Embedded Operating System Projects

Uwe Hentschel, Daniel Richter, Andreas Polze (Eds.)

Technische Berichte Nr. 90

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 90

Uwe Hentschel | Daniel Richter | Andreas Polze (Eds.)

Embedded Operating System Projects

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2014
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-296-4

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URL http://pub.ub.uni-potsdam.de/volltexte/2014/6915/
URN urn:nbn:de:kobv:517-opus-69154
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-69154

Contents

List of Figures

List of Tables

Listings

1 Introduction

This technical report presents results from the lecture "Operating Systems for Embedded Computing" that has been offered by the "Operating Systems and Middleware" group at HPI in Winter term 2013/14. Focus of the lecture and accompanying projects was on principles of real-time computing. Students had the chance to gather practical experience with a number of different OSes and applications. Three outstanding projects are at the heart of this technical report.

In today's life, embedded systems are ubiquitous. However, within our curriculum, the focus is still on principles, development techniques, best practices, and tools that are mainly targeted at traditional desktop systems. Embedded (operating) systems are different in many aspects. These include predictable timing behavior (real-time), the management of scarce resources (memory, network), reliable communication protocols, energy management, special purpose user-interfaces (headless operation), system configuration, programming languages (to support software/hardware co-design), and modeling techniques.

The lecture on "Operating Systems for Embedded Computing" has discussed design decisions and trade-offs of current embedded operating systems. We have studied algorithms to manage resources, such as CPU, memory, network – together with constraints imposed by the underlying hardware and environment. Configuration of operating systems was another important aspect addressed by the lecture.

In order to gain practical experience with real-time systems, the "Distributed Control Lab" has been set up and operated by the "Operating Systems and Middleware" group for a number of years. In line with the ideas of this Lab, numerous student projects have been carried out.

Within this technical report, authors present experiences with near-hardware programming. Projects address the entire spectrum, from bare-metal programming to harnessing a real-time OS to exercising the full software/hardware co-design cycle.

Project 1 focuses on the development of a bare-metal operating system for LEGO Mindstorms EV3. The new EV3 is a fascinating piece of machinery. While still a toy, it comes with a powerful ARM processor, 64 MB of main memory, standard interfaces, such as Bluetooth and network protocol stacks. EV3 runs a version of

Linux. Sources are available from Lego's web site. However, many devices and their driver software are proprietary and not well documented.

Developing a new, bare-metal OS for the EV3 requires an understanding of the EV3 boot process. Since no standard input/output devices are available, initial debugging steps are tedious. After managing these initial steps, the project was able to adapt device drivers for a few Lego devices to an extent that a demonstrator (the Segway application) could be successfully run on the new OS.

Project 2 looks at the EV3 from a different angle. The EV3 is running a pretty decent version of Linux. And the pioneering work on Real-time Linux from 10 years ago has found its way into today's standard Linux distributions. In principle, the RT_PREEMPT patch can turn any Linux system into a real-time OS by modifying the behavior of a number of synchronization constructs at the heart of the OS.

Priority inversion is a problem, that is solved by protocols such as priority inheritance or priority ceiling. Real-time OSes implement at least one of the protocols. The central idea of the project was the comparison of non-realtime and real-time variants of Linux on the EV3 hardware. A task set that showed effects of priority inversion on standard EV3 Linux would operate flawlessly on the Linux version with the RT_PREEMPT-patch applied. If only patching Lego's version of Linux was that easy...

Project 3 takes the notion of real-time computing more seriously. The application scenario was centered around our Carrera Digital 132 racetrack. Obtaining position information from the track, controlling individual cars, detecting and modifying the Carrera Digital protocol required design and implementation of custom controller hardware. What to implement in hardware, firmware, and what to implement in application software – this was the central question addressed by the project. Bottom line here: development of custom hardware and firmware programming are tedious tasks. However, once accomplished, they help immensely with the programming of (real-time) control software.

The usage of standard operating systems (Windows/Windows Embedded) to control real-time systems is quite possible with the help of some extra hardware and firmware – that has been demonstrated by project 3.

Addressing the entire spectrum, from bare metal system development to modification / adoption of existent OSes to real-time control with commercial off-the-shelf OSes – these three projects are good representatives for many problems and tasks developers of embedded systems have to face everyday. Our students had to understand the topic area, make design decisions, master tools and development techniques, implement, demonstrate, and evaluate their systems.

The result is outstanding!

2 Development of a Simple Operating System for LEGO Mindstorms EV3

2.1 Introduction

The *Mindstorms EV3* is *LEGO*s third generation of battery-powered embedded devices designed for explorative programming and robotics with LEGO bricks. It exposes a programmable interface to a number of different sensors and motors, allowing LEGO creations of arbitrary behavioural complexity. In contrast to its predecessors, which were shipped running custom firmware, the Mindstorms EV3 runs a modified *Ångström Linux Distribution*, the complete source code of which, including the source of the modified *U-Boot* boot loader, has recently been released as open source. In addition to the general-purpose Operating System, the Mindstorms EV3 has received a considerable upgrade in computing power when compared with the previous generations. While the *Mindstorms NXT 2.0* contained a 48MHz *ARM7*

CPU, the Mindstorms EV3 contains a more powerful 300MHz *ARM9 CPU*, more precisely the *AM1808 CPU* by *Texas Instruments*, as can be seen in figure 2.1. The datasheet [1] and the programmers reference manual [2] for the AM1808 CPU are available as resources.

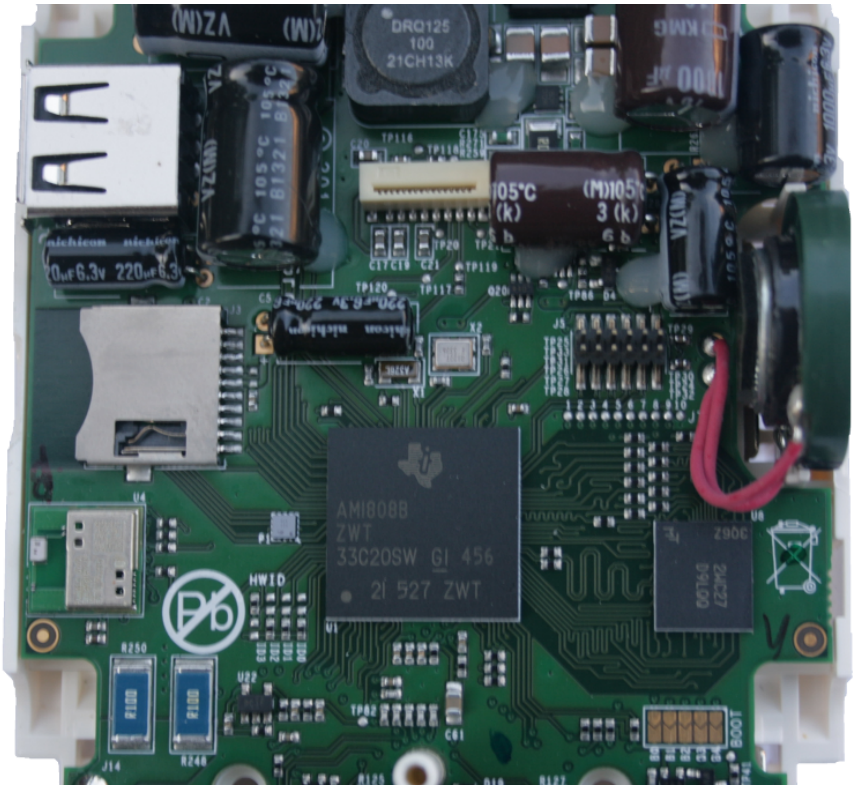

Figure 2.1: A view on the CPU and internals of the LEGO Mindstorms EV3

Based on this platform, we aimed at the creation of a simple Operating System capable of controlling the device's hardware, as well as a subset of the available periphery devices, to perform an example real-time task. The result of our endeavours was manifested as an open source repository, which is available on github via `http://github.com/ev3ninja/ninjastorms`.

In the following sections, we outline our exploration of the platform, as well as the creation of our development tools and deployment process, and give an insight in our experiences with the development in a bare-metal environment. Additionally, we describe the architecture of the resulting Operating System, and explain a subset of its more interesting aspects in detail. Finally, we outline an experiment we

have created based on our system, and discuss some of the possibilities of future extensions to our work.

2.2 Development and Deployment Process

During our research on the properties and capabilities of the Mindstorms EV3 platform, we discovered that the EV3 provides an interactive serial console interface on the first Sensor port [7]. In order to utilize this console, we soldered a *UART* to *USB* converter with 3V logic to a stripped EV3/NXT cable, and trimmed the unused wires, as illustrated in figure 2.2.

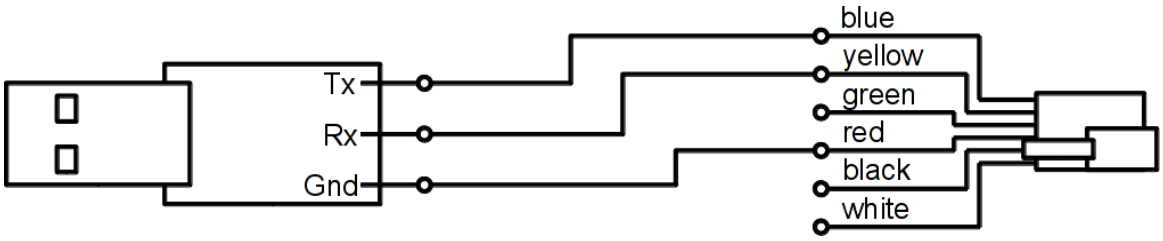

Figure 2.2: A UART to USB converter, where ground is connected to the third (red) wire, Rx is connected to the fifth (yellow) wire, and Tx is connected to the sixth (blue) wire of an EV3/NXT cable.

The resulting end-to-end coverter was used to connect the first sensor port of the EV3 to a USB port on the host machine, which exposed a serial interface device on the host, if configured properly. On a typical unix-like system, this device node is called `/dev/ttyUSB*`. To set up a stable bidirectional communication via this device, we configured it to a baudrate of 115200, 8 databits, no parity and 1 stopbit, by setting the appropriate options in a serial console emulator, for example *PuTTY*. Via this connection, it was possible to intercept the output of the bootloader of the EV3 once the device was powered on.

However, when we began testing the connector, we had the problem that our console only showed the first line of the bootloader output. Intrigued by the missing data, we decided to capture the potential between ground and the data send channel of the serial console using an oscilloscope, where we could clearly see a drop in the signals voltage. Using a multimeter, we measured the resistance between the EV3s ground and the channels of the first sensor port, and noticed that the guide we used to construct our connector indicated a wrong channel to be connected to ground. This resulted in a slow drop in the logic current of the serial consoles received data,

until the signal was no longer recognized by the UART converter, which happened near the end of the first line of output. After this discovery, we connected the ground to the right channel, which produced a connector that worked reliably.

The received output looked similar to the following:

```
EV3 initialization passed!
Booting EV3 EEprom Boot Loader

        EEprom Version:   0.60
        EV3 Flashtype:    N25Q128A13B

  [...]

Hit 'l' to stop autoboot:  0

  [...]
```

This implied to us that the boot process of the EV3 was interruptable. Upon pressing `l` during boot, we discovered that the boot loader exposed an interactive shell, allowing us to explore its behaviour and configuration. We identified the boot loader as a *U-Boot* instance, which is generally configured using predefined environment variables, which were accessible using the built-in command `printenv`. An excerpt of this configuration is included below:

```
U-Boot > printenv
  bootcmd=if mmc rescan 0; then if run loadbootscr; then run \
    bootscript; else if run loadimage; then run mmcargs; run \
    mmcboot; else run flashargs; run flashboot; fi; fi; else \
    run flashargs; run flashboot; fi
  loadbootscr=fatload mmc 0 ${bootscraddr} boot.scr
  bootscript=source ${bootscraddr}
  loadimage=fatload mmc 0 ${loadaddr} uImage
  mmcboot=bootm ${loadaddr}

  [...]
```

During boot, U-Boot evaluates the `bootcmd` variable as a set of instructions on how and what to boot, to conditionally bring up the desired system. The following pseudocode describes the preconfigured behaviour:

```
1 if sd card is inserted
2   if boot.scr file exists on sd card
```

```
    load and source boot.scr
  else
    if uImage file exists on sd card
      load and boot from sd card
    else
      load and boot from flash
    fi
  fi
else
  load and boot from flash
fi
```

This configuration implied that we could provide the EV3 with bootable sd cards injecting arbitrary code to the platform, just by inserting a properly deployed FAT32 formatted MicroSD card. This was very helpful in the creation of our operating system, as it allowed us to test code on the device without removing or damaging the existing operating system, as we did not need to access the flash memory at all to boot our application. We think the reason for this design decision may have been to provide a firmware update mechanism, or it may be a remainder of a development and deployment mechanism that we just happened to rediscover.

However, in order to deploy applications to the EV3, we needed to be able to compile code for the platform using our development machines, which required a cross-compile toolchain for bare-metal ARM targets. The maintainers of the EV3 stock image github repository suggest using a version of the CodeSourcery pre-compiled toolchain for ARM targets [5]. Unfortunately, this toolchain was intended for glibc based linux targets, and caused linker issues when used for bare-metal targets, where some glibc provided functions expected by the compiler were not present at link time. This forced us to create a custom cross-compile *GNU* toolchain from scratch later during the project, but fortunately this is a well documented and straightforward process. Based on a guide by Adam Kunen [6], we created an automated shell script to create and install a cross-compiler for bare-metal ARM targets on a GNU/Linux host system, which is included in the github repository of our project, as well as in the appendix of this document for further reference.

Using the C compiler of this toolchain, we were able to generate *ELF binaries* for ARM from C code. Unfortunately, the U-Boot instance on the EV3 was compiled without the `bootelf` command, so we needed to use the `objcopy` of our toolchain to convert the ELF binary to a plain binary file and used the `go` command of U-Boot to jump to the entry point of our code. From this, we created a batch script for U-Boot instructing the boot loader to load the binary to the proper adress in memory and `go` to our entry point. This batch script was then converted to a bootable `boot.cmd` file using the U-Boot utility `mkimage`. Copying the `boot.cmd` to the MicroSD card together with our compiled binary completes the deployment process.

The complete compilation and deployment process was slightly more complicated, as we needed to find out the entry point to our code dynamically at compile time, and to instruct the linker about the load address on the EV3, but is explained in more detail in Section 3, and also contained in the top-level Makefile of the project in its entirety.

2.3 Creating a Simple Self-Contained Application

Concurrently to the creation of our toolchain and deployment process, we also looked into the creation of a simple, self-contained example application to run on the EV3 platform, in order to validate that the assumptions we made during our exploration and research were correct. The canonical simplest possible application in the world of computer science is a program that writes `"Hello World!"` to a well-defined output sink and then terminates. To find out how to accomplish this, we searched through the source code of the Operating System running on the EV3, which was available on github [3]. Looking at the main Makefile of the project, which was located in the subdirectory `lms2012/open_first`, we found a section responsible of configuring and building a modified version of the U-Boot boot loader, which was also contained in the repository in the subdirectory `extra/uboot-03.20.00.13`. This section was completely independent of the rest of the source code, allowing us to extract the boot loader sources from the repository, and to use the information gathered from the Makefile to properly configure this instance of U-Boot for the EV3 outside the scope of the LEGO project sources.

Upon extracting the U-Boot sources from the LEGO project repository, we needed to configure the sources. U-Boot expects to be configured for a certain platform before initiating the build process, which is done by calling `make` and passing the toolchain to use and the target board as parameters. In our case, the appropriate configure and compile calls looked similar to the following:

```
$> make CROSS_COMPILE=arm-none-eabi- da850_omapl138_evm_config
$> make CROSS_COMPILE=arm-none-eabi- all
```

U-Boot provides a standalone application example, intended as a demonstration on how an application would behave if executed outside of an Operating System context, we utilized to create our example application. We extraced the sources of the `hello_world.c` file, the `stubs.c` file, and the configured U-Boot include tree into a separate project, as the standalone example heavily depends on U-Boot for system calls and clones of standard library functions like `printf`. From there on, we needed to break down this standalone example to the bare necessities, to be able to lay the foundations of our own project. We extracted the necessary build flags

from U-Boots Makefiles, eliminated unnecessary or unsed includes and removed unneeded files from the include tree, until we boiled it down to its base parts, which are described in detail in the following subsections.

2.3.1 hello_world.c

The `hello_world.c` file contains all the logic required to print the string "hello world" to `stdout` in the following function:

```
// entry point
int
hello_world (void)
{
  puts("hello world");

  return 0;
}
```

However, since we developed in a bare-metal context, providing an implementation for the `puts` function was non-trivial. For this, we needed a prototype declaration of the function, and we also needed to reference the actual implementation provided by the running U-Boot instance via a stub. The function prototype can be delared as expected:

```
void puts(const char*);
```

Providing the function stub is more difficult, and requires some ARM GCC inline assembly, as well as the layout of U-Boots main global datastructure. We could export more functions here, as U-Boot contains a large subset of the library functions as described by the C standard, but our goal was to reduce the amount of necessary code, hence we only exported `puts`, which is the fifth function in U-Boots function jump table.

```
// U-Boot global data layout
typedef struct  global_data {
  /*bd_t*/ void   *bd;     // simplified, assuming
                           // pointers of equal size
  unsigned long   flags;
  unsigned long   baudrate;
  unsigned long   have_console;
  unsigned long   env_addr;
  unsigned long   env_valid;
  unsigned long   fb_base;
```

```
  void            **jt;    // jump table
} gd_t;

// id of 'puts' in U-Boot function table
#define XF_puts 4

#include <stddef.h>     /* required for offsetof */

// provide the 'puts' asm stub
void __attribute__((unused)) dummy(void)
{
asm volatile (
  " .globl puts \n"
  " puts :\n"        \
  " ldr ip, [r8, %0]\n"   \
  " ldr pc, [ip, %1]\n"   \
    : : "i"(offsetof(gd_t, jt)),
        "i"(XF_puts * sizeof(void *)) : "ip");
}
```

This piece of source code exports a global symbol to the `puts` function as provided by U-Boot, by calculating the address of the memory address of the function in U-Boots in-memory jump table, which allowed us to directly use `puts` in our code. The complete file is contained in the Appendix of this document.

2.3.2 Building and Deployment

Despite the code of this hello world example being quite simple, the building process was fairly complex, due to the nonstandard way the application was going to be deployed and executed. Before we were able to compile, we needed to specify the prefix of the toolchain and store it in a variable, to decouple the build process from the toolchain. In our case the respective command in GNU make syntax was similar to the following:

```
PREFIX = arm-none-eabi-
```

Using this variable, we needed to find out the include directory of the toolchain's *gcc* by invoking:

```
LIBGCCDIR = $(shell dirname $(shell $(PREFIX)gcc \
              -print-libgcc-file-name))
INCGCCDIR = $(LIBGCCDIR)/include
```

The `INCGCCDIR` should be passed to the compiler using `-isystem` to specify the proper system include path for the compile calls. The complete listing for the compile call of the application including the compiler flags we used is as follows:

```
$(PREFIX)gcc -g -O2 -pipe -fno-common -msoft-float \
  -fno-builtin -ffreestanding -nostdinc -isystem \
  $(INCGCCDIR) -marm -mabi=aapcs-linux -march=armv5te \
  -mno-thumb-interwork -fno-stack-protector -Wall -Wextra \
  -Wstrict-prototypes -Werror -o hello_world.o \
  hello_world.c -c
```

Several of these flags, including `-msoft-float`, `-marm`, `-march=armv5te` are specific to the target ARM architecture, others including `-ffreestanding`, `-nostdinc` and `-fno-builtin` are due to the bare-metal nature of this application, to isolate the build environment from the libraries and header definitions present on the host system. We extracted most of the compiler flags from the U-Boot Makefiles, but altered them slightly during the process of our project, as our understanding of the architecture and the toolchain developed and new options opened up. For example, when we used the CodeSourcery Toolchain instead of our self-compiled one, it was not possible to use any optimization flags, like `-O2`, because there would be linker issues in the provided libgcc like the following:

```
/usr/local/bin/../lib/gcc/arm-none-linux-gnueabi/4.3.3/libgcc.a
  (_dvmd_lnx.o): In function `__aeabi_ldiv0':
(.text+0x8): undefined reference to `raise'
```

This was due to the fact that the bare metal linkage that we did would not work with the libgcc provided by CodeSourcery, when there was the possibility that the divisor of an integer division would be zero. The ARM architecture we were working on did not provide a hardware integer division, which is why the libgcc provided a software implementation as replacement, which relied on the `raise` capabilities of the *glibc* for divisions by zero. Since there was no glibc in our linkage, the errors were raised by the linker.

