Erfassung und Regelung der Leistungsaufnahme von Sensorknoten

Diplomarbeit im Fach Informatik

vorgelegt von
Simon Kellner
geboren am 27. Oktober 1978 in Roding

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. August 2005
Abgabedatum:	31. Januar 2006

Erklärung

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, 31. Januar 2006

Energy Accounting and Control for Sensor Nodes

Diploma Thesis

by

Simon Kellner

born October 27th, 1978, in Roding

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: August 1st, 2005
Submission: January 31st, 2006

Abstract

In recent years, the use of sensor networks has spread to a variety of fields, including biology and civil engineering. The sensor nodes forming these networks are expected to provide several years worth of data with the battery pack they are given at deployment. Therefore, one of the most important aspects of both software and hardware design for these nodes is the energy consumption.

Tuning sensor node applications to a lifetime this long is an important part of application development. As most of the sensor node applications to date are deterministic, their behavior can be simulated and their power consumption can thus be calculated before deployment.

In non-deterministic scenarios this a-priori evaluation might not be sufficient. Such scenarios may include the frequent adaption of the sensor network to environmental conditions, or the use of non-deterministic applications in which sensor readings can affect the application's actions.

Therefore, this work investigates how techniques of adaptive power management, which are already available for ordinary computers, can be implemented on sensor nodes. There are two prerequisites for adaptive power management: statistics on power consumption and methods of changing application behavior. On sensor nodes, information on both energy usage and remaining energy is required for useful statistics.

In this work, concepts and methods for energy accounting, battery lifetime estimation and application reconfiguration at runtime are presented. Evaluation results of their implied overhead are given where reasonable.

Contents

Chapter 1

Introduction

In recent years, technological advances led to a new form of programmable devices named sensor nodes or motes. Since then, the use of sensor nodes has spread to a wide range of fields. Today they form a useful tool for biologists as well as civil engineers and are usually used for long-term monitoring of a large area, examples of which include woods, buildings and the major part of an island.

Sensor nodes are small, battery-powered devices with two basic functions, reading sensor data and communication. These two functions are necessary to form a sensor network. Sources of sensor data are normally environmental data like temperature and brightness. The communication channels are mainly used for aggregating the sampled data to one central point where an ordinary computer can be used to store and later analyze the data. Sensor networks can pose interesting routing problems, particularly in dynamic networks where the location of the nodes is not fixed, e.g. for monitoring an animal herd. The area of routing protocols for sensor nodes is not a part of this thesis because it has already been under extensive research. Today, developers of sensor node applications can choose the most suitable one from a multitude of routing algorithms for their application.

One of the most confining requirements usually is the lifetime, i.e. the desired period of time in which the nodes can perform their duty. In many projects sensor nodes have to last as long as several years. As the number of nodes in a network can reach a few thousand units, recharging or replacing their power supply would be a time and money-consuming task and in some situations even impossible to perform, e.g. during the long-term monitoring of animals which must not be disturbed. So the nodes are expected to provide data for several years with the batteries they got at the time of deployment.

This constraint permeates all design aspects of a node. Hardware on a sensor node can either be completely turned off or provides fine-grained control over its power consumption, preferably both. The latter feature allows the node's application to select the mode of operation with the lowest

power consumption possible, in every situation.

When developing software for sensor nodes, time and energy-consuming algorithms have to be avoided wherever possible. This leads to software design different from common embedded systems with focus on reliability. Sensor node applications must also be reliable, of course, but a few failing sensor nodes have to be expected. This loss can usually be compensated by increasing the number of nodes.

To aid developers in designing their application to last the specified lifetime, various analysis tools based on hardware simulation have been developed. They can be used for detecting power-consuming hot-spots and for tuning the application towards the desired energy consumption rate. This is an established part of software design for sensor nodes and works well for the deterministic applications used to date.

As the utilization of sensor nodes is relatively new, it is not uncommon to change the node configuration after a few weeks of data sampling. These changes could include anything from simple parameter adjustment up to a major redesign, e.g. if it was decided to completely stop using one type of sensor while increasing the sampling rate of the other sensors. Depending on the severity of the change, the repeated process of simulation, tests and parameter tuning for the modified software can thus prove to be cumbersome.

