Entwicklung einer Infrastruktur für Energiesparverfahren

Diplomarbeit im Fach Informatik

vorgelegt von
Holger Wunderlich
geboren am 24. April 1980 in Kemnath

Institut für Informatik,
Lehrstuhl für Verteilte Systeme und Betriebssysteme,
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Dipl.-Inf. Andreas Weißel
	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 1. Juli 2005
Abgabedatum: 2. Januar 2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 13. Januar 2006

Operating System Support for Power Management of Mobile Robots

Diploma Thesis

by
Holger Wunderlich
born April 24st 1980 in Kemnath

Department of Computer Science,
Distributed Systems and Operating Systems,
University of Erlangen-Nürnberg

Advisors:	Dipl.-Inf. Andreas Weißel
	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat

Begin:	July 1th, 2005
Submission:	January 2th, 2006

Abstract

For battery-driven devices it is important to guarantee a defined runtime or to complete a given task with the available battery capacity.

Two prerequisites are needed in order to achieve this goal:
The energy usage of a device has to be known.
Fine grained limitation of the energy usage has to be possible.

Gaining knowledge of the energy usage of a system can be done by accounting the energy consumption of the devices. Thereby it is not enough to account the energy used by the running applications as this is only a part of the total consumption. Energy is also consumed during idle periods since the devices are not switched off.

To be able to efficiently limit the energy usage it is also necessary to know which applications consumed the energy. Hence it is not sufficient to only estimate the energy consumption but to accounted it to the responsible applications.

Reducing the energy consumption can be done by limiting the energy usage and by exploiting the different power management modes of the devices to also reduce the consumption during idle periods.

However, only reducing the energy of a system is not sufficient to be able to complete a given task.

A robot for example may have the following tasks to fulfill:

- Exploring the environment.
- Taking images and building a graphical map.

During normal operation both tasks are executed with equal priorities. If the robot runs on low batteries taking images and processing them should be delayed. It should continue exploring until only enough energy is left to return to the charging station. Then exploring should also be delayed and the robot should return. After charging the battery the robot can continue executing the tasks.

To accomplish this, it is necessary to change the priorities for the tasks during runtime. This can be done by restricting the available energy for one task, while not doing so for the second one.

The decision, which tasks to limit and how to limit them is done by dedicated programs, the so called energy-aware policies (these are not part of this work). A policy requires information about the energy consumption of each application and the remaining runtime (or battery capacity). Therewith it can calculate and set the limits for the applications.

The main objective of this work is to develop an infrastructure that provides the accounted data (consumed energy, device usage) and that sets and maintains the limits for the applications. Furthermore it provides feedback about the effects of the limitations. This feedback can be used by the policies to adapt the limits in case the energy consumption is still too high or the consumption is lower than expected. This can happen due to interdependencies between the devices. For example limiting the energy usage of the CPU also reduces the usage of other components if the limited application is very CPU intensive.

The infrastructure has to be implemented within the operating system layer since only there it is possible to get the required device information and to control the devices and applications as required.

It is shown that using such an infrastructure enables accurate accounting and fine grained limitation of the energy consumption while still being able to accomplish set targets.

Contents

Chapter 1

Introduction

Mobile devices have grown more and more powerful over the last years. This lead to higher energy consumption which cannot be countervailed by batteries with higher capacities. Thus it was necessary to develop new strategies and methods for saving energy.

New hardware was developed that supports different states of operation in which the energy consumption is reduced. Until now each device is managing the power for itself. Using an all-embracing power management would lead to higher energy savings and with an appropriate infrastructure energy usage may be controlled in such a way that given goals can be achieved though the energy would normally not suffice. For instance, for battery operated devices it is important that a defined runtime is reached with the available battery capacity. For a cleaning robot a goal that has to be fulfilled is not running out of energy before the chores are done. Another example is detecting and reacting to different kinds of power-related emergencies. If a robot runs out of battery all normal work should be suspended in favor of returning to the charging station. This can be done by prioritizing the different tasks into tasks that have to be fulfilled and into those which can be fulfilled if enough energy is available and changing the priorities of the tasks depending on the energy available.

Such an infrastructure is not only useful for mobile devices but also for other computers, especially high-end servers. For those, enough energy is available but the critical factor is the heat dissipation. Currently throttling the system if the temperature reaches critical values is done by the hardware itself. Using this infrastructure enables a fine grained control over the system which allows to throttle the affected components, by limiting the energy usage, as long as it is needed. It is also possible to detect emerging thermal emergencies and react to them prematurely.

Often the given tasks are not processed by one single application. Hence it is essential to distinguish applications and to handle them differently. This also incorporates the limitation of energy usage. For a robot for example the mobility has a higher priority than any other task (e. g. recording images) to assure that the robot reaches the charging station before the energy is depleted. In this case any other task is postponed, while during normal operation other priorities may apply. These for example may prefer applications using the camera over those that use the motors. Distinguishing applications also requires the knowledge of the energy usage of each single device of the system to be able to map the usage to the applications.

Controlling and limiting is only possible at the operating system layer since this is the only location where full access to the hardware and all applications is possible.

This work introduces services to determine this data, accounting it to the responsible applications during operation. Policies implemented in user-space or as operating system services make use of this information to manage and control the energy consumption.

To provide feedback and further details for the policies not only the energy but also the device usage is accounted since there is no point in limiting devices that are hardly used.

Feedback is also given about the effects of limiting. This enables the policies to correct the set limit in case the discrepancy of energy savings and planned savings is substantial.

The main objective of this work is to develop such an infrastructure that supports accounting and limiting. A sample implementation is done for Linux and is tested on Robertino, an autonomous mobile robot.

The work is structured as follows:
The first part introduces the resources that need to be accounted. This is followed by a discussion on the usable strategies for accounting and limiting. After that methods are shown how to control the energy usage and a sample policy is presented. Then a short look is taken at the implementation and thereby occurring problems.

The second part of this thesis is the evaluation of the measurements. Within this part the energy consumption of the different devices is measured and mapped to the resources. Thereafter the accuracy of accounting is compared to measurements and the effects of limiting are discussed.

The work is finished with some proposals for future investigations and a conclusion.

Chapter 2

Related Work and Background

While a lot of work has been done for resource accounting and scheduling, there are no papers dealing with full control over the energy usage of a system.

This chapter introduces the Robertino-Robotsystem which is the platform this work is implemented and tested on. Later on this work resource and energy accounting and scheduling is discussed.

2.1 Robertino-Robotsystem

The Robertino-Robotsystem, from now on called Robertino, was developed at the TU Munich. The goal was to design a small, easy to use, but fully operational, autonomous mobile robot.

Robertino consists of a PC104+-PC, a firewire camera, three motors for driving and six distance sensors. Additionally it is possible to mount an actuator which was not mounted on the Robertino used in this work.

The PC is equipped with a Pentium 3 M with 500 MHz, 128 MB RAM, a 20 GB hard disk, onboard graphics, PCMCIA, LAN, IEEE1394 and USB connectors and two CAN controllers.

Additionally there is a 802.11b WLAN card plugged into the PCMCIA slot.

The camera system was developed at the Fraunhofer Institut for the Volksbot and is connected via firewire. The AISVision is a omnidirectional camera system that is able to transmit 30 pictures per second with a maximum resolution of 640 x 480 pixels per picture [Wis04].

The motors are mounted on the round base-plate arranged in an equilateral triangle.

This means that Robertino can control every of the three degrees of freedom when moving on a flat surface independently. These are: movement in x-direction, movement in y-direction and rotation.

A drawback of this maneuverability is that relatively complex calculations have to be done to set the speeds for the motors. Figure 2.2 shows the schematics

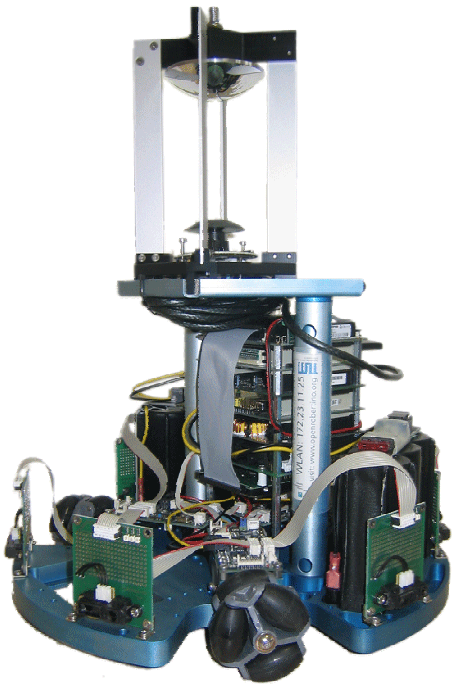

Figure 2.1: Openrobertino

of the motors and the driving direction. For each motor there are several vectors: One vector pointing into driving direction. This one is representing the actual speed of the motor.

If the known values for the movement are: *vx*, *vy* (velocity in x- resp. y-direction in mm/s) and *omega* (rotational velocity in counter-clockwise direction when looking on Robertino in mdeg/s) the speed for each motor $\vec{v_x}$ (in mm/s) can be calculated using Equation 2.1, where *rb* is the distance (in mm) from the center to the wheel's center (for the complete derivation see [url]).

$$
\begin{aligned}
\vec{vw} &= 2 * \pi * rb * \frac{omega}{360000} \\
\\
\vec{v_0} &= -0.5 * \sqrt{3} * \vec{vx} + 0.5 * \vec{vy} + \vec{vw} \\
\vec{v_1} &= -\vec{vy} + \vec{vw} \\
\vec{v_2} &= 0.5 * \sqrt{3} * \vec{vx} + 0.5 * \vec{vy} + \vec{vw}
\end{aligned} \tag{2.1}
$$

Each motor and two sensors are controlled by one Atmel ATmega8535 microcontroller which is also managing the communication with the PC over CAN[1].

[1]CAN means *Controller Area Network* this is an asynchronous serial bus-system which was de-

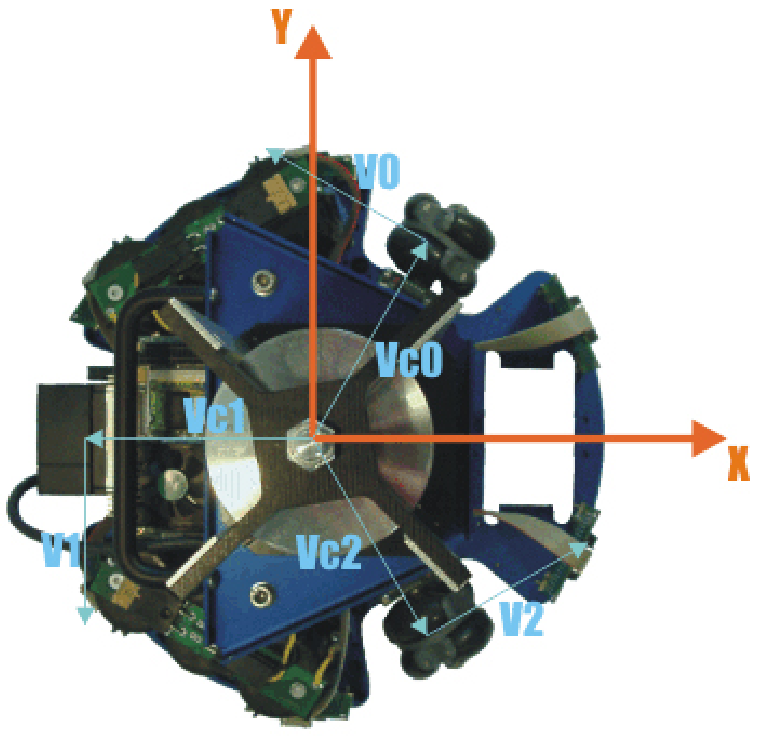

Figure 2.2: Definition of wheel motion vectors

For this purpose it is connected to a Intel 82527 CAN controller.

The motor-speed is regulated with pulse width modulated signals. When the motor is not running a braking circuit ensures that the wheel does not turn (which is needed so the robot does not slide away if it is being operated on an unlevel plane.

The power for the robot is supplied by two lead accumulators with 12 V and 6 Ah each which should lead to a theoretical runtime of 4 h [CV01]. Measurements in [Höl05] show that the real runtime under full load is about 2.5 h (respectively 2 h if runtime is defined as everything running - after about two hours the motors stop because of low voltage, whereas the PC keeps running for another half hour). Own measurements show a runtime of about 40 minutes because of defect accumulators[2].

For this work this is not a problem, because the goal is to have complete control over the used energy - no matter if there is energy for some minutes or a couple of hours available.

veloped for the automobile industry [Bos]

[2]Lead acid accumulators need a special charge current and voltage which the used power supply does not produce ([HLG05]).

Unfortunately Robertino was developed with pattern recognition which means high performance, in mind. This raises problems because it is not very energy-efficient or at least energy-aware.

- The *batteries* cannot be controlled and monitored (they do not have any intelligence built in like smart batteries in modern notebooks have [FD98]).
- The *omni-directional drive* has a big disadvantage: Besides rotation the motors do not move in the direction the robot drives, there is always a movement to the wrong direction that has to be compensated by the other motors which leads to higher energy usage.
- The *motors* consume energy, even if they do not rotate, because of the braking circuit that ensures that the wheels do not turn if they are stopped.

2.2 Resource accounting and limitation

To be able to gain control over the energy usage of whole system, the energy usage per time of each component has to be known.