The next step after compiling `hello_world.c` to `hello_world.o`, was linking. The linker call was particularily interesting since we had to specify a custom entry point, because the entry point of our application was not necessarily called `main`. Additionally, we needed to provide the linker with the base address where the boot loader was going to load the application to, because there was no virtual memory management in the context of our bare metal application. The actual linker call we ended up using was similar to the following:

```
$(PREFIX)ld -o hello_world -e hello_world hello_world.o -g \
  -Ttext 0xC1000000 -L$(LIBGCCDIR) -lgcc
```

In this call, the `-e` switch defined the symbol name of the entry point, and `-Ttext` defined the starting address of the `.text` segment, where the application expected its instructions to start during runtime. This was for example important for the calculation of absolute jumps, which are calculated relative to the starting point of the program.

An examination of the resulting `hello_world` binary using the `file` utility yielded:

```
hello_world: ELF 32-bit LSB executable, ARM, EABI4 version 1
  (SYSV), statically linked, not stripped
```

This ELF binary had to be converted to a plain binary in order to work with the version of U-Boot running on the EV3, because it did not contain the `bootelf` command, as described before. The appropriate `objcopy` call for our example was similar to the following:

```
$(PREFIX)objcopy -O binary hello_world hello_world.bin
```

This yielded the binary that was eventually deployed and booted on the EV3, however, we still needed to create the `boot.scr` file before deployment. For this, we needed to find the entry point of the application, in other words the address of the function we want the program to begin with, which is usually called `main`. However, as there is no Operating System to load our application, and the information on the entry point was lost during the `objcopy`, we need to recover this information. There are several ways to do this, one of which is to disassemble the ELF binary, and to use the `grep` utility to find the address of the function we expect to be entry point. An example code for this could look similar to the following, again in Makefile syntax:

```
ENTRY = hello_world
ASM = hello_world.asm
$(PREFIX)objdump -d hello_world > hello_world.asm
ENTRY_ADDR = 0x$(shell grep '<$(ENTRY)>' $(ASM) | head -n1 | \
                  cut -d' ' -f1)
```

Using this information, it became possible to generate the `boot.cmd` file as follows:

```
echo "fatload mmc 0 0xC1000000 hello_world.bin" > boot.cmd
echo "go $(ENTRY_ADDR)" >> boot.cmd
```

which instructed U-Boot to load the `hello_world.bin` file to the specified address in memory, which in our case was located in the lower regions of the EV3 RAM, and jump to our specified entry point. An interesting side effect of this method is that after execution of our application, the EV3 will return back to the boot loader console, which will probably cease to work properly, because our code may have

clobbered the register containing the address of U-Boots global datastructure. If we wanted to preserve this, we could add the appropriate flag to the compiler flags, but since the dependencies to U-Boots C library functions were only temporary, we decided to ignore this issue for the time being.

After generating the `boot.scr` from the `boot.cmd` file, and copying both files `boot.scr` and `hello_world.bin` to a FAT32 formatted MicroSD card, and booting the EV3 with this card inserted, the serial console captured output similar to the following:

```
  [...]
reading boot.scr

127 bytes read
## Executing script at c0600000
reading hello_world.bin

60 bytes read
## Starting application at 0xC100000C ...
hello world
## Application terminated, rc = 0x0
U-Boot >
```

Which implies that our application was loaded and executed correctly, as the string `hello world` is visible in the output of the serial console. These were very exciting results, because they showed that our understanding of the EV3 platform was correct, and that our build and deployment process behaved the way we intended them to.

Note that the size of the resulting binary file is a mere 60 Bytes. An application with similar functionality compiled for a standard GNU/Linux target may be up to 8 Kilobytes large. Also, the U-Boot console unexpectedly still worked after our simple example terminated, but this will probably not be the case for more complex applications that clobber the U-Boot global datastructure address register.

An example Makefile for the complete build and deployment process is contained in the repository of this project, and additionally a simplified version is included in the Appendix of this document, to complement the `hello_world.c` file in the Appendix.

2.3.3 Removing U-Boot dependencies

In its described state, the standalone example application still depends on the `puts` implementation of U-Boot, as well as the presence of U-Boots jump table in memory and the address of this jump table in the registers of the ARM CPU. To remove these

dependencies, we needed to provide a `puts` implementation of our own, like the following:

```
int
puts (const char *s)
{
  while (*s)
    {
      putchar(*s);
      ++s;
    }

  putchar('\n');

  return 0;
}
```

This simplified our problem, but did not solve it because we were now lacking a `putchar` implementation. Fortunately, implementing `putchar` for the EV3s serial console interface in a bare-metal context was fairly straightforward. The reference manual of the CPU stated that the AM1808 contained three UART devices, which exposed memory mapped registers in different areas of the EV3s memory. Upon reading the configuration of the U-Boot sources, we found out that the UART device used in the serial console on the first sensor port was mapped to the memory region beginning at `0x01D0C000`. The manual also states that the offset of the *Transmitter Holding Register*(THR) to this base address is zero, and that the offset of the *Line Status Register*(LSR) to the base address is `0x14`. Using this information, we were able to implement a very simple `putchar` as follows:

```
#define UART_THR (volatile char*)(0x01D0C000)
#define UART_LSR (volatile char*)(0x01D0C014)

int
putchar (int c)
{
  if (c == '\n')
    putchar('\r');

  while (!(*UART_LSR & (1 << 5)));

  *UART_THR = c;
  return c;
}
```

This version included the output of an addition carriage return upon encountering a line break, because we had some issues with our terminal emulators, where the terminal would not acknoledge an implied carriage return upon line break, and instead kept the indentation of the previous line.

Using these two functions, we removed the U-Boot dependency, effectively creating the simplest possible independent example application for our bare-metal ARM target, including a working toolchain and deployment process. This enabled us to move on to more complex tasks with maximum freedom as well as a solid understanding of the platform fundamentals, and to theoretically gain complete control of the platform. Again, a complete version of the `hello_world.c` file without U-Boot dependencies is contained in the appendix of this document, and can be compiled using the same Makefile as before.

2.4 Architecture Considerations

Building upon the simple example described before, we developed a fairly complex system, the architecture of which is going to be described in detail in this section. Our system runs directly on the device hardware, using memory mapped registers as means of communication and interaction with the hardware. After the boot process is complete, we assume to be the only code operating the device, and expect the hardware to react appropriately.

From an abstract point of view, our system consists of three basic components, each of which is described in detail in a dedicated section below. Interfacing with the hardware via the memory mapped registers is the *Periphery Control Libray*(libp). The libp contains very basic versions of the device drivers and device interface providers we needed, and was intended to provide a level of abstraction to the underlying hardware, and a procedural interface to basic hardware functionality. Complementing the libp, we implemented a subset of the C standard library functions we needed as described by various standards. We updated this rudimentary libc with new functions as they were required, but while we tried to keep to the standards as much as possible, the specific use-case of our system allowed some simplifications, which are outlined below. The third aspect of our system is a simple C runtime which in the current state of the system fulfills some of the roles of the C runtime, but is also used as a container for the application logic. The libc and libp are compiled into shared libraries, which are linked statically to the application code and the C runtime, resulting in a single binary file for deployment. This implies that for an updated application code, the operating system has to be recompiled and -linked as well, which creates some deployment overhead. However, the overall compilation time was short enough to allow for that, and it simplified the system layout

significantly when compared to implementing a persistant operating system with a dedicated application loader.

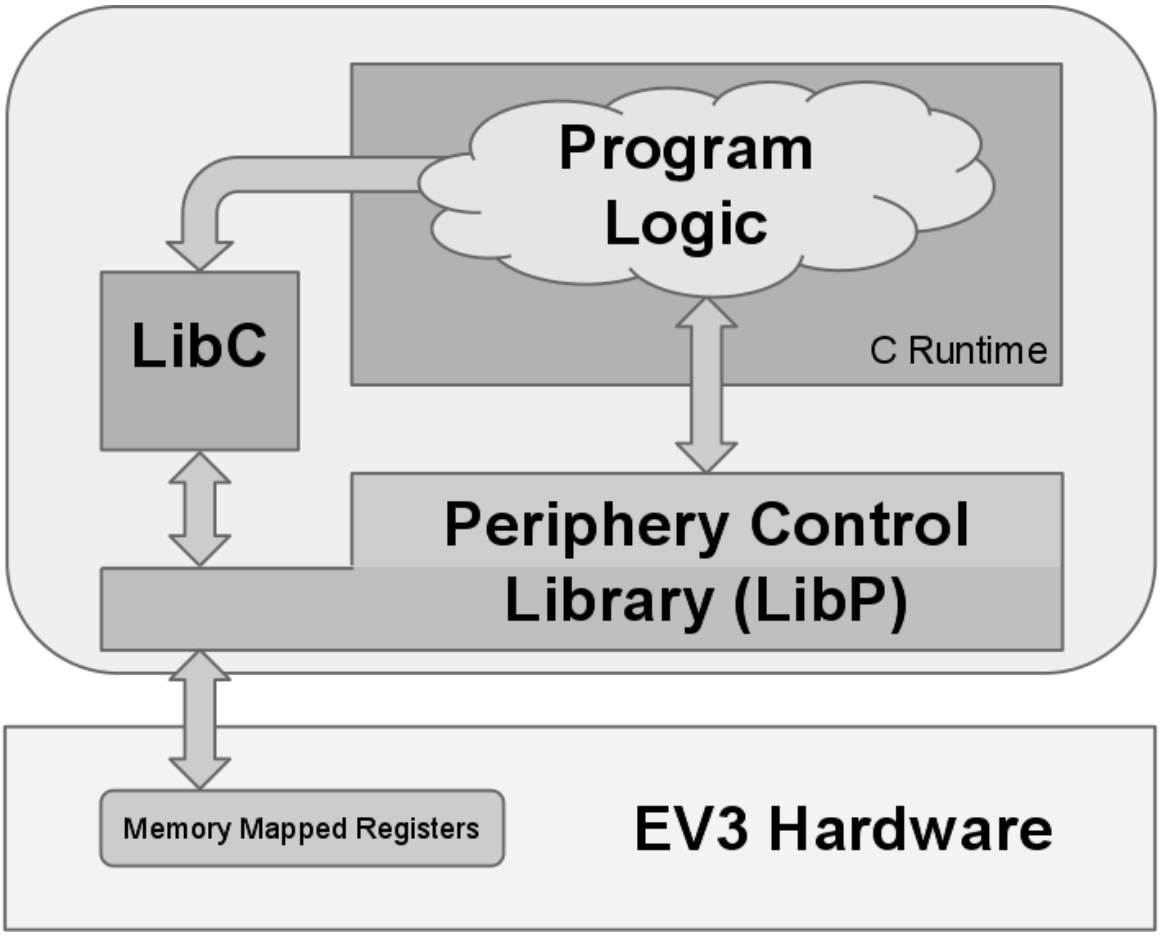

Figure 2.3: The architecture of the Operating System, coarsely divided into functional blocks, the large upper box represents the project source code separated into application logic, libc and periphery control library, while the lower box represents the platform hardware, where memory mapped registers are used as communication medium between hardware and software.

2.4.1 C Library Implementation

The C standard defines a lot of functions, and most C library implementations contain extensions to a certain degree. However, since our system was intended to be quite simple, we did not necessarily need the full power of a feature-complete C

standard library, allowing us to omit most of the more advanced functions and concepts. One of the more notable aspects we left out it the concept of file descriptors, and files in general. We also left out anything regarding virtual memory management. This was due to the fact that file systems and files in general are non-trivial concepts, that we did not need to introduce into our system yet, and that would take a considerable amount of work to implement properly. The same statement holds for virtual memory management. Since our application is the only code running on the platform, the concept of virtual memory is not essential to the execution of our code, as there is no need to separate processes or hide physical memory functionality, like memory mapped registers, from user code.

What we did need to implement was a subset of `stdio.h`, specifically character and string output functions. Since the serial console via the UART adapter was initally the only means of communication we established with the device, it was essential to have a more sophisticated output method than `putchar` and `puts`, which is why we implemented a feature-incomplete version of `vprintf` and `printf`. This allowed us to print numerical values as well as formatted strings to the serial console, instead of mere character arrays using `puts`. During the development of our simplified `vprintf`, we noticed that we were initially lacking a true understanding of the inherent complexity of the functions of the `printf` family. Our `vprintf` clone was only capable of printing characters, strings, signed integers, and hexadecimal integers, excluding formatting like leading zeros or length specifiers, and already spanned more than 150 lines of code. We found this astonishing and it lead us to develop a new kind of respect for the convenience provided by some parts of the C standard library.

Beyond these output functions, we also implemented a subset of the functions of the `mem*` family, in particular `memcmp`, `memcpy` and `memset`. These functions may be expected by the compiler to exist at compile time and may calls to them may be sliently added when manipulating stack based array. An example for this can be seen in the following code, where the compiler silently inserted a call to `memset`, ignoring the possibility that memset may not exist:

```
void
function (void)
{
  int arr[1000] = { 0 };
}
```

This is evident in the disassembly of this function:

```
c1000030 <function>:
c1000030: e1a0c00d  mov ip, sp
```

```
c1000034: e92dd800  push {fp, ip, lr, pc}
c1000038: e24cb004  sub fp, ip, #4 ; 0x4
c100003c: e24ddefa  sub sp, sp, #4000 ; 0xfa0
c1000040: e24b3efa  sub r3, fp, #4000 ; 0xfa0
c1000044: e243300c  sub r3, r3, #12 ; 0xc
c1000048: e3a02efa  mov r2, #4000 ; 0xfa0
c100004c: e1a00003  mov r0, r3
c1000050: e3a01000  mov r1, #0 ; 0x0
c1000054: ebffffe9  bl c1000000 <memset>
c1000058: e24bd00c  sub sp, fp, #12 ; 0xc
c100005c: e89da800  ldm sp, {fp, sp, pc}
```

The absence of these functions caused occasional linker errors, which is why we chose to add them to our C library implementation. Generally speaking, we tried to follow our usual workflow when developing the system and application code, disregarding the fact that several C library functions were missing, and each time a function required by us or the compiler was undefined, we added it to our C library implementation, growing it in the process.

We tried to minimize the amount of direct interaction of C library functions with the hardware, and instead to use libp functions for hardware access, to keep the layered structure of our system as clean as possible. In fact, the only function of the C library thar interferes with the hardware directly is `putchar`, when writing directly to the UART console memory registers. We decided against abstracting from this, because the actual abstraction is going to be part of the libc as soon as a means of defining input and output streams is added, and the `putchar` implementation is forced to adhere to the properties of the given `stdout`.

2.4.2 Reverse Engineering Platform Device Drivers

The final goal of our project was to be able to control the LEGO sensors and motors from our system in a way that would enable us to set up a simple real-time based experiment. However, the road to this goal was considerably longer and harder than we expected, and we only partially succeeded.

To create the drivers, we used mainly two resources. For the devices of the AM1808 CPU, we found all required information in the reference manual of the CPU, but for the higher level devices like the LEDs on the front of the EV3, we needed to reverse-engineer the necessary information from the original LEGO source code and to use a lot of trial-and-error debugging. This was aggravated by the fact that the LEGO source code was partly obfuscated and overly complex which was due to the way it was developed and probably carried over from earlier hardware revisions and generations which were not running the Linux that was shipped with the EV3.

The LEGO drivers were implemented as kernel-space devices using high-resolution timer callbacks, and exposed their interfaces via device nodes to a higher level layer of LEGO code, which then interfaced with the frontend on the device that was exposed to the user via the LCD screen on the front of the EV3. Another resource we had access to were the circuit plans of the Mindstorms NXT 2.0 hardware as well as of some of the simpler sensors, and motors.

As a proof of concept, the first periphery device we wanted to be able to control were the LEDs on the front of the EV3. The communication of our code with the AM1808 CPU hardware was implemented using memory mapped registers. Some of these memory mapped registers control a section of the chip called *General-Purpose I/O*(gpio), which is one of the interfaces used to communicate with hardware outside of the CPU, for example the LEDs. Interfacing with the gpio ports is fairly straightforward, except for the fact that these ports are multiplexed. In addition to writing the correct values to the registers corresponding to the port data, we also needed to properly configure the ports as input- or ouput-ports, as well as properly configuring the pulldown resistors of the ports.

Beyond the difficult task of understanding the elctrical engineering aspects of hardware interaction, we also had to reverse engineer the LEGO source code to find out behind which ports the LEDs were listening. This was especially difficult, because the LEGO sources contained different port descriptions for multiple hardware revisions, which meant that in case of an error we did not know if our gpio interface code was the problem or if we simply interfaced with the wrong ports. In actual fact, we spent several days after getting one of the two front LEDs to work debugging our gpio implementation, while the real problem was that we used the wrong gpio ports, and just got the first LED working out of sheer luck. Debugging this kind of problem can be very frustrating, as there initially was no point of reference that could be taken as a ground truth, and the hardware continuously degraded to a blackbox whose state was not known, and whose behaviour was not fully understood. Following the LEDs, we implemented support for the buttons on the front of the EV3, which included extending our gpio driver from a write-only to a read/write driver, but apart from this was a very straightforward process.

After finishing the LED and button control, we started looking into the development of control code for the LEGO sensors and actuators. Again, we faced the problem of reverse engineering thousands of lines of code to find out the main points of sensor and motor control, but fortunately we had an additional resource. We had access to the circuit schematics of the Mindstorms NXT 2.0 and a set of sensors and motors of the NXT 2.0 generation. These schematics showed us how the devices interfaced with the I/O ports on the EV3, and the LEGO sources described which sensor port was mapped to which gpio ports. Using this information we were able to capture the state of the appropriate gpio ports during the sensor operation. However, there were generally two kinds of sensors. One kind of sensors exposed its

state as a continous value of voltage on a specific gpio port, requiring us to write a driver for the *analog/digital converter*(adc) of the AM1808 to be able to interface with these devices. Examples for sensors of this kind were the touch sensors, and the NXT light/distance sensor. However, the sensors differed in the interpretation of these signals, which meant that we needed to write different drivers that interpreted the values captured by the gpio and the adc differently. Additionally, the LEGO sources also identified which kind of sensor connected to a port, using a complex scheme of port states and voltages. We chose to omit this functionality, and instead use dedicated interfaces for each sensor type, which imples that the application needs explicit knowledge on the kind of sensor or actuator attached, and risked invoking undefined behaviour in case a wrong function is used, which was acceptable for our use case.

The second kind of sensors, which were mainly the newer sensors of the EV3 generation, were more complicated. They did not expose their values via continuous voltages on the gpio ports, but instead contained an I^2C bus controller connected via gpio to the linux kernel modules of the LEGO source. The AM1808 contains an I^2C bus controller abstracting away from the timing constraints of the bus protocol, but unfortunately LEGO decided to use the software bus driver implementation contained in the linux kernel instead. We were unable to create an I^2C bus driver implementation due to time limitations and the functionality requirements of such a driver, which meant that we could not interface with the modern EV3 sensors.

After we finished working on the NXT style sensor drivers, we implemented the motor driver which worked by similarily to the LED controller, just by switching gpio pins on and off. To implement different levels of speed, we made it the applications responsibility to turn the motor off and on in the appropriate intervals to basically implement pulse width modulation.

The full source code of all of the device driver implementations can again be found in the project repository on github.

2.4.3 Emulating GCC's Constructor Function Attribute

Application and library state initialization is a non-trivial process. Optimally, initialization code for unused modules coule be omitted, and the initialization should be processed before the program enters the application logic to ensure correct behaviour during runtime. Additionally, sometimes a library module requires another module to be required before it starts its initialization.

This functionality is provided by an extension of the GNU toolchain to the C language, which are the `constructor` and `destructor` *function attributes* [4]. The syntax is as follows:

```
void
```

```
__attribute__ ((constructor))
initialization (void)
{
  // this function is executed before main
}
```

The ELF binary contains dedicated sections to implement this feature. Functions declared `constructor` are placed in the `.init_array` section of the binary, and functions declared `destructor` are placed in the `.fini_array` section. Upon execution of the binary, the C runtime as provided by the C library may honour these section by executing the functions from the `.init_array` in unspecified order before jumping to the applications entry point, and executing the functions from the `.fini_array` once the application returns from `main` or calls `exit`. If the application terminates abnormally, for example via a call to `abort` or through a signal, the `.fini_array` is ignored. Additionally, it is possible to give the constructurs a well-defined order by providing an integer argument to the attribute, where constructors with lower argument are executed before constructors with higher argument.

This mechanism is very useful for application and library state initialization and finalization for a number of reasons. The linker may discard the constructors of unused compilation units, effectively eliminating unused code from the binary. If the constructor code would be invoked explicitly, the linker could not detect unused compilation units und would not be able to optimize this. Additionally, it is possible to specify complex dependency graphs of constructors by giving them appropriate precedence arguments to ensure that higher-level constructors are always called after the constructors they depend on. If these constructors would just invoke the low-level constructor explicitly, there would be problems when several modules depend on the same constructor, becasue it may be executed multiple times.

The constructor and destructor function attributes are useful features for initialization and finalization, but only if they are honoured by the loader of the target platform. Unfortunately, the makeshift loader we implemented using U-Boot scripts does not support this. To remedy this, we needed to add another level of indirection above the applications entry point that emulates this functionality of the linker. This is what we called our *C Runtime*.