Also, more complex applications showing a more probabilistic behavior can be envisioned, e.g. with environmental parameters that influence the control flow of the application. An analysis of this kind of software before deployment has to work with average or worst case scenarios. Worst case analysis might lead to an undesirably low rate of data collection while average case analysis, in addition to being difficult to calculate correctly, could be too optimistic.

In the world of fully-fledged PCs, the concept of adaptive power management has already started to gain ground. Here, the PC user or server administrator can control the power consumption of the computer to some extent. For adaptive power management to work, a system must have information on energy usage and supply, combined with a way of exercising control over its power consumption.

Consequently, this work investigates how the two basic parts of adaptive power management, energy accounting and control, can be transferred to the field of sensor nodes. For the control of energy consumption various methods of reconfiguration and their overhead on the application are presented in chapter 5. This includes widely used methods as well as some rarely encountered ones. Energy accounting on sensor nodes not only has to consider the used, but also the remaining energy in form of battery capacity. Methods for both tasks are examined in chapter 4, after a description of our chosen target platform in chapter 3. A brief description of possible future work in chapter 6 and a summary in chapter 7 conclude this thesis, while we now turn to already existing related work.

Chapter 2

Related Work

2.1 Energy Accounting

As offline power consumption analysis of sensor node applications is an established part of application development, various tools already exist for this purpose.

A very accurate tool for estimating the energy consumption of a sensor node is AEON [10]. It uses an emulation of the sensor node hardware in combination with a detailed energy model.

PowerTOSSIM [13] on the other hand sacrifices some accuracy for scalability. Applications are compiled for the x86 platform. Parts of the application code are changed to facilitate simulation of a large number of nodes in one process, typically thousands.

These methods in their current form are useful only for analysis before deployment. The energy models, however, can also be utilized in runtime energy accounting as proposed in this work.

2.2 Reconfiguration

For software reconfiguration on sensor nodes, previous research mostly focused on the flexible method of uploading new applications.

Deluge [7] and Xnp [9] are two implementations for TinyOS which focus on uploading complete application binaries. In addition, Deluge features a switching operation between applications which is also discussed in this thesis.

Other methods try to keep the size of the upload small. SOS [5] is an operating system which introduces indirections on inter-module calls. A kernel which is exempted from update operations provides functions for loading and unloading modules. Here, applications can be updated at a granularity level of modules. Contiki [3] has similar properties.

Even smaller updates are possible with Maté, a virtual machine for sensor nodes. Primitive instructions on this machine include arithmetic operators as well as instructions for reading sensors

and communication. Applications are comprised of capsules, each capsule containing up to 24 instructions. This is sufficient for a simple function. Application updates operate on the granularity level of capsules. As each instruction of this virtual machine has to be interpreted, low-level instructions like `xor` cost more energy and high-level instructions like `sendr` save energy when update and runtime costs are compared.

Chapter 3

Experimental Setup

To implement and evaluate various methods presented in this thesis, the BtNode platform [4] from the ETH Zürich was selected for its Bluetooth capability. TinyOS [6] was the operating system of choice because of its widespread use and its interesting implementation language, NesC. What follows is a description of each component's characteristics.

3.1 The BtNode

A BtNode (rev 3.22) includes an Atmel ATMega128L microprocessor with 128 KB of flash memory, 4 KB of EEPROM and 4 KB of RAM clocked at 7.3728 MHz. Throughout this work it is referred to as ATMega128, for there are no differences between these two microprocessors except the supported clock rate, and ATMega128 is the name used in various build tools. This microcontroller provides enough room even for complex applications. For power management, the ATMega128 features six easily selectable sleep modes.

As the 4 KB of internal RAM are insufficient for storing sampled data, 256 KB of external RAM are provided. Every byte of external RAM is usable through a combination of manual control over some pins and masking out certain address lines.

The BtNode can use two methods of communication which differ in power consumption and protocol layers.

