University of Kaiserslautern
Department of Computer Sciences
Software Engineering Research Group
Prof. Dr. Dieter Rombach

Fraunhofer Institute for Experimental Software Engineering
Department of Component Engineering
Fraunhofer-Platz 1, 67663 Kaiserslautern

Project Thesis

Design and Implementation of a Framework for Automatic Measurement of Embedded-Software Energy Consumption in an XScale Platform

Jonas Mitschang *(jonas@mitschang.net)*

January 19, 2007

Supervisor:
Samir Amiry *(samir.amiry@iese.fraunhofer.de)*
Prof. Dr. Dieter Rombach *(rombach@informatik.uni-kl.de)*

I hereby declare that I have self-dependently composed the student research project report at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

January 19, 2007 – Kaiserslautern

Abstract

Embedded systems are becoming more and more important in today's life in many ways. They can be found in dishwashers, mobile phones, coffee machines, PDAs, etc. Although there is no common definition of what an embedded system is, it can be generally defined as a special-purpose information processing system, containing both: software and hardware. Embedded systems are integrated in a larger systems which interact with environment for achieving a set of predefined tasks or applications.

In general, embedded systems are characterized by resources scarcity, among which energy is becoming more and more important (especially the energy consumed by the processor). The energy consumed by an embedded system is strongly influenced by the software running on it (the embedded software). That is why it is crucial to explore the software characteristics that have an influence on the energy consumption, and to understand how this influence could be represented. In order to realize this task, there is a need for the construction of a reliable measurement platform for energy consumption by embedded devices.

The target of this work is to design and implement a framework for measuring energy consumption of embedded software. This framework is based on the XScale[1] architecture, a popular Intel[14] platform designed for energy aware applications.

The framework has a software repository which contains a number of programs (user-defined) that are supposed to run on the mentioned platform. These program codes are the input of the framework. Automated measurements for energy consumption are performed on all programs for gathering the required information. In the context of this work, a first evaluation of the framework was performed to make an initial check its quality.

Keywords: Embedded software, Intel XScale, power / energy consumption, measurement, framework

Contents

1. Introduction **6**

2. Embedded Systems and Energy Consumption **8**
2.1. Embedded Systems 8
2.2. Resources Scarcity 8

3. Related Work **11**
3.1. Simulators 11
3.1.1. Avrora 12
3.1.2. Platune 12
3.2. Real Platforms 13

4. Framework Design **15**
4.1. Framework Requirements 15
4.2. Framework Overview 15
4.3. Energy Measurement Theory 16

5. Framework Implementation **18**
5.1. Development Board 18
5.1.1. JTAG-Adapter and Software 19
5.1.2. Measurement Points 20
5.1.3. Universal Bootloader 21
5.2. Amplifier module 21
5.3. Data Acquisition Module 23
5.3.1. Acquisition Hardware 23
5.3.2. Acquisition software 24
5.4. Program-Loader Module 25

6. Framework Evaluation **27**
6.1. Benchmark Repository 27
6.1.1. Repository Requirements 27
6.1.2. Sorting algorithm repository 28
6.2. Measurement Results 28
6.2.1. Evaluation of the windows boot process 32

7. Conclusion and Future Work **34**

7.1. Conclusion 34
7.2. Future work 34

Appendices **38**

A. U-Boot Command Set **38**

B. Acquisition Software Details **41**

List of Figures

2.1. Embedded systems overview 8
2.2. Example of an embedded system 9
2.3. Development of the battery price per watt hour since May 2003[27] . 9

3.1. Generic simulator 11
3.2. System-on-Chip 13
3.3. Real platforms 13

4.1. Overview over the energy measurement platform 15
4.2. Measuring the core current 16

5.1. DevkitIDP system diagram taken from the datasheet. 18
5.2. DevkitIDP development board based on the Intel PXA255 Application Processor 19
5.3. JTAG interface PCB (left) and schematic (right) 20
5.4. Measuring the core current of the DevkitIDP board 21
5.5. Schematic of the instrumentation amplifier module 22
5.6. Instrumentation amplifier printed circuit board 23
5.7. National Instruments data acquisition card NI-PCI-6032E 24
5.8. Screenshot of the oscilloscope software. 25
5.9. Program-loader module 26

6.1. Screenshot of the oscilloscope application capturing the energy consumption for the Windows boot. 29
6.2. Screenshot of the oscilloscope application capturing the energy consumption for suspending and resuming windows. 30
6.3. Samples of the analogue digital converter card while booting windows. 32
6.4. Power consumption of the PXA255 processor while booting windows. 32
6.5. Energy consumed by the PXA255 processor while booting windows. . 33

List of Tables

5.1. DevkitIDP Specification Highlights 20

6.1. GNU C compiler optimization levels (from gcc manual page) 28
6.2. Power Consumption Specifications for PXA255 processor 31

1. Introduction

Nowadays mobile applications are becoming more and more important. In the year 2005 79.2 million mobile phones were registered in Germany according to the CIA World Facebook[3]. These mobile applications are mostly battery powered and have to be recharged sooner or later. In order to reduce the energy consumption of a system, it is necessary to know the behavior of the system and its components concerning energy consumption. To investigate this topic, a reliable measurement platform for energy consumption is needed. The energy consumption depends on the used hardware and on the used software. It is important to determine the way software influences the energy consumption of the whole system for being able to improve the software for better usage of energy.

This framework is based on the XScale[1] architecture, which is already optimized for low energy consumption and which is primarily designed for mobile applications like PDAs, mobile phones, mobile gaming consoles etc. In these applications a high battery lifetime and thus low wattage is desired. If a system had energy aware software and thus consumed less energy, batteries with less capacity (which are cheaper) would be used and production costs would decrease. Taking into consideration the huge number of products, a lot of money could be saved and a vantage on the market could be achieved.

If it is possible to estimate the energy consumption of a system while developing it, and if it is possible to reduce its energy consumption by optimizing the software, there would be a huge potential of saving production costs. Companies with energy-aware software have a competitive advantage against companies that do not put efforts into energy efficiency.

Existing platforms have limitations concerning for example the accuracy, so a measuring framework fulfilling the requirements has to be built. The goal of this work is to develop a reliable energy measurement platform (energy consumption of different parts of the system) for a specified platform (XScale architecture) which includes the construction of the needed measuring hardware and the implementation of the required software. The platform is extensible in its qualities and may easily be migrated to other systems.

This work is organized as follows: As this work deals with measuring energy of given systems chapter 2 gives an introduction on embedded systems and why it is of fundamental relevance to improve their energy performance. Chapter 3 provides information about different approaches for gathering information by simulating a system or by direct measurements on the real platform. The design of the energy measurement framework is discussed in chapter 4. For implementing the measurement platform different parts were required. Chapter 5 presents the used framework parts and their implementation. In chapter 6 the framework is evaluated concerning the previously introduced challenges. Finally chapter 7 summarises the achievements of this work and provides an outlook on future work.

2. Embedded Systems and Energy Consumption

2.1. Embedded Systems

Embedded systems are becoming more and more important in today's life. An embedded system is a special-purpose information computing system that is integrated in a larger system that interacts with its environment:

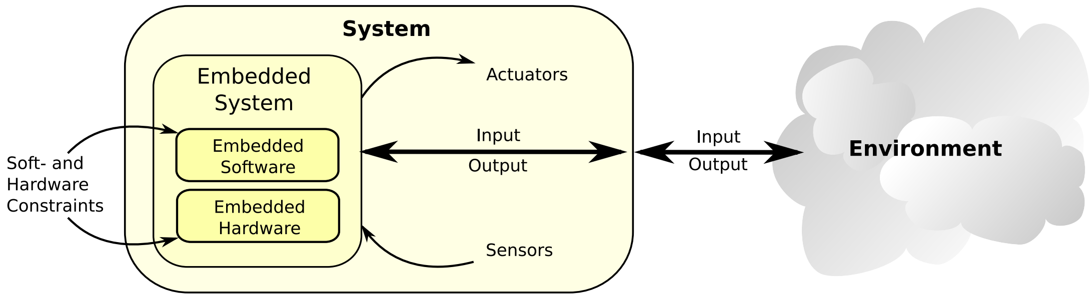

Figure 2.1.: Embedded systems overview

Nowadays, they are used to provide complex add-in values for many systems in different fields of application like e.g. industrial automation, communication technology, consumer electronics, automotive industry and signal processing.

Often these embedded systems (especially for consumer electronics and automotive industry) are battery powered. Certainly a high lifetime is desired for the battery powered embedded systems. The limitations embedded systems have, are described in the next section.

2.2. Resources Scarcity

In general embedded systems are produced in a large number of units (high cumulative costs) or with limitations concerning size or weight. These are the reasons

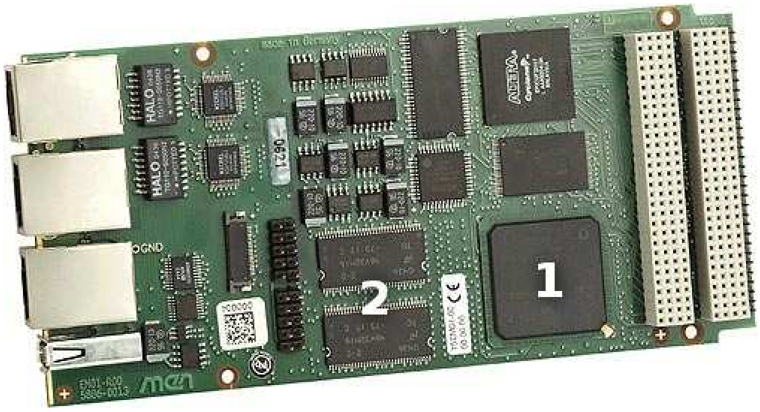

Figure 2.2.: Example of an embedded system
Railway Technology[26] Embedded System Module with PowerPC processor(1) and memory(2).

for resources scarcity of embedded systems: the computing power (processor speed), the available memory and the battery capacity are limited.

To increase the lifetime of a battery powered system there are two options:

- On the one hand it is necessary to have powerful batteries.
- On the other hand the energy consumption of the system should be as optimal as possible.

In order to optimize the energy consumption one should be aware of the effects software has on the energy consumption of the whole system. If the device consumes less energy, it will result into higher lifetime or alternatively smaller (weaker and thus cheaper) batteries will be used and the device will be produced cheaper. As additional bonus this will also reduce the systems physical dimensions.