For this reason a data structure is needed that holds the information about available resources and their energy consumption. A method for accounting resources to processes, respectively independent activities, and handling them as new abstraction, called resource containers, has been presented by Banga et al. in [BDM99]. A powerful, yet easy to use accounting model based on resource containers has been developed by Martin Waitz [Wai03].

Banga et al. developed the resource containers as a model for fine-grained resource management. A resource container (RC) is an abstract OS entity that logically contains all system resources (e. g. CPU time, sockets, protocol control blocks) that may be used by an application to execute a particular independent activity. All resources used by user and kernel level processing for such an activity are accounted to the appropriate RC. Each RC has attributes that are used to provide scheduling parameters, like QoS or limits. The scheduler can access this information and uses it to schedule the processes associated with the container.

An independent activity is not necessarily a process. Often one process handles many independent activities (like for example a webserver (= process) and a single client connection (= independent activity)). Or the activity consists of more than one thread, because the application is split into different parts, for example to provide fault isolation. Additionally resources are consumed when the kernel is working for the activity which, until now was not accounted correctly, or not accounted at all.

To be able to control the resource usage of the entire system, or a subsystem, the resource container form a hierarchy. The usage of a child is also applied to the parent container. If one container is limited it and all its children combined must not exceed these limits.

Martin Waitz used and improved the resource containers by introducing a new energy aware resource model which is based on the RCs in [Wai03]. With this model it is possible to account and limit the power consumption of the different processes. Unlike Banga et al. he split resource accounting off the processes and used a separate structure. Another novelty is that a process may be attached to more than one resource container

These changes were needed, because one of the targets of his work was to account client-server-relationships correctly. Decoupling resource accounting from processes also has the advantage that the OS is able to correctly account for work done within the kernel, as the kernel can change the resource container for accounting independently of the current process.

Mostly the clients access servers via pipes or sockets. These requests can be detected by the OS and used to determine the correct RC for accounting the resources used by the server. The server does not need to provide own resources. When working for a client it gets scheduled with the priority of the client RC. Servers have own resources as a fallback if the clients resources are depleted and the request is not finished yet, and for work that has to be done for the server itself.

2.2.1 Accounting

Originally the only supported resource was the CPU-time which was changed in this work by introducing a generic resource structure to which new resources may be attached simply. The used resources are CPU- and energy usage.

The during activity used resources are accounted to the thereby active process respectively its resource container. Idle times and energy used during that time is not accounted.

CPU accounting is done by periodically sampling the running process. In this case sampling is done every timer-interrupt. If the resource container is changed, the sampling function has to be called before and after the switch to update the values of both containers. Sampling means that the time stamp counter (TSC) is read and the difference to the last update is calculated. This difference is the CPU usage of the currently active process in the last timeslice. The TSC is a monotonically increasing counter.

Energy accounting is accounting the energy a process needs during activity. This is done by using a method introduced in a 'Studienarbeit' by Simon Kellner [Kel03]. He uses a linear combination of performance counters. These count several events that are generated while executing code and allow to obtain information about how the code is executed to estimate the power consumption of the CPU.
For estimation the change of the counters in the last interval is multiplied with configurable weights and then summed up. The result is the consumed energy which is then accounted to the RC.

2.2.2 Limiting

For limiting the time is split in epochs. These epochs are called timeslices. Within each timeslice the RC and therefore the process, has a given amount of resources that it may consume. This limit is configurable. To reduce peaks in usage very short timeslices can be used. To set coarse limits long timeslices can be used. For fine grained limitation short timeslices are used. For this reason it is possible to use more timeslices concurrently.

Each resource needs separate limits and accounting information stored in the RCs because on the one hand the limits may differ from application to application and on the other hand it is not possible to set one limit for all resources.

If more resources than allowed are consumed within a timeslice the processes are temporarily stopped. This is done by setting the state of the process to TASK_UNINTERRUPTIBLE. When they are allowed to run again the state is set to TASK_RUNNING. This is similar to waiting for hardware to complete a task.

To determine if the resources of a RC are depleted, the resource usage of the timeslice and the set limit are compared if the usage is higher than the limit the resource container is marked as out of resources. This check has to be done for every available resource container that consumed resources since the last check. To assure that the limits are held, checking if the RC is depleted has to be done frequently.

2.2.3 Refreshing

At the beginning of each timeslice the limits are calculated. This is done using $max_{new} = min(max_{old}, usage) + limit$, where max_{new} is the limit, max_{old} the old limit, $usage$ the actual usage of the last period and $limit$ the amount of resources that may be consumed per epoch. Because an over-consumption in the last timeslice has to be considered $min(max_{old}, usage)$ instead of $usage$ is used. This guarantees that bursts in the usage are limited correctly. If the process used to many resources in one timeslice it has less or even none resources left in the next timeslices.

2.3 Further work about accounting and limiting

In this section further work is presented that deals with accounting and limiting energy.

2.3.1 ECOSystem

Heng Zeng et al. developed a system called *ECOSystem* that is using a *currentcy model* for managing energy as a first class operating system resource [ZFE+02, ZELV02].

The subjects of this development were unified energy accounting over different hardware components, fair allocation of available energy among applications and extending battery lifetime by limiting the average discharge rate

ECOSystem is a modified Linux that incorporates the currentcy[3] model. For accounting the resources (energy, disk, CPU) the resource containers from Banga were modified.

On contrary to Banga or Waitz applications do not get a certain amount of energy per time interval for each resource but they get a certain amount of Currentcy that they may consume.

The applications have to pay for resource usage, whereas the price for each resource is variable. The price reflects the power management mode of the resource. To switch the mode a certain amount of Currentcy has to be payed. If there is not enough Currencty for changing the mode, the application must not use the resource.

The amount of Currentcy is determined at the beginning of each time interval and reflects the total energy usage (discharge rate) that is desired within that interval. Limiting the usage is done by not distributing 100 % of the Currentcy that is needed to drive a fully active system.

The energy of not considered devices and the base energy (energy consumed in the low-power modes of the managed devices) is separately accounted to the processes . This energy cannot be limited and on top of it it reduces the opportunity of improvements.

Since this work is based on the RCs by Banga, server-client-relations are not observed, and that is why it is possible that resource usage is accounted to the wrong processes, resulting in delays because server processes run out of Currentcy earlier than normal processes which then have to wait until the server receives new Currentcy.

Like in the other papers, only the energy usage of the processes is accounted while idle energy is ignored. And although the power management modes of the

[3]Currentcy is a coined term, combining the concepts of current (i. e. amps) and currency (i. e. $)

devices are recognized for accounting, they are not taken into consideration for limiting.

2.3.2 Nemesis

In [NM01] it is shown that with an appropriate operating system structure, energy may be managed as just another resource.

It is investigated how energy management could be implemented into *NemesisOS* which was designed to provide detailed and accurate resource accounting capabilities and QoS to applications, and how a decentralized resource management architecture can be used to manage energy consumption.

Limits have to be set by the user, who defines all goals that have to be fulfilled. Goals are for example a defined battery runtime, maximum discharge rate. The user has no options to assign different priorities to the goals, for example the runtime has to be obtained in every case, while it should be tried to reach another goal.

Other than before, the applications running on this OS know how much resources they consume. The operating system defines charges for the different resources (higher if more resource contention) which are announced to the applications. By limiting the amount of credits available the resources are encouraged to adapt themselves to the current resource situation (decentralized resource management). This adaption means not to access resources that are heavily used or consuming a lot of energy, while only little energy is available.

If an application does not support this adaption, the user has to intervene and define rules how the application should be treated if it should run unlimited, or if it should be stopped until enough resources are available.

The problem with anonymous resource usage has also been taken care of. Device drivers are implemented as privileged user-level processes which register interrupt handlers with the system. The driver only performs the single demultiplexing function for the hardware device. All other functionality is performed at the user level using shared libraries.

Thus the resources are used directly by the application and not by the kernel.

Taking advantage of the different power management modes of the devices is not provided by the OS. The applications could be modified to use the modes themselves which leads to high development costs, since each application has to be changed if new hardware is used.

Also it is not possible to determine, how much energy may be saved when limiting a resource, except the applications themselves know it.

Porting the resource management and accounting system to Linux would lead to a implementation similar to the resource containers, since the drivers are structured completely different. Also one of the prerequisites of this work was that no modifications to the applications are needed.

Chapter 3

Gaining Control over the energy usage

To be able to control the energy usage of a given system the usage of each component in each possible state has to be known.

First of all every component consuming energy has to be identified and the possibilities for saving energy have to be evaluated. This will be done while not disregarding side effect that occur because of limiting components (For example performance loss, or decreased speed).

Thereafter it is shown how, when and to whom used resources should be accounted.

Finally it is discussed which data needs to be stored, how long and where it needs to be stored respectively who stores it.

3.1 Resource Classification

In this section the different resources of the Robertino are introduced.

But first of all a brief definition of *resource* is given to understand, why the resources are chosen as they are.

Commonly *resources* are defined as the contrivances that may be used by a process during execution. Resources are CPU-time, energy usage, disk usage. His perspective was looking from the process layer to the devices providing the resources. Energy was modeled as an own resource, and all energy spent for the process[1] was accounted to this resource. Limiting is simply done by not scheduling the process.

This work uses another approach. Gaining full control over the energy usage implies knowledge about which device is using how much energy and therefore using one resource for accounting the all energy is not applicable. For this reason

[1]Although a resource container may have several processes attached here it is assumed that every process has its own resource container. For this reason 'resource container' and 'process' are used interchangeably.

another definition of *resources* is used in this work. Resources are the devices[2] that consume energy, and that provide the contrivances for operation. The energy a process uses, is accounted to the originating resource in the resource container (RC) of the process. The figures 3.1 and 3.2 show the connections of the resources. Details about how the implementation changed compared to the original resource containers are shown in chapter 4 Implementation.

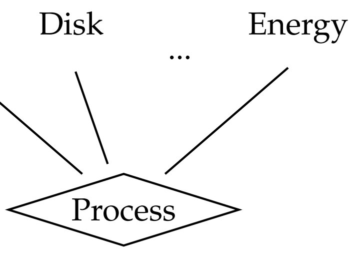

Figure 3.1: Resource definition according to M. Waitz

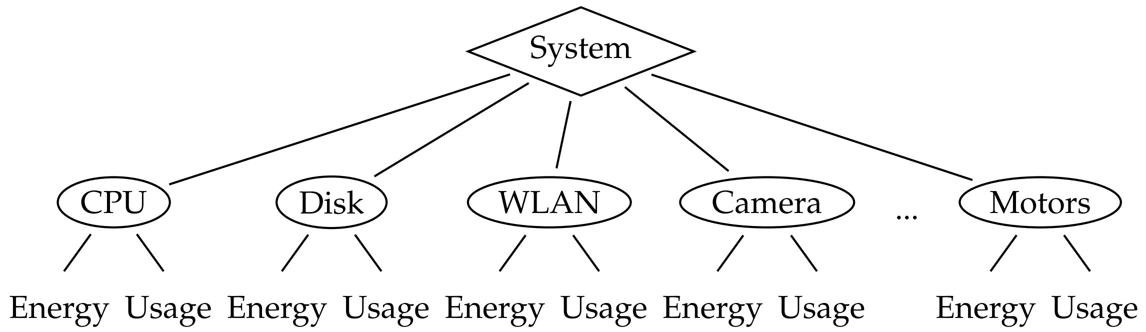
System
CPU
Disk
WLAN
Camera
...
Motors
Energy Usage Energy Usage Energy Usage Energy Usage Energy Usage

Figure 3.2: Current resource definition

3.1.1 CPU

The unquestionable most important resource of a PC is the CPU. No process may run without using the CPU.

For this reason the CPU may be used to limit the energy usage of the whole system, whereas limiting other resources has only impact on the energy usage of the limited resource.

[2]More specifically the device classes but at the moment it is assumed that only one device for a device class is attached (e. g. one hard disk). Supporting more than one device per device class may be done by introducing more resources (e. g. disk0, disk1, etc.) or by enhancing the resource structure to support more devices per resource.

The CPU used for Robertino is an old Pentium III mobile which does not support any power management modes (except idle-state, when the CPU is just doing nothing). Modern CPUs (like newer Intel Pentium/Centrino M [3] or Mobile AMD Athlon/Sempron[4]) support different power management strategies such as dynamic frequency scaling (the frequency is reduced during periods of sparse activity) or dynamic voltage scaling (the voltage is reduced during such periods).

3.1.2 Hard disk

Another resource using a lot of energy is the hard disk. A peculiarity of the disk is that the firmware has its own power management algorithm built in which yields in some unpredictability of the energy usage. But this power management can often be disabled and 'manually' set from the computer.

The disc supports different power management modes (shown in Table 3.1) but it may only be accessed if it is in the active mode.

Mode	Description
active	The disk is spinning at full speed and instantly accessible
idle	The heads are moved into parking position. Everything else is active.
standby	The mechanics of the disk are powered off.
sleep	Additionally to the mechanics the electronic for request handling is turned off. The device has to be reset in order to leave this state.

Table 3.1: Power management modes of the disk

3.1.3 Network - especially WLAN

Though Robertino is equipped with both 10/100 Mbit ethernet controller and IEEE 802.11 b WLAN this work focuses on accounting the energy usage of WLAN because during normal work the robot will communicate over WLAN and not trail a cable for communication.