The ChipCon CC1000 is a basic digital radio receiver and transmitter with configurable frequency (433/868/915 MHz). It provides fine-grained control of its parts like the frequency synthesizer and mixer as well as the transmit power, so it is very suitable for power management. The communication protocol is very simple, as the chip transmits exactly the series of bits it receives from the microcontroller. Apart from some preamble suggested by the manual, no particular

communication format is enforced. Error detection or correction has to be implemented by the application.

The other method of communication with the BtNode is a Zeevo Bluetooth controller with a range of 10 meters. In several aspects this chip is quite the opposite of the CC1000. It is probably the most complex chip on the BtNode. For power management it offers the modes specified in the Bluetooth standard (connected, hold and sniff mode as well as the parked state), which are much more course-grained than the controls of the CC1000. It takes a bit of effort to create a Bluetooth ACL connection to another node as several commands have to be sent. In exchange, the communication is connection-oriented and error free. Another advantage of this communication method is the possibility to connect to a PC equipped with a Bluetooth dongle. While it takes only a few commands on the PC (like creating a TCP/IP connection), the application on the BtNode has to implement at least the L2CAP layer on top of the ACL connection, which seriously affects both code size and energy consumption.

The microcontroller has the possibility to completely turn off power for the Bluetooth and the CC1000 separately. This feature is beneficial as these chips consume a considerable amount of energy, even when in idle mode, the Bluetooth chip in particular.

The BtNode itself does not have sensors on board. Instead, a connector can be used to plug in a sensor board. A description of a suitable sensor board follows in the next section.

The BtNode alone allows to sample different data sources. The batteries are connected to the microprocessor's A/D converter, allowing an application to read out the current voltage. Both the Bluetooth and the CC1000 chip can return RSSI (Receive Signal Strength Indicator) values which could be used for estimating distances between nodes.

3.2 The Sensor Board

The TecO *ssimp* sensor board is designed to be plugged into a node, and therefore contains nothing but sensors. It is equipped with acceleration, brightness and temperature sensors as well as a microphone.

Sampling acoustic data from the microphone is not complicated. The necessary steps consist of setting up the A/D converter and start reading data from it. As the 260 KB of RAM are quickly filled with this kind of data, it is advisable to compress it or to store only analysis results of it.

The most prominent chips on the sensor board are the two acceleration sensors, each covering two axes. One of these sensors is mounted at 90° towards the board to cover the remaining axis. Together they can detect acceleration in every direction up to 10 g, which is communicated to the microprocessor via the length of a duty cycle, a value of 50% meaning 0 g. The duty cycle is readily sampled using the input capture feature of the microprocessor. In theory these sensors can be used

to compute the movement and location of a sensor node relative to others. The acceleration sensors are not used for this work because the chosen interface method does not provide the number of connections needed to read acceleration values for all three axes.

A very useful sensor for the laboratory environment is the TSL2550 sensor for brightness. Since most brightness sensors are sensitive to infrared light but are supposed to report brightness in the visible spectrum, this sensor actually consists of two brightness sensors, one for infrared light only and one for infrared and visible light. So the value for visible brightness is the difference between readings from both sensors. The sensor can be set up and read via the I^2C-bus on the microcontroller. What makes this sensor useful for a laboratory environment is the ease of changing the sensor input with a flash light, for example.

The most interesting sensor on this board for this work, however, is the MCP9800 temperature sensor. It is also accessed via the I^2C-bus, but unlike the TSL2550 it requires longer and more complex commands. It uses four registers, one control register, two for parameters and one for the current temperature. The existence of a control register already suggests that this sensor has multiple modes of operation, including one feature not provided by the other sensors on this board. Unlike the other sensors which require polling or use periodic interrupts to communicate with the microprocessor, the MCP9800 can send aperiodic interrupts when a configurable temperature limit is reached. This is the kind of aperiodic event which is hard to estimate before deployment and thus justifies the overhead of runtime energy accounting and control.

3.3 Software Environment

3.3.1 TinyOS

In this thesis nearly all of the software for the BtNode was written for TinyOS, particularly for the TinyBt contribution which ports TinyOS to the BtNode. TinyOS is a large collection of software modules for hardware commonly found in sensor nodes such as the mica nodes [8]. The software modules provide simple hardware abstraction, e.g. for LEDs, hardware virtualization of timers, and also implementation of higher-level algorithms like multihop-routing.