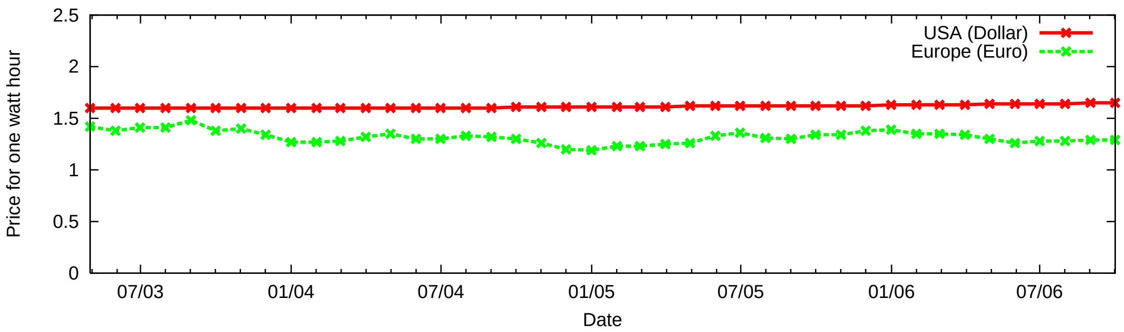

Figure 2.3.: Development of the battery price per watt hour since May 2003[27]

In the last years the price for one watt hour of battery capacity has not changed significantly (Figure 2.3). In May 2003 one watt hour cost 1.60$ while it costs 1.65$ nowadays (October 2006).

Due to the high number of up to hundreds of thousands of devices much money could be saved by optimizing the software for energy consumption. For analysing the effect of software on power consumption in detail, a reliable measurement platform is needed.

Non-functional properties

Non-functional properties are "the qualities we desire of a problem solution other than those concerning its functionality." [2]. Non-functional properties imply a number of design decisions, like e.g. synchronization, protection, reliability, scalability, isolation and last but not least the energy consumption. The non-functional property *production cost* depends on other properties (like energy consumption). It is of fundamental relevance for embedded systems.

3. Related Work

3.1. Simulators

Simulators (Figure 3.1) are used to study how a system works based on a model of this system. A model is a abstract mathematical description of the real system. The simulator is supplied by a given set of controlled inputs and behaves “almost like” the real system to generate the simulated results (simulated output). The simulator behaviour may be modified by parameters and constraints to perform tests and optimizations in detail.

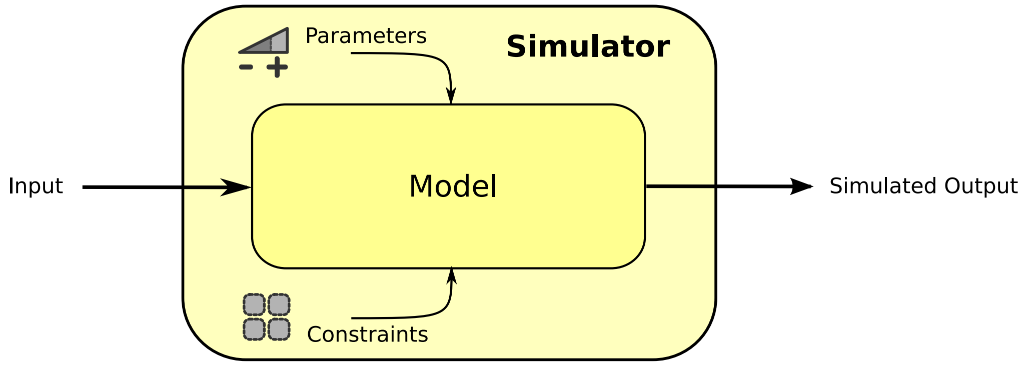

Figure 3.1.: Generic simulator

By changing parameters and inputs (like the software which is running on a specific device) of this system (in this case: embedded system) predictions can be made on how the system reacts to these changes and how the system behaves in this case. Computer simulations are a useful part in many domains but all results are based on the constructed model which is never perfect.

Concerning energy consumption nowadays there exist different simulators in this domain. I want to introduce two of them in this chapter: Avrora (3.1.1) and Platune (3.1.2).

3.1.1. Avrora

Avrora[4] is a research project of the University of California, Los Angeles (UCLA) Compilers Group[8]. It combines a set of simulations and analysis tools for programs written for specific architectures like the AVR microcontroller produced by Atmel[7] and the Mica2 sensor nodes. Avrora includes the following features:

- Simulator for testing programs with cycle accurate execution times
- Profiling utilities
- `gdb` hooks for source-level debugging
- Control flow graph tool for generating graphic flow diagrams
- Energy analysis
- Stack checker

Avrora is a well documented open source project which is written in Java and designed to be easily extendable. Besides all the advantages of avrora there is one major disadvantage: the supported micro controller is not compatible to the XScale instruction set. I want to analyse the energy consumption of complex XScale programs but avrora only supports AVR micro controllers with a maximum of up to 128kB program memory. So avrora is absolutely not suiting the needs in this case.

3.1.2. Platune

Platune [5] is a simulation and exploration environment for a parameterized System-on-Chip (SOC)[6] architecture (Figure 3.2).

System-on-Chip means that all electric components e.g. processor, IO (digital and analog), memory etc. and also the software are embedded into one device. SOC is typically used for embedded systems.

The Platune architecture simulates a MIPS[9] processor:

- MIPS CPU instruction set
- MIPS data and instructions cashes
- CPU to data cache buses
- Memory
- Instruction and data cache to memory bus

Platune comes with a C compiler that compiles the applications for the target MIPS architecture. Platune is limited to the MIPS architecture and does not support

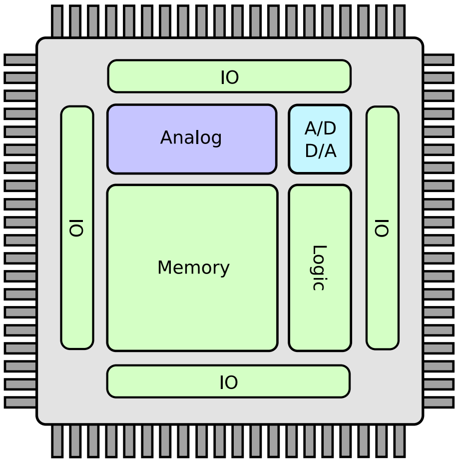

Figure 3.2.: System-on-Chip

the modern energy aware XScale platform yet. Platune can mainly be used for qualitative evaluation of energy consumption as it is not accurate enough and does not meet the demands.

3.2. Real Platforms

A real platform (Figure 3.3) with a hardware processor and peripherals is needed for measuring the real energy consumption without any simplifications that are made by each simulator model (as no model is as accurate as nature).

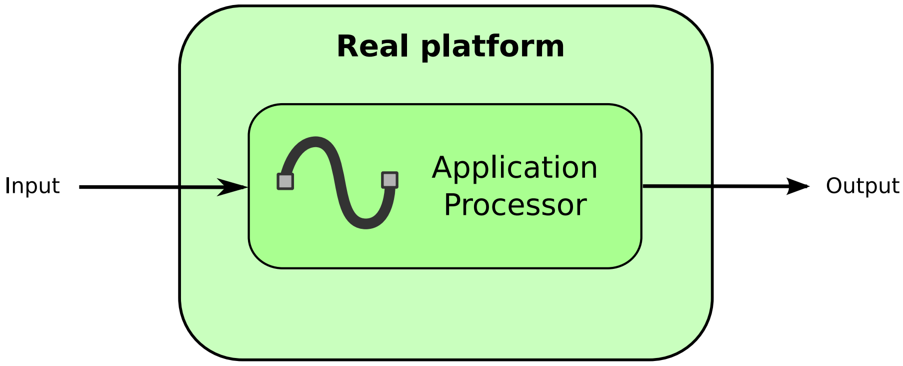

Figure 3.3.: Real platforms

The real platform, alike the simulator, is supplied by a given set of controlled inputs. The difference is that the results (outputs) can be directly measured with the suitable measuring equipment. These real (not simulated) data reflects the physical behaviour (accurate) and not just similarities.

The gathered system information could be used to improve the accuracy of a given power simulator but this is not the consideration of this work. In this case a real platform (XScale DevkitIDP, see 5.1 page 18) is used.

4. Framework Design

4.1. Framework Requirements

The framework measures the energy consumption of the core processor (CPU) of the embedded system (see 4.3). It should be extensible for capturing the energy consumption not only of the core but also of other modules in the future (see 7, page 34). The power consumption measurements made at any point of time have to be written to a log file for being able to analyze the energy of each section of a program.

The acquisition rate of the energy data has to be adjustable. On the one hand this allows to track the energy consumption of function calls of the program that are taking a really short time. And on the other hand this allows to evaluate energy consumption of programs with long runtime (of several hours or days).

4.2. Framework Overview

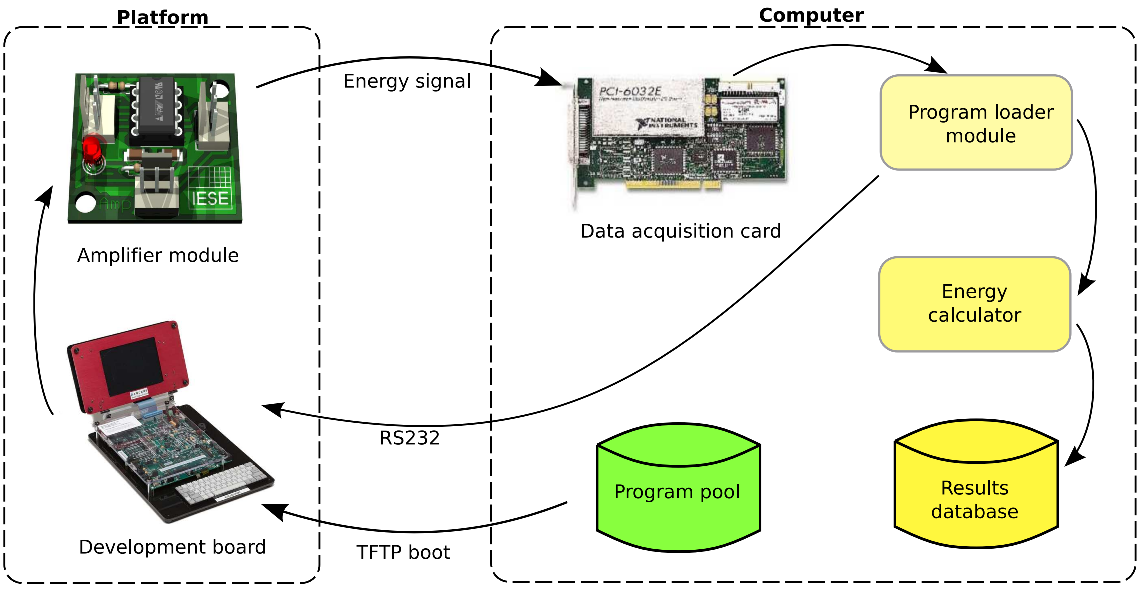

Figure 4.1.: Overview over the energy measurement platform

Figure 4.1 shows the synopsis of the energy measurement platform:

- The *development board* loads the image files from the *program pool* using *TFTP boot* [17].
- One task of the *program loader module* (see 5.4, page 25) is to control the development board using the serial line (RS232) [18]. It ensures that the board loads all program images (images are the compiled program binaries in a specific format) successively and starts them.
- While programs are running, the amplifier module (see 5.2, page 21) amplifies the measured signal and transmits it to the data acquisition card.
- The acquisition card (see 5.3.1, page 23) samples the signal and passes it to the program loader module.
- Inserting the calculated energy values into the result database is done by the energy calculator.