The energy characteristics of WLAN are different from other resources: The difference between idle-energy and active-energy is very small and since the WLAN-interface is idle most of the time, most energy is not used during activity (sending and receiving) but during idle-times. To be able to reduce the idle energy usage the IEEE 802.11 standard[Com99] defines two power management modes.

These are the *Active Mode* (*AM*, or *CAM* (for continuously-aware mode)), in which the interface is always awake and ready for transmitting and receiving

[3] `http://www.intel.com`

[4] `http://www.amd.com`

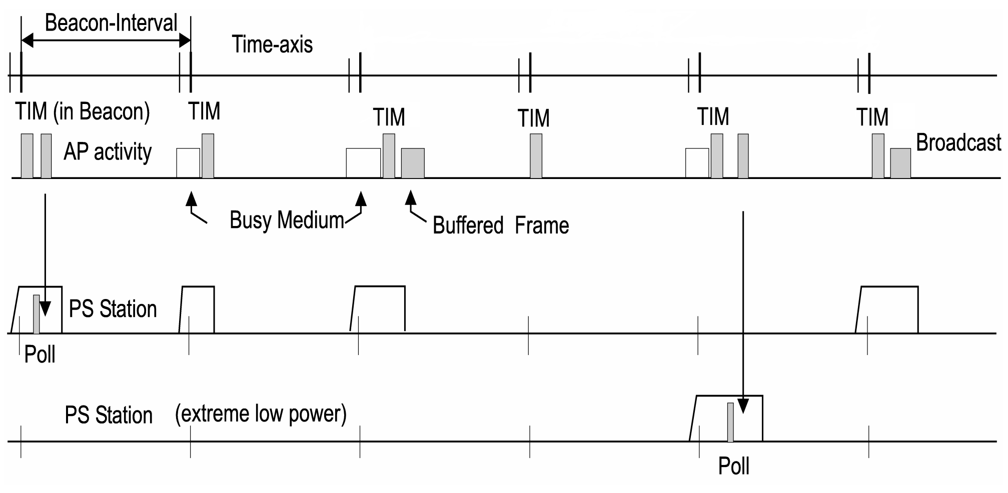

Figure 3.3: Reception of packets for wireless network interfaces working in *Power Save* mode, from [Com99]

packets and the *Power Save* (*PS* or *PSM* (for power saving mode)) mode in which the device is dozing and not able to receive packets. The data sent to a device in *PS* has to be stored by the access point. The WLAN device is periodically waking up and checking if there are packets waiting at the access point. In this case the packets are received and the device goes back to doze again afterwards. If there are no packets the device enters doze instantly. The interval the device is dozing is called *beacon period*. The longer this period is, the more energy may be saved but the longer is the latency, the time a packet needs to be transmitted, because it has to wait until the device wakes up again. This may lead to a couple of problems, for example if there are streaming applications running it may come to slow downs because the buffers run empty. Or, again, deadlines won't be met which may yield in retransmissions and though higher energy usage, or even failure of the application because the data cannot be transmitted in time. If the interval is too short, more energy may be needed than staying awake all the time, because after each wake-up a so called traffic indication map (TIM) has to be sent which informs the device about buffered packets and also energy is used to re-sync the device. An example of two wireless network interfaces in PSM using different beacon periods and their access point is shown in Figure 3.3.

3.1.4 Camera

The firewire camera is an important device for determining the position and for detecting obstacles. Thus on Robertino it will be used often and consume energy which has to be accounted.

The camera supports different image resolutions and different framerates. Higher resolution means higher image quality but also more data that needs to be processed which needs more energy. Higher framerate means that more images per second are taken. This leads to increased energy usage, too.

3.1.5 Motors and sensors

Two other resources that have not been researched yet are the motors and the sensors. Robertino is equipped with three identical motors. Each motor is managed by a microcontroller which also manages two distance sensors.

The sensors always use an almost constant amount of energy, as does the microcontroller[5].

When talking of the motor in future, the combination of motor, two sensors and a microcontroller is meant.

Because of the arrangement of the motors most of the time it is necessary to use all three motors when driving into one direction. All motors are accounted to one resource, and therefore also have a combined limit. Accounting and especially limiting the motors separately could lead to bogus movement if not all limits are equal and there is no advantage in accounting the motors separately.

Another thing that has to be considered is that the motors are attached via CAN-bus which has only a limited bandwidth. Since the microcontrollers stop the motors if after about 100ms no message was received, the software at the PC has to send the control commands continuously which leads to a constantly high traffic on the CAN-bus (because this is necessary for every motor). It has to be assured that the normal traffic on the bus is not affected, while accounting and limiting messages should always be possible and preferably not delayed.

3.1.6 Battery - A Special Resource

A resource that must not be forgotten is the battery. It is not consuming energy but it is providing the needed energy for the whole system. It is important to account the remaining energy and build a history to be able to calculate the remaining runtime of the system. Most modern laptops use smart batteries which provide detailed information about remaining capacity, actual voltage and drain rate [FD98]. With this data it is possible to calculate the energy that may be used per time to reach the desired targets.

[5]The energy usage of the sensors varies a bit, reflecting the measured distance. The energy used by the microcontroller is almost constant, because the software is not using interrupts and for this reason cannot benefit from the different power management modes the controller supports. The energy for both parts is assumed to be constant (an average is used), because the variations are negligible compared to the variations when using the motor.

On Robertino unfortunately no smart battery is available. The only possibility to get information about the battery are the ADCs (Analog-to-Digital Converter) of the microcontrollers. One of them can be used to measure the current voltage of the batteries. This voltage represents the capacity of the batteries, once it falls below a certain threshold, the battery is depleted.

3.1.7 Other possible resources

There is still some energy that is not accounted, and there are also some devices that are used by processes but until now not mapped to resources.

One large source of energy consumption that has not been discussed is dissipation loss. Because no device has a efficiency of 100 %, energy is lost (more precisely transformed into heat). The power supply for example has a maximum efficiency of 95 % ([TM03]).

To account this energy correctly, it has to be split according to the energy usage of the resources and the shares have to be accounted to the processes using these resources which is an expensive calculation that yields no advantage.

For other devices it is simply impossible (or only with very high effort) to determine the energy usage, or the energy usage of a device must not be accounted to one single resource.

The devices that may be accounted to separate resources are:

- The *RAM* and *mainboard* are accounted together with the CPU. This is because the costs of measuring the CPU and the other components separately are very high, and the RAM cannot be limited.

- The energy consumed by the *firewire controller* should not be accounted to the resource camera, although it is currently the only device attached to this bus. It is possible that other devices are attached and then the accounting has to be changed. If the camera is removed, the accounting to this resource is not possible and the energy use would not be accounted.

- For the *CAN controller at the PC* applies the same. The device cannot be switched off, or in any other way limited.

3.2 Accounting

While in [Wai03] a separate resource for energy accounting was used now the device-usage and energy-usage are accounted to one resource. Doing this allows to distinguish the energy needs of the different resources which only can be done when the energy usage of the device is exactly known.

The values that have to be accounted can be split into two groups: *Usage* and *energy*. While accounting usage is quite simple accounting energy is more complex. The energy that has to be accounted consists of three different classes: *idle-energy*, *transition-energy* and *active-energy*.

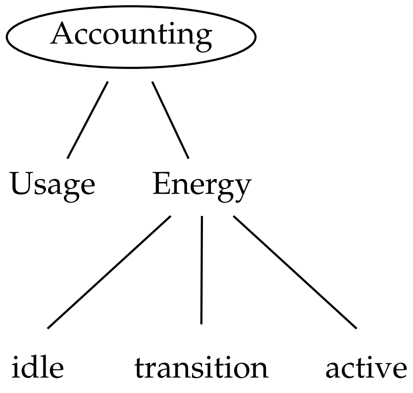

Figure 3.4: Accounting

Idle-energy is the energy that is used if the device is doing nothing. If there are different power management states there is a different idle-energy for each state. The idle-energy may also be 0 if no energy is consumed (if for example it is possible to switch the device on/off using software).

A novelty is the accounting of the different states a resource may be in. These states represent the different power management modes of the device. When a device is put into another mode (for example from active to sleep) the state of the resource is changed and all henceforth used energy is accounted to the new state until the device switches the mode again. Switching from one power management mode to another uses *transition-energy*. If no energy is used its value is 0. An example for transition-energy is the energy needed to spin up the motors of a hard disk when it switches from standby to active. This energy has to be accounted when changing the state.

The energy used during device accesses is the *active-energy*. There can be different values for the different types of usage, for example read and write. Accounting active-energy can be done together with accounting usage. Often the active-energy is directly calculated with the usage ($usage * energy\_per\_usage$).

It has to be observed that active-energy may not only be accounted in the state *active* but in every state the device may be accessed. Figure 3.5 shows the idle and active energy usage in the different power management modes of a device.

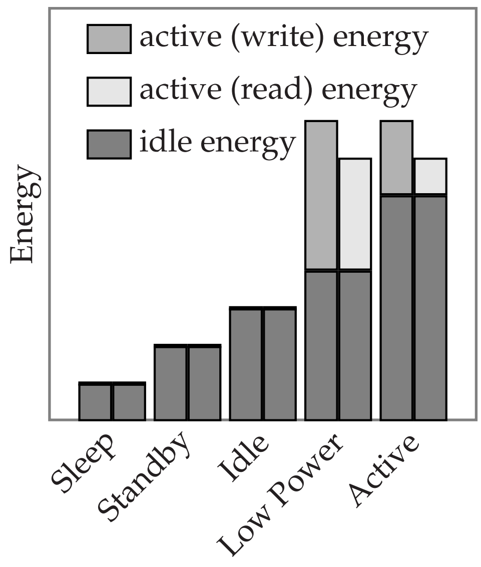

Figure 3.5: Composition of the total energy usage of a device

The idle energy has to be accounted periodically and additionally the transition-energy and the active-energy have to be accounted whenever they occur. This poses the problem that the idle energy must not be accounted, when the device is active or changing states. To enforce this there are three options.

The first option is not to account idle-energy when the device is active or changing states. This is problematic since every time the idle-energy should be accounted it has to be checked if the device is idle which produces a lot of overhead.

The other two options take advantage of the fact that active-energy[6] is higher than the idle-energy. For this reason the idle-energy can be accounted no matter what the device is doing (of course it has to be adapted if the state changed). When the device is active the difference of the active-energy and the idle-energy has to be accounted. This difference may be determined using two different methods. The first method is that the active-energy is calculated using the equation $usage * energy\_per\_usage$ and then the idle-energy for the time the activity lasted is subtracted. This has the advantage that the active-energy needs not to be saved for every state in which the device may be used but only once. The disadvantages are

[6]transition-energy is handled equivalently. For this reason only active-energy is discussed here

that the calculation has to be performed whenever the device is accessed, and the length of the usage has to be known which is in most cases a problem to determine since the devices often have own buffers, or the drivers optimize the accesses at a level where it is no longer known who induced the access.

Another possibility is to calculate the differences offline and to save them together with the other values. This needs more memory, because for every state in which the device may be accessed the values have to be saved. The advantage of this method is that the energy per access (without idle-energy) can be calculated very exactly, because the length of the access, the used energy and the idle-energy per time are known. Another advantage is that the calculations are done offline and for that, no additional calculations to determine the energy during accounting are needed which saves energy and time.

Regardless of how the energy is accounted, it has to be decided where to account the used energy.

3.2.1 Accounting activity

The device usage and active-energy are accounted to the resource container that uses the resource. This is normally (synchronous access) the current RC. When accesses are done asynchronous the RC that initiated the access has to be charged with the usage and energy.

The energy used for the activity can be calculated by multiplying the resource usage (e. g. written bytes) with a factor (e. g. μJ/ byte) that has to be determined for each activity that a resource may perform first.

3.2.2 Accounting state changes

Accounting transition-energy is not as simple as accounting activity, since often more than one process benefits from the change.

Accounting to the resource container that caused the state change

Simply accounting to the RC that caused the change is the solution that uses the least resources itself (CPU-time and energy) but it is also the unfairest solution because if more processes are running, and all want to use the resource, it is coincidental which one is first. If the whole energy for the state change is accounted to this process, the other processes which may be using the resource immediately after the first are the winners, because no transition energy is accounted to them although they benefit from the state change.

Accounting to all the resource containers that use the resource in a certain time period after state change

This may be avoided if the transition energy is accounted to all processes that use the resource in a certain period after the state is changed (for example 30 sec). If only one process is using the resource, all energy is accounted to it. The problem thereby is that it is not known how many processes and which processes use the resource during this period. Because of that the accounting has to be delayed until the period passed, or the energy is redistributed every time a new process uses the resource.

Delaying the accounting causes problems if a process terminates before the period ends, because then its share has to be accounted to another RC (for example the parent RC, or it is accounted to the other RCs that used the resource).
Another problem is that if the accounting is delayed, limiting the usage may be inaccurate during that period, because the processes have more energy left than they may consume, and then, after accounting, because they used more than they were allotted, they are suddenly delayed for a long time until they have enough resources to run again. This may lead to bursts in resource usage and to long idle periods.

Redistributing the energy during the period causes computational overhead and also has impacts on the limiting.
If redistribution should be avoided, it is possible to split the transition energy in shares, and account one share per time to the processes. The apportionment is recalculated every time splitting the share in as many parts as processes have been using the device until now (since the state changed) and account one part to each process. This also solves the problem if a process terminates and its RC is no longer existent, because then the energy may be shared amongst the other processes.