We altered our build and deployment process to point to a new entry point, which was a function that calls all functions declared constructor, calls the old entry point function and stores its return value, calls all functions declared destructor, and returns the old entry points return vaule. The linker of our toolchain exposes the addresses of the constructor and destructor array as magic constants, which means that the implementation of the initialization and finalization routines is fairly straightforward, for example a function for the initialization looked similar to the following:

```
static void
```

```
ev3ninja_runtime_init (void)
{
  size_t count;
  size_t i;

  count = __preinit_array_end - __preinit_array_start;
  for (i = 0; i < count; i++)
    __preinit_array_start[i] ();

  count = __init_array_end - __init_array_start;
  for (i = 0; i < count; i++)
    __init_array_start[i] ();
}
```

It should be noted here that the toolchain would not complain if we were to use the constructor function attribute, but not call these functions in our actual code, because the toolchain expects the execution environment of the compiled application to honour these conventions. The fact that our deployment process did initially not provide this functionality could have lead to silent failures of critical functionality, which would have been hard to debug. However, since we did implement this feature, we restored a functionality to the toolchain that it expected to have in the first place, and we also improved the optimization capabilities of the linker in the context of eliminiating unused code from the binary.

2.5 Experiment

After implementing the sensor and motor control in our system, we decided on a proof of concept experiment to demonstrate the capabilities of our system. Despite our problems with the I^2C bus controller and the resulting inability to interface with the gyroscope sensor or infrared distance sensor, we decided to create a self-balancing two-wheeled robot based on the idea of a segway, or the inverted pendulum, the physics of which is described in figure 2.4. For this, we created as robot, using a set of two light sensors, which we used as distance sensors assuming even ambient light, uniformly reflective surfaces and low direted and specular light. To be able to balance, we moved the center of mass of the segway a bit to the top, allowing the wheels to balance below the center of mass.

The robot was controlled using a simple PID controller implemented in C. A PID controller is a basic building block capable of stabilizing different kinds of systems by providing different paramters. The main idea of a PID controller is that it is possible to stabilize a system by controlling an actuator based on the measurement

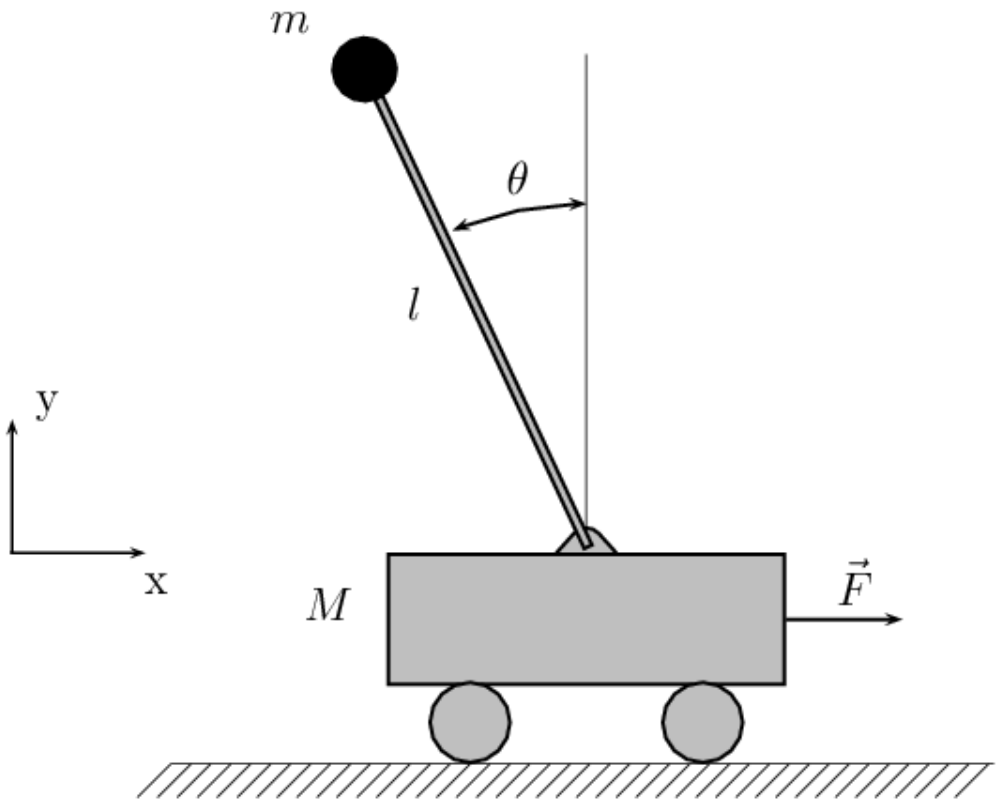

Figure 2.4: The inverse pendulum in the context of a segway - The mass of the segway is concentrated at the top of the pendulum, while the bottom is free to move and keep the surface area connecting to the ground below the center of mass to keep the balance.

between the current system state and a desired system state. The actual difference between the system state and the desired state is called p, the differential over the difference is called d, and the integral over the difference is called i. Each of these aspects of the PID controller is assigned a constant K_p, K_d and K_i, and the output is then calculated as $K_p * p + K_d * d + K_i * i$. This simple principle is capable of stabilizing a surprising variety of systems, if configured properly.

The p part of the controller brings the actual difference in state into the equation. The d part of the controller brings the differential of the difference, that is, the rate of change into the equation. This allows a properly configured controller to overshoot less, as high rates of change can be translated to less output force if the controllers state is near the desired state. The i part of the controller brings accumulated state difference into the equation, which allows the segway robot to not only stabilize, but theoretically also should allow for second order stability, where the robot tends to stay close to the spot it started on. Unfortunately, the ligh sensors we used were not precise enough for reliable segway control, however we managed to have it roughly keep its balance for a limited amount of time, which can be seen in the demo video we created, and which can be found at `https://www.youtube.com/watch?v=-t5TSZjHqMg`. At the end of the video, it can be seen that one of the light sensors moves from the brighter wooden tiles to a darker wooden tile, which introduced a shift of the brightness measurements between the two light sensors, as the reflective properties of the ground changed. This sudden change caused an unrecoverable balance failure for our robot.

Another problem we faced was that the speed control of the wheels was not very precise, and depended a lot on the current charge of the EV3s batteries.

The actual implementation of the PID controller in C is straightforward. Firstly, the controller needs to capture a difference between its current state and its desired state in a continuous loop. Then, the controller can approximate the differential of the difference as the difference beween the current and the last measurement, and can approximate the inegral over the difference as the continuous sum over the difference. Depending on the frequency of measurements, the value for K_i will probably need to be small as the sum over the differences grows quickly.

Generally, we considered the experiment to be successful, as we managed to build a self-balancing robot using LEGO bricks, although it was still very much dependent on environmental influences.

2.6 Conclusions

This project provided a very diverse and valuable learning experience, not only in hacking proprietary hardware. We dived into several topics, including but not

limited to reverse-engineering device drivers, creating PID controllers, electrical engineering, driver development and C standard library development, all of which most members of our team never touched before. Or knowledge of the C language intensified, and we developed a better understanding about the complexity of and the convenience provided by the C standard library functions. We deployed source code to a bare-metal ARM platform, for which we compiled a cross-compile toolchain by hand. We learned about hardware/software interfaces and debugging, and about specific tools, bootloaders and Operating System features

We found out that the Mindstorms EV3 is a surprisingly hackable platform, that can be applied to a very versatile range of applications, merely by properly provisioning a MicroSD Card. This property makes the EV3 a very valid platform for developing with, and learning about embedded systems, in a feature-complete operating system context, as well as on the bare metal.

At the beginning, our goals were the creation of a bare-metal Operating System capable of controlling the Mindstorms EV3s hardware and periphery, while being independent of existing source code on the device, and while also keeping it as simple as possible. Looking back, we think we achieved this goal. Our system is capable of controlling a self-balancing robot, only based on source code we wrote and that we kept as simple as possible. We also persisted the existing state of the project and the development and deployment process with various scripts, Makfiles and pieces of documentation, such that following generations of students can understand what we did and hopefully also why we did it.

We feel that this learning experience was definitely worth the effort and want to thank the Operating Systems and Middleware Chair at the Hasso Plattner Institute for its continuous support. In particular, we want to thank Jan-Arne Sobania for sharing his understanding on the ARM platform, and the U-Boot boot loader, Uwe Hentschel for his deep knowledge of electrical engineering and hardware debugging, Frank Feinbube for providing answers to any questions we dared asking, and of course to Prof. Andreas Polze for providing the Lecture, the Project, the Hardware and possibilities only limited by our abilities and timeframes.

2.7 Future Work

Our system was not designed to be a feature-complete general-purpose Operating System. Instead, we developed it to be as slim as possible, tailored to a specific purpose, while being as easily extendible as possible. As such, there are numerous possibilities for future projects to extend upon our system, some of which we want to list as suggestions below.

One large field we did not explore is the area of job control. Real Operating Systems have more than one task, and hence have to protect the operating system from the tasks, and the tasks from each other. This requires a lot of work and abstraction that we intentionally left out because of a lack of time. But generally speaking, the system would need a way to manage virtual memory, and to do context swiching between tasks and also to do scheduling. Also, depending on the level of interactivity desired the system might need some kind of shell or job control system, as well as an application loader.

All of this would probably require proper interrupt handling, which is also a field we have chosen not to look too deep into. If it would be possible to use the dedicated real-time timers of the AM1808 CPU for high-precision timers, the creation of a software I^2C driver would be possible, allowing for interaction with the more complex EV3 sensor devices.

Another field we did not have the time to explore is proper host-based remote debugging using gdb hooks connected via the serial UART console. This would enable more efficient debugging than the debug printing approach we used. Additionally, the system could use some amount of hardware independence, as it is currently very much limited to a single platform.

Generally, the possibilities to extend on the system we developed are almost unlimited, and the hardware is affordable. To enable anyone interested to extend upon our work, and as a complete reference to this document, we decided to release our work as open source on github via `http://github.com/ev3ninja/ninjastorms`. We hope that someone will find this useful, and would be glad to hear from anyone deciding to continue where we left off. Please consider writing us an email.

References

[1] *AM1808 ARM Microprocessor Datasheet*. URL: http://www.ti.com/lit/ds/symlink/am1808.pdf.

[2] *AM1808/AM1810 ARM Microprocessor Technical Reference Manual*. URL: http://www.ti.com/lit/ug/spruh82a/spruh82a.pdf.

[3] *EV3 modified Ångström Linux and U-Boot github Repository*. URL: https://github.com/mindboards/ev3sources.

[4] *Function Attributes - Using the GNU Compiler Collection (GCC)*. URL: http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html.

[5] *GitHub, (2013). mindboards/ev3sources*. URL: https://github.com/mindboards/ev3sources/blob/master/README.md.

[6] A. J. Kunen. *Building the GNU ARM Toolchain for Bare Metal*. URL: http://www.kunen.org/uC/gnu_tool.html.

[7] X. Soldaat. *EV3: Creating a Console Cable*. URL: http://botbench.com/blog/2013/08/15/ev3-creating-console-cable/.

2.A Appendix

2.A.1 cross-compile toolchain creation

```
#!/bin/bash

# this script should be able to generate an arm-none-eabi toolchain
# on your system. if not, try to fix it. :)
# tested on Gentoo Linux, and Ubuntu 12.04 LTS

# you may need to run this as root, depending on your configuration.
# basically, you just need full write permissions to the path
# specified in PREFIX.

# third party dependencies (probably incomplete):
#   - GNU make, gcc, ... (build-essential)
#   - texinfo

# this script was generated using this guide:
#   http://www.kunen.org/uC/gnu_tool.html

```

```
18 set -e
19 set -u
20 set -x
21
22 BUILDROOT=${BUILDROOT:-/tmp}
23 PREFIX=${PREFIX:-/usr/local}
24
25 # set make flags in environment, e.g. -j2 for parallel builds
26 MFLAGS=${MFLAGS:-}
27 # clean up previous build, if any
28 rm -rf $BUILDROOT/{src,build}
29 # create build directories
30 mkdir -p $BUILDROOT
31 mkdir -p $BUILDROOT/{orig,src,build}
32 # create install directory
33 mkdir -p $PREFIX
34
35 # fetch required packages, if necessary
36 cd $BUILDROOT/orig
37 if [ ! -f gcc-4.3.3.tar.gz ]; then
38   wget ftp://ftp.gnu.org/gnu/gcc/gcc-4.3.3/gcc-4.3.3.tar.gz
39 fi
40 if [ ! -f gcc-core-4.3.3.tar.gz ]; then
41   wget ftp://ftp.gnu.org/gnu/gcc/gcc-4.3.3/gcc-core-4.3.3.tar.gz
42 fi
43 if [ ! -f gmp-4.1.tar.gz ]; then
44   wget http://mirror.anl.gov/pub/gnu/gmp/gmp-4.1.tar.gz
45 fi
46 if [ ! -f mpfr-2.3.0.tar.gz ]; then
47   wget http://www.mpfr.org/mpfr-2.3.0/mpfr-2.3.0.tar.gz
48 fi
49 if [ ! -f gdb-6.8.tar.gz ]; then
50   wget http://mirrors.usc.edu/pub/gnu/gdb/gdb-6.8.tar.gz
51 fi
52 if [ ! -f binutils-2.19.tar.gz ]; then
53   wget http://mirrors.usc.edu/pub/gnu/binutils/binutils-2.19.tar.gz
54 fi
55 if [ ! -f newlib-1.17.0.tar.gz ]; then
56   wget ftp://sources.redhat.com/pub/newlib/newlib-1.17.0.tar.gz
57 fi
58
59 # unpack tarballs
60 cd $BUILDROOT/src
```

```
tar xzf ../orig/gcc-4.3.3.tar.gz
tar xzf ../orig/gcc-core-4.3.3.tar.gz
tar xzf ../orig/gmp-4.1.tar.gz
tar xzf ../orig/mpfr-2.3.0.tar.gz
tar xzf ../orig/gdb-6.8.tar.gz
tar xzf ../orig/binutils-2.19.tar.gz
tar xzf ../orig/newlib-1.17.0.tar.gz

mv gmp-4.1 gcc-4.3.3/gmp
mv mpfr-2.3.0 gcc-4.3.3/mpfr

# build binutils
mkdir $BUILDROOT/build/binutils-2.19
cd $BUILDROOT/build/binutils-2.19
../../src/binutils-2.19/configure --target=arm-none-eabi \
  --prefix=$PREFIX --enable-interwork --enable-multilib \
  CFLAGS="-g -O2 -Wno-unused-but-set-variable \
  -Wno-unused-but-set-parameter -Wno-format-security"
make $MFLAGS all
make install

export PATH="$PATH:$PREFIX/bin"

# build gcc compiler
mkdir $BUILDROOT/build/gcc-4.3.3
cd $BUILDROOT/build/gcc-4.3.3
# only enable C here
../../src/gcc-4.3.3/configure --target=arm-none-eabi \
  --prefix=$PREFIX --enable-interwork --enable-multilib \
  --enable-languages="c" --with-newlib \
  --with-headers=../../src/newlib-1.17.0/newlib/libc/include
make $MFLAGS all-gcc
make install-gcc

# build newlib
mkdir $BUILDROOT/build/newlib-1.17.0
cd $BUILDROOT/build/newlib-1.17.0
../../src/newlib-1.17.0/configure --target=arm-none-eabi \
  --prefix=$PREFIX --enable-interwork --enable-multilib
make $MFLAGS all
make install

# finish gcc build
```

```
cd $BUILDROOT/build/gcc-4.3.3
make $MFLAGS all
make install

# build gdb
mkdir $BUILDROOT/build/gdb-6.8
cd $BUILDROOT/build/gdb-6.8
../../src/gdb-6.8/configure --target=arm-none-eabi \
  --prefix=$PREFIX --enable-interwork --enable-multilib \
  CFLAGS="-g -O2 -Wno-unused-but-set-variable \
  -Wno-unused-result -Wno-enum-compare"
make $MFLAGS all
make install

# if you get here, you should have a working toolchain.
echo "all done."
```

2.A.2 U-Boot puts based standalone hello_world.c

```
// U—Boot global data layout
typedef struct  global_data {
  /*bd_t*/ void   *bd;     // simplified, assuming
                           // pointers of equal size
  unsigned long   flags;
  unsigned long   baudrate;
  unsigned long   have_console;
  unsigned long   env_addr;
  unsigned long   env_valid;
  unsigned long   fb_base;
  void            **jt;    // jump table
} gd_t;

// puts id in U—Boot function table
#define XF_puts 4

// puts prototype
void puts(const char*);

#include <stddef.h>     /* offsetof */

// puts asm stub
void __attribute__((unused)) dummy(void)
{
asm volatile (
  " .globl puts \n"
  " puts :\n"        \
  " ldr ip, [r8, %0]\n"   \
  " ldr pc, [ip, %1]\n"   \
    : : "i"(offsetof(gd_t, jt)),
        "i"(XF_puts * sizeof(void *)) : "ip");
}

// entry point
int
hello_world (void)
{
  puts("hello world\n");

  return 0;
}
```

2.A.3 U-Boot independent standalone hello_world.c

```
#define UART_THR (volatile char*)(0x01D0C000)
#define UART_LSR (volatile char*)(0x01D0C014)

int
putchar (int c)
{
  if (c == '\n')
    putchar('\r');

  while (!(*UART_LSR & (1 << 5)));

  *UART_THR = c;
  return c;
}


int
puts (const char *s)
{
  while (*s)
    {
      putchar(*s);
      ++s;
    }

  putchar('\n');

  return 0;
}

int
hello_world (void)
{
  puts("hello world");

  return 0;
}
```

2.A.4 Makefile to build the standalone hello_world.c

```
PREFIX = # arm-none-eabi-
CC = $(PREFIX)gcc
AR = $(PREFIX)ar
LD = $(PREFIX)ld
OBJCOPY = $(PREFIX)objcopy
OBJDUMP = $(PREFIX)objdump

SDDEV = # /dev/sd??
SDMNT = # /mnt/?

LOADADDR = 0xC1000000
ENTRY = hello_world

LIBGCCDIR = $(shell dirname $(shell $(CC) -print-libgcc-file-name))
LDFLAGS = -g -Ttext $(LOADADDR) -L$(LIBGCCDIR) -lgcc

INCGCCDIR = $(LIBGCCDIR)/include
CFLAGS = -g -O2 -pipe -fno-common -msoft-float -fno-builtin \
  -ffreestanding -nostdinc -isystem $(INCGCCDIR) -marm \
  -mabi=aapcs-linux -mno-thumb-interwork -march=armv5te \
  -fno-stack-protector -Wall -Wextra -Wstrict-prototypes -Werror

OBJ = hello_world.o

ELF  = hello_world
BIN  = $(ELF).bin
SREC = $(ELF).srec
ASM  = $(ELF).asm

ifeq ($(PREFIX),)
$(error Please set PREFIX to the prefix of your toolchain.)
endif
ifeq ($(SDDEV),)
$(error Please set SDDEV to the device node of your MicroSD card.)
endif
ifeq ($(SDMNT),)
$(error Please set SDMNT to an existing mount point of your choice.)
endif

.PHONY : all boot.scr boot.cmd deploy clean disas

```

```
42 all: $(ELF) $(SREC) $(BIN)
43
44 $(ELF): $(OBJ)
45     $(LD) -o $(ELF) -e $(ENTRY) $(OBJ) $(LDFLAGS)
46
47 $(BIN): $(ELF)
48     $(OBJCOPY) -O binary $(ELF) $(BIN)
49
50 $(SREC): $(ELF)
51     $(OBJCOPY) -O srec $(ELF) $(SREC)
52
53 boot.scr: boot.cmd
54     mkimage -C none -A arm -T script -d boot.cmd boot.scr > /dev/null
55
56 boot.cmd: disas
57     echo "fatload mmc 0 $(LOADADDR) $(BIN)" > boot.cmd
58     echo "go 0x$(shell grep '<$(ENTRY)>' $(ASM) | head -n1 \
59       | cut -d' ' -f1)" >> boot.cmd
60
61 deploy: $(BIN) boot.scr
62     mount $(SDDEV) $(SDMNT)
63     cp $(BIN) $(SDMNT)/
64     cp boot.scr $(SDMNT)/
65     umount $(SDDEV)
66
67 clean:
68     rm -f $(OBJ) $(ELF) $(ASM) $(BIN) $(SREC) boot.scr boot.cmd
69
70 disas: $(ELF)
71     $(OBJDUMP) -d $(ELF) > $(ASM)
72
73 %.o: %.c
74     $(CC) $(CFLAGS) -o $@ $< -c
```

3 Real-Time Linux on Lego EV3

3.1 Introduction

When building a real-time system, every part of the system has to ensure predictable timing. This also includes the operating system and, as a specific part of it, its scheduling mechanisms. One classic problem of priority-based scheduling mechanisms in real-time operating systems is priority inversion. This problem needs be be solved by an operating system used in a real-time system.