4.3. Energy Measurement Theory

The power P_{CORE} that is consumed by the application processor is the product of the core voltage U_{CORE} and the current flow I_{CORE}:

$$P_{CORE} = U_{CORE} \cdot I_{CORE} \tag{4.1}$$

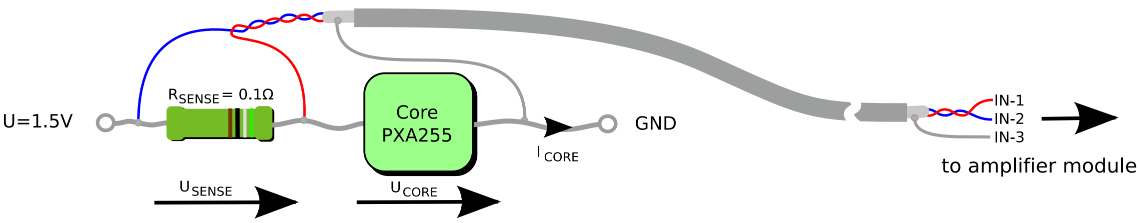

Figure 4.2.: Measuring the core current

According to Kirchhoff's laws, the current through the core I_{CORE} equals the current through the sense resistance I_{SENSE} plus the current through the amplifier module I_{AMP}. As the amplifier module (chapter 5.2) has a very high input impedance of $10G\Omega$ the current I_{AMP} through the amplifier module is very small ($I_{AMP} < 0.1nA$) in comparison to the current through the sense resistance ($I_{AMP} << I_{SENSE}$) and can be ignored in the calculation. The supply voltage U_{CORE} of the core can be calculated based on the voltage drop U_{SENSE} over the resistance (Figure 4.2).

$$U_{CORE} = U - U_{SENSE} \tag{4.2}$$

According to Ohms law, I_{SENSE} equals the ratio of the voltage U_{SENSE} which is measured and the resistance R_{SENSE}:

$$I_{CORE} = I_{SENSE} + I_{AMP} \approx I_{SENSE} = \frac{U_{SENSE}}{R_{SENSE}} \tag{4.3}$$

Inserting formula 4.2 and 4.3 in formula 4.1 yields:

$$P_{CORE} = (U - U_{SENSE}) \cdot \frac{U_{SENSE}}{R_{SENSE}} = \frac{U \cdot U_{SENSE} - U_{SENSE}^2}{R_{SENSE}} \tag{4.4}$$

The energy that is consumed by the processor is the integral of the power P_{CORE} over the time t. Inserting formula 4.4 leads to:

$$E = \int P_{CORE}(t)dt = \int \frac{U \cdot U_{SENSE}(t) - U_{SENSE}^2(t)}{R_{SENSE}} dt \tag{4.5}$$

Because the amplifier module has a gain of $G = 99.8$ (see chapter 5.2, page 21), the measured voltage U_{AD} results to $U_{AD} = G \cdot U_{SENSE}$. The consumed energy can be computed by measuring the voltage drop at the sense resistance over the time t:

$$E = \int \frac{U \cdot G^{-1} \cdot U_{AD}(t) - G^{-2} \cdot U_{AD}^2(t)}{R_{SENSE}} dt = \frac{1}{G^2 R_{SENSE}} \cdot \int (U \cdot G \cdot U_{AD}(t) - U_{AD}^2(t)) dt \tag{4.6}$$

$$= \frac{1}{99.8^2 \cdot 0.1\Omega} \cdot \int (1.5V \cdot 99.8 \cdot U_{AD}(t) - U_{AD}^2(t)) dt \tag{4.7}$$

The data acquisition hardware acquires up to $N = 100000$ samples per second. Hence, the time interval for one sample results to $\Delta t = \frac{1s}{N}$. So the calculation of the energy consumption can be transformed into the following sum:

$$E = \frac{1}{99.8^2 \cdot 0.1\Omega} \cdot \Delta t \cdot \sum_{n=0}^{\frac{T}{\Delta t}} 1.5V \cdot 99.8 \cdot U_{AD}(n \cdot \Delta t) - U_{AD}^2(n \cdot \Delta t) \tag{4.8}$$

Keeping this in mind one can see that the intrinsic calculation of the energy consumption is straightforward and can be really simply realized:

Listing 4.1: Pseudocode of energy calculation for a given voltage log

```
$E = 0                          # Initial energy = 0
read $delta_t from <logfile>    # Samples per second setting
for $U in <logfile> do
    $E += 1.5 * 99.8 * $U - $U^2
done
$E *= $delta_t/(99.8^2 * 0.1) # Unit: volt^2 * sec / ohm =
   joule
print "Consumed energy: $E Joule"
```

5. Framework Implementation

5.1. Development Board

The DevkitIDP from BSquare [11] was chosen as energy consumption measurement platform. It is based on the Intel[14] XScale[1] PXA255 Application Processor (Figure 5.2) and provides all necessary features:

- It is a platform designed for low power consumption and will be used in the future in this context. Nowadays the Intel XScale application processor is already used for example in mobile phones and Personal Digital Assistants (PDA).
- JTAG[19] interface for debugging and writing flash images (see 5.1.1).
- Measurement points for acquiring the power consumption of different parts of the board (see 5.1.2).
- Ethernet interface for loading the programs to the board (see 5.1.3).
- Serial line for receiving the outputs of programs that are running on the board.

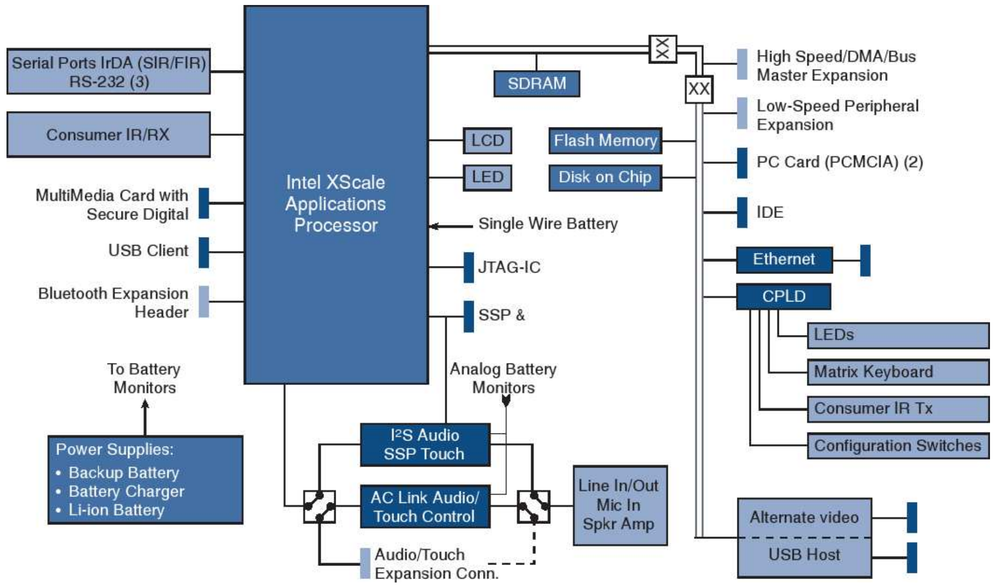

Figure 5.1.: DevkitIDP system diagram taken from the datasheet.

Figure 5.2.: DevkitIDP development board based on the Intel PXA255 Application Processor

5.1.1. JTAG-Adapter and Software

JTAG (Joint Test Action Group) was standardized in 1990 as the IEEE Std.1149.1-1990 and is used for testing printed circuit boards using boundary scan. For burning software images on the development board flash memory and for debugging purposes a JTAG-Adapter is required. Figure 5.3 shows the schematic of the JTAG-Adapter. It connects the printer port (LPT) of the computer with the JTAG header J_{15} of the development board.

Size (H x W x D)	10 in x 12 in x 16 in (25.5 cm x 30.5 cm x 40.6 cm)
Weight	8 lb (3.6 kg) with power supply
Processor	Intel PXA255 running at 400 MHz
	32 MB Intel Strataflash® NOR flash
	32 MB M-Systems Millenium Plus NAND flash
Touch screen	Two resistive 4-wire touch controllers
Video	Full VGA color Sharp LM8V302
Audio	AC '97 (Philips UCB1400)
PCMCIA	Two PC Card Type II slots
Secure Digital	Via Secure Digital connector USB
Radio	Bluetooth serial connector
Serial ports	Three configurable ports
Ethernet	10/100 BaseT
Expansion Buses	Multiple
Power Management	Supports Five Modes
Debugging	JTAG

Table 5.1.: DevkitIDP Specification Highlights

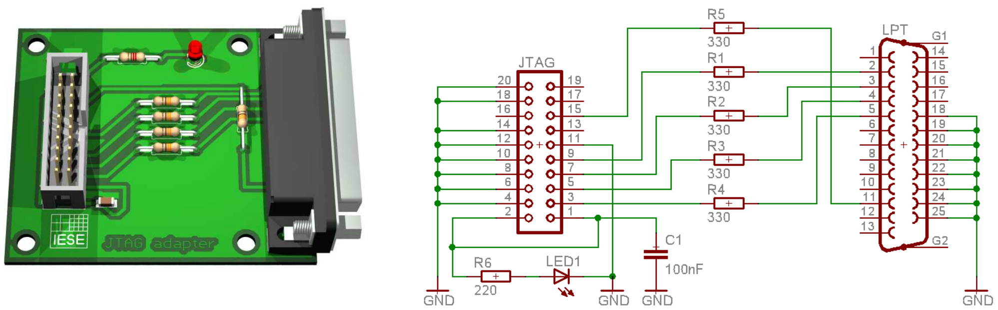

Figure 5.3.: JTAG interface PCB (left) and schematic (right)

In this case the *Universal Bootloader* [20] is used (see 5.1.3) as boot image. On the original BSquare [11] DevkitIDP CD is a JTAG software called `FLASH255` which programs ROM images into the flash memory using this JTAG adapter.