Accounting to a separate resource

If the computational overhead of the last method should be prevented, and if it is not important that the transition energy is accounted to each process as accurate as possible but the global accounting should be accurate, it is possible to account it to a separate resource container.

Deciding which method should be used depends on the transition energy itself. If it is only a very small value, it is probably better to account to the separate resource, because calculating the distribution may use more energy than the transition energy originally was. If, however, the states are changed infrequently and the energy usage is high, it may be worthwhile to distribute the energy because eventually the policies that limit access to the resources may need this information.

Accounting state changes into a better power management mode

Another problem is the transition energy if the resource switches back to the previous state, or if it switches from active mode into a low-power mode (e. g. sleep). This energy has to be accounted before it is used, because if the state is changed, no processes are using the resource. The best solution is, to sum up the transition energies that are needed to switch to the state that uses more energy and to switch back again and account this energy with one of the beforehand introduced methods.

It has to be observed that the energy is accounted in the state that has the higher energy usage, because this is the one into which is only switched if the resource is used. Summing the transition energies up in sleep mode (switching from active to sleep-mode and back) yields nothing, because there are no active processes the energy may be accounted to, whereas doing it vice versa (sleep-active and active-sleep) allows to account the energy to processes.

3.2.3 Accounting idle-energy

Idle-energy represents a large part of the overall used energy but accounting it is not as easy as accounting transition-energy or active-energy.

The idle-energy is not finite like the transition-energy and accounting it is problematic, because there may be no RCs available. This is the case if no processes run, or if no processes used the resource, or all processes that used the resource terminated.

Below some more or less fair methods of accounting idle energy are shown. Accounting the idle-energy to any RC that is available is no solution, because then limiting is not possible.

Account to the resource container that used the resource at last

This approach is quite simple: If a process uses a resource this is saved and when the resource is idle the idle energy is accounted to the process that used the resource last.

The first problem of this approach is that if the resource has not been used at all the idle energy cannot be accounted to a resource container. This may be solved by using a special resource container as fallback which will be used, when no other RC is available.

A similar problem occurs if the process terminates to which the idle energy is accounted and for this reason the resource container no longer exists. But instead of using the fallback-RC here it is possible to use the parent RC (which poses problems if the resource container has more than one parent).

Besides this approach is not fair, because if a process uses the resource for a long time and then directly afterwards another one uses the resource very short no idle energy is accounted to the first process but everything would be accounted to the second one.

Account to every resource container that used the resource

To avoid the problem that all idle energy is accounted to one resource container although other processes have also accessed the resource, the idle energy is split and accounted to all resource containers that accessed the device. Splitting the energy may be done using different methods. The most important are described later in this section.

First a look is taken at *how long* the idle energy should be accounted to a process that used the resource once.

One possibility is that if a process used the resource, idle energy (whereas idle energy is here not the complete idle energy but the share that is determined by the later on described methods) is accounted to it, until it terminates. Accounting continuously poses problems, because calculating the shares will get more and more complex over time, when more processes have been using the resource. Also it may become inaccurate because of rounding differences due to many divisions and small numbers.

Another possibility is that the idle energy is only accounted to the process for a certain time after it used the resource the last time. This will assure, assumed a well chosen interval, that the idle energy is accounted to only few processes. Also a process using the resource only once for a short time is not charged forever. A problem of this approach is that after some idle time, there is no process left, to which the idle energy may be accounted to which leads to the need of accounting the idle energy to a special resource.

The other question is, how the complete idle energy should be *shared among the processes*. The arguable simplest approach is to equally share the energy between all processes (processes are the processes the idle energy has to be accounted to as selected by the beforehand described methods). Although very simple, this is not fair, because it does not incorporate the time the processes have been active. A process that used the resource for just a very short time is treated equally as a process that used the resource for a long time, or repeatedly uses the resource. This may be improved by introducing weights which are used to compute the shares that are accounted to the processes.

The weights can be determined using the time since the last use of the resource of a process. The longer it has been, the smaller is the weight, i. e. the smaller is the share of idle energy that is accounted to the process. If a process uses the resource again, the weight is set to the maximum and then decreasing again over time. The advantage of this method is that to processes which have not used the resource for some time only a small share of the idle energy is accounted, whereas the most is accounted to recently active processes. However this method does not reflect the device-usage of the processes. To a process that uses the resource for the first time the same amount of energy is accounted as to a process that uses the resource regularly and for a long time.

This can be avoided by not using the idle-time but the active-time to compute the weights. The longer a process has been using the resource the higher is the weight and the more idle energy is accounted to the process. Though it is the most expensive method, because the active-time of each process has to be saved for each available resource, and a lot of computation has to be done to get the shares that are accounted to the processes, the fairest accounting of idle energy is achieved.

A further issue to be considered is the time period over which the weights should be calculated. It is possible to recalculate the weights using all data since the system started which means that it is likely that accounted energy has to be redistributed.

For example if one process used the resource all idle energy is accounted to it. Then after some time a second process uses the resource which means that half of idle energy since the system start has to be accounted to the first and the other half to the second process. As however the idle energy has been accounted to the first process, it now has to be subtracted and added to the second process. This has to be done for each process that uses the resource which leads to a very high overhead while accounting (values for idle-/active-time have to be saved and the weights, how the energy has been distributed until now). There is another problem: limiting is no longer possible, because it may happen that a process suddenly 'loses' a lot of accounted energy which it can use again, whereas another process that gets this energy accounted, exceeds the set limits and is not allowed to run for a long time, until the limits are met again.

Another problem occurs when a process terminates. It is no longer possible to account to this process and the weights must be reconfigured. The energy must not be subtracted from this process and added to the others but new weights would exactly do that. A solution would be to begin a new time period, where the accounted energy until now is fixed and only the new energy is distributed.

This will be quite similar to calculating the weights from the moment the resource is idle again. With this method idle-energy is accounted and not redistributed. The weights are calculated after the resource is idle again, and then it is constant (or may decrease, depending on the method used but the proportions stay constant) until the resource is used again which then leads to a recomputation of the weights.

Using this method, the idle- or active-times of the processes have to be saved for each resource also but no other additional data has to be saved, because distributed energy cannot be redistributed. This leads to a reliable limiting, because it does not have to be considered that large amounts of energy are transferred from one process to another.

For both methods there is yet another problem: If no one was using the resource or if all processes the idle energy is accounted to have terminated the idle energy cannot be accounted to a process. To not lose this energy amount, it may be accounted to a special resource container. This also applies if there are no processes left to account the idle energy to, for example because to much time has passed and the resource has not been used again, or if not the complete idle-energy is distributed amongst the RCs.

3.2.3.1 Account to a separate resource

If it is not important, to accurately share the idle energy[7] but to account the total used energy accurately a simple solution may be used: The complete idle-energy is accounted to a special resource container.

Actually there is an advantage using a separate resource for accounting the idle energy: The other resources only show the real usage of the device which may be

[7]Which is the case in this work, since limiting the idle-energy is hardly possible

used to calculate accurate limits and to perform better estimations, because the values are not tainted with the idle energy.

Another advantage of this method is that a lot of time and energy is saved, because no complex calculations need to be performed in order to find out how much energy is charged to what RC.

3.2.4 Accounting the resources

3.2.4.1 CPU

The power consumption of the CPU can be determined by counting how long it stays in the different states (active, idle, halt, ...). More detailed information can be obtained analyzing which code is executed or by reading the performance counters. They count events inside the processor and can be used to determine how much energy was used because it is possible to draw conclusion from the counters to the activity of the different parts of the processor. The runtime of a process may be calculated using the time stamp counter (TSC) which has a higher resolution than the normal timer (which has only 100 or 1000 ticks per second depending on the architecture and the kernel version).

Finally, to get a correct global energy usage, the idle consumption of the CPU has to be accounted. This is done by counting the idle time of the CPU and multiplying this time with a weight. Using the performance counter is not possible because during idle periods no events are counted.

3.2.4.2 Hard disk

Accounting the hard disk is possible by counting the number of read and write accesses and accounting the time the hard disk was in the various idle-modes. Accounting the transitions between the states is important, since a lot of energy is used for it.

3.2.4.3 WLAN

Accounting WLAN is done similarly to the way of accounting the disk. The bytes sent and received are counted. Additionally it is important to accurately account the idle times since the device is normally idle for most of the time and a lot of energy is used during that period.

3.2.4.4 Camera

Accounting the camera can be done by counting the number of pictures taken. Accounting the used energy is done by multiplying this number with the energy that is needed for taking one picture.

3.2.4.5 Motors and sensors

Robertino is equipped with three identical motors which will be accounted to one resource. There are different possibilities for accounting since not only the PC is involved but also the microcontrollers to which the accounting may be sourced out.

Accounting directly at the PC: This has the advantage that there is no additional traffic on the CAN-bus and accounting can be done as often as needed. But there are also problems using this method: The speed has to be extracted out of the packages that are sent over the CAN-bus. This delays the messages and may lead to problems if the robot has to react quickly, for example to evade an obstacle. The next difficulty is that the speed of the motor does not represent the used energy. This is because energy depends also on the underground the robot is driving on. This may be the same underground for all motors but it may be different for each motor, or at least two of the motors. This may happen if the robot drives along the transition of two different undergrounds. The frictional resistance of these undergrounds is different and for that reason the motors need a variable amount of energy. The needed adaption of the speed is done automatically by the microcontrollers. These are also limiting the maximum speed of the motors to not risk damage of too high current.

For these reasons the accounting at the PC using only the speed of the motors is only a rough estimate. It is not countervailed by accounting the data as often as needed.

Accounting at the microcontrollers and then sending the data to the PC: In this case all needed information is available and the used energy can be accounted exactly. The accounted data may be summed up for some time (e. g. 100ms) and then send to the PC where it is added to the resource. Collecting the data and sending it after some time is done to reduce the traffic on the CAN-bus. A drawback of this method is that the accounted energy is always behind the actual energy used.

3.2.4.6 Battery - a unique resource

Accounting the battery is very important, because without accurate battery-data no valid estimation of the runtime can be made.

Since on Robertino no SMBattery is available one of the ADCs of a microcontroller is used to measure the current voltage. Measuring the remaining capacity is not possible without discharging the battery due to the measurement. This is because for a short time the maximal current has to flow which then may be measured using a current-to-voltage converter attached to the ADC.

A simpler method is to measure the voltage and to determine the runtime from it. This is possible because the voltage drops when the battery is discharged[8].

The remaining runtime of the robot can be estimated with two different methods:

- *Comparing* the current value with a representative discharge curve.
  This is quite accurate if the discharge curve has no parts where it is parallel to the x-axis. Unfortunately these curves do not exist but have to be determined. The next problem is that there is not only one curve but there are many for the different discharge currents. This complicates the runtime estimation, because the current drain of the robot is not constant and therefore there would be a couple of points where the current voltage fits the reference curves which results in inaccurate runtime.

- *Estimating* the remaining runtime. This may be done in different ways:

  Simple extrapolation v1: The easiest solution is to divide the remaining voltage ($V_{remaining}$) by the voltage drop of the previous (V_{prev}) and the current interval (V_{curr})[9] and multiply the result with the length of the interval .

  This is shown in Equation 3.1, where RT is the remaining runtime, V_{crit} is the critical voltage, where the robot cannot operate anymore and $T_{interval}$ the length of an interval.

$$RT = \frac{V_{curr} - V_{crit}}{V_{prev} - V_{curr}} * T_{interval} \tag{3.1}$$

  A big problem of this extrapolation is that if the voltage of the last and the current interval are the same, the runtime is undefined.

  This problem may be avoided using a longer interval. But then the estimation is less precise as with a shorter interval. On the other hand a longer interval is better, because prior to dropping to a lower value the voltage is oscillating between the current and the lower voltage for some time which is smoothed in a longer interval, whereas with shorter intervals it is possible that one is the low and the next is the high voltage again.

  With these problems in mind, the method has been extended to:

[8]There are of course more sophisticated methods to get the remaining capacity but this one is quite simple and sufficient for this work. Using another method is possible by adapting the code of the resource battery (see subsection 4.2.4.

[9]During one interval (e. g. 1 sec) the voltage is measured periodically (e. g. every 100ms) and then an average for this interval is calculated.

Simple extrapolation v2: Instead of always using the last interval the first interval which has a higher voltage than the current one is used (search direction is from the last interval to the first interval measured). If no matching interval is found the runtime cannot be determined.

This solves the problem that occurred if the last interval had the same voltage as the current. The oscillating problem still exists but it is eased, as the last higher value is used. At worst the runtime is determined shorter that it is in fact which will be corrected withing the next few intervals but a runtime can be calculated every time.

Weighted simple extrapolation: To avoid huge fluctuations of the remaining runtime also older estimates may be used. A problem may be that if the runtime has changed a lot because new limitations are set or older ones were canceled, this will not be noticed immediately but only after some intervals.

A simple way of calculation is e. g. (V_x is the voltage before x intervals; $V_{curr} = V_0$):

$$RT = \frac{V_0 - V_{crit}}{\frac{(V_3-V_2)*1+(V_2-V_1)*2+(V_1-V_0)*3}{6}} * T_{interval} \tag{3.2}$$

Sophisticated extrapolation: A completely different approach is also possible.

Instead of using constant timeslices and observing the voltage drop the time ($Period_{drop}$) is measured that passed until the voltage has dropped by a certain value.

With this time it may be calculated how long the energy may last:

$$RT = \left(\frac{V_{curr} - V_{crit}}{V_{drop}} * Period_{drop} \right) - Period_{curr} \tag{3.3}$$

Here $Period_{curr}$ is the time that has passed since the start of the current period.