Depending on the financial and temporal constraints, it can be reasonable to use a general-purpose operating system like Linux for building a real-time system. However, in order to use such a system we need to ensure that the scheduling mechanisms of the general-purpose operating system have predictable timing behaviour, for example, by implementing solutions to priority-inversion. For Linux, the `RT_PREEMPT` patch[1] changes the kernel in such a way that it has predictable timing, for example, by implementing priority inheritance for kernel synchronization primitives.

To demonstrate the real-time behaviour of a `RT_PREEMPT` Linux, we conducted an experiment. As our target platform, we chose the Lego Mindstorms EV3[2], since it is an embedded platform that is nevertheless easy to program. In detail, our experiment focuses on the mechanisms introduced into Linux to handle priority inversion. Therefore, we constructed a mission program, which is based on a task set that suffers from priority inversion. We evaluated this scenario by running the program on an ordinary Linux and a patched version of it.

3.1.1 Lego EV3

The Lego Mindstorms EV3 is the third generation of programmable bricks produced by Lego. In the academic context, it is a versatile platform for prototyping and researching embedded, robotic systems. The brick has the following hardware specifications:

[1] `RT_PREEMPT` patch, also `CONFIG_PREEMPT_RT`: Kernel patch adding real-time support to Linux [10]

[2] Lego Mindstorms EV3: Lego robotics platform, see `http://mindstorms.lego.com/`

- AM1808 System-on-a-Chip (SoC) with a 300MHz ARM-9 CPU, 64MB RAM and controllers for SDHC-Card, USB, I²C, UART, GPIO and display.
- 4 motor and 4 sensor ports using a modified RJ-12 jack with the locking tap on the side of pin 1. One port can be used as a serial console.
- A-Type USB host port and micro B-Type USB client port.
- Monochrome 178 × 128 LCD connected to the SoC graphics unit.
- 6 buttons with background LED.
- A speaker.

The platform is running Ångström Linux[3], a Linux-2.6-based distribution specialized on embedded systems but not supporting hard real-time.

3.1.2 Linux and Real Time

The 2.6.x series of the Linux kernel are interesting for embedded systems due to zero licensing costs, their wide range of supported hardware and low resource consumption. They already provide basic real-time facilities, such as a constant-time scheduler. Pages can be locked to prevent page faults, which would otherwise delay memory accesses indeterminately. Real-time activities can suspend other running tasks when certain preemption points in the kernel code are reached. User-mode mutexes support the priority inheritance protocol and hence are not prone to priority inversion.

However, there are still many non-preemptable code paths in the kernel, especially interrupt handlers and scheduling facilities. Moreover, the OS does not exploit the full resolution of hardware timers, thereby limiting scheduling and deadline precision. Synchronization mechanisms are (at least partly) prone to the priority inversion phenomenon with potentially catastrophic failure of the deployed facility. One example showing the severity of such problems is the Mars Rover NASA Mission [5].

The RTLinux project and its successor, the `RT_PREEMPT` configuration of Linux, drastically improve the real-time capabilities of Linux by introducing (nearly) full kernel preemptability and a protection against priority inversion in kernel synchronization primitives through priority inheritance. Implementation details are given in section 3.2.

[3]The Ångström Distribution: Linux distribution for embedded devices [11]

Figure 3.1: EV3 Front.

Figure 3.2: EV3 left side with host USB and microSD slot.

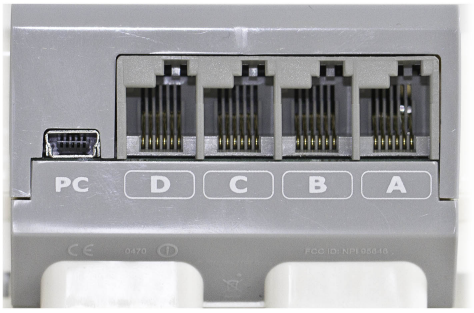

Figure 3.3: EV3 top side with client USB and sensor ports.

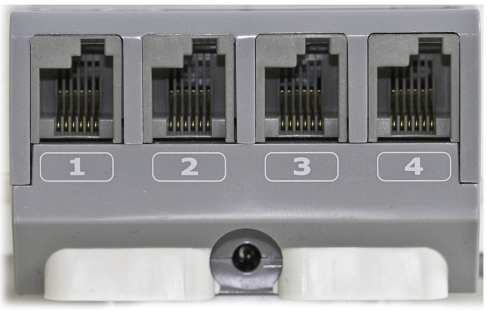

Figure 3.4: EV3 bottom side with motor ports. Port 1 can be used as UART console.

3.1.3 Experiment Setup

The default operating system on the Lego EV3 cannot achieve hard real-time given the limitations implied by Linux 2.6.

We approached these drawbacks by applying the `RT_PREEMPT` patch to the original operating system running on the EV3. This patch left us with a fully preemptable kernel and support for priority inversion in frequently used kernel synchronization primitives, such as kernel mutexes. Subsequently we constructed a task set which is likely to miss deadlines due to priority inversion. The task set controls the movement of a robot and simulates sensor and communication activities. We show the effectiveness of our change by implementing and running this task set on both the original EV3 OS and the real-time variant. Using the original EV3 OS, the highest priority task eventually misses its deadline and causes the robot to stop. The robot running the task set on top of the patched OS never stopped as no task misses its deadline.

3.1.4 Structure of this Chapter

Section 3.2 is concerned with the historical developments in the field of real-time UNIX and Linux operating systems and highlights the reasoning behind important real-time mechanisms. Section 3.3 describes the steps necessary to patch the EV3 OS and elaborates on challenges and the solutions we found. Section 3.4 explains concept, implementation and observations concerning our experiment in detail along with implementation challenges and code snippets. Section 3.5 reflects on the current situation of Linux in the real-time context and on our development process. Finally, section 3.6 concludes.

3.2 History and Mechanics of Real-Time Linux

To understand what constitutes a real-time operating system, and particularly the one we are developing our experiment for, we researched the historical reasoning that has led to the current design of the `RT_PREEMPT` patch. The following section will give an overview over the influential predecessors *QNX* and *RTLinux*, as well as the changes introduced by the `RT_PREEMPT` patch in the context of Linux 2.6 and how they contribute to an overall real-time behavior.

3.2.1 History

Successful implementations of UNIX-like real-time systems date back to the early 1980s [7]. UNIX [9] had already become an attractive multipurpose operating system, because it supported a wide range of hardware and enabled easy programming and testing of software using C. However, due to its time-sharing nature, it was not suitable for any hard real-time applications. Many of those applications were still being developed using assembly language on bare metal. QUNIX, later named *QNX*[4], successfully bridged this gap by implementing a UNIX-compatible microkernel OS with hard real-time capabilities.

The microkernel of QNX only provided interrupt handling, scheduling, memory mapping, synchronization, signaling and message passing between threads. The kernel code was structured in a way that allowed to place upper bounds on the time spent in non-preemptable kernel code paths. Most of the kernel code itself was preemptable and message passing used a software-implemented bus, where message sends can be preempted and resumed. Priority inheritance was put in place at synchronization primitives. As any other concern, for example file system access, had to be implemented as a thread, its execution was subject to priority-based scheduling and preemption.

In the mid and late 1990s, the *RTLinux* project [14] initiated by Yodaiken, Dougan and Barabanov aimed at adding QNX-inspired real-time capablities to the increasingly popular Linux kernel. However, not the full Linux kernel was rewritten, but instead it was put on top of a new microkernel. This architectural "shortcut" retained all of the Linux kernel and user functionality but made it fully preemptable. Real-time tasks operate directly on top of the microkernel which ensures that they could meet their deadlines without the risk of interference with kernel or even user code.

The architectural decisions in RTLinux are motivated by the following five *threats to predictable timing* identified by the RTLinux team:

1. **Disabling interrupts.** Linux makes heavy and inconsistent use of `CLI` (disable interrupts) and `STI` (enable interrupts) instructions, which causes unpredictable interrupt response latencies. RTLinux replaces these instructions by macros, which either forward interrupts directly to the Linux kernel (invoking the ISR[5]) or delay them for later delivery. Delayed interrupts are delivered when `STI` is reached. To decide, whether to delay an interrupt delivery, the macros lookup the respective interrupt ID in a software-implemented bitmask. This abstraction is named *soft interrupts* and allows the interrupting device to

[4]QNX: Commercial UNIX-like operating system for embedded systems, see `http://www.qnx.com/`

[5]Interrupt service routine, also referred to as interrupt request handler or IRQ handler

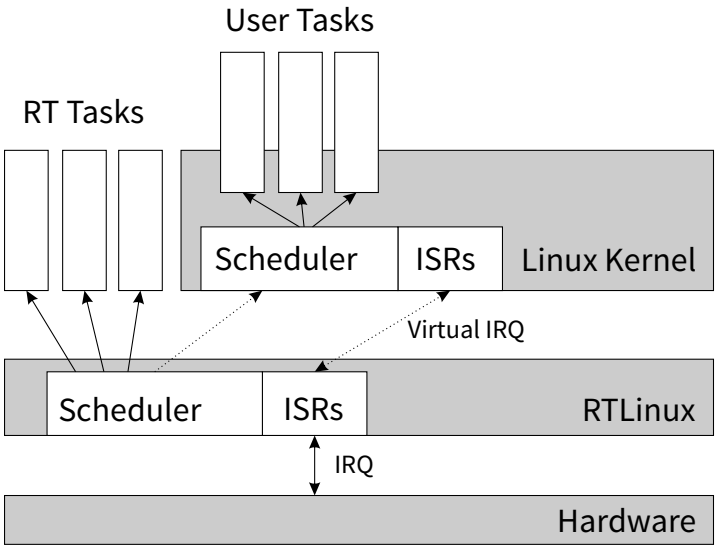

Figure 3.5: Architecture of RTLinux and its interactions with real-time tasks and the original Linux kernel running on top.

receive a response from the CPU immediatley. Moreover, RTLinux-based code can handle the interrupt regardless of the interrupt state of the Linux kernel.

2. **Protected-mode changes and context switches** are both expensive, as they move registers to memory and back, and cause unpredictable delays due to cache flushes and TLB[6] invalidations. RTLinux puts real-time tasks into *kernel modules* to avoid protected mode changes and limit context switches to kernel threads only. While this avoids cache invalidations, it sacrifices isolation between real-time threads completely.

3. **The default scheduler** in Linux aims at maximizing average throughput and response times and does not care about deadlines. RTLinux maintains this scheduler inside the original Linux kernel and adds a simple priority-based *rate-monotonic scheduler* for real-time tasks as loadable microkernel module.

4. **Periodic clock interrupts** trade timer precision for interrupt load, for example a 10 ms precision requires the clock to emit 100 interrupts per second. RTLinux eliminates this tradeoff by reprogramming the CPU hardware timers to trigger an interrupt when needed. This way the precision can be as high as the hardware timer's precision, but timer interrupts only happen if some event was scheduled beforehand (for instance, switching tasks).

[6]TLB: Translation lookaside buffer

5. **Inter-process communication.** As the Linux kernel can be preempted at any time (in any inconsistent state) it is unsafe to call kernel code from real-time tasks. RTLinux adds a "safe", non-blocking FIFO channel, which can be read and written by both real-time tasks and Linux threads. Its maximum buffer size is set at compile time, so it can be allocated in constant time.

The overall *kernel preemptability* results from the co-operation of timers and soft interrupts: The microkernel's ISR handles timeouts and hands over control to a real-time task, while the original kernel can neither prevent nor notice this preemption as long as real-time tasks do not violate certain isolation rules (for example, not to call kernel code). The RTLinux scheduler resumes the kernel when there is no pending real-time task and also emulates clock interrupts required by the original kernel to do its own scheduling.

Moreover, RTLinux offers POSIX-style I/O for devices (`read/write`), shared memory, priority-inheriting mutexes and semaphores as loadable microkernel modules - thus duplicating some of the original Linux functionality.

The original RTLinux distribution was developed at the *New Mexico Institute of Mining and Technology*. It was later continued by *FSMLabs* and *Wind River Systems* as commercial RTOS. The facts that RTLinux was patented from the beginning and commercially discontinued in 2011 accelerated the development of a free alternative based solely on modifying the Linux kernel: the `RT_PREEMPT` configuration of the Linux kernel, which is maintained and distributed as separate patch and can be applied directly to the mainline Linux kernel source.

3.2.2 Architecture of the `RT_PREEMPT` Patch

The `RT_PREEMPT` configuration of the kernel adresses the same concerns as RTLinux: kernel preemptability, soft interrupts, context switches, scheduling and timers. In contrast to the RTLinux architecture, `RT_PREEMPT` does neither implement nor require duplicated kernel facilities. It patches the existing kernel code to be safely callable from real-time threads.

Threaded IRQ Handling

`RT_PREEMPT` does not fully virtualize interrupts and interrupt control instructions as RTLinux does. It registers a default interrupt handler for each *IRQ* (interrupt request) instead, which spawns original IRQ handlers inside a kernel thread. The default handler itself has a predictable timing behavior. IRQ threads serve as fully preemptable interrupt service routines. The default priority of an IRQ thread is above user priorities but below standard real-time priorities, so interrupt servicing will still preempt normal user code as usual. However, ISRs can decide to change their thread priority to satisfy real-time constraints.

Unfortunately, threaded IRQ handling comes at the price of configuring (pre-allocated) task structs and invoking the scheduler, which introduces significant, but predictable overhead during device interaction compared to direct interrupt handling. RT_PREEMPT could theoretically reuse one thread for all IRQs of the same priority but is, to the best of our knowledge, not doing so [12].

Preemptable Synchronization and Priority Inheritance
The original Linux kernel provides and uses a range of synchronization primitives: semaphore, mutex, spinlock and RT-mutex. A mutex is prone to priority inversion while RT-mutexes implement priority inheritance. Spinlocks are not preemptable. User space programs do not use these primitives directly. They mainly rely on the Pthread API[7], which also supports priority inheritance. Internally this API is implemented using kernel primitives. So finally, the pthread priority inheritance relies on the RT-mutex kernel API.

RT_PREEMPT replaces the general kernel mutexes with RT-mutexes to obtain kernel-wide priority inheritance, a feature formerly rejected by Linux creator Linus Torvalds [6]. Moreover, spinlocks are now implemented using RT-mutexes which makes them preemptable regarding higher priorities. RT_PREEMPT introduces the new primitive raw_spinlock, which is implemented like the original spinlock. These are still necessary for some multi-processor operations: For instance, when a process is pushed to another CPU, the scheduler on the receiving CPU waits behind a spinlock until the pushing CPU has left the run queues in a consistent state. Additionally

Primitive	Default	RT_PREEMPT
semaphore	semaphore	semaphore
spinlock	spinlock	rt_mutex*
mutex	mutex	rt_mutex*
rt_mutex	rt_mutex*	rt_mutex*
raw_spinlock	(not present)	spinlock

Figure 3.6: Implementations of different kernel synchronization primitives in the unmodified (default) kernel and a RT_PREEMPT variant. *) supports priority inheritance.

[7]POSIX thread: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html

Predictable Scheduling
While the RTLinux microkernel used its own scheduler to manage real-time tasks, `RT_PREEMPT` uses the Linux kernel scheduler for both real-time and non-real-time activities.

Linux natively supports "soft" real-time scheduling (as of version 2.6), so `RT_PREEMPT` does not actually change the scheduling algorithm. Real time threads are supported through the priorities 0 to 99, and scheduling strategies within those priorities include first-in-first-out `SCHED_FIFO` and round-robin `SCHED_RR`. For the real-time priorities and scheduling strategy `SCHED_FIFO`, the kernel chooses an optimized memory layout for the run queue: A bitmap keeps track of currently active priorities and for each priority, a FIFO run queue is stored. Searching the bitmap can be done in constant time using a `BSF` (bit scan forward) or `CLZ` (count leading zeros) instruction if present.

During scheduling, the kernel needs to give up preemptability. This causes interrupts to be actually disabled while the next thread is retrieved from the run queue and the current thread is put back. `RT_PREEMPT` deferrs freeing memory (for example, when threads finish) until no real-time activity is about to run. This way, this period is kept as short and predictable as possible. `RT_PREEMPT` also disables the system call `yield()`, which yields control to the next thread in the run queue.

3.2.3 Mainline Integration

Linux 2.6 already integrated many real-time capabilities into the official kernel, for instance RT-Mutexes, constant-time preemptive FIFO scheduling strategies and user-space priority inheritance support. The tendency to make mainline Linux better suited for real-time application has continued as kernel preemptability was continuously increased with each subsequent release, based both on `RT_PREEMPT` code and refactorings thereof.

Linux 3.14 further adapted RTLinux' `raw_spinlock` and natively uses priority-inheritance protocols in all other kernel synchronization primitives. Threaded IRQ handling is available and scheduling algorithms have evolved: The `SCHED_DEADLINE` scheduler class implements an *earliest deadline first* (EDF) algorithm. It preempts all other real-time priority tasks when necessary to meet a deadline. Instead of explicitly specifying priorities, a task is defined by its period, expected runtime and deadline relative to the start of a period. Based on these properties, the `SCHED_DEADLINE` scheduling policy dynamically assigns negative priority values to these tasks. Tasks exceeding their configured runtime are de-scheduled immediately and given a new time slice in the next period.

3.3 Patching and Deploying Linux on EV3

Before we were able to work on the actual experiment, we had to prepare the tool chain and apply the `RT_PREEMPT` patch. We will explain our process and the single steps in detail, so you can reuse our findings for your own projects. Hopefully, you will be able to prevent running into the same problems we did.

The EV3 sources already include a configured tool chain. Nevertheless, we had to prepare our local tool chain and hardware in order to be able to successfully build an operating system for the EV3. To debug the build components, we used a serial console. Using this environment, we applied the `RT_PREEMPT` patch to the EV3 operating system. As there is no matching `RT_PREEMPT` patch for the specific kernel version of the EV3 operating system, we had to choose between porting EV3 software and drivers to another kernel version or manually resolving conflicts arising during patching.

3.3.1 The EV3 Toolchain

We developed for the EV3 in a cross-platform fashion. We compiled our programs using an ARM cross-compiler on an x86-based host machine running a Linux distribution. Then we transferred the resulting files to a microSD card. From this card, we could boot our own operating system on the EV3. In this section, we will describe each of those steps in detail.

To use a microSD card as a boot medium for the EV3, the card needs a specific partition layout. The repository of the original EV3 operating system contains scripts, which create this layout. However, even those scripts expect an initial layout of two arbitrary partitions, which we have to create first. After creating those partitions, we can use the script `lms2012/open_first/format_sdcard.sh` to create the final correct partitions.

For building the application for ARM on an x86-based system, we used a cross-compiler toolchain. We used the *code sourcery lite* tool chain [1] as required by the online setup documentation of the original Lego OS [2]. You can install the cross-compiler toolchain either at the place specified in the setup documentation. Alternatively, you can install them anywhere and create symbolic links in your `/usr/bin` directory. With those tools, one can use the make targets specified in the original Makefile provided with the Lego OS in the following order to build a complete stock image:

1. **u-boot:** Creates the boot loader u-boot, which will be used to boot the new OS from the microSD card.

2. **kernel:** Creates the kernel in `extra/linux-03.20.00.13`.

3. **modules:** Creates EV3 specific kernel modules, such as drivers, based on the previously compiled kernel. Those modules are build from the sources in `lms2012/d_*/source`.

4. **lms2012:** Creates the *lms2012* virtual machine which executes Lego programs by interpreting byte codes. The source code can be found in `lms2012/lms2012/source`.

After building each single part of the system, we create a complete image including all of those parts and deploy it to the microSD card. First we execute the script `lms2012/open_first/make_image.sh`. Then, we copy the resulting images to the microSD card. Therefore we first mount the first and the second partition of our card to our earlier created mount points `/media/LMS2012` and `/media/LMS2012_EXT`. Now, we can start the script `lms2012/open_first/update_sdcard.sh` which transfers all necessary data to the partitions. Afterwards, we use `sync` to complete all I/O to the card and unmount the partitions. If one inserts this card into the EV3, it will start the u-boot boot loader from the first partition, which will then load the OS on the second partition.

Building custom kernel modules or applications with this tool chain requires specific folder structures. So if you want to create your own application compiled for the EV3, you have to create a new folder in `lms2012/`. It has to include a folder called `source` which includes the source files. Further, it has to include a folder called `Linux_AM1808`. It will be used for the build process and contains the final executable. You should put a `Makefile` there resembling those which can be found in the folders prefixed with `c_`. You can then create a make target for your application using a line like this one for an application called motor_control:

```
control:
    $(MAKE) -C $(BASE)/motor_control/Linux_$(ARCH)
```

To create a kernel module, you have to recreate the structure of one of the folders prefixed with `d_*`. You can then add the module to the list `MODULES` in the `Makefile`, so it will be build when you build the `modules` target. After building your application or module, you still have to deploy it to the microSD card. We chose a pragmatic approach and created a make target that takes care of these steps. It basically mounts the partitions and copies the kernel modules and applications onto the EV3 OS file system mounted via `LMS2012_EXT`. This way, we could also update our application without reinstalling the whole EV3 OS.