As hardware like the ATmega128 and CC1000 can also be found on other sensor nodes, parts of TinyBt are in fact adaptations of these modules for use on the BtNode. It also comes with a module for the most prominent feature of the BtNode, its Bluetooth chip. This module, however, only provides some communication functions between microprocessor and Bluetooth controller. Code to connect to another sensor node or a PC had to be implemented for this work.

TinyOS is an event-based operating system, a property which is not common among resource constrained embedded platforms. It is ideal for sensor nodes, however, because it allows the mi-

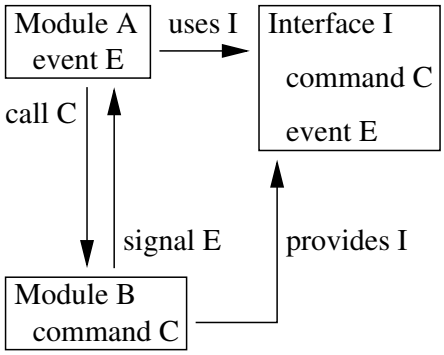

Figure 3.1: Relationship of interfaces and modules in NesC

crocontroller to frequently enter one of its sleep modes, thus reducing power consumption. This design is supported by certain characteristics of the chosen implementation language which will be discussed in the following sections.

3.3.2 NesC

The language of TinyOS and TinyBt is NesC, an extension of C. It does not provide new primitives like new data types or operators, but new flags for variables and functions in addition to the higher-level constructs called `interface`, `module` and `configuration`.

Interfaces in NesC are used similarly to other languages as named collections of function prototypes. Modules, on the other hand, are named collections of function implementations. Each module should provide one or more interfaces (`provides`) and is free to use (`uses`) any number of interfaces.

Each prototype function of an interface has to be flagged with either a `command` or an `event` keyword, thereby indicating a direction of the function call. While a command function has to be implemented by the used module, an event function must be implemented by the using module.

The relationship of interfaces and modules is shown in Figure 3.1. In this example, module A uses the interface I while module B provides I. Module B has to implement all the commands specified in I which can then be called from all modules using I. Conversely, module A must implement all events listed in I, so that the used module B can send events to the using module by `signal`ling an event E.

Note that an event can occur at any time. It is quite usual for modules to successfully return from a command and later signal a failure to complete this command. For example, a `send` command in a communication module could return successfully if the message was appended to its output buffer, but a timeout could cause the signaling of a failure event.

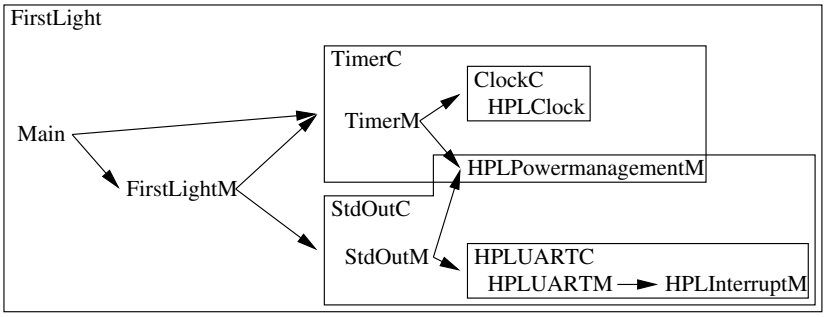

Figure 3.2: Wiring of modules for application `FirstLight`

As modules can only use interfaces, one purpose of a configuration is to specify the actual connections between modules. This operation is called *wiring*. It obviously only works if the interfaces match. An application is a configuration which wires the module `Main` to other modules.

Another purpose of configurations is to hide complexity by connecting a set of modules and providing one or more of their interfaces. Such configurations can be used in other configuration instead of modules.

In short, a configuration is a list of components, i.e. modules or other configurations, and a description of how these components are wired together. An example can be seen in Figure 3.2 which shows part of the wiring for a simple application named `FirstLight`.