The problem of voltage fluctuation does also exist therefore a smoothing should be used to obtain more stable values. The advantage of this method is the need of less storage than with the previous method for a new interval is not started until the defined change in voltage occurred. Hence there do not exist empty intervals. This simplifies the calculation because intervals without a voltage change do not have to be considered.

The voltage loss may not only be used to calculate the remaining runtime but also to trigger events (e. g. change limits) if the voltage drops below a threshold.

3.2.4.7 Global overhead

Accounting the in subsection 3.1.7 mentioned energy consumption to the special resource *global overhead* is required to cover all used energy, and concurrently be able to account the usage of the other resources as detailed as possible to be able to limit them correctly. Although this energy cannot be reduced, accounting it is required since the information may be necessary for example for thermal management. Limiting is also possible if the herein accounted energy is accounted to the other resources in a fair way which is only possible with high complexity which is again using time and energy.

The simplest possible solution is to introduce a resource that is located beyond the resource container hierarchy (and exists only once) where this amount of energy is accounted to.

This resource is for accounting only and the total energy usage that cannot be accounted to other resources for any reason is stored here which simplifies the accounting a lot without posing any downsides.

3.3 Limiting

This section analyses if resources may be limited and which side effects may occur. Refreshing the consumed resources is done as described in subsection 2.2.3.

3.3.1 Limiting basics

Limiting is possible using two different methods. The common method is to not schedule the process anymore if one resource is depleted. The disadvantage hereby is that the process must not run anymore until it has enough resources gained to run again.

The second method is to limit the energy usage of devices by utilizing the different power management modes of the device. Doing this allows the processes to continue using the resource but with a reduced performance (speed, quality, response) while the energy usage is lower, too. Of course, if a process runs out of resources it still has to be stopped but it may run for a longer period of time since its resources do not drain as fast as with an unthrottled device since this is now using less energy. This may necessarily lead to a longer runtime of the process but since the process may not use the full capacity of the resource it may not notice the throttling.

3.3.2 CPU

Limiting CPU-usage and hence the energy-usage may be done by not allowing the process to run. So it does not need resources and the CPU uses less energy.

Problems like exceeding deadlines and therefore needing extra energy due to recalculations may always happen when limiting. For this reason one has to act with caution when using this system on real-time systems.

Additionally, to be able to calculate the total energy usage, the idle consumption of the CPU has to be accounted. Since the CPU used in this work does not support any power management the energy consumption during idle periods is constant and cannot be reduced.

3.3.3 Hard disk

Limiting the disk is not as simple as limiting CPU because restricting the access could result in data loss.

Read accesses may be delayed by halting the process until enough resources are available because it will wait until the access is completed. A problem is that the process probably may never gain enough resources to complete the read request and for this it will never run again, or it needs a long time until it gets the resources and for that reason it may miss deadlines, needed for successful work of the robot.

Write accesses are normally done asynchronously. That means the process writes the data into a buffer and after some time the data is written to disk by another process. Delaying this write will yield nothing, because the process that issued the write is no longer involved and will not be detained from issuing more write accesses. Only the process writing the buffers to disk would be delayed which is critical, because other write accesses could have enough resources left and they would be delayed too when they are queued after the one that is out of resources. Another possibility is to cancel such an access which is not an opportunity, because this leads to data loss. Rearranging the accesses, so the one that has not enough resources will be processed later and the others are preferred, is possible but then a lot of memory may be used if a process has not enough resources to write but schedules a lot of writes before it is stopped. The computational overhead (time and energy) for resorting the accesses is also very high which renders this option useless.

Therefore writing always should be possible, no matter if there are enough resources left. If, after writing, the resources are depleted, the process will not be scheduled until there are enough resources again. So the process just gets a credit, and over a longer period of time the energy-usage of the complete system will be within set limits although there probably will be peaks overshooting the limits.

Limiting energy usage by changing the state is only possible for the idle energy. As shown in subsection 3.1.2 there is only one state where the disk may be accessed. To reduce the idle energy the disk may be put into *standby* mode but it has to be observed that switching from *standby* to *active* needs time and additional energy. For this reason switching to *standby* may not always lead to energy savings.

3.3.4 WLAN

Limiting the WLAN interface may be done similar to the disk. Writing (sending) may be delayed until enough energy is available. Receiving data may be delayed by putting the device into *Power Save (PS)* mode.

Determining the optimal *beacon period* is problematic, because a too high interval may lead to a couple of problems, for example if there are streaming applications running it may come to slow downs because the buffers run empty. If the interval is too short, more energy may be needed than staying awake all the time, because after each wake-up a TIM has to be sent.

Another possibility to save energy would be reducing the speed of the device but as stated in subsection 3.1.3 the most energy is used in idle-mode and for that reason the savings would be minimal and not worth the effort changing the speed. Also reducing the speed raises the energy needs of the device since the transmissions take longer.

3.3.5 Camera

Limiting the energy needs of the camera could be done by reducing the image size (resolution) of the taken image, or reducing the rate the images are taken.

Changing the image size is not a good solution. Because the applications using the camera are expecting a specific size, namely the one they configured the camera for. Changing the size could yield in crashing the application, or in erroneous image processing which again may lead to other errors, like colliding with an obstacle. Whereas limiting the framerate should not pose any problems, because the application has to be able to deal with missing frames because this may happen if the resolution and the framerate are very high, or the traffic on the bus is very high (for example if other applications are using hardware also attached to the firewire bus). Of course limiting the framerate has a drawback: the video possibly made of the pictures will buckle and have dropouts. But for image processing, for example to build maps or to detect obstacles, the lowest framerate the camera supports which are 3 FPS, should be enough, also because if the camera is limited there is a high possibility that the motors are limited, too, and for that reason the robot is not driving as fast that obstacles may not be detected.

3.3.6 Motors and sensors

Limiting the sensors is not possible, because they do not support any power management. When limiting the motors it has to be observed that the energy is not increasing proportionally to the speed. As it is shown in Figure 5.1 (on page 55) the energy increases and then, after a certain speed threshold is passed it decreases.

Limiting like accounting (see page 28) is possible at the microcontroller or at the PC.

Limiting at the PC

Limiting the motors at the PC is affected by not certainly knowing the speed and thus the energy usage and the speed may be changed by the microcontroller. This may lead to a higher usage than desired and in worst cases to undesired movement if not all motors are limited equally.

To be able to limit all motors equally, the limit should set as a percentage of the original speed. For example if the limit is 100 % the speed is not limited. If the limit is 60 % the speed is set to 60 % of the given speed. This ensures that the ratio of the speeds stays constant and the robot is always able to drive. A percentage rate has the disadvantage that the energy needs after throttling cannot be determined easily. When limiting other resources it is known how much energy is saved, here it can only be estimated, because the speeds of the motors vary and for this reason the energy usage and savings vary, too. Thus the limit has to be corrected if the limit is not kept, i. e. the usage is higher or lower.[10]

An advantage of limiting at the PC is that all motors can be limited simultaneously.

Limiting the speeds has to be done before the speed is send to the microcontrollers. To ensure that all motors are limited equally, the messages that are sent have to be delayed until there is one for each motor. After the speed has been modified the messages may be sent. This may lead to critical delays if for example the robot has to evade obstacles.

Another possibility is, to save the current unlimited speeds of the motors and if the limits change, the new speed is calculated with the saved values and then sent to the motors. This reduces the delays but the traffic on the CAN bus is increased.

Limiting at the microcontrollers

When limitation is done at the microcontrollers the energy usage and the speed are known but the speeds of the other motors are unknown. Throttling the motor without this knowledge may lead to different limits on each motor and thence driving in the desired direction will no longer be possible. The only possibility is that the motors communicate with each other and try to find a limit themselves but then again the base energy limits have to be sent from the PC which is the only device that knows all speeds and then at least three messages are needed before the limit can be applied.[11]

Limiting using both, the PC and the microcontroller

Probably the best solution is to combine both methods:
The calculation of the limit is done on the PC, whereas the limit is a percentage of the original speed which is then sent to the microcontrollers that are responsible for applying it to the speeds. With this method no delay (except the calculation of the new speed) is occurring so the robot does not lose its ability to respond quickly and there is only one broadcast message sent over the CAN bus to distribute the limits.

[10]A lower energy usage means more savings but also higher performance loss, so it should be tried to get as close as possible to the limit.

[11]Three messages, because one microcontroller has to get the energy usage from the other two, calculating the limits and then distributing the limits which can be done broadcasting the limit with one message.

3.3.7 Interdependencies

A side effect that may happen while limiting one resource is that more energy than expected is saved because other resources need less energy, too.

If for example the CPU is limited and their resources are depleted the process is not allowed to run until the resources are refreshed. During this time no energy can be consumed by the other resources. This leads to higher energy savings than intended.

In this section interdependencies are analysed and their assets and drawbacks are discussed.

It is granted that the application(s) attached to the limited RC are using all resources available. Decreasing performance and increasing response times are not mentioned for each resource. It should be clear that limiting normally leads thereto.

		Limited resource				
		CPU	*disk*	*WLAN*	*motor*	*camera*
Interdependencies	*CPU*	-	√	√		√
	disk	√	-	(√)		√
	WLAN	√	(√)	-		√
	motor	√	(√)		-	
	camera	√	(√)			-

Figure 3.6: Overview of interdependencies when limiting resources.
√ means direct dependencies, while (√) denotes indirect dependencies.

CPU If the limits of the *CPU* are exceeded, the process may not run anymore which leads to further savings:

Energy is saved in the resource *disk*, because no more read accesses are made, and only the already made write accesses will be executed.

The resource *WLAN* is also using less energy, because it is not accessed. But data for this application may be received and for that energy may be consumed.

The *motors* stop after about 100ms of CPU limiting, because they need a drive command periodically.

No energy is saved by the *camera*, because the camera delivers pictures with the set rate, no matter if they are used or not.

For this reason the longer the CPU is limited, the less resources are consumed.

The drawbacks are that data received over *WLAN* is not processed and may be lost.

DISK Exceeded limits of the resource *disk* result in delayed read accesses. The write accesses will always be executed for safety reasons (If the robot runs out of energy it is better to use some more energy to save the results of the work done, than to lose all data but drive some more centimeters).

If a read access is delayed, the process is stopped and the same savings as with exceeded *CPU* limits apply.

WLAN Limiting *WLAN*, like the *disk*, also saves *CPU* energy, because transmitting (writing) is also delayed by scheduling away the process. Since receiving data may be delayed, further energy may be saved, depending on the applications and what they do with the received data (write it to disk, process it, ...).

But it is also possible that more energy is used, because data has to be retransmitted, or the application has to recalculate data.

The *motors* and the *camera* are not directly affected by the limitation.

MOTOR Limiting the *motors* has no direct effect on the energy needs of the other resources. Depending on the running applications it is possible that the *CPU* and *camera* also use less energy if the motor-speed is used to adapt the framerate of the camera, or to adapt the algorithm that detects obstacles with the sensors.

CAMERA Limiting the *camera* may save *CPU* as well as *disk* and *WLAN* energy. This is because the framerate is limited, and less pictures are available per second. Processing fewer images means less *CPU* usage and therefore less energy consumption. Depending on the application it also means less data written to *disk* and/or transmitted over *WLAN* but only if the application just writes new images and not one image per interval, no matter if it is new or not.

The energy usage of the *motors* is not affected by limiting the *camera*.

3.4 Controlling energy usage

To process the accounted data and to limit energy consumption an interface is needed that provides the data and possibilities to set the limits.

3.4.1 Required data

Here the data required for adaptive policies is presented.

At first it is useful to query the global data, like the *number of resources* and the *number of special resources* available and *number of timeslices* in combination with the *length of the timeslices*. The data of the special resource is shown later in this section. For each resource the following global information can be queried:

- The *name* of the resource.
- The *device state*, i. e. if it is online, or switched off.
- The *available power management states* and the
- *current power management state*.
- If the resource is *limitable* or for accounting only.
- The *unit* in which the usage is accounted.

Additionally for each resource the **energy-profile** is available. This consists of the *transition energies* from each state into each other state (if a state change from one state to another is not possible the energy is 0), the *idle energy* which is consumed in each state and the *active energy*. The *active energy* is the energy that is consumed if the device is accessed. This may consist of different values, for example energy for reading and energy for writing. Depending on the strategy used for idle-accounting it is possible that there is separate *active energy* value (or more) for each state. This value may also be only the difference between idle and active (for details see subsection 3.2.3). If the device cannot become active in a state the *active energy* is 0[12].

Except the *device state* and the *current power management state* this information is constant and it is sufficient to query it once.

[12]It is assumed that activity is always consuming an additional amount of energy. If this is not the case 0 may be replaced by another value that is not used as energy value (like NULL or -1).

For each resource container the following information is available:

- Amount of *energy used* and
- device *usage* since the start of the accounting.
- The *limits* for energy consumption and the usage for each timeslice.
- Additionally the *PID*s of the processes attached to this RC
- if they are *servers*
- and the *children* of the RC are available.

The resource container reference that is needed may be obtained by reading it from another RC, searching for a PID and its respective RC, or by opening a file in the resource container file system introduced in [Wai03] which is also used in this work.

For the special resources other values are provided.
These are the measured *unit*, the *current value*, the *critical value* which is for example when the threshold where the measured value is too low to guarantee operation (in this case, the voltage is too low to drive).