5.1.2. Measurement Points

The DevkitIDP board provides several measurement points (TP_x) and jumpers (J_x) for testing the board (core, memory, peripherals). In this case the jumper J_{29} and test point TP_2 are used for measuring the core current.

J_{29} supplies the PXA255 core with power. The board is shipped with jumper J_{29} closed. It was replaced by a sense resistance $R_{SENSE} = 0.1\Omega$ resulting in a voltage drop which depends on the core current (Figure 5.4). This voltage drop signal is connected over a shielded twisted pair cable to the amplifier module (see 5.2). The shield of the twisted pair cable is linked with the board signal ground on TP_2.

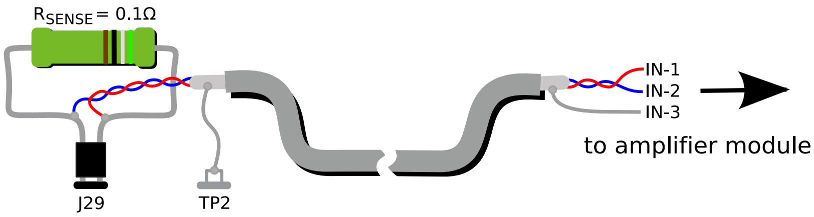

Figure 5.4.: Measuring the core current of the DevkitIDP board

The IN-1, IN-2 and IN-3 pins are connected to the amplifier module (see below) for amplifying the applied signal.

5.1.3. Universal Bootloader

U-Boot [20] is an open source bootloader for several architectures including Intel XScale PXA255 under General Public License. The U-Boot image (`uboot.img`) is transfered to the development board flash memory using a JTAG adapter (see 5.1.1).

The communication with U-Boot is done over console (which is attached to the development board) or in this case over serial line[18]. Using U-Boot makes it possible to transfer programs to the XScale board using the network interface (protocol: TFTP [17]) and run them out of the volatile memory. Appendix A (page 38) shows an overview of the U-Boot command set.

5.2. Amplifier module

The voltage drop signal that is measured with about 20 millivolts is very weak. The voltage is a differential signal, so the amplifier has two important tasks:

- Amplify the signal by a factor of 100. So the resulting signal has the dimension of 2 volts.
- Making an absolute signal out of the differential signal. The absolute signal can be sampled using standard methods without using acquisition cards with differential inputs.

The type of amplifier that performs these two tasks is called *instrumentation amplifier*.

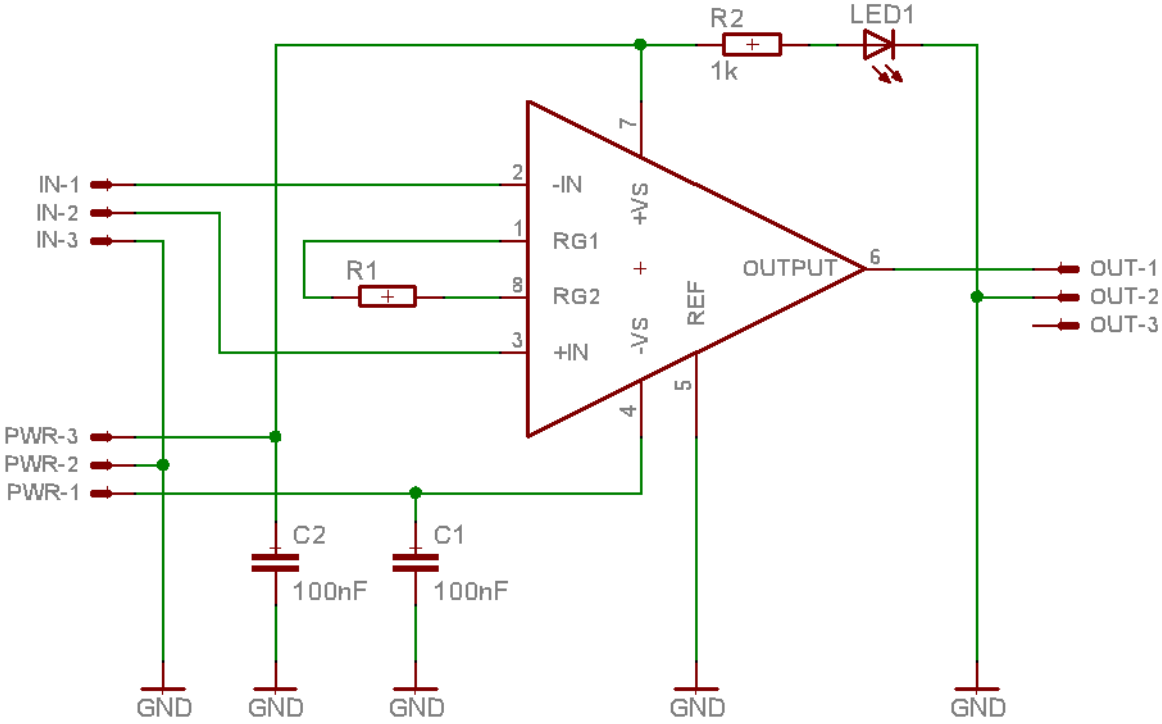

Figure 5.5.: Schematic of the instrumentation amplifier module

Figure 5.5 shows the schematic of the instrumentation amplifier module. The IN-pins are connected to the development board while the PWR-pins supply the amplifier with a symmetrical supply voltage of $\pm 12V$. The OUT-pins are directly connected to the ADC-Card.

Resistor R_1 sets the gain G of the module. The gain equation is:

$$G = \frac{49.4k\Omega}{R_1} + 1 \Leftrightarrow R_1 = \frac{49.4k\Omega}{G-1}$$

With a resistance of $R_1 = 500\Omega$ in the framework the gain results to $G = \frac{49.4k\Omega}{500\Omega} + 1$ which at least enable us to capture the signal.

$$Gain\ G = 99.8 \Rightarrow U_{OUT} = 99.8 \cdot U_{IN}$$

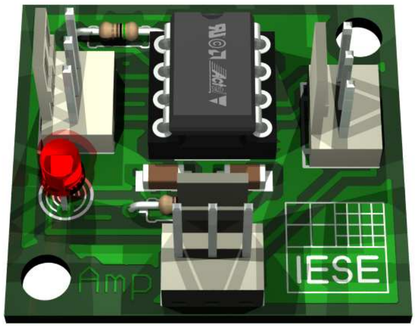

Figure 5.6.: Instrumentation amplifier printed circuit board

Figure 5.6 shows the PCB layout of the amplifier module. The core of the module is the Analog Devices [16] AD620. The amplifier needs a symmetric power supply which is provided by two 12 volts lead batteries.

5.3. Data Acquisition Module

5.3.1. Acquisition Hardware

For capturing the amplified signal the high speed data acquisition card NI-PCI-6032E (Figure 5.7) from National Instruments [15] is used. It performs up to 100 thousand samples per second at a resolution of 16 bit simultaneous on 16 analog input channels. At present only one channel is used for measuring the PXA255 core current.

In future it is possible to extend the platform for concurrent measuring of the energy consumption of the memory and peripherals.

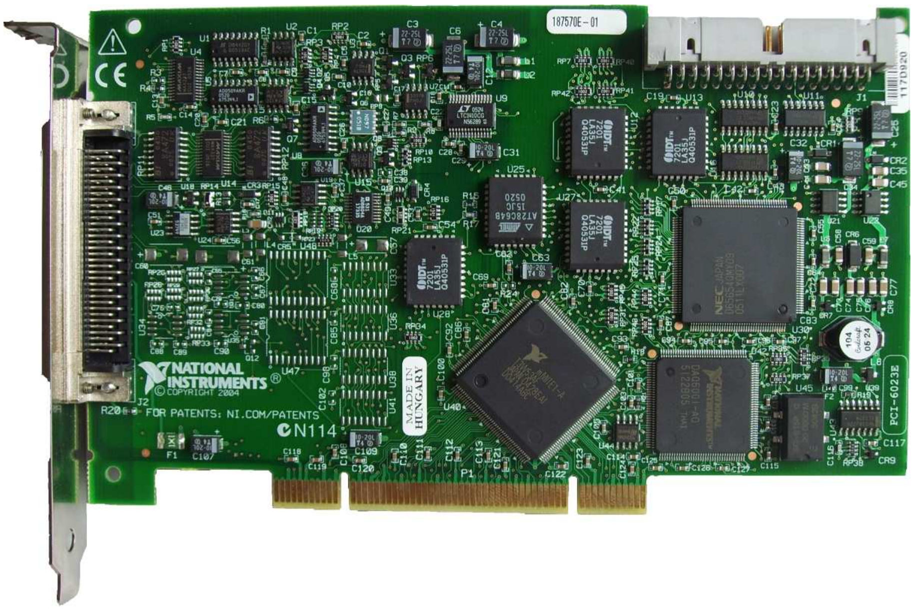

Figure 5.7.: National Instruments data acquisition card NI-PCI-6032E

5.3.2. Acquisition software

For sampling the signal the *Linux Control and Measurement Device Interface* (COMEDI [10]) is used by the acquisition software `myscope`. It is based upon the Linux kernel 2.6.12.2 [21] with the *RealTime Application Interface* (RTAI [12]) from *Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano* (DIAPM [13]).

RTAI provides hard realtime functionality for writing programs with strict timing constraints like data acquisition. It consists of a patch for the Linux kernel, API and testsuites.