In order to start your own applications, you also have to modify the start up process of the operating system. Originally, it starts the *lms2012* virtual machine. This leads to two problems. First, as this process allocates device resources, such as the screen, your application can't use them anymore. Second, *lms2012* loads the

device drivers during its start up phase. As some of the device drivers are currently not working with a real-time patched OS (section 3.3.2), the system is halting at loading those faulty drivers. So we had to disable the start of *lms2012*. You can change it permanently by modifying the tar-archive containing the original content of the file system for the EV3. Therefore, you edit the file `/etc/init.d/lms` in the file `lms2012/open_first/lmsfs.tar.bz2`. You can also change it on every deploy by modifying the file on the mounted partition `LMS2012_EXT`. We "misused" the `lms` file to start our own applications.

The amount and configuration of all of those steps grew continuously during the development of our project. In order to speed up development and prevent mistakes during the build process, we implemented custom make targets for our most common tasks. You can find them in `lms2012/open_first/Makefile` in our fork of the official Lego EV3 OS [3].

3.3.2 Applying the RT_PREEMPT Patch

The EV3 system is based on an Ångström Linux distribution with EV3-specific changes. The same goes for the Linux kernel deployed on the EV3. It is based on a Linux 2.6.33-rc4 kernel with changes both by the Ångström distribution and the EV3 developers.

To explain the problems that we were facing when trying to apply a `RT_PREEMPT` patch to this kernel, let us first look at the development process of the `RT_PREEMPT` team. While a new Linux kernel version is developed, the `RT_PREEMPT` team will work on making the `RT_PREEMPT` patch compatible. During this development, several patch revisions will be published, but compatibility is only maintained for the respective latest release of the Linux kernel. Of course, the early revisions might not feature the full support of all `RT_PREEMPT` features the patch will eventually provide. For the kernel version of the EV3 kernel (2.6.33) the `RT_PREEMPT` team's first release was for the 8th release candidate version of the kernel, while EV3 runs only the 4th. Consequently, none of the `RT_PREEMPT` patch releases was fully compatible with the EV3 kernel, and even the first (incomplete) versions of the `RT_PREEMPT` patch were only available for a newer kernel version.

At this point we had two choices: (a) stay with the current kernel and try to apply a patch that is not fully compatible with the kernel release, fixing problems where necessary; and (b) deploy and patch a different Linux kernel on the EV3. While a newer kernel version would probably be compatible with the EV3 hardware, the Lego software might not be compatible with it. As we wanted to be able to reuse at least parts of the EV3 software (for example, the motor drivers), we had to ensure the compatibility of our patched system with it. The changes to the EV3 kernel, however, are undocumented, and so switching to a different Linux kernel version would

probably have required a lot of reverse engineering and code portation. Because of this, we chose to stay with the current Ångström kernel and to manually fix any problems that arose during the patching of the kernel.

Patching Lego's Ångström Linux
We've subsequently tried to apply a number of different patch versions to the original EV3 kernel, and decided to use the `RT_PREEMPT` patch version 2.3.66-rt8 (8th release of the patch), because (among its newer releases) it gave the least merge conflicts. Most of these conflicts could be solved easily, as they were only caused by minor changes between the kernel revisions (for example, renamed variables or reordered code sections). However, there was one slightly more tricky case in the file `kernel/timer.c`. Between the EV3 kernel version and the kernel version the patch expected, a few function calls were moved between the `update_process_times()` and `run_timer_softirq()` functions. The changes that the patch made affected that same ordering as well. As it was not perfectly clear which ordering should be preferred, we chose to change the ordering to the way that the `RT_PREEMPT` patch expected it. As we did not see any problems caused by this reordering, we assume that it was successful.

As a result, we obtained a bootable version of the EV3 kernel with the `RT_PREEMPT` patch applied to it. It can be obtained from our GitHub repository[3]. In order to test whether the running kernel is a successfully patched one, you can execute `uname -a` on the command line and look for the kernel version number. It should end with a *rtX* suffix. Another method is to execute `sudo ps x` and look for the IRQ handler threads. On a `RT_PREEMPT` kernel, they should be listed as kernel threads (that is, in square brackets, for example as `[ksoftirqd/0]`), while on the old EV3 kernel, they show up as userspace threads (that is, without square brackets).

USB Driver Problems in the Patched Linux
Initially, when booting the patched kernel, we experienced a system freeze during the activation of the USB driver kernel module on startup. As we were not relying on USB support for our application scenario, we solved this problem by disabling the USB kernel module. However, in the scope of the Embedded Operating Systems lecture, another team tried to solve this problem differently. They were hoping for USB support in order to use a WLAN dongle on the `RT_PREEMPT` patched EV3-OS. They found out that the system freeze was a result of an interrupt that the USB host device throws when it is activated. Apparently, the patched interrupt handling mechanism does not mask the interrupt fast enough and because of that, the interrupt fires continuously. This results in an interrupt storm and the ISR of the driver (which masks the interrupt) consequently won't ever be executed. The team solved this problem by adding a masking command to the default irq handler (`irq_default_primary_handler()`). Sadly, they later encountered a similar but

less easily debuggable problem with the drivers for the WLAN USB dongle they wanted to use, and subsequently gave up on the topic. We would attribute these problems to the `RT_PREEMPT` patch version we used, and to the fact that it was not designed for the particular kernel version we are using. If USB and WLAN driver support is a necessity, it might be useful to investigate the alternative route, to port the EV3 software to a newer kernel version, as presented earlier.

3.3.3 Debugging

To conduct our experiment, we wanted to execute our patched OS on the EV3 itself. However, we also wanted to get information about our running applications and even interact with them. As described in the introduction, the EV3 hardware provides a UART console on the first sensor port. Based on a tutorial, one group managed to create USB-to-UART cables, so we could use those to get a serial console connection to the EV3. To establish the connection we used Putty [8] with the following parameters:

Serial line: /dev/ttyUSB0

Speed/Baud: 115200

Data bits: 8

Stop bit: 1

Parity: None

Flow control: None

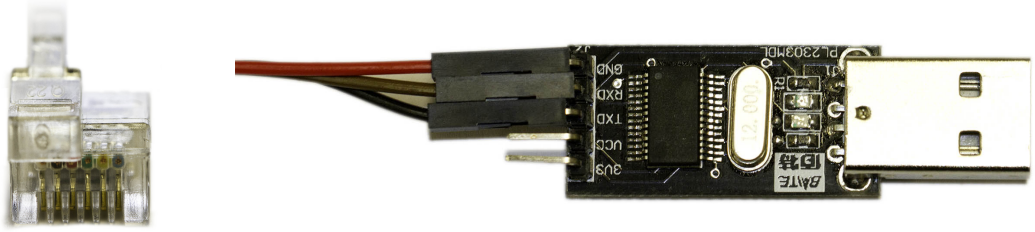

Figure 3.7: Modified RJ plug (connector tap above the white wire) and UART-to-USB adapter with yellow wire connected to RX, blue to TX and red to ground. Colors in this image are different.

On the software side, as we were interested in the actual timing of events, we used print debugging. We were developing a kernel module, and, therefore, used the `printk` procedure with the default log level. After executing our program, we could then read the output from the kernel log using the `dmesg` command.

3.4 Experiment: Real-Time Schedule on EV3

Based on the tool chain and the `RT_PREEMPT` patched Linux, we were able to implement our experiment. We already described the general idea of our experiment in section 3.1.3. The concrete implementation was a kernel module, starting three kernel threads running the control program. Each of the threads serves a specific purpose, such as reading the sensors.

We implemented the experiment as a kernel module, as user space programs are not prone to priority inversion in the original Linux kernel version running on the EV3. We could simply use the corresponding Pthread API, which implements priority inheritance. In kernel space, the situation is different, as synchronization primitives only support priority inheritance through the `RT_PREEMPT` patch.

The central part of the experiment was the construction of a task set, which leads to priority inversion. During the initialization of kernel threads, we faced ambiguous information on the actual ordering of thread priorities. To implement the periodic control threads, we used the timer API and implemented a simple watchdog to recognize missed deadlines.

We wanted to reuse existing modules for accessing Lego devices, such as sensors and motors. As the implementation of device modules are intertwined with a virtual machine executing user programs, we had to refactor some of these modules.

3.4.1 Priority Inversion

One of the classic real-time problems the real-time patch addresses is priority inversion. It is defined by a high priority thread being delayed by a lower priority thread. While there are numerous ways this can happen in a real-time system, the classical example is illustrated below.

In our example, we use three threads of different priorities in a preemtable system with shared ressources. Figure 3.8 illustrates the normal way in which the system should handle a thread becoming runnable that has a higher priority then the one currently running. While this seems straight forward, it gets complicated as soon as shared resources (in our example, a mutex) are involved. Figure 3.9 shows how priority inversion can happen: The highest priority thread shares a mutex with the lowest priority thread. The lowest priority thread holds the mutex when it gets inter-

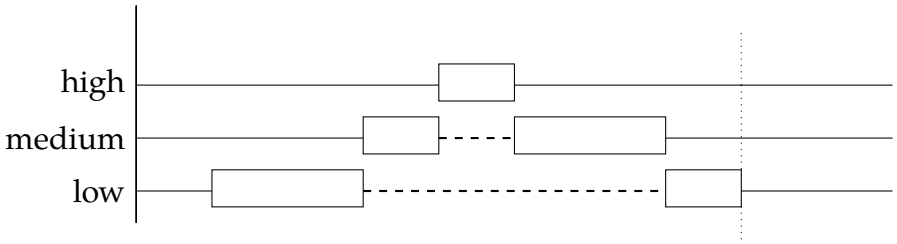

Figure 3.8: A schedule period of a task set without interaction.

rupted by the medium priority thread. When the highest priority thread becomes runnable, it cannot accquire the mutex and waits for the lowest priority thread to release it, which in turn cannot run, because the medium priority thread is running. Therefore, the medium priority thread delays the high priority thread from running and priority inversion occurs.

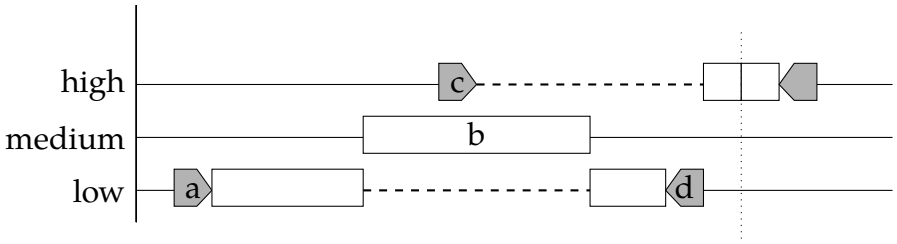

Figure 3.9: Priority Inversion: A schedule period of the task set with task "low" acquiring a mutex (a), being interrupted by the medium priority task (b) and therefore preventing "high" from entering the mutex (c) until "low" can release the mutex (d).

One common solution for priority inversion is illustrated in figure 3.10. When a system with priority inheritance is faced with the above problem, it will boost the priority of the lowest priority thread temporarily to the level of the highest thread that also shares the mutex. Hence, the lowest priority thread cannot be interupted by the medium priority thread while it holds the mutex and priority inversion cannot occur. This principle is known as *priority inheritance*.

3.4.2 Modeling the Task Set

One of the reasons that the priority inversion problem is so well known is the publicised occurance within the Mars Rover project [5]. Our task set and algorithm design

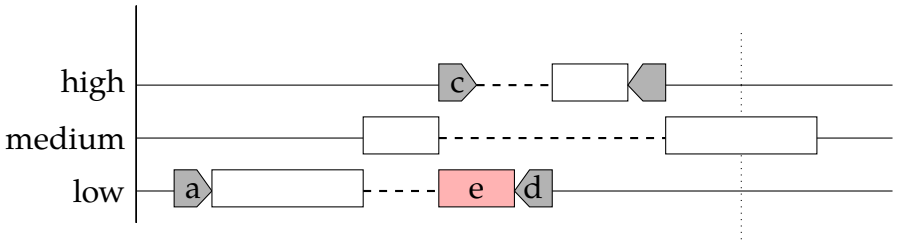

Figure 3.10: Priority Inheritance: Same task set with mutex as in figure 3.9, but the mutex being acquired by "high" (c) elevates the holding task "low" (e) to its own priority to avoid the interruption caused by "medium".

is therefore in the style of the parts of the Mars Rover that contributed to the priority inversion.

The priority inversion of the Mars Rover was ultimately solved by activating priority inheritance in its operating system, which is precisely what we hoped to do with our example application and real-time enabled version of the EV3 Linux kernel.

The following task set is designed to show the effects of priority inversion and inheritance and therefore illustrates one difference between the EV3 default kernel and our patched variant. As explained above, at least three threads with different priorities are required:

Table 3.1: Task set with timings relative to the start of a period. Period length: 100 ms

Task	Priority	Start	Duration	Deadline
High	30	40 ms	10 ms	60 ms
Medium	20	30 ms	30 ms	100 ms
Low	10	30 ms	100 ms	100 ms

3.4.3 Real-Time Task Set

The task set described above is naturally implemented with three threads running inside the kernel with the specified priorities. The synchronisation primitive we chose as described in section 3.2.2 is `Mutex` from <*mutex.h*> and the runtimes are as follows:

Task	Priority	Start	Duration
High	60	200 ms	200 ms
Medium	30	200 ms	800 ms
Low	10	0 ms	400 ms

The most interesting part of the implementation was validating that a priority inversion had actually occured. We observed several different executions of the task sets that led to some unexpected conclusions.

We observed the execution by having each thread measure the time between starting and terminating and inferred the execution orders accordingly.

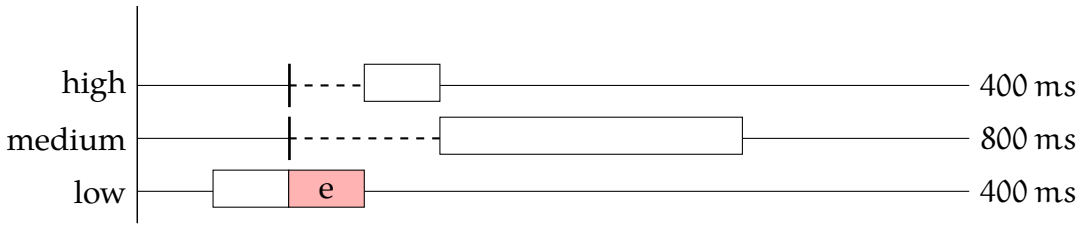

Figure 3.11: Intended schedule with priority inheritance. Thread *low* runs on elevated priority from the moment *high* tries to enter the critical section thereby preventing *medium* from interfering.

Figure 3.11 shows our intended execution order with priority inheritance preventing priority inversion to occur. The run time of each thread is given at the right side of the diagram. The low thread has its intended runtime (disregarding context switch overhead) while the high threads execution time is extended by the time the low thread runs elevated. The medium thread has its intended runtime because it does not get to record a timestamp until both other threads have finished. Figure 3.12 shows the same task setup without priority inheritance enabled: Only the medium thread has its intended runtime, because it does not get preempted. The low thread has to wait for the medium thread to finish, while the high thread has to wait for both (although only half of the low threads runtime) - and priority inversion occurs.

When we started to meassure the outcome of the scheduling problems, we expected one of the cases shown in figure 3.11 and figure 3.12 to occur, however neither did. In fact the runtimes we recorded are shown in table 3.2. The only schedule we could construct from these times made no sense and is shown in figure 3.13.

Lacking any explanation for this schedule, we experimented with the priorities inside the Linux kernel and found that the documentation in the kernel sources had mislead us (see section 3.4.5). After ensuring that the priorities we gave to the threads were actually what we expected them to be, we observed the schedule

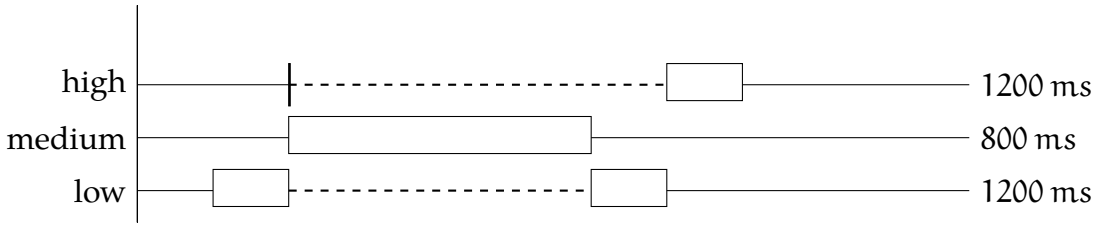

Figure 3.12: Expected schedule with priority inversion occurring. Thread *high* gets delayed.

Table 3.2: Recorded runtimes on first tries.

Task	Runtime
High	200 ms
Medium	800 ms
Low	400 ms

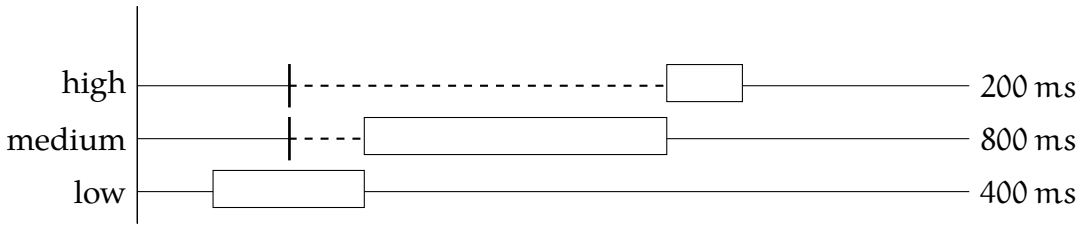

Figure 3.13: Actual schedule inferred from the timings measured in the first run.

from figure 3.12 when running the regular EV3 Linux kernel and the schedule from figure 3.11 with our real-time enabled version of the EV3 Linux kernel.

3.4.4 Implementing the Tasks

As explained above, the RT_PREEMPT patch patches the mutex interface transparently. The Mutex API stays the same both as in the original kernel, but is transparently replaced by the RTMutex implementation in the patched kernel. Hence the code shown in listing 3.1 will use priority inheritance on the shared mutex in our patched operating system, but in the original kernel will use the default Mutex implementation without priority inheritance support.

Listing 3.1: Excerpt of code for the high priority thread.

```
sched_setscheduler(this_thread, SCHED_FIFO, high_priority);
mlockall(...);

while(true) {
    start_us = get_current_time_us();

    mutex_lock(sensor_mutex);
    /* Actual control code */
    mutex_unlock(sensor_mutex);

    duration_us = get_current_time_us() - start;
    sleep_us = (MOTOR_CONTROL_PERIOD_US - duration_us);
    schedule_hrtimeout_range(sleep_us, 0,
           HRTIMER_MODE_REL);
}
```

3.4.5 Linux Kernel Priorities

The source code documentation in `include/linux/sched.h` suggests that a lower priority value represents a higher priority, even for real-time priorities.

Listing 3.2: Comment excerpt from sched.h.

```
/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL/SCHED_BATCH
 * tasks are in the range MAX_RT_PRIO..MAX_PRIO-1. Priority
 * values are inverted: lower p->prio value means higher priority.
 * [...]
 */
```

However, when examining the Linux kernel code however, one will also find the following code:

Listing 3.3: Code excerpt from sched.h.

```
static void
__setscheduler(struct rq *rq, struct task_struct *p, int policy, int
    prio)
{
        BUG_ON(p->se.on_rq);

        p->policy = policy;
        p->rt_priority = prio;
        p->normal_prio = normal_prio(p);
        /* we are holding p->pi_lock already */
        p->prio = rt_mutex_getprio(p);
        if (rt_prio(p->prio))
                p->sched_class = &rt_sched_class;
        else
                p->sched_class = &fair_sched_class;
        set_load_weight(p);
}
```

While the `normal_prio` function contains the following line:

```
prio = MAX_RT_PRIO-1 - p->rt_priority;
```

Therefore, we were forced to conclude that, internally, the Linux kernel treats lower numbers as the higher priority, while values set through `sched_setscheduler()` work in the opposite way.

3.4.6 Implementing a Watchdog

Missing a deadline in a hard real-time system is considered a failure. In order to minimize the consequences of such failure, many real-time systems implement fail-safety measures, such as self-checks, emergency stops, etc. A watchdog is a periodically running self-checking routine, which ensures that either important tasks have met their deadline or an emergency routine (alarm, system stop, recovery) is invoked.

Implementing a simple watchdog first requires the tasks in question to log a timestamp once per period. The watchdog is a periodic task with highest priority, which confirms that the task has correctly updated its timestamp by comparing it with the current time. If the difference of the timestamp to the current time exceeds twice the task's period, the task has definitely missed one period. In our example, this case is handled by stopping all motors and triggering an acoustic alarm.