Hence the runtime cannot be calculated without having older values and extrapolating it from these values, it is necessary to store a history of the values. This poses the problem of where to store the history. Below the pros and cons of both possible locations are discussed.

Keeping it inside the kernel has the advantage that the data can be updated whenever a new battery value is accounted.

Updating the history in the user-space means periodical querying the battery value. If the policy has a low priority it is not guaranteed that the values are updated periodically. This leads to more complex calculations to generate the history.

An advantage of the user-space is that unlimited space for storing and time for calculation is available. This means that more sophisticated methods for generating and handling the history are possible.

A severe disadvantage of storing the history in the user-space is that on switching the policies the history is lost since a policy is an application and switching implies terminating the application and starting another one. This may be solved in various ways, for example implementing a daemon that stores the history and the policies query that history, or one policy saves the history and the other loads it. However, all of these proposals consume additional resources.

For these reasons the history be maintained within the kernel and additionally to the beforehand mentioned values the *remaining runtime* has to be provided.

3.4.2 Controlling resource usage

The energy usage can be controlled by setting other (needing a smaller amount of energy) state for the resources. The energy savings can be calculated using:

$$energy_{savings} = energy_{idle_{current\_state}} - energy_{idlenew\_state}$$

This is only effective for a longer period of time if the resource supports becoming active in the state it is switched to, otherwise the next access to the resource will yield to a change back to the current state. It is possible to determine this, by checking the energy-profile. If the active energy of the new state is 0 it cannot become active in this state.

The benefit of this method is that the policy needs no knowledge about how to set the states. The needed data which are the costs for changing a state and the energy needed within the new state are known, because they are part of the energy-profile.

Another way of controlling the energy usage is to limit the device usage and therefore the energy that is consumed additionally to the idle energy. The energy used for activity is either saved directly as the usage, or it may be calculated using (for one resource):

$$energy_{active} = energy_{total} - energy_{idle} * time_{device\_online}$$

Which of the two possibilities is applicable depends on which strategy for accounting shown in subsection 3.2.3 is used.

With the usage in the current state and a given total energy limit the policy is able to calculate the usage limits for the resources using Equation 3.4. If the resource is not limited before, $limit_{resource_{old}}$ has to be replaced by $usage_{resource}$.

$$\begin{aligned} limit_{percent} &= \frac{energy_{active} - energy_{limit}}{energy_{active}} \\ limit_{resource_{new}} &= (100\,\% - limit_{percent}) * limit_{resource_{old}} \end{aligned} \tag{3.4}$$

If the energy usage exceeds the set total limit the limits have to be adapted.

Interdependencies may be detected automatically, by monitoring the energy usage of all resource before and after setting limits, or switching states of one resource. If the energy used in an other resource changes instantly after the first resource was changed, then there is an interdependency which leads to other energy savings than expected. Normally they are higher, because other resource cannot work all the time as they may depend on data or calculations done by the limited resource. But the contrary may also happen, the energy usage of one resource is limited and the usage of another device increases. This may happen if an application uses for example an attached network drive (via WLAN) and the hard disk for data storage and detects that the access to the network drive is very slow due to limitation and stores all data to the local disk which leads to a higher energy usage of the hard disk.

3.4.3 Example policy

In this section a sample policy is introduced. It is shown that it is possible to easily adapt existing applications for adaptive power management to benefit from the infrastructure. First a brief synopsis of self-tuning wireless network power management is given. Thereafter the modifications that are needed that it takes advantage of using the infrastructure are shown.

In [ANF03] self-tuning wireless network power management is introduced. This is needed, because in different situations different power management strategies are needed and that cannot be achieved with a 'one fits all'-approach. The self-tuning power management (STPM) adapts to the characteristics of the hardware and the applications. A novelty of STPM is that energy and time needed for changing power modes and the base power usage of the devices are considered. To be able to adapt to the applications STPM provides a simple interface that allows applications to disclose hints about their intended network usage. For unmodified applications (they do not disclose hints) a special hinting module was developed that attempts to identify these applications and to predict their network usage.

STPM uses hints from applications to switch the power management mode of the WLAN device. The applications inform the module of the maximum latency they accept and the amount of data that is to be transferred. STPM then decides if the device should be put to CAM (continuously-aware mode) or stay in PSM (power saving mode). After finishing the transmission the applications informs the STPM again which then may switch back to PSM.

For unmodified applications the hinting module tries to predict the usage. This is done by observing all network traffic and mapping the network ports to applications. If the applications disclose hints the traffic is ignored by the hinting module. At this point the first big problem arises. If an unmodified application uses the port that was formerly used by a hinting application the traffic and therefore the application is not considered.

The hinting module informs the STPM of transfers and the device is depending on the traffic put into CAM. If for a defined period of time no more packets are sent, the STPM is informed again and the device may be put back to PSM.

STPM can be modified to take advantage of the resource accounting and limiting infrastructure. To be able to effectively manage a device STPM has to know the energy profile of this device. This information can be obtained querying the infrastructure. Setting the power management modes of a device is also possible. This enables the STPM to be used much more flexible because it adapts itself to new hardware since it can gather the needed information and no changes to the STPM are needed if devices are changed.

Adapting the hinting module not only simplifies the hinting but renders it more accurate.

This is because the detection of unmodified application can be improved by not mapping the applications to network ports but by checking whether the RC is already known and processed. If it is unknown it has to be checked whether the application supports hinting or not.

An unmodified application is similarly handled as with the original module. However, the data needed for determining when to switch power management modes (transmitted bytes, last transmission) can be queried from the infrastructure.

Chapter 4

Implementation

The accounting and limiting system described in this work was implemented for the Robertino-Robotsystem running a Linux system as introduced in section 2.1. The kernel version is 2.6.7 modified to support the original resource container structure which was then enhanced and adapted to fit the needs of this work.

Lots of the original implementation from [Wai03], like for example the core system, the resource container structure and the propagation of resource bindings, could be reused while the structure of the resources had to be changed extensively. This was required because with the original implementation it was not possible to account the different states and more values (energy and usage) per resource.

Changing the resource structures leads to changes in the accounting procedures. For example it is no longer possible to only account to the current resource container, or switch the resource container in certain situations. Additionally idle time also has to be accounted which neither Banga nor Waitz did.

This chapter describes the new resource structures followed by a detailed view on the implementation of the different resources and their particularities in accounting and limiting.

4.1 Resources

While the original resource structure was kept very simple, this is no longer possible, because additional data has to be saved to be able to account and limit the system efficiently.

Also by changing the point of view the definition of resource has changed (as described on page 13). While earlier the view was towards the processes now it covers the complete system. Back then the question was: Which resources will a process use? Now it is: Which resource requires how much energy and what is this energy used for?

The new resource structure is split into two parts: A global structure that exists once for each resource, holds the static data and the data that must not be managed for each resource container. The second structure exists once for each resource container. Within this structure the data per RC, like usage, and limits, is saved.

The global structure holds the following data: The *resource description* which consists of a name, the units and scales[1] to convert the accounted data to the unit.

The needed *energy profile* which consists of idle energy for each state and transition energy from each state into each state.

The energy needed in active periods is recorded separately, because the amount of energy values may vary in different resources. For example in the resources WLAN and disk the read and write accesses use different amounts of energy which also depends on the amount of data transferred. Contrary to that the motors need no values for active periods because all calculations are done directly at the microcontrollers which transmit the totally used energy to the computer.

Additionally different *flags* and the *current state* of the resource are saved. The current state needs to be saved to determine the correct energy values, and to account to the adequate states. There are flags to determine if the resource is online, or offline (i. e. switched off or removed) and if the resource is used for accounting only. Amongst others these flags have been implemented for optimization purposes, because, for example checking if the resources are depleted for a device that is not online is not useful. It is also not reasonable to assign resources to offline resources, or to resources that are accounted but not limited. A resource may only be accounted to determine the energy usage of the device, or it cannot be limited, because it only supports one state, or the states cannot be switched[2].

Listing 4.1 shows the data structure used in this implementation. The current state and flag if the device is online have been put into a separate structure because these values change during operation, while the other values are constant.

The other part of the resource structure (shown in Listing 4.2 is saved within the resource container. In this structure that exists for each resource the limits for the resource are stored. Limiting is only possible for the resource but not the different states. This is because limiting each state separately is of no avail but would produce high computational overhead.

The energy usage is accounted for each state separately, so the policies can determine in which state the most energy is used and if it is of use to limit the resource by setting it into another state. The amount of energy that is needed to switch from one state to another is accounted independently of the states. This is done to be able to determine, how much energy is used due to state changes. If for example most of the energy usage of a resource is caused by state changes it would be reasonable not to switch the resource anymore but leave it in one state that probably uses more energy than the other states, however, the switching overhead would be avoided.

[1]Scaling is necessary, because within the kernel floating point arithmetic is not easily available, whereas within user-space the real units are preferred to abstract integers (like timer ticks).

[2]The CAN-bus could be represented by such a resource however, the effort needed to determine the energy needed by device is very high but leads to no advantages.

```
struct res_driver_info {
    const char *const name;
    const char *const unit[RES_CNT_COUNT];
    const char *const unit_per_time[RES_CNT_COUNT];
    const u32 scale[RES_CNT_COUNT];
    const u32 scale_per_time[RES_CNT_COUNT];
    const u32 idleEnergy[RES_STATE_COUNT];
    const u32 transitionEnergy[RES_STATE_COUNT][RES_STATE_COUNT];
    const u8  isLimitable[RES_CNT_COUNT];
    const u8  availStates;
};

struct res_driver_state {
    u8 currState;
    u8 devOnline;
};
```

Listing 4.1: Global values for a resource

4.2 Accounting

To be able to schedule the processes according to the information saved within the resource containers the accounted data has to be correct.

A generic interface for resource accounting is provided which then internally distributes the data to the accounting function of the according resource. Additional interfaces are provided for adding/removing devices and for changing power management states.

Accounting activity is done by recording the difference from idle to active-usage as described in subsection 3.2.3. In the following particularities of the implementation and arising problems of the accounting for the different resources are described.

4.2.1 CPU

Accounting CPU usage (time) is done by sampling the TSC as introduced by Martin Waitz. Measurements (see subsection 5.1.1) show that the smallest deviation between measured and accounted energy is achieved when only sampling the TSC and multiplying it with an energy weight. It turned out that using performance counter is far more inaccurate.

```
struct rc_states {
    u64 stateChanges;
    u64 used[RES_CNT_COUNT];
};

struct rc_limits {
    u64 max_effective;
    u64 max[RES_TIME_SLICES];
    u64 last_used[RES_TIME_SLICES];
    u64 avg_usage[RES_TIME_SLICES];

    u64 limit[RES_TIME_SLICES];
    u8 limit_scale[RES_TIME_SLICES];
};

struct rc_resource {
    u64 overheadEnergy;
    struct rc_states state[RES_STATE_COUNT + 1];
    struct rc_limits limits[RES_CNT_COUNT];
};
```

Listing 4.2: New resource structure

4.2.2 Hard disk

Accounting the (energy) usage of the disk is done by counting all bytes read from and written to disk. The problem thereby was that write accesses are done asynchronous, and therefore the RC-binding has to be saved with every write access. The automatic power management of the hard disk is disabled. This implementation uses standby as only power management mode available. The user sets the timeout after which the disk is put into standby. The resource disk has an internal timer which also counts the time since the last disk access and which switches the state of the resource to standby after the set timeout because there is no notification if the disk really switched to standby.

Another possibility would have been to periodically check the mode of the disk and to set the state of the resource accordingly. But this check must not be performed too often since it needs about 500 ns to get a response and possibly the mode of the disk is changed due to this request.

4.2.3 Motors

Accounting the motors is done at two locations. The data is gathered and processed by the microcontrollers which then send a message with the used energy to the PC every 100 ms. At the PC the data is accounted to the respective RC.

4.2.4 Battery

The current voltage of the battery is measured every 2.2 ms at one of the micro-controllers. An average over 100 ms is calculated which is sent together with the motor-data to the PC. At the PC the data is saved in the resource battery.

The resource battery is not within the resource container structure. It only exists once since everything else is only overhead. The resource also has another structure than the other resources. It is only possible to 'account' the actual voltage. It is also not possible to limit this resource.

The resource battery includes a history of the last voltages and a simple estimation algorithm (described in 3.2.4.6) for the remaining runtime. The runtime and the current voltage of the battery may be queried from the outside.

4.2.5 Idle-energy

Idle energy accounting is done by a function that gets called every timer interrupt which is 1000 times a second with this kernel version. This function is also responsible for CPU accounting.

The idle energy is accounted to a separate idle-resource-container.

4.2.6 State changes

Every time the power management mode of a device is changed by the system, the resource container structure is informed and accounts the state change to the currently active RC. This is done to a separate value (overheadEnergy) within the RC to not taint the other accounted energy-data.

4.3 Limiting

Limiting the resources is done by temporarily stopping all processes that are out of resources as introduced by [Wai03]

Additionally the resources may be limited by putting them into other power management modes. This has to be done by the policies which set the modes using a generic interface. The request is then distributed to the concerned resource which changes the internal state and the mode of the device.

4.4 Refreshing

Refreshing the resources is done periodically as introduced in subsection 2.2.3.

A novelty is that the resources support a flag which determines if they should be accounted only or if they may be limited. If they are for accounting only, no limiting is possible, and refreshing the resources is not needed.