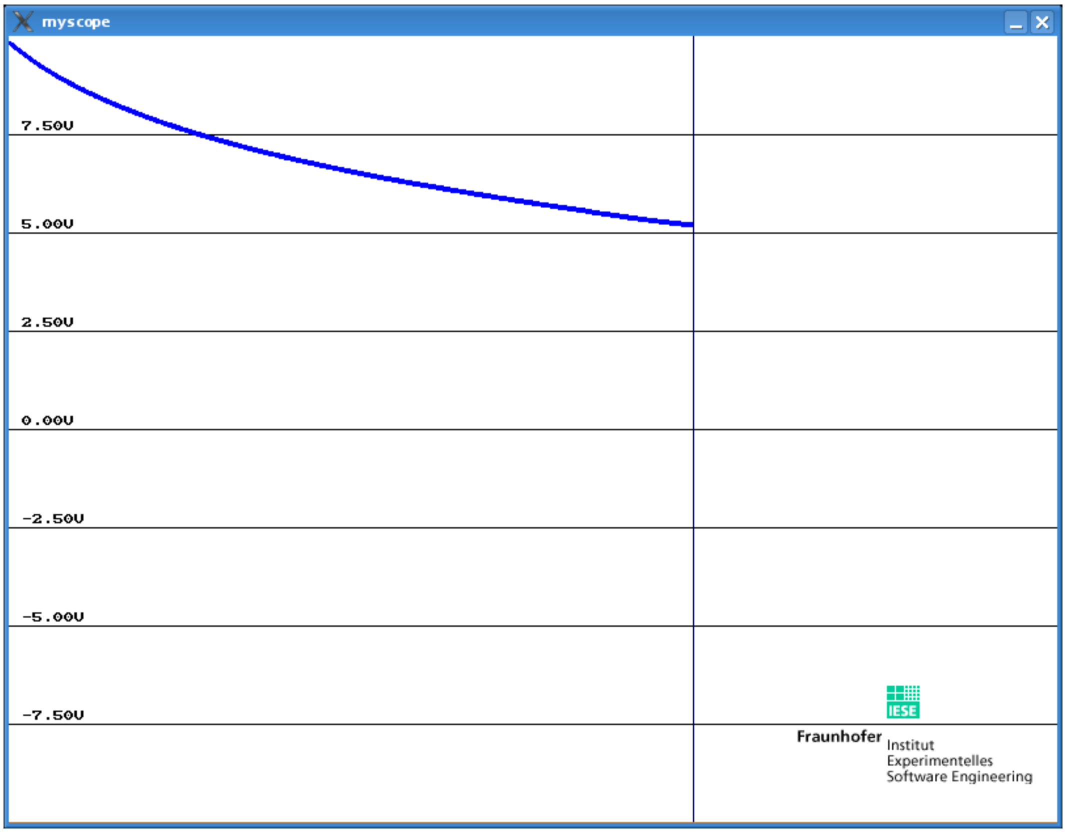

Figure 5.8.: Screenshot of the oscilloscope software.

The acquisition software reads the sampled values from the digital analog converter and has two modes to output the data. See appendix B, page 41 for more details.

- In the display mode (Figure 5.8) the program shows the voltage signal in realtime to get a coarse impression of the power consumption.
- The second mode logs the voltage with a given conversion time (e.g. 10.000 samples per second) to a logfile. In this operation mode `myscope` provides a named pipe (FIFO) that allows to insert labels in the logfile. The program-loader module (see 5.4) sends the outputs of the board to this named pipe for associating the sections of the XScale program to the acquired data.

5.4. Program-Loader Module

For automatically analyzing multiple programs the program-loader module performs the following actions (Figure 5.9):

- Compiling all source codes in the specified directory

- Creating bootable image files for the compiled binaries and putting them into the TFTP (see 5.1.3) directory e.g. `/tftpboot`
- The module communicates with the Universal Bootloader over serial null modem and issues the commands to load (`tftpboot`) and run (`bootm`) the images
- While the program is running the acquisition software (see 5.3.2) records the core current of the PXA255
- The output (`stdout`) of the program running on the board is forwarded through the serial line and is inserted in the core current log file at the proper position for being able to track the power consumption of several sub sections of the program
- For a specific section of the executed program the energy consumption can be calculated from the voltage drop over R_{SENSE} using specific formulas (see 4.3 on page 16)

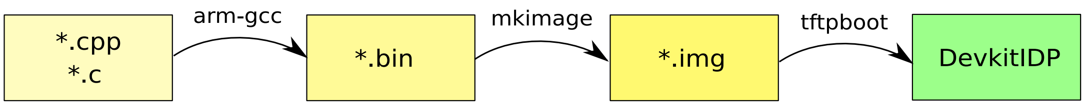

Figure 5.9.: Program-loader module

6. Framework Evaluation

6.1. Benchmark Repository

6.1.1. Repository Requirements

Focusing on high level programming languages (C in this case), the GNU[22] C compiler `gcc`[23] is used. It provides different optimization levels and also optimization for the binary size or program runtime. A `gcc` optimization scheme for energy consumption in future is imaginable (see 7, page 34).

Energy consumption of assembly instructions and machine code are not topic of this framework evaluation.

The programs that are evaluated should be available in different versions to ease the comparison:

- Program flow: Iterative or Recursive
- Used data types and structures: Simple types (`int`, `long`, `float`, `double` ...), lists, trees, graphs etc.
- Algorithm complexity class: Runtime complexity and memory complexity
- Different optimization levels (compiler optimization: see table 6.1)

Option	Description
`-O` or `-O0`	Do not optimize. This is the default.
`-O1`	Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
`-O2`	Optimize even more. `gcc` performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify `-O2`. As compared to `-O`, this option increases both compilation time and the performance of the generated code.
`-O3`	Optimize yet more. `-O3` turns on all optimizations specified by `-O2` and also turns on the `-finline-functions` and `-frename-registers` options.
`-Os`	Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Table 6.1.: GNU C compiler optimization levels (from gcc manual page)

6.1.2. Sorting algorithm repository

For testing the power measurement platform several sorting algorithms from [24] can be used because they are described well and are already classified to complexity classes and program flow style. I got 41 of these algorithms to compile for the target PXA255 processor. These 41 algorithms can be executed and evaluated using different data sizes.

Caused by the limited time of this project thesis, it is unfortunately impossible to run the whole software repository that was intended and already prepared. So there is no analysis performed in this project thesis; instead the energy measurement framework is verified to match the development board data sheet power characteristics.

6.2. Measurement Results

The framework is qualitatively evaluated to check its functionality and its accuracy. Figure 6.1 shows a screenshot of the oscilloscope application capturing the energy consumption for the Windows boot. Labeled positions are:

- Switching on the supply power of the development board.

- Windows booted successfully and is in idle state (showing the desktop on the display)

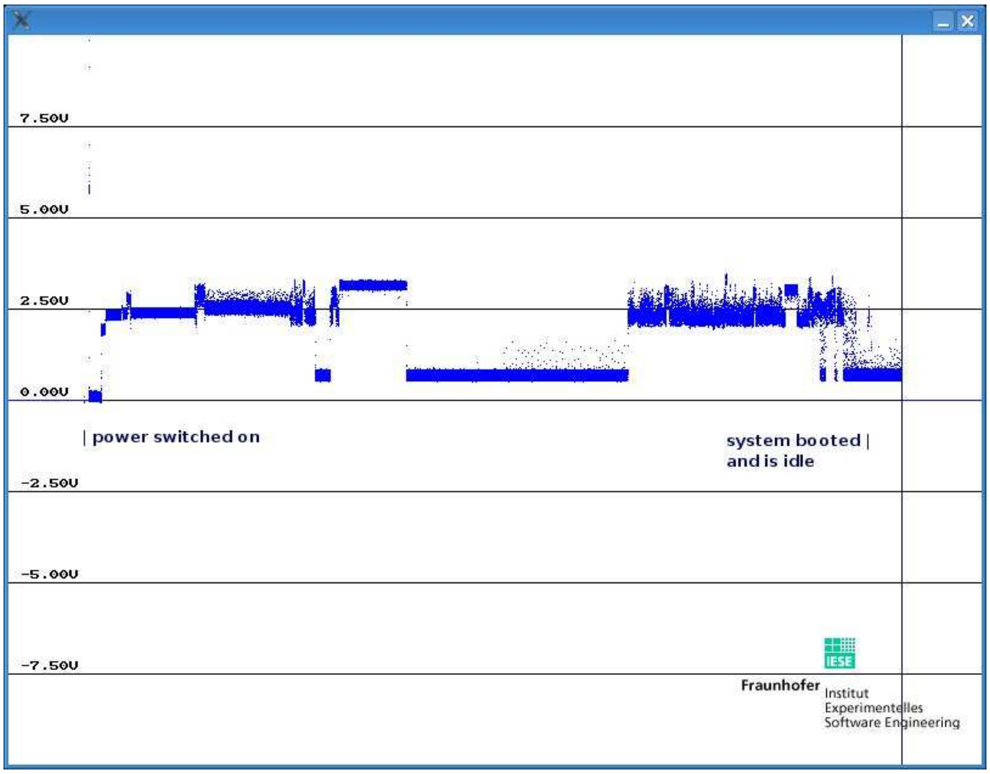

Figure 6.1.: Screenshot of the oscilloscope application capturing the energy consumption for the Windows boot.

The windows boot process shows many different power states for the PXA255 application processor. The boot process is evaluated and compared to the PXA255 data sheet below (chapter 6.2.1).