Each box represents a configuration with its name in the upper left corner. All other names are that of modules. The names mostly follow the TinyOS naming convention in which module names end with an “M” and configuration names with a “C”. All names starting with “HPL” indicate platform dependent code, usually in a module.

The `HPLPowerManagementM` module and the `TimerC` configuration are used by more than one module, but code and variables of these components exist only once. This can be done with two different kinds of multiplexing.

HPLPowerManagementM is unaware of multiplexing. Calls from `TimerM` and `StdOutM` are directed to the same command functions in HPLPowerManagementM, and if any event from HPLPowerManagementM was generated, it would be sent to both of the using modules, combining the return values of both event functions.

TimerC however is fully aware of multiplexing. A configuration wiring some module to TimerC has to assign a unique numeric id to this wire. This id has to be unique for each `Timer` interface used. All commands and events of TimerM, which is responsible for providing the exported multiplexed Timer interface, have an additional id parameter, which allows to signal events selectively. This hardware virtualization allows more than one module to share one hardware timer.

Concepts like interfaces and modules clearly stem from the world of object-oriented programming languages. However, they are adapted to an embedded environment, especially with regard to simple memory management. While modules and objects have some common features such as encapsulation and interchangeability, a module (code *and* data) can exist only once. This avoids potentially complex memory management code in an environment with limited memory.

3.3.3 Entanglement

Apart from these language concepts the NesC compiler itself has an interesting feature. When compiling an application, all of the source files are read prior to processing to allow a very thorough examination of the code. This is only feasible because the code size of an embedded application is very small compared to PC applications.

One part of this examination is for concurrent access of variables. For this analysis the compiler needs a thread model, i.e. which instructions can be interrupted by which other parts of the application. The NesC thread model is clearly designed for an operating system with its synchronous and asynchronous parts.

All synchronous operations are based on *tasks*, in NesC defined as functions with neither parameters nor return value but tagged with the keyword `task`. These tasks are stored in a queue and executed sequentially, without any form of multi-tasking. The scheduler implementing this simple scheme is supplied by TinyOS, but is essentially required by the thread model of NesC. Instructions in tasks are only safe from code in other tasks, but they can be made non-interruptible by making use of the `atomic` statement.

Every asynchronous operation begins with an interrupt. In NesC, an interrupt handler can be declared by attributing a function with `interrupt` and `spontaneous`. The first flag causes the compiler to wrap the function accordingly (saving and restoring registers, `reti`) while the latter one informs the compiler that this function is called despite the absence of an explicit call in the source code and therefore must not consider this fuction to be “dead code”, i.e. code that can be removed because it is never used.

Functions directly called in this asynchronous context have to be attributed with `async`. Instructions in asynchronous functions outside `atomic` blocks can be interrupted by every other asynchronous function. The only way an asynchronous function can call a synchronous one is to `post` a task, i.e. the task’s function pointer is inserted in the task queue.

As tasks can post other tasks, too, it is possible to implement a simple form of cooperative multi-threading by prematurely returning from a task after posting the desired successor task. In an embedded environment, however, a task taking more than a few microseconds to complete is rarely used, due to the usually small amount of available memory.

Chapter 4

Energy Accounting

Today, applications for sensor nodes are based on periodic tasks, be it the normal duty cycle in which data sampling is done or the periodic setup of an ad-hoc network. This design makes it easy for the developers to tailor the application to some desired lifetime using simulation techniques in combination with an energy model of the sensor node.

For more dynamic applications this kind of static analysis before deployment is not enough. This chapter therefore focuses on methods for estimating the node's power consumption as well as its remaining energy at runtime.

4.1 Energy Accounting

4.1.1 Environment

Sensor nodes are distinctively less complex than an ordinary PC, so energy accounting methods also do not have to be as complex. For example, the power consumption of a Pentium 4 chip heavily depends on the instructions it executes and can easily vary by more than 20 W. In [1], performance counters on the Pentium 4 were (ab)used to account energy in spite of this effect. On the other hand, the power consumption of instructions on the ATMega128 is relatively constant. This can be seen in Figure 4.1 where consumption values for four different instructions are listed.