Additionally resources can be switched on and off dynamically. This is for example possible for the WLAN interface which can be removed from the PC-CARD slot. If a resource is offline no refreshing is necessary either.

Chapter 5

Measurement and Evaluation

All measurements were done on the Robertino-Robotsystem which was described in section 2.1 using Linux with a modified kernel version 2.6.7.

5.1 Resources

First of all the resources had to be measured to get the energy profiles. For all resources active and idle energy of all relevant states and the energy needed for state changes were measured. The results are presented in this section.

5.1.1 CPU

The energy for the CPU is calculated using the equation

$$perfCtr_0 * weight_0 + perfCtr_1 * weight_1$$

where $perfCtr_x$ is the change of the performance counter x within the accounted interval. Together with the CPU the mainboard and the memory are measured since it is not possible to measure the CPU independently.

To determine the weights the energy is measured and the performance counters are monitored. To simulate many different operations on the CPU the benchmarks of the MiBench suite [Gut] except the telecom benchmark (which did not run on Robertino) were used.

For the values of one run the equation is solved and the minimal error for the weights is searched. This was done using a simple brute force algorithm that tests all possible solutions.

Several passes with different combinations of performance counters were done but finally the smallest error was achieved by simply using the TSC as one value. This simple accounting method can be stated with the old Pentium III M used for testing that is consuming an almost constant amount of energy no matter which instructions it is processing. New processors disable parts that are not needed for the current work. Therefore the power consumption varies depending on the executed instructions. Determining the energy usage of these processors is only possible using the performance counters. Since a performance counter only counts the appearance of one event and the number of concurrently accountable performance counters is limited not all events are monitored. This leads to not being able to account the total consumed energy, because energy is used in parts of the CPU which are not monitored. For this reason the average deviation when using performance counters to account the energy usage of modern CPUs is higher than it is using the TSC to account older CPUs.

benchmark	measured	TSC	weight	calculated	deviation
automotive	203.848 J	8740174215		203.847 J	-0.000573 %
consumer	101.486 J	4378626555		102.123 J	0.626933 %
network	53.540 J	2284627950	$23.323 * 10^{-9}$	53.284 J	-0.477379 %
office	180.376 J	7786109460		181.595 J	0.676043 %
security	112.324 J	4735399315		110.444 J	-1.674122 %
average absolute error					0.612803 %

Table 5.1: Measured energy, calculated energy ($TSC * weight$) and deviation

The detailed results of the calculations are shown in Table 5.1. These values were confirmed running the same test again, and more important, running four other tests scenarios:

- Image processing
  The program *convert* was run successively three times while one more *convert* was running in the background. A JPEG-image of 2.2 MB was processed with the following options: *-resize 400x400 -charcoal 30 -mosaic -flatten*

- Robomon
  Robomon is the sample application developed by members of the TU Munich to demonstrate the functions of Robertino. It has a GUI which allows to display the values of the distance sensors graphically. Additionally it is possible to control the motors and watch the images taken by the camera. This scenario consists of opening all windows (sensors, camera, control panel) and then let Robertino drive for some time.

- 'Real world'
  This scenario consists of logging into Robertino using *ssh*, then doing some normal work, like *ls* and *cat*, after that for some files the md5-sums are calculated using *md5sum*. Thereafter the session is closed and after a new ssh-login a file is edited using *vim* and then the session is closed again.

- Compiling a kernel
  Finally a kernel was compiled using *make -j3 all*[1].

Table 5.2 shows the results. The deviation[2] is under 2 % which is a result that confirmed the used weight.

scenario	measured	TSC	weight	calculated	deviation
Image processing	6147.448 J	266684428220	$23.323 * 10^{-9}$	6219.443 J	-1.171 %
Robomon	429.732 J	4784311063		426.097 J	0.846 %
'Real world'	1906.742 J	7435320646		209.959 J	0.383 %
Compiling a kernel	6040.884 J	254046770868		520.038 J	0.169 %

Table 5.2: Results of the scenarios to confirm the weight for calculating the CPU energy

5.1.2 Hard disk

The energy consumption of the disk was measured for the various power management modes and while reading and writing different amounts of data. The results of the measurement are shown in Table 5.3 and Table 5.4.

Since it is not known how much data is transmitted because accounting is done at a level where only one block (512 byte) at a time is transferred, an average was calculated. For *writing* the energy consumption per byte is set to *559 μJ/B* (averaged out of transmitting 1, 10 and 100 blocks). For *reading* the average is *558 μJ/B*.

During the final measurements it was figured out that using such a simple method is not sufficient because obviously it is important to know the exact amount of data transmitted to account the correct amount of energy. It may be an improvement to account only once a second and account blocks, not bytes. Relocating the accounting to a position where the amount of data that has to be transferred is known is another alternative. However, this poses problems when accounting since at this level it is not known whether the data is really transferred to or from disk. It is not read if it is already in the memory. No writing takes place if, for example the file the data should be saved to was deleted, the data was changed again which renders the current data useless.

The other possibility is to use a similar strategy as it is proposed below for WLAN.

[1]*-j3* means that three make-processes are running concurrently. This is normally used for multiprocessor machines on which the kernel was normally compiled.

[2]The deviation is calculated using $\frac{measured-calculated}{measured} * 100$

[3]The active consumption without idle consumption is calculated as follows:
$consumption_{active} = consumption_{total} - \frac{consumption_{idle}}{1\,s} * duration$

		State change to			
	Energy	Energy consumption		Duration	
	consumption	→ standby	→ idle	→ standby	→ idle
idle	0.7006 J	4.487 J		3.59 s	
standby	0.177 J		3.6222 J		1.58 s

Table 5.3: Energy consumption during idle periods and for state changes

			Energy consumption		
	Blocks (512 b)		Meanwhile	per Byte	
action	transferred	Duration	total	total	diff. to idle[3]
idle		100 ms	0.07006 J		
write	512	572.87 ms	1.17241 J	2289.8633 μJ	1505.9616 μJ
write	5120	585.08 ms	1.20517 J	235.3847 μJ	155.3244 μJ
write	51200	577.47 ms	1.19487 J	23.3367 μJ	15.4353 μJ
write	512000	631.50 ms	1.34264 J	2.6223 μJ	1.7582 μJ
read	512	560.07 ms	1.16115 J	2267.8711 μJ	1501.4926 μJ
read	5120	571.47 ms	1.19611 J	233.6152 μJ	155.4167 μJ
read	51200	566.07 ms	1.19641 J	23.3674 μJ	15.6215 μJ
read	512000	592.08 ms	1.28785 J	2.5153 μJ	1.7051 μJ

Table 5.4: Energy consumption during sending and receiving

5.1.3 WLAN

For WLAN the idle energy consumption in CAM, PSM and switching from one mode to the other was measured. In PSM a beacon period of 100 ms is used. Additionally in both modes sending and receiving different amounts of data was quantified. The results are presented in Table 5.5, Table 5.6. The values shown in the last two columns represent the amount of energy accounted for each transmitted byte.

During the final tests of the infrastructure it turned out that it is not enough to account the usage per byte. A better solution may be accounting a fixed amount of energy at the beginning of a transfer and only a small amount for each byte. The problem thereby is that it has to be determined when a transfer is started, and when it is completed. Accounting a fixed amount of energy at the beginning of a transfer can be represented as a state change. When a transfer is started the resource switches[4] into a special activity mode. The transition energy for switching is the fixed amount of energy that has to be accounted. Switching back to the cur-

[4]In this case *switching* causes no mode change of the device. It is only required to assure that a fixed amount of energy (transition energy) is accounted.

rent state, this means ending a transfer, is done when no data is transmitted for a given period of time.

	Energy consumption	State change to Energy consumption → PSM	→ CAM	Duration → PSM	→ CAM
CAM	1.064 J	0.139 J		153 ms	
PSM (100ms)	0.295 J		0.161 J		173 ms

Table 5.5: Energy consumption during idle periods and for state changes

action	Bytes transferred	Duration	Energy consumption Meanwhile total	per Byte diff. to idle CAM	PSM
idle CAM		10 ms	10.64 mJ		
idle PSM		10 ms	2.95 mJ		
send	10000	32.8 ms	45.95 mJ	1105 nJ	3626 nJ
send	100000	291 ms	422.30 mJ	1127 nJ	3364 nJ
send	1000000	2570 ms	3839.77 mJ	1105 nJ	3081 nJ
receive	10000	24.6 ms	32.53 mJ	635 nJ	2526 nJ
receive	100000	168 ms	231.77 mJ	530 nJ	1822 nJ
receive	1000000	1540 ms	2112.19 mJ	474 nJ	1657 nJ

Table 5.6: Energy consumption during sending and receiving

5.1.4 Camera

It was not possible to measure the energy usage of the camera separately. Measurements had to be done while also sampling the CPU. For this reason the energy had to be adjusted by removing the energy needed for the CPU. This is possible without major problems since the CPU energy was accounted by the resource container system.

Measuring the camera was done by taking images with a given framerate (30 FPS or 3 FPS) and image-size (320x240), using a simple program that only copies the image from the firewire kernel-modules and then deletes it. The energy usage was captured for each combination of framerate and image size for 30 seconds. Evaluating the data showed that the image size has no impact on the energy usage, while the framerate has.

The data shown in Table 5.7 represents the average usage of a test, where for each framerate images were taken for 500 seconds. The idle-energy which has to be subtracted from the acquired data was measured, and verified using the CPU energy accounting.

framerate	measured	idle	diff. to idle	diff. to idle per picture
idle	10.44307 J	10.44307 J	0.0 J	0 μJ/pic
3 FPS	10.45651 J		0.01343 J	4479 μJ/pic
7 FPS	10.47091 J		0.02784 J	3978 μJ/pic
15 FPS	10.50790 J		0.06483 J	4322 μJ/pic
30 FPS	10.57708 J		0.13400 J	4467 μJ/pic

Table 5.7: Results of the measurements of the camera

5.1.5 Motors

Since the three motors are similar, only one was measured. The energy consumption at the different speeds is shown in Figure 5.1 and in Table 5.8.

By the time the motor is running faster than 300 ticks/second ($\approx 2,89\frac{cm}{sec}$) the energy consumption decreases.

Apparently limiting the motors will not always yield to energy savings. This is not given if the motor speed is reduced from v=600 to v=300 which represents a speed reduction of 50 %. However, the energy usage will increase from 2.78J to 3.17J (approx. 14 %).

Speed	Energy consumption	
	total	without idle
0	0.7945 J	0 J
100	1.1616 J	0.3670 J
200	1.4297 J	0.6351 J
300	1.5749 J	0.7803 J
400	1.5007 J	0.7062 J
500	1.3406 J	0.5460 J
600	1.1796 J	0.3847 J
700	1.2012 J	0.4067 J

Table 5.8: Energy consumption of a motor

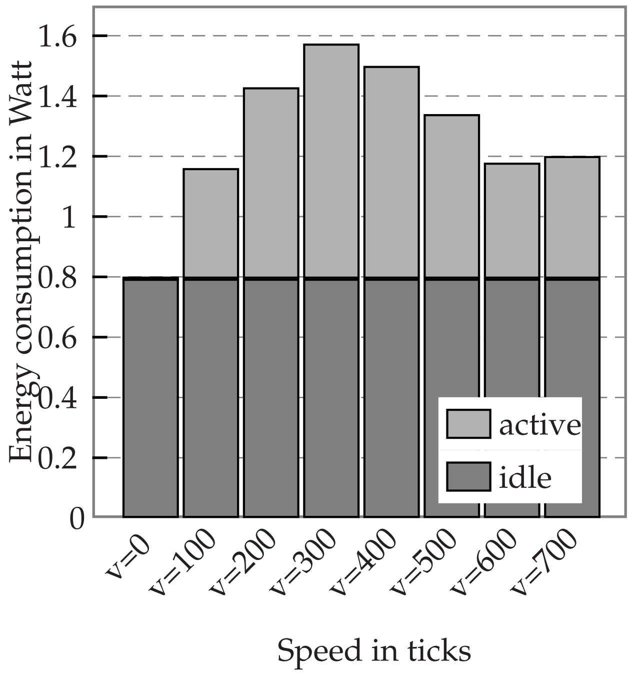

Figure 5.1: Energy usage at the different speeds

5.1.6 Battery

The battery is measured during operation using a voltage divider to reduce the voltage from max. 13 V (during charging, normally max. 12 V) to max. 5 V which is the highest value that the ADCs support and sampling the voltage with one of the ADCs. An ADC has a resolution of 10bit which allows a range from 0 to 1024. Some sample conversions are shown in Table 5.9. The ADC values may be converted to voltages using $Voltage = ADC * 0.0148$.

ADC-	Voltage		
value	measured	calculated	deviation
882	13.06 V	13.05 V	-6 mV
810	12.00 V	11.99 V	-12 mV
708	10.50 V	10.49 V	-21 mV
674	10.00 V	9.98 V	-25 mV

Table 5.9: Sample values for ADC->Voltage conversion.

5.2 Accounting and Limiting

To evaluate the accuracy of accounting and limiting a couple of test were conducted. Below the different test scenarios are introduced, followed by the results and, if needed, proposals for improvements.

The hard disk was excluded from the measurements and an average consumption of *0.72 J* is assumed. This value is a bit higher than the energy consumption during idle periods since the measurement data is written to disk periodically and the power management of the disk is disabled. Excluding it was necessary because accounting the energy consumption failed. This is because of an disadvantageous implementation of the energy accounting. In contrary thereto accounting the usage is working properly.