Figure 6.2 shows a screenshot of the oscilloscope application capturing the energy consumption for suspending and resuming windows. Labeled positions are:

- Pressing *suspend* in the Windows start menu
- The system is completely suspended and consumes about $60\mu A$ (see Table 6.2)
- Pressing the *wake up* button on the development board
- Windows resumed normal operation successfully and is in idle state (showing the desktop on the display)

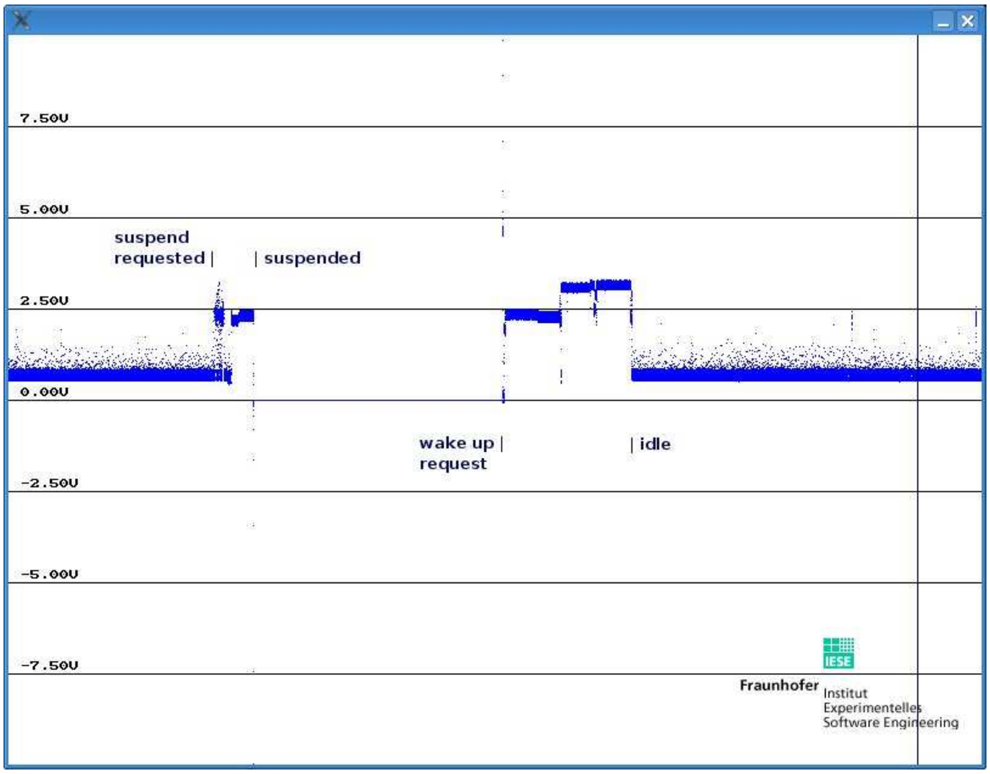

Figure 6.2.: Screenshot of the oscilloscope application capturing the energy consumption for suspending and resuming windows.

Symbol	Description	Typical	Maximum	Units
400 MHz active mode				
V_{cc}		1.3	1.65	V
V_{ccq}/V_{ccn}		3.3	3.6	V
Temp	Operating temperature	21	100	℃
I_{ccc}	V_{cc} Current	245	800	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	28	355	mA
P_{TOTAL}	Total Power	411	2598	mW
300 MHz active mode				
V_{cc}		1.1	1.43	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	185	570	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	24	345	mA
P_{TOTAL}	Total Power	283	2057	mW
200 MHz active mode				
V_{cc}		1.0	1.32	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	115	340	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	19	330	mA
P_{TOTAL}	Total Power	178	1637	mW
400 MHz idle mode				
V_{cc}		1.3	1.65	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	95	460	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	9	50	mA
P_{TOTAL}	Total Power	121	939	mW
300 MHz idle mode				
V_{cc}		1.1	1.43	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	43	335	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	9	50	mA
P_{TOTAL}	Total Power	77	659	mW
200 MHz idle mode				
V_{cc}		1.0	1.32	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	33	205	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	9	50	mA
P_{TOTAL}	Total Power	63	451	mW
33 MHz idle mode				
V_{cc}		1.0	1.32	V
V_{ccq}/V_{ccn}		3.3	3.6	V
I_{ccc}	V_{cc} Current	15	70	mA
I_{ccp}	V_{ccq} and V_{ccn} Current	9	50	mA
P_{TOTAL}	Total Power	45	272	mW
Sleep mode				
V_{cc}		0	0	V
V_{ccq}/V_{ccn}		3.3	3.3	V
I_{ccp}	V_{ccq} and V_{ccn} Current	45	75	μA

Table 6.2.: Power Consumption Specifications for PXA255 processor

6.2.1. Evaluation of the windows boot process

Figure 6.2.1 shows the digital analogue converter card values sampled while booting windows mobile. The noisy peaks at the start and the end of the sampled data are produced by the DevkitIDP power switch.

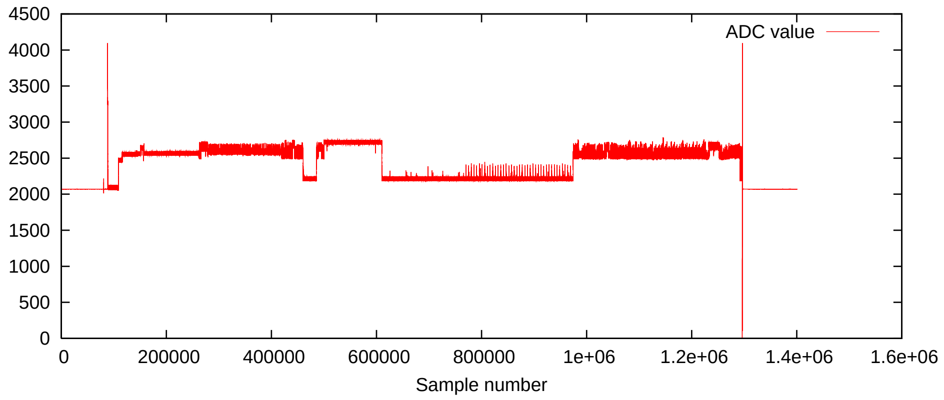

Figure 6.3.: Samples of the analogue digital converter card while booting windows.

In figure 6.2.1 the samples are translated into the power dissipation using formula 4.4 discussed in chapter 4.3:

$$P_{CORE} = \frac{U \cdot U_{SENSE} - U_{SENSE}^2}{R_{SENSE}}$$

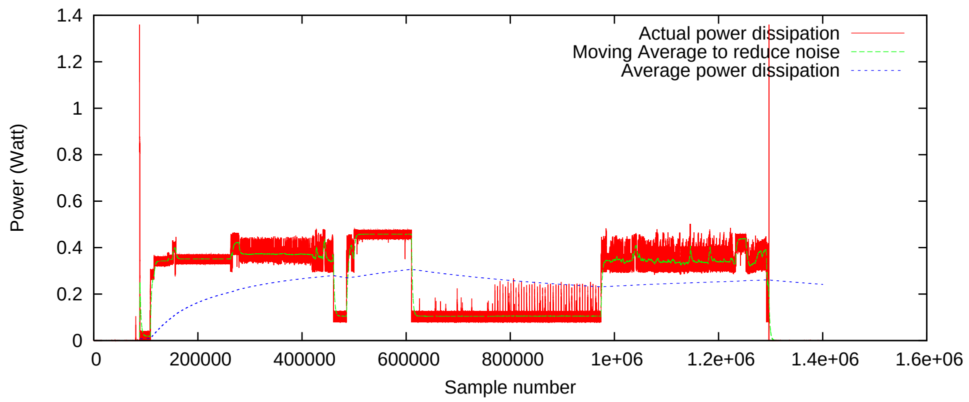

Figure 6.4.: Power consumption of the PXA255 processor while booting windows.

Figure 6.2.1 finally illustrates the energy consumption of the application processor. The energy consumption is calculated out of the previously calculated power dissipation using formula 4.5:

$$E = \int P_{CORE}(t)dt$$

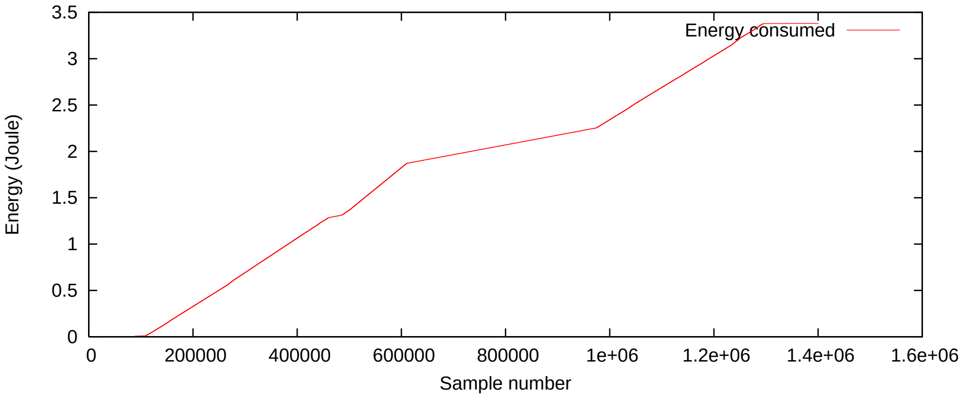

Figure 6.5.: Energy consumed by the PXA255 processor while booting windows.

Listing 6.1: Analysis of Windows boot

```
------------------ Configuration -----------------
Ref 0      = -10.00    Volt
Ref 1      = 10.00     Volt
Frequency = 100000.00     Hz

------------------ Calculating -------------------
sample   65536: Uad=0.00V power=  0.7mW  energy=0.01uJ  total energy=0.000J=0.00mWh
sample  131072: Uad=2.25V power=326.2mW  energy=3.26uJ  total energy=0.083J=0.02mWh
sample  196608: Uad=2.35V power=340.1mW  energy=3.40uJ  total energy=0.315J=0.09mWh
sample  262144: Uad=2.36V power=341.5mW  energy=3.41uJ  total energy=0.545J=0.15mWh
sample  327680: Uad=2.57V power=371.4mW  energy=3.71uJ  total energy=0.795J=0.22mWh
sample  393216: Uad=2.38V power=345.0mW  energy=3.45uJ  total energy=1.039J=0.29mWh
sample  458752: Uad=2.88V power=415.7mW  energy=4.16uJ  total energy=1.280J=0.36mWh
sample  524288: Uad=3.23V power=464.6mW  energy=4.65uJ  total energy=1.477J=0.41mWh
sample  589824: Uad=3.08V power=443.3mW  energy=4.43uJ  total energy=1.777J=0.49mWh
sample  655360: Uad=0.65V power= 95.1mW  energy=0.95uJ  total energy=1.918J=0.53mWh
sample  720896: Uad=0.57V power= 83.0mW  energy=0.83uJ  total energy=1.987J=0.55mWh
sample  786432: Uad=0.85V power=124.2mW  energy=1.24uJ  total energy=2.055J=0.57mWh
sample  851968: Uad=0.96V power=139.9mW  energy=1.40uJ  total energy=2.124J=0.59mWh
sample  917504: Uad=0.58V power= 85.1mW  energy=0.85uJ  total energy=2.193J=0.61mWh
sample  983040: Uad=2.33V power=337.3mW  energy=3.37uJ  total energy=2.283J=0.63mWh
sample 1048576: Uad=2.27V power=328.9mW  energy=3.29uJ  total energy=2.514J=0.70mWh
sample 1114112: Uad=2.28V power=329.6mW  energy=3.30uJ  total energy=2.738J=0.76mWh
sample 1179648: Uad=2.29V power=331.7mW  energy=3.32uJ  total energy=2.960J=0.82mWh
sample 1245184: Uad=3.17V power=457.1mW  energy=4.57uJ  total energy=3.198J=0.89mWh
sample 1310720: Uad=0.00V power=  0.7mW  energy=0.01uJ  total energy=3.378J=0.94mWh
sample 1376256: Uad=0.00V power=  0.7mW  energy=0.01uJ  total energy=3.378J=0.94mWh

-------------------- Summary ---------------------
Filename:          myscope/windows_start.log
Numer of samples: 1400859
Program runtime:  14.008590 sec = 0.23 min
Consumed energy:  3.378288 Joule
Average power:    241.158335 mW
--------------------------------------------------
```

The energy calculator module evaluates for the windows boot process an average power consumption of $241mW$, which is fits the specification of the PXA255 data sheet (table 6.2).