Listing 3.4: Modification of a thread to update the variable `last_ktime` and implementation of the watchdog thread checking if the thread has run in the last period.

```

// supervised thread ——————————————————————————————
int threadHigh_fn(void* params) {
   sched_setscheduler( ... ); // set priority and SCHED_FIFO
   while(true) {
      last_ktime = ktime_get();
      /* ... work ... */
      usleep_range(MOTOR_CONTROL_PERIOD, MOTOR_CONTROL_PERIOD);
   }
}

// watchdog thread ——————————————————————————————
int threadWatchdog_fn(void* params) {
   sched_setscheduler( ... ); // set priority and SCHED_FIFO
   while(true) {
        now_ktime = ktime_get();
        deltaNs = ktime_to_ns(now_ktime) - ktime_to_ns(last_ktime);

        if(deltaNs > (2 * MOTOR_CONTROL_PERIOD)) {
            /* Failure */
            motor_command(MOTOR_CMD_STOP);
            sound_command();
            return 1;
        }

        /* Put yourself to sleep */
        usleep_range(WATCHDOG_PERIOD, WATCHDOG_PERIOD);
    }
}
```

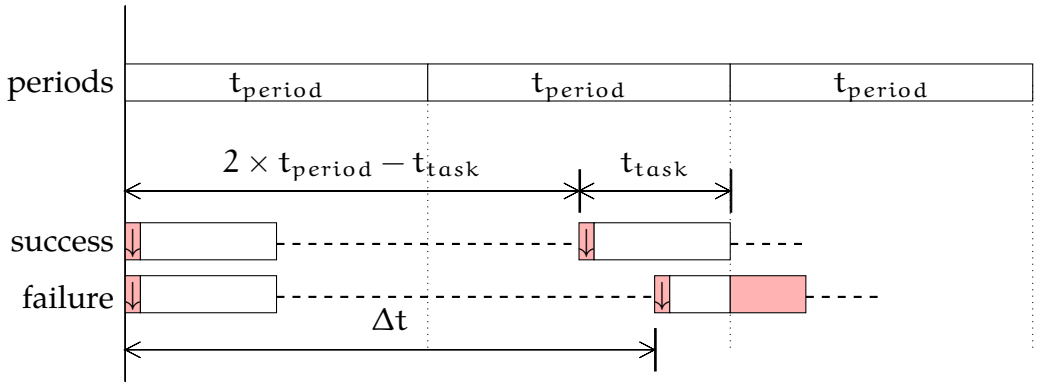

Figure 3.14: Timestamps are logged (indicated by ↓) at the beginning of task execution. Let Δt be the time between timestamps. The rate-monotonic schedule **fails** when $\Delta t > 2 \times t_{period} - t_{task}$. For simplicity we use $\Delta t > 2 \times t_{period}$ as a **sufficient condition**.

In theory, if the task's exact execution time is known in advance, the watchdog should be alarmed whenever the difference in timestamps exceeds $2 \times t_{period} - t_{task}$ with t_{period} and t_{task} being the period and the task's execution time respectively. Hence exceeding $2 \times t_{period}$ is a sufficient condition for unknown t_{task}.

3.4.7 Controlling the Hardware

To build our robot, we wanted to use stock Lego sensors and actuators. On the software side, this allowed us to reuse the existing drivers and libraries in order to access these devices. However the architecture of the EV3 software stack made it necessary for us to refactor every library we needed. As the supplied libraries are user space libraries, we also had to find a way to access the device API from kernel mode.

EV3 Software Architecture
To understand the challenges regarding the device libraries, we first have to understand the software architecture of the EV3 as depicted in figure 3.15. Usually, the EV3 is programmed with a graphical programming language designed for education. Those programs are then translated into byte codes. Therefore, the central part of the software stack running on the EV3 is the *lms2012* program, which is basically an interpreter for those byte code programs. The interpreter itself uses several user space libraries, which provide access to input and output devices. They convert data into the right format and write it to the corresponding device files. The device files are again implemented through drivers provided by Lego.

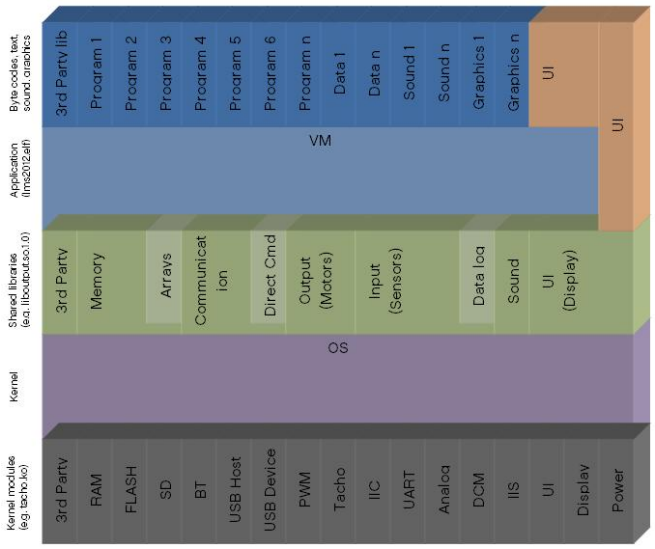

Figure 3.15: The layered software architecture as depicted in the source code documentation of the public EV3 operating system [2]. The VM layer and the user space library layer are depicted as separated.

Creating Self-Contained Device Libraries

The conceptual architecture omits one fact about the device libraries: They are not implemented as self-contained modules. They rely on the existence of the byte code interpreter. However we wanted to solely use the device, without instantiating an interpreter, as the interpreter could introduce all kinds of side-effects changing the timing behaviour of our applications. Still we wanted to reuse the existing libraries as they already implemented all necessary data conversions to control the devices. So in order to make them self-contained we refactored the device libraries by removing all calls to the interpreter. We managed to extract a self-contained library from the motor and the sound library. We struggled with doing so for the sensor library as several layers of abstraction seem to be intertwined in this library. There exist procedures for interpreting input byte codes, using the *Inter-Integrated Circuit* (I^2C) bus protocol and decoding sensor values depending on the type of sensor attached.

Using the Device Libraries from Kernel Mode

As discussed, we implemented our experiment in a kernel module. However, our refactored libraries are only usable from the user space, because they write to the device files using the user space file API. As this API isn't available in kernel mode, we thought of two solutions for using the libraries anyway. The first and best solution would be to create a kernel device library using the kernel API to access the device file or directly communicate with the driver object. As this would have meant to

rewrite the library, we used a second provisional solution. We used the concept of an up-call to start a user space application, which then used the device libraries to control the physical actuators (listing 3.5).

Listing 3.5: Upcall to call a user mode program, which uses the device libraries to control the motors.

```
ret = call_usermodehelper("/home/root/motor_control",
                          argv, envp, UMH_WAIT_EXEC);
```

Of course, this can only be provisional, as an upcall introduces all kind of timing challenges. For example, the priority of the called process has to be set correctly to interrupt the running kernel process. Additionally, the time needed for the context switch becomes part of the delay in controlling the actuators, which is undesirable in a real-time system.

3.4.8 Observations and Results

Our goal was to evaluate the effectiveness of applying the `RT_PREEMPT` patch regarding the handling of priority inversion. Therefore, we constructed a task set which is likely to suffer from priority inversion. Additionally, we implemented a watchdog, which observed the running tasks. In the case of a missed deadline, it stopped all actuators and terminated the control program.

As an experiment, we ran our task set on a robot running an ordinary Linux and one running the patched RTLinux. We observed that the robot running an ordinary Linux in all cases stopped after approximately five seconds. The one running RTLinux continued its journey. These physical results of one robot stopping and the other one carrying on is rather unremarkable. Actually, the internal behaviour of the mission program is of interest. The ordinary Linux encountered the intended priority inversion. The watchdog, running reliably with the highest priority, detected the error and stopped the robot. Finally, the overall result of the experiment is visible from an architectural point of view. By only patching the underlying operating system kernel, we successfully solved a timing problem of a user-provided application.

3.5 Discussion

The experiment demonstrates the advantage of patching a general-purpose operating system to allow real-time timing constraints: An established operating system API can be used and timing behaviour is guaranteed. However, we had to force the problem of priority inversion by implementing our experiment with mechanisms

which were not yet real-time enabled. Considering all this effort to break the system, the question arises whether it is actually still necessary to patch a Linux kernel to gain real-time behaviour.

Regarding the implementation of our experiment, we mainly struggled with the cross-platform development process. In particular, we encountered most problems when communicating with the actual physical EV3 running our system. In retrospect, we should have evaluated other options of testing the system.

3.5.1 Future of Real-Time Features in Linux

During our research on the internals of Linux and `RT_PREEMPT`, we found ourselves in the midst of a changing real-time world. The mainstream availability of embedded systems ranging from rapid-prototyping technologies (*RaspberryPi*, *Lego Mindstorms*) to commercially distributed *smart devices* is rapidly increasing due to low hardware costs — and so is the demand for real-time capabilities in Linux.

As we have already seen in section 3.2.3, more and more of the real-time features of RTLinux and the `RT_PREEMPT` patch were integrated into the official Linux kernel during the past few years. By now, almost all features of `RT_PREEMPT` have been successfully merged into the Linux kernel:

- Kernel preemtability can be activated by compiling with `CONFIG_PREEMPT`.
- Priority inheritance is always active, even inside the kernel.
- Threaded IRQs are available, when the configuration option `CONFIG_IRQ_FORCED_THREADING` is activated.
- Hardware high-resolution timers are used, when the option `CONFIG_HIGH_RES_TIMERS` is chosen.

Although it may seem that `RT_PREEMPT` is not necessary anymore from a feature perspective, it still adds about 300 changes to the 3.14 version of Linux. Many of those changes replace spinlocks and thus improve preemptability; some refactorings aim at better predictability and shorter interrupt latency (response time a device experiences after triggering an IRQ). Recent benchmarks confirmed that the `RT_PREEMPT` patch still exhibits significantly better timing behavior than just `CONFIG_PREEMPT` in terms of both mean and variance of interrupt or task switch latency [4, 13].

Considering these observations, we are confident that `RT_PREEMPT` will be maintained alongside Linux for a few more years by a growing community. It is also quite reasonable to think of mainline Linux as a place for thoroughly tested and matured parts of `RT_PREEMPT` and not as a future replacement thereof.

3.5.2 Remarks on the Development Process

As we presented in section 3.3.2, our RT_PREEMPT kernel does not support the drivers for the EV3 USB WLAN dongle. For our development process, this meant that deploying and debugging of our application via WLAN or network connection was not possible. Therefore, we had to resort to deployment via SD card (see section 3.3.1) and debugging via serial console (see section 3.3.3). This process proved to be quite effortful and time-consuming. After every change, we had to recompile the application, insert the SD card, mount it, copy the changed files, unmount the card, plug it into the EV3, reboot it and watch for the results on the serial console. Initially, many problems arose due to us forgetting to mount or unmount the SD card, or similarly stupid things. As described in section 3.3.1, we eventually modified the build scripts to prevent these mistakes.

Of course, we also experienced a number of errors in our application sources, and, as we were mostly dealing with concurrent execution and thread synchronization, these were incredibly hard to debug with the means of the serial console only. For us, this meant debugging was only possible via kernel prints. This was a huge effort, because we had to redeploy our application via the SD card whenever we wanted to add debug statements.

If we were to do the project all over again, we would definitely reevaluate the possibilities of emulating the EV3 operating system on a development machine in order to provide a better means of testing and debugging the application before actual deployment to the EV3. Even if emulating the exact EV3 OS and/or hardware was not possible, setting up a virtual machine with a patched RT_PREEMPT kernel and testing the hardware-independent parts of our application (for example, the synchronization mechanisms) on this VM beforehand could have helped us save a lot of time. We would, therefore, highly recommend any future groups working with the EV3 to investigate any potential virtualization and emulation methods before beginning application development. You cannot always prevent making mistakes and having to debug. We underestimated the time we would have to put into debugging — and in hindsight, we should have put more time and effort into our development tools from the beginning.

3.6 Conclusion

Our goal was to evaluate the handling of priority inversion in a RT_PREEMPT patched Linux, deployed on an EV3. To understand the foundations we would build on, we first investigated on the history and architecture of real-time Linux, in particular the RT_PREEMPT patch.

On the practical side, we implemented the mission program in a cross-platform approach. Therefore, we adjusted the EV3 toolchain to build our extensions. For debugging we used a console connected via a serial connection. The first step towards our real-time experiment was to patch the the Linux kernel running on the EV3. However, we encountered problems as there was no suitable patch for the running kernel version. Consequently, we solved this problem by applying the patch manually. Based on the patched and the original Linux, we implemented a mission program in a kernel module using three processes. To implement the kernel process synchronization we used the standard kernel mutex API. The correct implementation of a task set leading to priority inversion turned out to be difficult due to ambiguous information about the ordering of real-time priorities in Linux. Finally, we conducted the actual experiment. We observed that it was sufficient to run our mission program on a patched Linux system to get a solution for priority inversion.

Through describing this experiment with all its details, we hope to also provide a guideline for everyone who wants to develop embedded software on the EV3 or work with the `RT_PREEMPT` patch, or even both.

References

[1] *Code Sourcery Lite Toolchain version 2009q1-203*. Apr. 24, 2014. URL: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/.

[2] *EV3 modified Ångström Linux and U-Boot GitHub repository*. URL: https://github.com/mindboards/ev3sources.

[3] *EV3 operating system patched with RT-Preempt GitHub tepository*. Apr. 24, 2014. URL: https://github.com/amintos/ev3sources.

[4] Felipe Cerqueira and Björn B. Brandenburg. *A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUS RT*. 2013.

[5] M. B. Jones. *What really happened on Mars*. 1997.

[6] *Linus Torvalds on priority inheritance*. Apr. 24, 2014. URL: http://lwn.net/Articles/178258/.

[7] S. Murrel and T. Kowalski. "A real-time satellite system based on UNIX". English. In: *Behavior Research Methods and Instrumentation* 12.2 (1980), pages 126–131. ISSN: 1554-351X. DOI: 10.3758/BF03201588.

[8] *PuTTy release 0.63*. Apr. 24, 2014. URL: http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[9] O. Ritchie and K. Thompson. "The UNIX Time-Sharing System". In: *Bell System Technical Journal, The* 57.6 (July 1978), pages 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x.

[10] *RT_PREEMPT Patch on the Real-Time Linux Wiki*. May 26, 2014. URL: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO.

[11] *The Ångström Distribution*. May 26, 2014. URL: http://www.angstrom-distribution.org/.

[12] *Threaded IRQs on Linux PREEMPT-RT*. 2009. URL: http://www.artist-embedded.org/docs/Events/2009/OSPERT/OSPERT09-Henriques.pdf.

[13] H. Toyooka. *Evaluation of Real-time Property in Embedded Linux*. May 26, 2014. URL: http://events.linuxfoundation.org/sites/events/files/slides/toyooka_LCJ2014_v10.pdf.

[14] V. Yodaiken. *The RTLinux Manifesto*. 1999.

4 Carrera Racing Track

4.1 Introduction

Carrera's slot car racing track has been around for a couple of years now. Starting with an analog system where a slot car can only drive on one lane of the track, the racing tracks have evolved into a digital system with junctions, intersections and other special track segments. The use of a digital protocol to control the slot cars also allows for more than two players to race against each other. This project aims at providing a framework for building arbitrarily complex control applications, ranging from rather simple ones like determining the maximum speed one can use to safely drive around the entire track to more sophisticated ones like finding the optimal strategy for a given track and driving even faster than a human being.

The following sections briefly describe the racing track and the slot cars and how we determine a car's position on the track.

4.1.1 Track

The slot car racing track consists of single segments, each with two lanes of metal. The slot cars are held by these metal lanes and obtain their supply voltage from them. There are straights in different lengths and curves in several radii and angles available. With these segments it is possible to build very complex race courses. In contrast to the analog slot car racing track, the digital one allows to drive multiple slot cars on one lane.

Besides these passive segments there are also active ones such as junction segments which are controlled by a photo transistor and a decoder. The photo transistor receives the infrared light from an infrared LED on the bottom of each slot car. The decoder can identify the ID of the passing car by evaluating the time the LED is turned on (the longer the LED is turned on, the higher the car ID). With this car ID the decoder can listen to the digital track signal and decide whether it should switch the junction for the passing car or not.

4.1.2 Slot Cars

The slot cars have an electric motor which drives the wheels. The supply voltage of 14 V is obtained from wiper contacts from the metal lane. Each car has a built-in decoder which receives the digital track signal.

The decoder controls the electric motor with pulse width modulation. There are 15 different speed levels and one brake level available. It has been measured that the decoder only controls the motor's force and not the rotation velocity. Therefore, the resulting velocity of a car is mainly influenced by rolling resistance and air drag.

A decoder can be programmed to listen to exactly one ID and to interpret the digital track signal only for that ID. This signal contains information about every ID, such as the desired speed level. In addition to the decoder each car has an infrared LED which is located on the bottom of the car facing the track. The LED emits a pulsed light signal whose period depends on the programmed ID of the decoder. It is possible to recognize the ID of the slot car on special points on the track, such as junctions or the starting and finishing straight.

4.1.3 Sensors

There are position sensors installed by Carrera at certain points on the track, which can recognize the programmed ID of a passing slot car. These sensors are placed in front of each junction in that way, that the decoder in the junction is able to decide with the help of the track signal if the junction should switch or not. The start finish straight has also a position sensor for each line to determine the lap times for every slot car. Assuming an ordinary usage of the Carrera slot car racing track these position sensors are engough. But for a more precise position and velocity estimation it was necessary to develop and install more position sensors. The additional position sensors consists of a photo transistor which receives the LED's emitted infrared light. The signal of the transistor is reinforced by another transistor and will be transmitted to a central sensor board by wire. We placed the additional sensors on both lanes before and after each curve.

4.2 Related Work

The introduction of Carrera's digital racing tracks included a so-called ghost car, i.e. an automatically controlled slot car. Although it is possible to specify the maximum speed of these ghost cars, the cars are not able to change their speed but always use the specified one. Thus, one has to make sure that the specified maximum speed is suitable for all track sections or the car will jump off of the track.

Several people tried to create an improved ghost car for digital racing tracks as well as initial ghost car implementations for analog racing tracks. Of course the basic idea is to provide some kind of control software with information about the car's position on the track and the course of track ahead. Based on these information the control software then sets the car's speed for the upcoming track section.

While some of the following implementations modify the slot car and control it autonomously, others use external hardware and leave the slot car more or less unmodified.

4.2.1 Carrera Control Using Artificial Intelligence (2003)

In their diploma's thesis at University of Applied Sciences Regensburg, Dinauer and Dinauer described a system that controlled a slot car on an analog Carrera racing track using neural networks [1]. The rather small setup employed photoelectric sensors to detect the car's position on the track and used an unmodified slot car, i.e. all control commands were sent through the racing track, allowing the car to be used by both humans and the control system. Dinauer and Dinauer did not attempt to change the car's velocity in a continuous manner; their neural network only calculates new velocities at discrete points in time: whenever the slot car passes a photoelectric sensor.

4.2.2 Ghost Car with Photoelectric Sensor (2009)

Stephan Heß enhanced a digital Carrera slot car with an reflective photoelectric sensor that allows the car to detect markers on the racing track [6]. Heß used these markers to encode the maximum speed levels for different track sections. This allowed him to improve the ghost car's performance while keeping its ability to switch lanes. Additionally, with only a small amout of changes made to the original slot car, it can still be used by human player.

4.2.3 BlueRider (2009)

At the Department of Computer Sciences at Konstanz University of Applied Sciences, Nagel and Urbanietz developed a prototype of Carrera slot car that is able to communicate with a host computer per Bluetooth [16, 22]. The host computer receives sensoric data from the slot car, which was enhanced with a tachometer on the its rear axle and an accelerometer, and sends control information, e.g. engine output, back to the car.

In his bachelor's thesis Nagel later extended this prototype to receive infrared signals from the track that provide the car with information about its current position,

the state of the track ahead, e.g. if there is another car blocking the following track section, and if it possible to change to a different lane [15]. In addition to the infrared receivers that are already on the racing track and receive information from the car, Nagel installed infrared emitters on the track that allow the slot car to receive the aforementioned information about its position on the track. Moreover, photoelectric sensors in the slot of the track provide another way of determining the position of a passing car.

4.2.4 Controlling a Slot Car with an Android Device (2010)

Grant Skinner used his Android phone to accelerate and decelerate a slot car on an analog racing track [21]. The phone connects per Wi-Fi to a host computer which in turn is connected over USB to a PhidgetMotorControl. The PhidgetMotorControl then increases the eletrical current on the racing track as the phone is tilted which effectively accelerates the slot car.

4.2.5 Carrera Project Lab (2012)

The Department of Electrical Engineering and Computer Sciences at Münster University of Applied Sciences has a Carrera Project Lab that follows two approaces for controlling a slot car.

Similar to the GhostCar project, one approach replaces the car's control unit with a custom microcontroller that directly controls the car's speed depending on the centrifugal forces acting upon the car [19]. This mapping is proportional, i.e. the car has no memory of track sections it has already passed.

The other approach uses a camera above the racing track. Using image processing techniques a computer observes the slot car's position on the track. In this scenario a separate "microprocessor black box" controls the speed of the car by emulating a Carrera handset controller [20].