In Table 5.10 the results of measuring the system for *289.11 seconds* while it was idle are shown. During this time the average battery voltage was *11.68 V* . For the disk an energy consumption of *208.16 J* is assumed. The deviation of *-536.40 J* $(-10,4\,\%)$ can be reduced by tuning the energy weights for accounting. This is possible because until now for example the dissipation loss of the power supply has not been considered. As mentioned before (subsection 3.1.7) its maximum efficiency is 95 %. Since the energy consumed by the system is measured directly at the batteries the dissipation loss has to be regarded. If 5 % loss is considered the deviation is only -278.9 J. Due to the fact that the efficiency changes with the load and 95 % is an optimistic value a higher dissipation loss can be assumed. Therefore the accounting is correct, but missing the loss. Accounting this amount of energy is done by adapting the weight for the resource global overhead.

	Idle
CPU	2659.34 J
WLAN	319.36 J
Camera	1200.58 J
Motor	238.32 J
Accounted	4417.59 J
Disk	208.16 J
Total	4625.75 J
Measured	5162.15 J
Deviation	-536.40 J

Table 5.10: Comparison between accounted and actual energy consumption

The next two tests were made to verify the limiting. Therefore the motors were measured while running with a speed of 300 ticks/second and then they are measured again with a set usage limit of 50 % which means they are only allowed to run at a speed of 150 ticks/second. Both test were executed for about *300 s* with an average voltage of *11.55 V* . The results are presented in Table 5.11. The deviation between measured (*155.11 J*) and accounted (*154.49 J*) is only *0.62 J* which is about 2 mJ/ s. This shows that throttling the motor usage yields to the desired results.

	v=300	v=150	difference
Motor	683.88 J	532.77 J	151.01 J
Measured	5923.87 J	5768.76 J	155.11 J

Table 5.11: Comparison between unlimited (v=300) and limited (v=150) energy consumption

The validation of the accounting and limiting of the camera usage and energy consumption was done using a program that displays the taken images in an window of the X Window System. The framerate of the application was set to 30 FPS which was throttled to 3 FPS in the second test. Since the tests were performed using an ssh connection the data has to be transferred to the remote computer and displayed there (which is automatically done by ssh). This resulted in high CPU load. Another test limited the CPU load by setting the energy limit of the CPU to *1.5 J* while the camera was taking images at a rate of 30 FPS. As the other programs these were executed for *300 s* . The battery voltage was *11.60 V* .

				Differences	
	30 FPS	3 FPS	CPU-limit	30 FPS ⇔3 FPS	30 FPS ⇔CPU-limit
CPU	3499.87 J	2888.19 J	3070.19 J	611.68 J	429.68 J
Camera	1200.96 J	1170.26 J	1201.57 J	30.7 J	-0.61 J
Measured	6438.85 J	5673.44 J	5964.02 J	765.41 J	473.98 J

Table 5.12: Comparison between unlimited (30 FPS) and limited (3 FPS) camera and limited CPU energy consumption (limited to 1.5 J)

Table 5.12 shows that limiting the energy consumption of the CPU to 1.5 J is successful. The consumptions is in fact reduced to ≈ 1.43 W $(= \frac{429.68 J}{300 s})$. The savings of limiting the camera should be about 36.2 W within the 300 seconds (0.12 J). These values are calculated using the beforehand (subsection 5.1.4) determined energy consumption of the camera. The accounted value is below this theoretical value. This is because it is not guaranteed that really 30 pictures per second are taken one missing picture every six seconds[5] has to be expected.

The evaluation of measurements shows, that it is possible to account and limit the energy consumption of a system with the developed infrastructure. Although improvements are needed for accounting the hard disk and WLAN. Additionally the weights have to be fine tuned to enable a more accurate accounting. No changes are needed for accounting the motors, the camera and the CPU.

It is also shown that accounting the energy consumption during idle periods is possible and accurate.

[5] 6 J are 45 pictures; the program was executed for 300 s; $\frac{300}{45} \approx 6.667$

Chapter 6

Future Work

Additionally to the beforehand mentioned improvements the following enhancements should be investigated.

6.1 Multiple devices per resource

Having resources for all devices poses the problem, what to do if there is more than one device per resource. Today it is not uncommon to have two disks, or, for robots, it is possible to have two cameras, for example for stereoscopic vision, or one holonomic camera for obstacle-detection and one that is versatile for tracking objects. In most cases the devices will not be identical (same disk size, same buffer, same resolution for example). So the needed energy is not identical. For that reason the energy characteristics for each device have to be stored and used adequately. Multiple devices per resource yield to the problem where to store the device characteristics. Should they be stored within the resource container structure, like now, or should they be put into the driver which calculates the energy usage and calls the resource container structure for accounting? Another question is, how to identify the different devices during runtime and how to assign the appropriate energy values. This may, for example, be done by using device pointer which are also used by the kernel internally, and therefore being able to assign each usage to the right device. Using device pointers is rudimentarily implemented in this work, because they were needed to enable setting the states for the device. Adapting the resource containers to be able to handle more devices per resource should not be of a problem for this reason.

6.2 Uniform configuration interface

Currently the energy profiles are implemented into the kernel. If a device is exchanged by another the kernel has to be modified, compiled and installed. This is a lot of work which can be avoided by providing an interface that allows to set the energy profiles dynamically.

A similar interface still exists for the configuration of the CPU-weights. A drawback of the dynamic configurability is, that the profiles have to be saved and restored when rebooting the computer. As accounting and limiting is currently set up very early this may be a problem, since no access to disk or other storage is possible. It is possible to set up accounting later but then the energy consumed during start-up is not accounted.

Another problem is that there is a different number of energy values needed for each resource.

6.3 Observe length of accesses and state changes

In this work activity and transition energy are accounted at once. This means, whenever the state is changed or an activity occurs a defined amount of energy is added to a resource container. In fact the activity/transition last for some time, for this reason it should be investigated if a better limiting can be achieved by not accounting the complete amount of energy, but to split it up and account it over the same period of time the activity/transition lasts.

Chapter 7

Conclusion

This work introduces an infrastructure for accounting and limiting the total energy consumption of for example a robot.

It is shown that the total energy consumption can be accounted to the resources consuming it and therewith to the processes.

Thereby not only the energy usage during activity but also the consumption during idle periods has to be taken into consideration. Different methods for accounting the idle energy are discussed.

Limiting on the one hand is done by restricting the amount of energy available for the different processes and resources. On the other hand advantage is taken of the different power management modes which allow to not only reduce the energy consumption in idle periods but also during activity.

When using the different power management modes it is crucial to account the energy consumed during mode changes. Since accounting this amount of energy differs from accounting energy consumed during activity or idle periods several strategies are discussed that allot this amount of energy to processes in a fair way.

To allow fine grained limiting it is necessary to provide feedback of the effects of set limits. This is done by not only accounting the energy consumed but by also accounting the device usage.

Such a feedback can be used to adjust limits and to detect interdependencies. These occur, when limiting one resource also influences the energy consumption of another resource.

Using this infrastructure allows it to develop policies which are able to control the system in such way that given tasks and goals are fulfilled although this would not be possible under normal circumstances.

As proof of concept, the infrastructure has been implemented for Robertino, an autonomous mobile robot. This robot is controlled by a computer using the Linux operating system.

List of Figures

List of Tables

Bibliography

[ANF03] Manish Anand, Edmumd B. Nightingale, and Jason Flinn. Self-Tuning Wireless Network Power Management. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking (MOBICOM '03)*, September 2003.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation OSDI'99*, pages 45–58, February 1999.

[Bos] Bosch. Controller Area Networks.

[Com99] IEEE Computer Society LAN MAN Standards Committee. *IEEE 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications*, August 1999.

[CV01] Alois Knoll Christian Verbeek, Franz Murr. Das Robertino-Robotersystem: Ein autonomer mobiler Roboter für Forschung und Lehre. Technical report, Technische Universität München - Institut für Informatik, 2004-10-01.

[FD98] D. Friel and R. Dunstan. Smart Battery Data Specification, December 1998.

[Gut] Matthew Guthaus. MiBench version 1.

[HLG05] Thomas Hirth, Nestor Lucas, and Javier Gutierrez. Akkuproblematik am Robertino, 2005.

[Höl05] Kurt Höller. Energieversorgung und -optimierung von Roboterplattformen, January 2005.

[Kel03] Simon Kellner. Event-driven temperature control in operating systems, April 2003.

[NM01] Rolf Neugebauer and Derek McAuley. Energy Is Just Another Resource: Energy Accounting and Energy Pricing in the Nemesis OS. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 67, Washington, DC, USA, 2001. IEEE Computer Society.

[TM03] Tri-M. *High Efficiency PC/104 Vehicle Power Supply*. Tri-M, rev. 05/03 edition, 2003.

[url] OpenRobertino.org.

[Wai03] Martin Waitz. Accounting and Control of Power Consumption in Energy-Aware Operating Systems, 01 2003.

[Wis04] Dipl.-Ing. Thomas Wisspeintner. AISVision - A Cost-effective Omnidirectional Camera System, 2004.

[ZELV02] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: Unifying Policies for Resource Management, 2002.

[ZFE$^+$02] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

Entwicklung einer Infrastruktur für Energiesparverfahren

Batteriebetriebenen Geräten fallen heutzutage immer komplexere Aufgaben zu, deren Lösung viel Energie benötigt. Da der Mehrverbrauch trotz besserer Batterien nicht erfüllt werden kann wurde neue Hardware entwickelt, die verschiedene Arbeitsmodi unterstützt in denen unterschiedlich viel Energie verbraucht wird. Allerdings verwalten die einzelnen Geräte diese Zustände selbst.

Mit einer übergeordneten Instanz, die das Energiemanagement des kompletten Systems kennt und kontrollieren kann ist es möglich zusätzliche Energie einzusparen. Gezieltes Limitieren des Energieverbrauchs einzelner Geräte ermöglicht es, Aufgaben zu erledigen, deren Erfüllung sonst aufgrund mangelnder Energie gescheitert wäre. Mit dem Wissen über den Energieverbrauch können Aufgaben priorisiert werden, während andere verzögert werden.

Ein Beispiel für die Anpassung von Prioritäten ist der Transportroboter der erkennen kann, dass seine Energie nicht mehr ausreicht um weitere Güter zu transportieren und deshalb umgehend zur Ladestation zurückkehrt. Sind die Batterien wieder voll aufgeladen, kann er mit seiner normalen Arbeit fortfahren.

Um allerdings zu Wissen, wie lange die Energiereserven noch reichen, und wie viel Energie gespart werden kann ist es nötig den Energieverbrauch des Systems exakt zu kennen. Dies kann dadurch erreicht werden, dass der Verbrauch jeder Komponente separat gemessen und von einer übergeordneten Instanz verwaltet wird.

Zusätzlich zum Erfassen muss auch eine gezielte Limitierung des Energieverbrauchs möglich sein. Mit dieser können Geräte und dadurch die Prozesse, die diese Komponenten verwenden, verzögert werden. Durch das Verzögern von Prozessen können andere Prozesse ihre Aufgaben unter Umständen schneller erledigen, auf jeden Fall aber steht ihnen mehr Energie zur Verfügung. Damit ist es realisierbar, dass Aufgaben auf jeden Fall erledigt werden und bestimmte Ziele sicher erreicht werden können, während andere weniger wichtige Aufgaben unerledigt bleiben können.

Um die Kontrolle über den Energieverbrauch zu erlangen wird in dieser Arbeit eine Infrastruktur entwickelt, mit der genaues Erfassen und feingranulare Steuerung des Energieverbrauchs möglich ist.

Dazu wird der Energieverbrauch aller Komponenten erfasst und an sogenannte "Policies" weitergeleitet. Diese Policies können aus diesen Daten die Limitierungen für einzelne Komponenten berechnen, die dann mithilfe der Infrastruktur gesetzt werden.

Es reicht nicht aus, den Energieverbrauch im aktiven Betrieb zu erfassen, da ein nicht unerheblicher Teil der Energie im Leerlauf verbraucht wird. Zur Erfassung dieses Verbrauchs werden in dieser Arbeit verschiedene Ansätze vorgestellt und diskutiert.

Ein weiterer Posten des Energieverbrauchs sind die Zustandswechsel von einem Arbeitsmodus zu einem anderen. Auch für die Erfassung dieser Energie werden verschiedene Verfahren vorgestellt.

Zum Limitieren wird nicht nur die Energiemenge, die einem Gerät zur Verfügung steht beschränkt, sondern es werden auch die verschiedenen Arbeitsmodi der Komponenten mit einbezogen. Nur mithilfe der diversen Arbeitsmodi ist es möglich den Energieverbrauch im Leerlauf zu beeinflussen.

Die beschriebene Infrastruktur wurde für Robertino implementiert. Robertino ist ein autonomer mobiler Roboter, der sich mithilfe dreier Motoren fortbewegt. Zusätzlich verfügt er über mehrere Abstandssensoren und eine omnidirektionale Kamera. Der Steuerrechner ist ein herkömmlicher Industrie-PC, der mit einem Pentium III M, 128MB Speicher, WLAN, Firewire und einer 20 GB Festplatte bestückt ist.

In dieser Arbeit wird, anhand der Implementierung für Robertino, gezeigt, dass die Infrastruktur das Erfassen des kompletten Energieverbrauchs des Roboters und die gezielte Steuerung des Verbrauchs einzelner Komponenten ermöglicht.