7. Conclusion and Future Work

7.1. Conclusion

The task of this project thesis was to design and implement a framework for automatic measurement of embedded-software energy consumption in an XScale platform. Different parts required for the framework were composed for achieving the task of automatic measurement. The instrumentation amplifier printed circuit board was manufactured and checked for its quality. Universal Bootloader was installed on the target board using a home-made JTAG adapter for loading the program images automated over the network interface. A big software repository including many different sorting algorithms was generated but could not be evaluated caused by the limited time of this project thesis. Different tests with several programs were made to prove the reliability of the framework. It is still too early to provide a final conclusion covering the whole energy measurement topic but the composition of all relevant subsystems is tested and the results acquired so far seem very promising.

7.2. Future work

Migration to other target platforms: This project thesis only targets the DevkitIDP platform. But the migration to any other platform is imaginable. As described in chapter 4.3 the current floating through a sense resistance into a part of the system (e.g. CPU) is measured. Caused by the instrumentation amplifier with adjustable gain this sense resistance can be placed next to each power drain.

Multiple measurement channels: The current version of the platform only uses one analog input channel of the NI-PCI-6032E analog converter card. This channel measures the energy consumption of the application processor. For several applications it is desirable that also the energy consumption of other parts of the system can be measured. Additional measurement channels require additional amplifier modules. For this purpose amplifier integrated circuits with more than one output can be used. The second important part after the processor is the volatile memory (RAM and external cache). Most program variables and also the program code itself is located in the memory, thus the memory is frequently accessed. It is possible that the next version of an energy measurement framework has multiple channels: CPU, Caches, RAM, Chipset and Non-volatile memory (FLASH, Hard discs).

GCC power optimization: Like described in chapter 6.1.1 the GNU GCC compiler[23] is capable of optimizing the generated binaries in different ways (see table 6.1). In normal operation it does not optimize the binary at all but it can be instructed to optimize it on the one hand for execution speed or on the other hand for the binary file size. In case of the execution speed optimization (`gcc -O2`), the code is tweaked for runtime. Thus the program runtime decreases and the processor (maybe other components too) can switch to an energy save idle or standby mode earlier which may result in less energy consumption than without optimization (assumed that the optimized code does not consume much more energy than the original version, this is something to evaluate). The program size optimization (`gcc -Os`) results in smaller binaries. Thus there are less memory (hard disc, flash etc.) accesses required and the program can be loaded faster. The shorter program-loading time and less memory accesses may also lead to less energy consumption. GCC and most other compilers like the Intel C++ compiler[25] are absolutely not capable of optimizing the program specially for the energy consumption. It would be desirable that compilers provide options for generating energy aware executables (e.g. `gcc -Oe`). For providing a efficient power optimization scheme a multi channel power measurement platform is needed for improving and verifying the results of the compiler.

References

[1] Intel XScale Technology
http://www.intel.com/design/intelxscale

[2] Derek Bridge - Software Development - Glossary
http://www.cs.ucc.ie/ dgb/courses/swd/glossary.html

[3] The CIA World Facebook
https://www.cia.gov/cia/publications/factbook/print/gm.html

[4] Avrora is a set of simulation and analysis tools
http://compilers.cs.ucla.edu/avrora/

[5] Tuning framework for System-on-Chip platforms
http://www.cs.ucr.edu/~dalton/Platune/

[6] Embedded Processor and System-on-Chip design
http://linuxdevices.com/articles/AT4313418436.html

[7] Atmel Corporation
http://www.atmel.com

[8] UCLA Compilers Group
http://compilers.cs.ucla.edu

[9] MIPS Technologies
http://www.mips.com

[10] Linux Control and Measurement Device Interface
http://www.comedi.org

[11] BSquare - Trusted Solutions for Smart Devices
http://www.bsquare.com

[12] RealTime Application Interface
http://www.rtai.org

[13] Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano
http://www.aero.polimi.it

[14] Intel Corporation
http://www.intel.com

[15] National Instruments
http://www.ni.com

[16] Analog Devices
http://www.analog.com

[17] RFC 1350: The TFTP protocol
http://www.faqs.org/rfcs/rfc1350.html

[18] The RS232 standard
http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html

[19] Joint Test Action Group
http://en.wikipedia.org/wiki/JTAG/

[20] U-Boot - Universal Bootloader
http://sourceforge.net/projects/u-boot/

[21] The Linux Kernel Archives
http://www.kernel.org

[22] GNU's Not Unix! - Free Software, Free Society
http://www.gnu.org

[23] The GNU Compiler Collection
http://gcc.gnu.org

[24] Sorting- and Search-Algorithms
http://www.sortieralgorithmen.de

[25] Intel C++ compiler
http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm

[26] Railway Technology
http://www.railway-technology.com

[27] Development of the battery price per watt hour
http://www.solarbuzz.com/Batteryprices.htm

A. U-Boot Command Set

The Universal Bootloader has an extensive command set. In the framework only the commands `tftpboot` and `bootm` are used.

Command	Description
autoscr	Run script from memory
bdinfo	Print Board Info structure
ds	Disassemble memory
as	Assemble memory
break	Set or clear a breakpoint
step	Single step execution.
next	Single step execution, stepping over subroutines.
where	Print the running stack.
rdump	Show registers.
bmp	Manipulate BMP image data
info imageAddr	display image info
go	Start application at address 'addr'
reset	Perform RESET of the CPU
bootm	**Boot application image from memory**
boot	Boot default, i.e., run 'bootcmd'
bootd	Boot default, i.e., run 'bootcmd'
iminfo	Print header information for application image
imls	List all images found in flash
icache	Enable or disable instruction cache
dcache	Enable or disable data cache
coninfo	Print console devices and information
date	Get/set/reset date and time
getdcr	Get an AMCC PPC 4xx DCR's value
dcrn	Return a DCR's value.
setdcr	set an AMCC PPC 4xx DCR's value
dcrn	Set a DCR's value.
diag	Perform board diagnostics
doc	Disk-On-Chip sub-system
info	Show available DOC devices
tftpboot	**Boot image via network using TFTP protocol**
eeprom	EEPROM sub-system
bootelf	Boot from an ELF image in memory
bootvx	Boot vxWorks from an ELF image
ext2ls	List files in a directory (default /)
fatload	Load binary file from a dos filesystem
fatls	List files in a directory (default /)
fatinfo	Print information about filesystem
fdcboot	Boot from floppy device
fdosls	List files in a directory
flinfo	Print FLASH memory information
erase	Erase FLASH memory
protect	Enable or disable FLASH write protection
fpga	Loadable FPGA image support

imd	i2c memory display
imm	i2c memory modify (auto-incrementing)
inm	Memory modify (constant address)
imw	Memory write (fill)
icrc32	Checksum calculation
iprobe	Probe to discover valid I2C chip addresses
iloop	Infinite loop on address range
isdram	Print SDRAM configuration information
chip	Print SDRAM configuration information
ide	IDE sub-system
reset	Reset IDE controller
siuinfo	Print System Interface Unit (SIU) registers
sitinfo	Print System Integration Timers (SIT) registers
icinfo	Print Interrupt Controller registers
carinfo	Print Clocks and Reset registers
iopinfo	Print I/O Port registers
iopset	Set I/O Port registers
dmainfo	Print SDMA/IDMA registers
fccinfo	Print FCC registers
brginfo	Print Baud Rate Generator (BRG) registers
i2cinfo	Print I2C registers
sccinfo	Print SCC registers
smcinfo	Print SMC registers
spiinfo	Print Serial Peripheral Interface (SPI) registers
muxinfo	Print CPM Multiplexing registers
siinfo	Print Serial Interface (SI) registers
mccinfo	Print MCC registers
loads	Load S-Record file over serial line
loads	Load S-Record file over serial line
saves	Save S-Record file over serial line
saves	Save S-Record file over serial line
loadb	Load binary file over serial line (kermit mode)
hwflow	Turn the harwdare flow control on/off
log	Manipulate logbuffer
info	Show pointer details
md	Memory display
mm	Memory modify (auto-incrementing)
nm	Memory modify (constant address)
mw	Memory write (fill)
cp	Memory copy
cmp	Memory compare
crc32	Checksum calculation
crc32	Checksum calculation
base	Print or set address offset
loop	Infinite loop on address range
loopw	Infinite write loop on address range
mtest	Simple RAM test
mdc	Memory display cyclic
mwc	Memory write cyclic
mii	MII utility commands
device	List available devices
mii	MII utility commands
device	List available devices
irqinfo	Print information about IRQs
sleep	Delay execution for some time
mmcinit	Init mmc card
nand	NAND sub-system
info	Show available NAND devices

nboot	Boot from NAND device
printenv	Print values of all environment variables
setenv	Set environment variables
saveenv	Save environment variables to persistent storage
askenv	Get environment variables from stdin
run	Run commands in an environment variable
pci	List and access PCI Configuration Space
pinit	PCMCIA sub-system
on	Power on PCMCIA socket
out	Write datum to IO port
in	Read data from an IO port
reginfo	Print register information
scsi	SCSI sub-system
reset	Reset SCSI controller
sspi	SPI utility commands
usb	USB sub-system
reset	Reset (rescan) USB controller
usbboot	Boot from USB device
usb	USB sub-system
reset	Reset (rescan) USB controller
vfd	Load a bitmap to the VFDs on TRAB
version	Print monitor version
echo	Echo args to console
test	Minimal test like /bin/sh
exit	Exit script
help	Print online help
kgdb	Enter gdb remote debug mode
cls	Clear screen

B. Acquisition Software Details

This appendix offers a deeper look into the implementation of the data acquisition module `myscope` like described in chapter 5.3.2 on page 24.