4.2.6 GhostCar Project (2013)

The GhostCar project [3] modified a slot car by adding a gyroscope to the car's chassis and a reflective photoelectric sensor to its rear axle. The gyroscope is used to detect the course of the track and to identify curved and straight segments. The reflective photoelectric sensor counts rotations of the car's rear axle which is proportional to its speed – assuming the wheels do not slip.

Using these two measurements the GhostCar project is able to continuously estimate the car's position on the racing track and adjust its speed appropriately. However, the GhostCar can be considered as a stand-alone product as it does not under-

stand any of the Carrera track's digital messages, i.e. the car can not be controlled by a human player anymore.

4.3 Signal Detection

For the control software to be able to make informed decisions, we need to detect to kinds of signals:

- The Carrera Control Unit transmits a **track signal** on the track's supply voltage [5]. This signal includes control information for every slot car, e.g. the car's speed level and whether it should switch lanes, and some status information for the entire track, for example if the pace car is driving around the track [5].
- When a slot car passes a **position sensor**, the sensor's photo transistor emits a signal that encodes the ID of the passing car.

With the combination of the track signal and the position sensor signals we are able to create a model of how fast the slot cars are travelling and where they are on the track. The following sections deal with the detection and decoding of these signals.

4.3.1 General Approach

Both the signal lines of the photo transistors and the track's supply voltage are connected to the I/O pins of the microcontroller on the sensor board. The microcontroller interprets the analog signals from the signal lines and presents them as digital signals, i.e. logical ones and zeroes, to the firmware. Also, the microcontroller can generate interrupts when the digital signal changes. This allows for three different approaches for decoding the information in the digital signals.

Round-robin, no interrupts. Each input pin is queried and processed periodically. The processing contains a comparison with a previous stored value. This is necessary to detect input signals which have changed since the last processing. Afterwards, further processing which depends on the concrete input signal can be performed.

Listing 4.1: Signal detection using a round-robin approach.

```
1 int main() {
2     int i;
3     while (true) {
4         for (i=0; i<position_sensor_count; i++)
```

```
5               if (position_signal_has_changed(i))
6                   process_position_signal(i);
7
8           if (track_signal_has_changed())
9               process_track_signal();
10      }
11  }
```

The advantage of this approach, which is shown in listing 4.1, is a simple and easy to understand programming structure. A disadvantage could be the high frequency in which every input pin has to be handled to avoid a miss of a rising or falling edge. If one period takes too long it can happen that the signal goes from high to low and then to high again without recognizing it. Furthermore, the concrete point in time of the signal edge can not be determined.

Interrupts only. Using this approach, an interrupt service routine (ISR) is registered on every necessary input pin. These ISRs are executed when the input signal changes on that pin. The processing in the ISRs is similar to the round-robin approach.

Listing 4.2: Signal detection with interrupts. The interrupt service routines are called when the digital signal of one of the specified pins changes.

```
1  int main() {
2      enable_interrupts();
3
4      while (true)
5          ; /* nothing to do here */
6  }
7
8  ISR (PORTR_INT0_vect) {
9      /* decode the track signal here... */
10 }
11
12 ISR (PORTC_INT0_vect) {
13     /* decode position signals here... */
14 }
```

Like the round-robin approach, the code shown in listing 4.2 is relatively easy to follow. The ISR approach has the additional advantage that the ISR code is only ever executed when the signal on one of the input pin changes. During an interrupt, other interrupts are masked and executed afterwards, but it is not guaranteed that the signal has not changed until the masked ISR is ready to execute. Thus, if the interrupt service routines take too long, we might miss other signal edges.

Hybrid. Round-robin with interrupts. This approach is a combination of both the round-robin and interrupt approach. The ISRs are executed whenever a signal edge occurs but we try to keep them as short as possible. As shown in listing 4.3, every ISR only stores the significant data about the interrupt in a buffer which is later processed by the main-loop in a round-robin manner.

With the short execution times of the interrupt service routines it is rather unlikely that we miss a signal edge. Unfortunately, the code is now harder to follow. Moreover, since the main-loop may be interrupted by the ISR execution, we need to implement a rudimentary sychronization scheme in order to prevent the ISRs from overwriting state variables that are currently processed in the main-loop.

Listing 4.3: Signal detection using a hybrid approach. The interrupt service routines store relevant information about the interrupt in a buffer. The main loop later evaluates the contents of this buffer.

```
1  int main() {
2      enable_interrupts();
3
4      while (true) {
5          if (!event_queue_is_empty())
6              process_next_event();
7      }
8  }
9
10 ISR (PORTR_INT0_vect) {
11     store_relevant_information_in_event_queue();
12 }
13
14 ISR (PORTC_INT0_vect) {
15     store_relevant_information_in_event_queue();
16 }
```

4.3.2 Track Signal

Manchester Code

The detection of the track signal is done with a combination of interrupt and round-robin approach. Inside the ISR, the Manchester code is decoded bit wise and stored in a temporary buffer.

The Manchester code can be decoded by saving the last decoded bit and measure the time until the next signal edge occurs. If this measured time is equal to known period (100 µs), the new bit is the inverted from the previous saved one. The other

way is having two edges with a short measured time length 50 µs each. Then, the new decoded bit is the same as the prevoius one. Each decoded bit is inserted in the temporary buffer for later processing. The end of a decoder word can be determined with an elapsing timer. If there is no signal edge for twice the base period, the decoder word has ended and this can be indicated with a flag.

This flag is queried in the main loop and if a new complete decoder word is recognized it will be analyzed to figure out the type of the decoder word depending on the length. Afterwards, it is appended to the USB transmitting queue.

4.3.3 Position Sensors

The signal data from the position sensors are also processed with a combination of round-robin and interrupt. The signals are falling and rising edges which correspond to the pulsed infrared light of the slot cars's id. With a mapping of input pins to specific points on the track it is possible to determine the exact location of a slot car at a time.

To detect the signal of the pulsed light, the timespan between the signal edges have to be measured. ISRs are registered on the falling and rising edge of each input pin. If a rising edge occurs, the current timestamp is inserted into a queue. Using this approach, a further processing can be made with a list of timestamps.

4.4 Data Transmission

Data between the PC and both microcontroller boards is transmitted over USB. There is a separate protocol for the sensor data traffic as well as the control traffic.

4.4.1 Sensor Data Protocol

The sensor board receives position sensor and track signals from the racing track. It parses the signals and creates data packets that eventually will be sent to the PC. The information included in a packet are the packet type, a timestamp and the packet payload.

There are currently four packet types:

Position sensor. The packet payload contains the ID of the position sensor that was passed over and the ID of the controller that passed over the specified sensor.

Controller state. The packet payload contains the decoded 9-bit controller state word that is sent by the Carrera Control Unit.

Control information. The packet payload contains the decoded 12-bit control word that is sent by the Carrera Control Unit.

Activity information. The packet payload contains the decoded 8-bit active word that is sent by the Carrera Control Unit.

But for the PC to be able to decode the received data packets, it has to identify the start of a packet. We looked into two possible packet layouts that provide this framing information: packets with a start bit and packets with a payload length field.

Start Bit Packet Layout

The start of a packet is identified with a byte whose most significant bit is set. All other bytes in the packet payload must not have their most significant bit set. As figure 4.1 shows, this packet layout makes it pretty easy to identify the start of packet. Unfortunately, the most significant bits separate multi-byte values which makes sending arbitrary byte streams rather cumbersome. Since we do not need to send arbitrary data but only have four kinds of messages, we could have adapted the actual data protocol to fit in this packet layout.

Payload Length Packet Layout

With the packet layout that is shown in figure 4.2 we use a length field in the packet header to determine the end of a packet rather than the start of the next packet. Since the control information are only stored in the packet header, there are no restrictions on the actual packet payload, which allows us to send multi-byte values like the timestamp continuously. Thus, we can simply interpret a packet as a C structure in order to decode it.

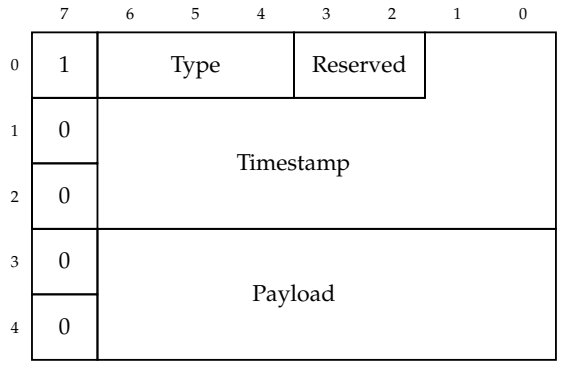

Figure 4.1: Start bit packet layout

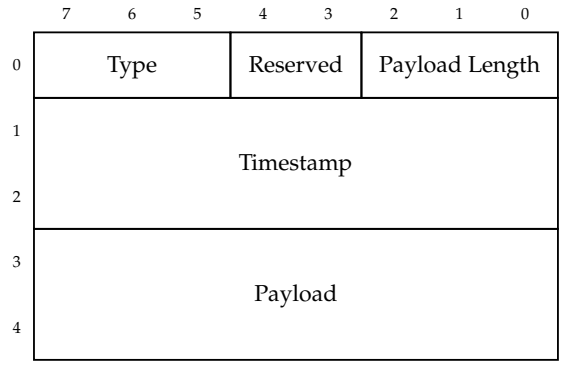

Figure 4.2: Payload length packet layout

Listing 4.4: Function prototype for sending sensor data

```
void usb_send_sensor_data(
    sensor_type_t type, const uint8_t *payload, size_t length
);
```

Comparison
As a first observation, the start bit layout can be considered to be more extensible than the payload length layout. Both layouts allow us to define new packet types by declaring new type identifiers, possibly by using some of the currently reserved bits. Concerning the payload extensibility, however, we could easily append additional payload bytes to the end of a message in the start bit layout without breaking previous implementations. The payload length layout, on the other hand, allows us to append up to six additional bytes to the payload of an existing packet type. If we wanted to add more bytes to an individual message, previous implementations may no longer be able to parse those messages.

However, the restrictions on the packet payload (payload bytes must not have their most significant bit set) make the start bit layout rather difficult to use in software.

Sending side. Assume we have the interface shown in listing 4.4. There are (at least) two possibilities to implement this interface.

1. The function masks the most signifcant bit of each payload byte to adhere to the packet layout restrictions. This would create unnecessary high coupling between the calling and the implementing code, i.e. the calling code would have to know that the most significant bit of each byte will not be sent. Also, the calling code would probably have to do some marshalling on its own, e.g. reordering the bits of a 16-bit value (a timestamp for example) to only use the lower seven bits of each byte.

   Although changing the `payload` data type from `uint8_t*` to `uint7_t*` would better communicate that the most signifcant bit of each byte will not be sent, the aforementioned adjustments to the calling code would still be necessary.

2. The function takes the full bytes into account and reorders the bits in a way that does not use the most signicant bit of each byte. Although this provides us with a cleaner interface than the previous method, it still requires additional code that reorders “normal” bytes into “7-bit” bytes.

Receiving side. Regardless of how the function shown in listing 4.4 is implemented, we would again need code that unmarshals “7-bit” bytes into “normal” bytes.

The payload length packet layout, on the other hand, provides us with a cleaner interface and more readable code on both the sending and the receiving side.

Additionally, the payload length layout allows us to process packets as soon as we have received the last packet byte while the start bit layout requires us to wait for the first byte of the next packet. This leads to lower processing latencies which is why we decided to use the payload length packet layout to send packets from the sensor board to the PC.

4.4.2 Control Protocol

The Control Protocol is a single byte protocol that is used to transport information from the PC to the actuator board. The relevant information are which car – or controller – should be affected by the command, which speed level it should use and whether the controller's lane change button should be pressed.

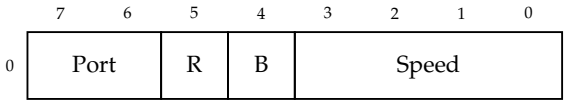

Figure 4.3: Control Protocol for the analog actuator board. The port value determines whether the values in the speed and lane change button (B) fields should be sent to the first, the second or both ports. Which port of the actuator board controls which slot car depends on which port of the Control Unit the actuator board ports are connected to.

During the course of the lecture we worked with two separate actuator boards. The first one emulates Carrera's analog handset controllers and is able to control two different slot cars. This actuator board is connected to two ports of the Carrera Control Unit and the ports it is connected to determine which slot cars the board can actually control. To assign the correct controller ids to a port the application software needs to perform some bootstrapping steps. We send commands to the both ports separately and listen on the track which controller id is affected by which port. Figure 4.3 shows the protocol that is used to communicate with the analog actuator board.

The second version of the actuator board mimics the behavior of Carrera's wifi or infrared adapters which use a digital protocol to communicate with the Control Unit and are able to control six different slot cars. This board is connected to the Control Unit with a single port but because of the digital protocol that is used on this port it can directly tell the Control Unit which controller id should have which speed

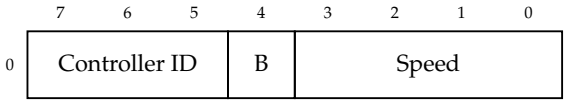

Figure 4.4: Control Protocol for the digital actuator board. The controller id field maps directly to the controller ids that are used on the Carrera racing track. The values 0 through 5 address actual controllers while 6 and 7 are reserved for future use.

level. There is no additional bootstrapping in software necessary since the concept of ports is hidden from the application software. All participants can talk about controller ids. The protocol that is used to communicate with the digital actuator board is shown in figure 4.4.

4.5 Implementations

We implemented two versions of control software that both communicate with the microcontroller to receive sensor data from the racing track but run on different platforms. The first version is a Windows Forms application written in C# running on a Windows 8 Professional. The other one is a Windows console application written in C++ running on a Windows Embedded Compact 7.

Both versions use the same sensor and actuator board and communicate with them with the messages described in the previous sections. In an experiment we tested the message roundtrip times with both versions. Section 4.6 compares the results from these measurements.

4.5.1 C# Application (Windows 8)

The C# application started out as spike to quickly test the microcontroller firmware and the protocol. As it is developed and run on a Windows 8, there is no special setup required except for providing a .NET runtime, and the FTDI drivers and libraries.

4.5.2 C++ Application (Windows Embedded Compact 7)

In addition to the C# application we also wrote a C++ application that would run on an ICOP eBox-3300 VESA-PC with a Windows Embedded Compact 7. The main purpose of the C++ application was to test whether an application would benefit

from running on a real-time operating system. Section 4.6 discusses the results of this test.

In order to conduct the experiment and deploy the application to the eBox, we had to build our own operating system image from the Windoes Embedded Compact sources, include the necessary FTDI drivers in this image, and connect the eBox to our developer machine. We used Platform Builder, which is included in Windows Embedded Compact 7 [11], to achieve all of these tasks.

Setting up Platform Builder

Platform Builder is a Visual Studio extension that is used to compile operating system images for embedded devices [11]. We decided to install Platform Builder in a virtual machine and basically followed the instructions in [13]. However, we had some trouble getting to the first step in the installation guide.

The disk image and key for Windows Embedded Compact 7 that we used were provided by Microsoft DreamSpark Premium. After we started the installation, we were prompted with the dialog shown in figure 4.5. Since the update apparently included some improvements, we decided to install it and the following updates. Unfortunately, this led to our license key being invalid. When we restarted the setup application and skipped the suggested updates, our product key was accepted and we could continue the installation.

We selected three components to be installed: Platform Builder, Shared Source and the x86 Architecture. The Shared Source component contains the source code of the entire Windows Embedded Compact kernel and allows stepping through the kernel source code during debugging [13]. The x86 Architecture includes Board Support Packages for CEPC, Microsoft Virtual PC and our ICOP eBox-3300 [12, 14]. MSDN tells us that "a board support package (BSP) is a set of software components that allows the OS to run on a specific hardware platform" [7].

Building an Operating System Image with a Hello World Application

After we had set up Platform Builder, we followed the guides in [8, 9] (while substituting Virtual PC with eBox-3300 of course) to create an operating system image with a sample application. To download the image to the device, we needed to connect the eBox with our development machine and make sure that Platform Builder recognized the device.

The eBox BIOS has a "Boot from LAN" option which, when enabled, causes the device to broadcast a series of DHCP messages across its local network. Figure 4.6 shows one of these messages in detail. According to [2], the bootfile name parameter indicates the name of a boot image the client can download via TFTP. This parameter seemed to us like a reasonable way to download the operating system image to the eBox. Although the eBox included a request for a bootfile name in its DHCP Discover

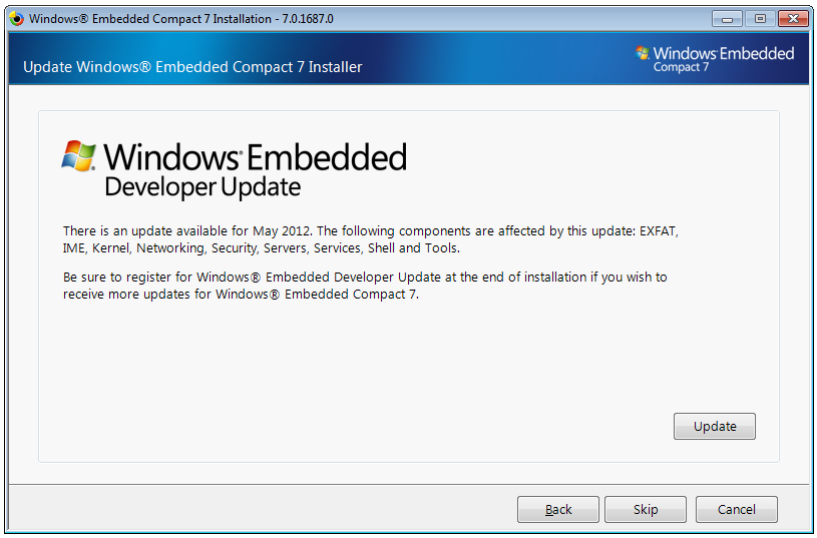

Figure 4.5: Windows Embedded Compact 7 Update Dialog. When the machine that we were trying to install Windows Embedded Compact 7 on was connected to the internet, the setup application asked us to install a series of updates including this update for EXFAT, Networking, Security and others.

messages, Platform Builder did not respond with appropriate DHCP Offer messages. In fact, Platform Builder did not respond to the DHCP messages of the eBox at all.

The guide in [10] describes the connection progress as follows:

> As the vCEPC starts, its boot loader obtains an IP address and then broadcasts BOOTME messages over the network. When Platform Builder receives the BOOTME messages, it recognizes the device.

Apparently, we had to mimic the behavior of the vCEPC and make the eBox send *BOOTME* messages (instead of DHCP or BOOTP messages) at boot time so that Platform Builder would recognize it. The eBox Windows Embedded Compact 7 Jump Start Kit provided by EmbeddedPC.NET contains a preconfigured operating system image with an ethernet bootloader which can be copied to a USB flash drive [17]. Using this flash drive we configured the eBox to boot from USB which loaded the preconfigured Windows Embedded Compact 7 image from the Jump Start Kit. In turn, this image requested an IP address from our local DHCP server, started the Ethernet bootloader, and broadcasted the BOOTME messages which were finally recognized by Platform Builder. Figure 4.7 shows the messages that were sent between the eBox and our development machine during a successful boot of our operating system image with the Hello World application.

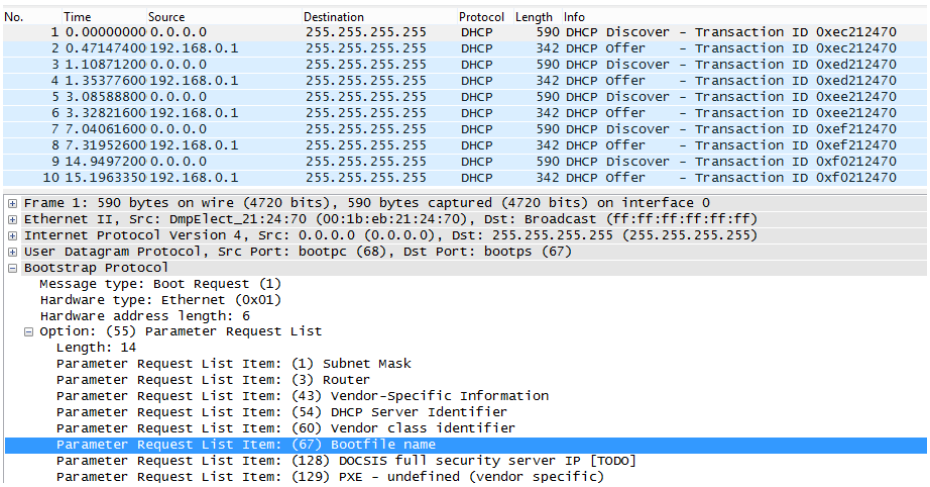

Figure 4.6: Bootstrap messages sent by the eBox (shortened). When the "Boot from LAN" option is enabled in the eBox BIOS, it sends a series of DHCP Discover messages across the network. The Discover messages include a request for a bootfile name which would be downloaded with TFTP [2, 4]. The DHCP Offer messages are *not* sent by Platform Builder but a local DHCP server, which only offers an IP address to the eBox but no bootfile name.