Listing B.1: myscope.cpp data acquisition and COMEDI implementation details

```
// comedi
#include <comedilib.h>

// comedi variables
sampl_t *map;
unsigned int chanlist[256];
int value=0;
int subdevice=0;
int channel=0;
int aref=AREF_GROUND;
int range=0;
int n_chan=1;
int n_samples = 0;
double freq=100000.0;
volatile bool quit;
double ref0 = -10.0;
double ref1 = 10.0;

/////////////////////////////////////////////////////////////
// prepare comedi command
int prepare_cmd(comedi_t *dev,int subdevice,comedi_cmd *cmd)
{
    memset(cmd,0,sizeof(*cmd));

    cmd->subdev = subdevice;

    cmd->flags = 0;

    cmd->start_src = TRIG_NOW;
    cmd->start_arg = 0;

    cmd->scan_begin_src = TRIG_TIMER;
    cmd->scan_begin_arg = (int)(1e9/freq);
```

```
    cmd->convert_src = TRIG_TIMER;
    cmd->convert_arg = 1;

    cmd->scan_end_src = TRIG_COUNT;
    cmd->scan_end_arg = n_chan;

    if (n_samples)
    {
        // limit sampes
        cmd->stop_src = TRIG_COUNT;
        cmd->stop_arg = n_samples;
    }
    else
    {
        // infinite sampling
        cmd->stop_src = TRIG_NONE;
        cmd->stop_arg = 0;
    }

    cmd->chanlist  = chanlist;
    cmd->chanlist_len = n_chan;

    return 0;
}

/////////////////////////////////////////////////////////////
// initialize comedi
comedi_t *init_comedi(
    int *size, char *filename="/dev/comedi0")
{
    comedi_cmd c,*cmd=&c;
    comedi_t *dev;

    int ret;

    dev = comedi_open(filename);
    if (!(dev)) {
        comedi_perror(filename);
        exit(1);
    }

    *size = comedi_get_buffer_size(dev,subdevice);
    printf("buffer size is %d\n",*size);
```

```
    map=(sampl_t *)mmap(NULL, *size, PROT_READ,MAP_SHARED,
        comedi_fileno(dev), 0);
    if (map == MAP_FAILED)
    {
        perror( "mmap" );
        exit(1);
    }

    for (int i=0;i<n_chan;i++)
    {
        chanlist[i]=CR_PACK(channel+i,range,aref);
    }

    prepare_cmd(dev, subdevice,cmd);
    ret = comedi_command_test(dev, cmd);
    ret = comedi_command_test(dev, cmd);
    if (ret != 0) {
        fprintf(stderr,"command_test␣failed\n");
        exit(1);
    }

    ret = comedi_command(dev,cmd);
    if (ret < 0) {
        comedi_perror("comedi_command");
        exit(1);
    }

    return dev;
}


///////////////////////////////////////////////////////////////
// read number of bytes in the streaming buffer
long int bufsize = comedi_get_buffer_contents(dev,subdevice);
if (bufsize < 0)
{
    // COMEDI error => finish program
    printf("ERROR:␣comedi_get_buffer_contents␣error\n");
    break;
}
else if (bufsize == 0)
{
    // no data received
}
else
```

```
{
    // data received in samples[i]
}

/////////////////////////////////////////////////////////////
// close comedi interface
{
    int ret = comedi_cancel(dev,subdevice);
    if (ret)
    {
        printf("ERROR: comedi_cancel: %d\n",ret);
    }

    ret = comedi_unlock(dev,subdevice);
    if (ret)
    {
        printf("ERROR: comedi_unlock: %d\n",ret);
    }

    ret = comedi_close(dev);
    if (ret)
    {
        printf("ERROR: comedi_close: %d\n", ret);
    }
}
```

Listing B.2: myscope.cpp graphics and SDL implementation details

```
// SDL library
#include <SDL/SDL.h>
#include <SDL/SDL_gfxPrimitives.h>
#include <SDL/SDL_imageFilter.h>

// display setup
#define WIDTH 800
#define HEIGHT 600
#define DEPTH 24

// ...

/////////////////////////////////////////////////////////////
// draw a horizontal line of specific voltage
void addVoltLine(SDL_Surface *surface, double volt,
    Uint32 col = 0x009000ff, char *description = "")
{
    int y = HEIGHT/2 - voltToPixel(volt);
```

```
    char str[64];
    sprintf(str, "%.2lfV %s", volt, description);
    hlineColor(surface, 0, WIDTH-1, y, col);
    stringColor(surface, 10, y-10, str, col);
}

/////////////////////////////////////////////////////////////
// draw background to screen
void DrawBackground(SDL_Surface *screen,
    SDL_Surface *background)
{
    SDL_BlitSurface(background, NULL, screen, 0);
}

/////////////////////////////////////////////////////////////
// update screen and view the actual curve
void DrawCurve(SDL_Surface* screen, SDL_Surface *average,
    sampl_t *samples, long int back, long int front,
    long int size, sampl_t triggerPoint, int mode)
{
    static int oldMin = 4095;
    static int oldMax = 0;
    static int oldY = -1;
    static int oldX = 0;

    long int sampleFront = front / sizeof(sampl_t);
    long int sampleBack = back / sizeof(sampl_t);

    int min = 4095;
    int max = 0;

    //
    min += 5;
    max -= 5;
    if (min > 4095) min = 4095;
    if (max < 0) max = 0;

    // find start sample: trigger rising edge
    sampl_t oldS = 4096;
    int startX = 0;
    // periodic mode
    for (int x = 0; x < screen->w; x++ )
    {
        sampl_t s = samples[(x+back/sizeof(sampl_t))
            % (size/sizeof(sampl_t))];
```

```
        // trigger detection
        if (startX == 0)
        {
            if ((oldS < triggerPoint) && (s >= triggerPoint))
            {
                startX = x;
            }
        }

        // max / min detection
        if (s > max) max = s;
        if (s < min) min = s;

        oldS = s;
    }


    if (mode == 1)
    {
        // periodic mode

        // show min and max-line
        oldMin += 5;
        oldMax -= 5;

        if (min < oldMin)
        {
            oldMin = min;
        }
        if (max > oldMax)
        {
            oldMax = max;
        }
        addVoltLine(screen,
            sampleToVolt(oldMin), 0x0000ffff, "min");
        addVoltLine(screen,
            sampleToVolt(oldMax), 0x0000ffff, "max");

        // make old graph darker
        boxColor(average,0,0,WIDTH-1,HEIGHT-1,30);

        oldY = -1;
        for (int x = 0; x < screen->w; x++ )
        {
```

```
            sampl_t s = samples[(startX+x+back/sizeof(sampl_t
)) % (size/sizeof(sampl_t))];
            int y = HEIGHT/2 - SampleToPixel(s);
            if (y < 0) y = 0;
            if (y > (screen->h-1)) y = screen->h-1;
            //pixelRGBA(screen, x, y, 255, 255, 0, 255);
            if (oldY != -1)
            {
                lineRGBA(average, oldX, oldY, x, y, 255, 255,
 0, 255);
            }
            oldY = y;
            oldX = x;
        }

    }
    else if (mode == 2)
    {
        // slow mode
        for (long int i = sampleBack; i < sampleFront; i+=20)
        {
            int x = (i / 2000L) % screen->w;
            int y = getSampleY(i, screen, size, samples);
            while (oldX != x)
            {
                oldX++;
                if (oldX >= WIDTH) oldX = 0;
                vlineColor(average, oldX, 0, HEIGHT-1, 0xff);
                if (x == oldX)
                {
                    if (x < WIDTH-1)
                        vlineColor(average, x+1, 0, HEIGHT-1,
 0x0000a0ff);
                }
            }

            pixelRGBA(average, x, y, 255, 255, 0, 255);
            //if (oldY != -1) lineRGBA(average, oldX, oldY, x
, y, 255, 255, 0, 255);
            oldY = y;
            oldX = x;
        }

    }
    else
```

```
    {
        printf("ERROR: unknown display mode\n");
    }

    // add graph to screen
    SDL_SetColorKey(average, SDL_SRCCOLORKEY, 0);
    SDL_BlitSurface(average, NULL, screen, 0);

    SDL_Flip(screen);
}

//////////////////////////////////////////////////////////////
// initialize sdl
void init_sdl(SDL_Surface **screen)
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0) exit(1);

    if (!(*screen = SDL_SetVideoMode(WIDTH, HEIGHT, DEPTH,
   SDL_SWSURFACE|SDL_HWPALETTE))) // |SDL_FULLSCREEN
    {
        SDL_Quit();
        exit(1);
    }
}

//////////////////////////////////////////////////////////////
// main function
int main(int argc, char *argv[])
{

    // ...

    // initialize SDL
    if (enableDisplay)
    {
        init_sdl(&screen);

        // generate the background image
        background = SDL_CreateRGBSurface(0, WIDTH, HEIGHT,
   DEPTH, 0xff0000, 0x00ff00, 0x0000ff, 0);
        if (!background)
        {
            printf("ERROR: SDL_CreateRGBSurface\n");
        }
        for (double volt=0.0; volt<=9.0; volt += 2.5)
```

```
        {
            addVoltLine(background, volt);
            if (volt != 0.0) addVoltLine(background, -volt);
        }
        if (enableDisplay == 1)
        {
            // periodic mode with trigger
            addVoltLine(background, trigger, 0x900000ff, "
trigger");
        }

        // generate "average"-surface
        average = SDL_CreateRGBSurface(0, WIDTH, HEIGHT,
DEPTH, 0xff0000, 0x00ff00, 0x0000ff, 0);
        if (!background)
        {
            printf("ERROR: SDL_CreateRGBSurface\n");
        }

    }
    // ...


    // main loop
    while((!quit) && ((n_samples == 0) || (n_samples >
samples_read)))
    {
        // ...

        // display data if requested
        if (enableDisplay)
        {
            DrawBackground(screen, background);
            DrawCurve(screen, average, map, back, front, size
, voltToSample(trigger), enableDisplay);
        }
        // ...

        // poll SDL events if display is enabled
        if (enableDisplay)
        {
            while(SDL_PollEvent(&event))
            {
                switch (event.type)
                {
```

```
                case SDL_QUIT:
                    quit = 1;
                    break;
                case SDL_KEYDOWN:
                    //quit = 1;
                    break;
                }
            }
        }
    }
    // ...

    SDL_Quit();
    return EXIT_SUCCESS;
}
```