Including FTDI Drivers and Libraries in the Operating System Image

Communicating with the Carrera Racing Track, i.e. with the sensor board and the actuator board, requires FTDI drivers and libraries.[1] To make the drivers available to the application, we need to include them in the operating system image. CEComponentWiz[2] allows us to add additional content to an operating system image as a subproject to an OS design. After we had extracted the zip archive with the FTDI drivers and archives, we used CEComponentWiz to create a thrid party catalog item that we added to our operating system image.

Table 4.1 shows the files we used from the extracted zip archive. `ftdi_d2xx.dll` contains the kernel mode driver and `ftdi_d2xx.inf` its corresponding INF file. The interface library to this driver is `ftd2xx.dll`. We included the development files `ftd2xx.lib` and `ftd2xx.h` so that we could compile and link our application with the FTDI component without having to explicitly add the development files to the application project. Once we had added the required files, we generated the FTDI catalog item with the menu entry Generate Component Project » Publish Component to 3rd Party (and catalog). Afterwards, the component was available in the Catalog Items View of Platform Builder below `Third Party▸Embedded101`.

[1] We used the Windows CE 6.0 drivers from `http://www.ftdichip.com/Drivers/D2XX.htm`.

[2] `https://cecomponentwiz.codeplex.com/`

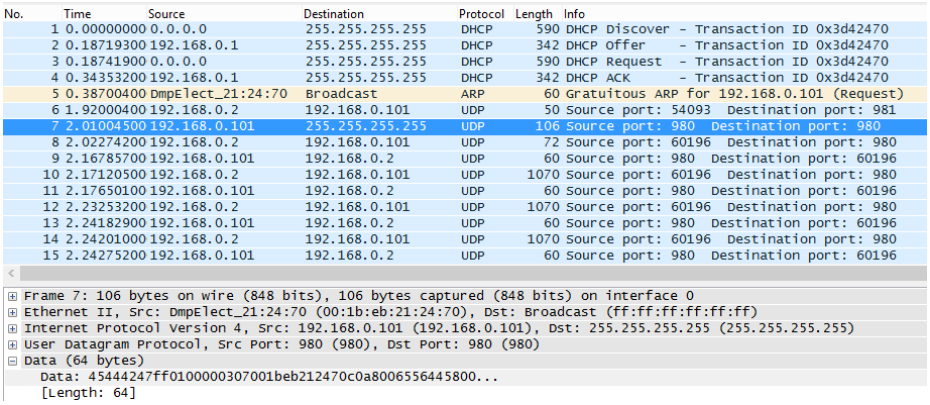

Figure 4.7: Messages sent by the eBox ethernet bootloader. This filtered capture contains messages from the eBox (host `192.168.0.101`), our development machine with a local DHCP server (host `192.168.0.1`), and the virtual machine that runs Platform Builder (host `192.168.0.2`).
First, the eBox acquires an IP address from the local DHCP server (messages 1 to 4). Afterwards it broadcasts a BOOTME message (7) to which Platform Builder responds with a TFTP write request (message 8). Finally, Platform Builder transmits the boot image with TFTP to the eBox (messages 10 and onwards).

Refer to [18, chapter 25] for a detailed explanation of how to add files to an operating system image.

4.6 Comparison

As described in section 4.5, we wrote two versions of a control software that receives messages from the sensor board and sends messages to the actuator board. Both versions use the same protocol stack, but one is written in C# and runs on a Windows 8 while the other one is written in C++ and runs on a Windows Embedded Compact 7. To compare both versions we measured the time it took each version to perform a full roundtrip.

4.6.1 Experimental Setup

In our experiment the control software drives two slot cars around the racing track with a fixed speed level. Whenever a slot car passes a position sensor, i.e. the sensor board sends a position sensor packet with the car id to the control software, the control software sends a command packet to the actuator board telling that particular

Table 4.1: FTDI files included in the OS design. The internal file name column shows the name of the file in the os image. The shortcut folder determines in which path – if any – the file will be accessible in the file system of the OS. The following flags were applied to the files: K – Kernel, S – System, U – Uncompressed, C – Compressed. The files can be found in the original file path; `$(Ftdi)` is the path of the extracted FTDI archive.

Internal file name	Shortcut Folder	Flags	Original file path
`ftdi_d2xx.dll`	None	KSU	`$(Ftdi)\ftdi_d2xx.dll`
`ftdi_d2xx.inf`	Windows	C	`$(Ftdi)\ftd2xx.inf`
`ftd2xx.dll`	Windows	SU	`$(Ftdi)\ftd2xx.dll`
`ftd2xx.lib`	None	C	`$(Ftdi)\ftd2xx.lib`
`ftd2xx.h`	None	C	`$(Ftdi)\ftd2xx.h`
`ftd2xx.reg`	None		`$(Ftdi)\inf files\ftd2xx.reg`

car to stop. After the software receives a controller state packet with the same car id, it measures the time for the whole roundtrip and tells the car to continue driving. This process was repeated 1000 times for each version of the control software.

The path of signals and messages for a single sensor passing as depicted in figure 4.8 is as follows:

1. A slot car passes a position sensor.
2. The position sensor generates a signal, which is interpreted by the sensor board.
3. The sensor board creates a position sensor packet, which contains a timestamp, and sends it to the application via USB.
4. The application stores the timestamp, creates a command packet that sets the speed of the slot car to zero, and sends it to the actuator board via USB.
5. The actuator board unmarshals the command packet and sends the appropriate signals to the racing track's control unit.
6. The control unit then puts the manchester-encoded signal on the racing track which again can be interpreted by the sensor board.
7. The sensor board creates a controller state packet, which again contains a timestamp, and sends it to the application via USB.

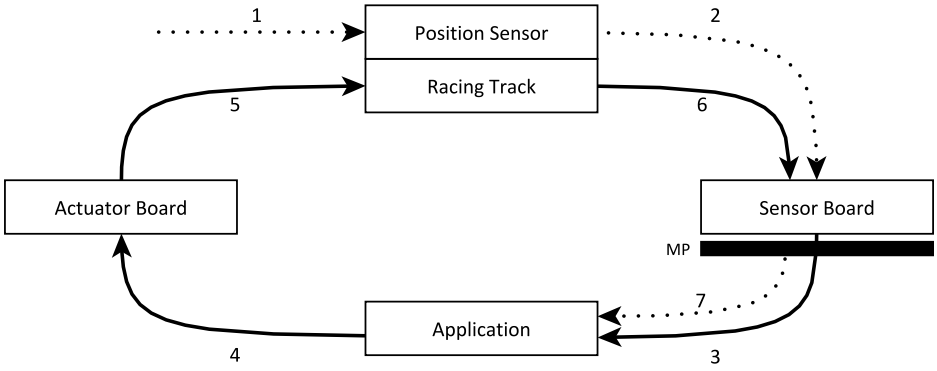

Figure 4.8: Experimental setup. After a slot car passes a position sensor, messages are sent through the entire system. The timestamps of those messages are measured by the sensor board just before sending them to the application where they are evaluated. The measuring point is represented by the black box labeled "MP".

4.6.2 Evaluation

First let us consider the message latencies that we would observe with an optimal system. This perfect system would have the same (constant) amount of processing time t_{fix} for every sensor passing of every slot car. t_{fix} is the time it takes the signals and messages to complete the steps 2, 3, 4, 5 and 7 in figure 4.8 because we can actually control the software and hardware that is involved in these steps. We consider the time it takes the position sensor to recognize a passing slot car, i.e. the time between step 1 and 2 and in extensions the time for step 1 itself, to be constant as well. However, the Carrera Control Unit sends commands to individual slot cars only once every 75 ms [5] and we do not know when exactly it does so. So, the message roundtrip time for our perfect system would be

$$t = t_{fix} + t_{var}$$

with $t_{var} \in [0\ \mathrm{ms}, 75\ \mathrm{ms})$ accounting for the variable delay caused by the Control Unit. Since we do not know when the Control Unit sends its commands to the slot cars, t_{var} is actually a random variable; and because every value of t_{var} is equally likely (a slot car can pass a position sensor at any given point in the Control Unit's command cycle), it has a uniform distribution.

Figure 4.9 gives some example of how t_{var} is affected when a car passes a position sensor at different points in time.

With these considerations in mind we can now compare both versions of our control software to the perfect system. Table 4.2 shows the minimum, maximum

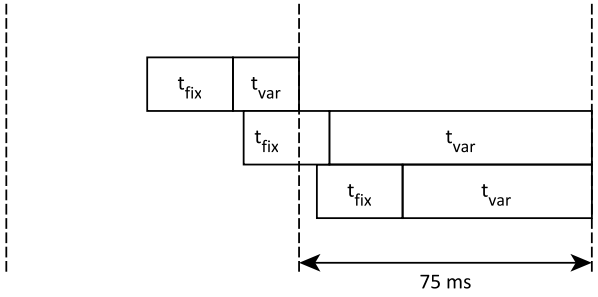

Figure 4.9: Possible delays introduced by the Control Unit. With t_{fix} being constant, the actual value of t_{var} depends only on the time when a slot car passes a position sensor. In the first line of this diagram the slot car passes the position sensor early enough so that we can send our control command to the Control Unit before it sends its command to the car causing a rather small value of t_{var}. In the second line we just missed that deadline and the Control Unit will send our command only after an additional 75 ms.

Table 4.2: Message latencies for both versions of the control software and their theoretical optimal counterparts. The values in the theoretical optimum column assume that $t_{fix} = \min(t) = 17$ ms.

	C++ (WEC 7)	C# (Win 8)	Theoretical Optimum
Minimum latency [ms]	17	17	17
Maximum latency [ms]	100	100	92
Average latency [ms]	58.4	57.9	54.5
Variance [ms²]	461.85	482.65	481.25

and average message latencies as well as the variance for both versions and the perfect system with t_{fix} being the minimum latency of the real ones. The entire value set is plotted in a bar chart in figure 4.10. Finally, figure 4.11 shows the cumulative distribution functions of roundtrip times for all three systems.

As we can see, both the C++ version and the C# version of our control software do not have a constant processing time for each sensor pass. Otherwise $\min(t) = \max(t) - 75$ ms would hold like it does for the perfect system. Unfortunately we could not yet determine which component causes this deviation but we suspect the USB connections between sensor board, actuator board and the control hardware to be the culprit.

Interestingly though, both versions show rather similar characteristics with respect to the cumulative distribution function in figure 4.11. We had expected to

observe a more layered image for the C# application because of scheduling and most importantly garbage collection. However, we could not determine whether garbage collection has any effect on the timing behavior of the C# application.

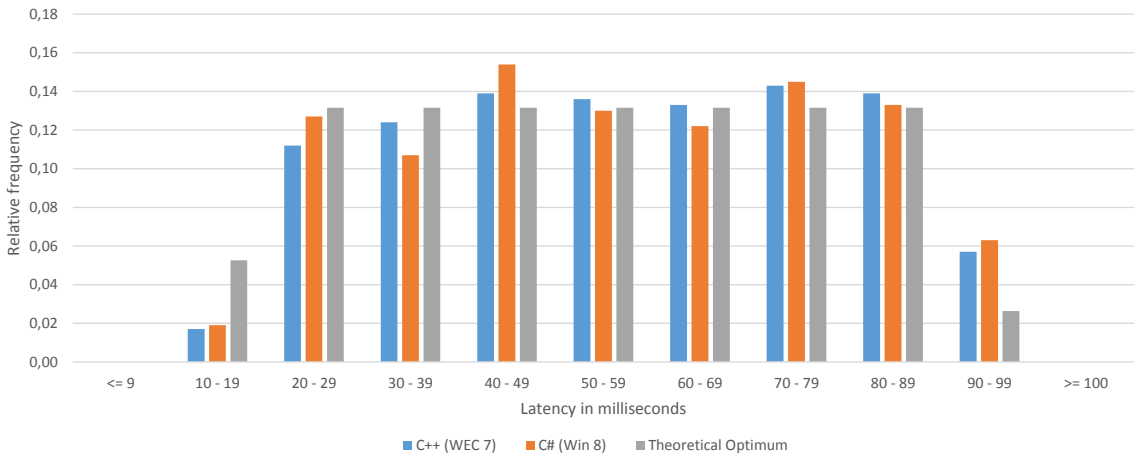

Figure 4.10: Probability mass function of message latencies sampled from 1000 messages. Blue bars show the relative frequencies of message latencies with the C++ application running on Windows Embedded Compact 7. Orange bars show the relative frequencies of message latencies with the C# application running on Windows 8. The gray bars indicate the probabilities of message latencies for a perfect system with the same minimum latency as the tested systems.

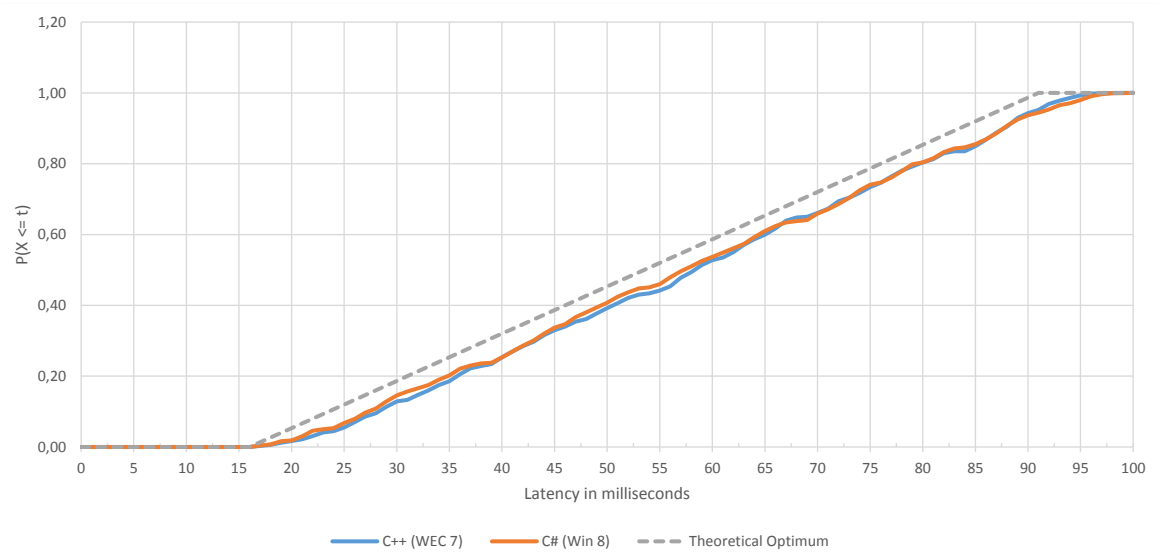

Figure 4.11: Cumulative distribution function of message latencies sampled from 1000 messages. The blue line shows the probability of a roundtrip being completed in less than t milliseconds with the C++ application running on Windows Embedded Compact 7. The orange line shows the same for the C# application running on Windows 8. The dashed, gray line indicates the probability for roundtrip times in a perfect system with the same minimum latency as the tested systems.

References

[1] S. Dinauer and C. Dinauer. "Aufbau einer Rennstrecke zur Kontrolle eines Carrera Rennbahnfahrzeugs mit Programmen der künstlichen Intelligenz und Steuerelektronik zur Simulation eines Mensch-Maschine-Systems". Diploma Thesis. Regensburg: Fachhochschule Regensburg, Aug. 2003.

[2] R. Droms. *Dynamic Host Configuration Protocol*. RFC 2131. Mar. 1997. URL: http://tools.ietf.org/html/rfc2131 (visited on 2014-05-24).

[3] A. Füsser. *GhostCarProjekt*. 2013. URL: http://www.mikrocontroller.net/articles/GhostCarProjekt (visited on 2014-04-30).

[4] J. Gilmore and W. J. Croft. *Bootstrap Protocol*. RFC 951. Sept. 1985. URL: http://tools.ietf.org/html/rfc951 (visited on 2014-05-24).

[5] S. Heß. *CU Daten-Protokoll*. 2007. URL: http://slotbaer.de/index.php/carrera-digital-124-132/9-cu-daten-protokoll (visited on 2014-05-19).

[6] S. Heß. *Ghostcar Etikettensteuerung*. May 2009. URL: http://www.slotbaer.de/index.php/slotbaer-projekte-digital/28-pd132-d124/38-ghostcar-etikettensteuerung (visited on 2014-05-06).

[7] Microsoft Corporation. *Board Support Package (BSP) (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/gg156127(v=winembedded.70).aspx (visited on 2014-05-24).

[8] Microsoft Corporation. *Create an Application (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/jj200344(v=winembedded.70).aspx (visited on 2014-05-19).

[9] Microsoft Corporation. *Design Your First OS (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/jj200351(v=winembedded.70).aspx (visited on 2014-05-19).

[10] Microsoft Corporation. *Download the OS to the Device (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/jj200350(v=winembedded.70).aspx (visited on 2014-05-19).

[11] Microsoft Corporation. *Getting Started (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/jj200349(v=winembedded.70).aspx (visited on 2014-05-19).

[12] Microsoft Corporation. *ICOP eBox 3300 Development Kit (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/gg155938(v=winembedded.70).aspx (visited on 2014-05-24).

[13] Microsoft Corporation. *Installation (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/jj200354(v=winembedded.70).aspx (visited on 2014-05-19).

[14] Microsoft Corporation. *x86 BSPs (Compact 7)*. Mar. 12, 2014. URL: http://msdn.microsoft.com/en-us/library/ee479172(v=winembedded.70).aspx (visited on 2014-05-24).

[15] M. Nagel. "Development of a Realtime Infrared Communication System, Connected to a CAN Bus". Bachelor's Thesis. Konstanz: HTWG Konstanz, Aug. 2009.

[16] M. Nagel and D. Urbanietz. *Platinendesign BlueRider*. Sept. 2009.

[17] S. Phung. *eBox-3310A-MSJK Compact 7 jump start kit*. Aug. 8, 2011. URL: http://www.embeddedpc.net/eBox3310AMSJK/ (visited on 2014-05-24).

[18] S. Phung and D. Jones. *Introducing Compact 2013*. June 17, 2013.

[19] P. Richert. *Autonomes Auto für eine digitale Carrera-Bahn*. 2012. URL: https://www.fh-muenster.de/fb2/labore_forschung/kt/projekte/autonomes_auto_pset.php (visited on 2014-04-30).

[20] P. Richert. *Inside Carrerabahn*. 2012. URL: https://www.fh-muenster.de/fb2/labore_forschung/kt/projekte/index.php (visited on 2014-04-30).

[21] G. Skinner. *gskinner.com | gBlog*. June 15, 2010. URL: http://gskinner.com/blog/archives/2010/06/air_for_android.html (visited on 2014-05-03).

[22] D. Urbanietz and M. Nagel. *Steuerung eines Autos über Bluetooth*. Aug. 2008.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
89	978-3-86956-296-4	**Embedded Operating System Projects**	Andreas Grapentin, Kirstin Heidler, Dimitri Korsch, Rakesh Kumar-Sah, Nicco Kunzmann, Johannes Henning, Toni Mattis, Patrick Rein, Eric Seckler, Björn Groneberg, Florian Zimmermann
88	978-3-86956-282-7	**HPI Future SOC Lab : Proceedings 2013**	Meinel,Christoph; Polze, Andreas; Oswald, Gerhard; Strotmann, Rolf; Seibold, Ulrich; Schulzki, Bernhard (Hrsg.)
87	978-3-86956-281-0	**Cloud Security Mechanisms**	Christian Neuhaus, Andreas Polze (Hrsg.)
86	978-3-86956-280-3	**Batch Regions**	Luise Pufahl, Andreas Meyer, Mathias Weske
85	978-3-86956-276-6	**HPI Future SOC Lab: Proceedings 2012**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Strotmann, Ulrich Seibold, Bernhard Schulzki (Hrsg.)
84	978-3-86956-274-2	**Anbieter von Cloud Speicherdiensten im Überblick**	Christoph Meinel, Maxim Schnjakin, Tobias Metzke, Markus Freitag
83	978-3-86956-273-5	**Proceedings of the 7th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch (Hrsg.)
82	978-3-86956-266-7	**Extending a Java Virtual Machine to Dynamic Object-oriented Languages**	Tobias Pape, Arian Treffer, Robert Hirschfeld
81	978-3-86956-265-0	**Babelsberg: Specifying and Solving Constraints on Object Behavior**	Tim Felgentreff, Alan Borning, Robert Hirschfeld
80	978-3-86956-264-3	**openHPI: The MOOC Offer at Hasso Plattner Institute**	Christoph Meinel, Christian Willems
79	978-3-86956-259-9	**openHPI: Das MOOC-Angebot des Hasso-Plattner-Instituts**	Christoph Meinel, Christian Willems
78	978-3-86956-258-2	**Repairing Event Logs Using Stochastic Process Models**	Andreas Rogge-Solti, Ronny S. Mans, Wil M. P. van der Aalst, Mathias Weske

ISBN 978-3-86956-296-4
ISSN 1613-5652