Another difference between PC and ATMega128 can be observed between their sleep modes. While the PC's power consumption during sleep mode is constant, the ATMega128 features more fine-grained control over its various clock signals during sleep. As clock signals are usually the most power consuming part of any chip, this results in six different sleep modes with widely varying consumption rates.

While it is already difficult to account the energy consumption of a CPU in an ordinary PC, accounting the energy consumption of the whole system is considerably more complex, due to the

multitude of relatively complex hardware designed to hide this complexity behind some interface. On a sensor node, most of the hardware has the opposite properties. There is few external hardware, which usually allows fine-grained control over its power-consuming features and is of simpler design. The Bluetooth chip on the BtNode, which is probably the most complex chip on this platform, is the exception to the rule.

When it comes to controlling the power consumption, PC and sensor nodes differ in the controlling targets for one. Should an operating system on a PC be forced to reduce power consumption of the system, it can do this by either throttling the system or by assigning more CPU-time to a less power consuming process. On a sensor node, however, there usually is only one application. Throttling the system by simply increasing the sleep time might not work here, because the node might miss a communication window and then spend more energy by trying to find a communication partner than the increased sleep cycle originally saved.

In the PC world, power consumption control is usually used to let the system continuously stay below some limit in order to require less expensive or noisy cooling. Short CPU bursts are allowed, as long as the overall consumption rate is not adversely affected. For a sensor node, the quality of power consumption control is determined by the lifetime a node can reach. So while the PC has an (assumed) endless supply of power, a sensor node is equipped with a limited amount of energy from the start.

4.1.2 Central Processor

Taking a first look from the point of energy consumption, there are only two states of a processor, active and idle. Differences in consumption rates within these states are typically small compared to the difference of the average consumption rates between them. We therefore examine those states more closely.

4.1.2.1 Active Mode

As stated above, the power consumption of the ATMega128 in active state is relatively constant which can be seen in Figure 4.1. The listed instructions were selected for their usage of different processor units: `nop` only makes use of the instruction decoder, `xor` and `fmul` both perform operations with different complexities on registers and `lds` finally exercises the memory interface.

All power consumption rates lie in the range of 30 mW $\pm$ 3.3%. Depending on the desired power consumption precision, energy accounting can take various forms:

One of those forms is to let the compiler generate accounting code, i.e. a few instructions of code that increase some counter by a fixed value. This is similar to simulation techniques of energy

instruction	power	current
`nop`	29.7 mW	10.0 mA
`xor`	29.5 mW	9.9 mA
`fmul`	29.0 mW	9.8 mA
`lds`	31.0 mW	10.5 mA

Figure 4.1: Power consumption and current draw of different instructions

accounting like in PowerTOSSIM, with the difference that this code is kept after development and deployment.

Care has to be taken in selecting the insertion points. A solution immediately crossing one's mind would be to insert this code at the beginning of each function. But since functions usually contain conditional instructions, this would lead to inaccurate results despite the amount of overhead.

A better place would be the beginning of each *basic block*. This term from the compiler world denotes a block of instructions, in intermediate as well as in assembler code, with two properties:

- It is entered at the first instruction only and
- every instruction in the block is executed exactly once.

Each instruction itself is a basic block, of course, but usually one is interested in basic blocks of maximal size. Such a block typically begins with a label and ends either before another label or after the first branch instruction.

By using this method it is possible to assign an individual energy consumption value to each instruction. For this reason, the method should be very precise, but it would also require a relatively large overhead, since even a small function with only two `if` ... `else` ... statements can easily have seven basic blocks of maximal size, one for each of the ... and one block before, between and after the two `if`s.

If precision is not the ultimate goal and the assumption that all instructions basically have the same power consumption rate is acceptable, another method of accounting becomes possible. Under this assumption, the only thing needed for accounting is the time the processor was active.

This was readily implemented using one of the timers. For the BtNode, timer/counter 3 was chosen because it allows precise counting if its prescaler is set to not scale down the main clock of the BtNode, and its 16 bit counter register generates an overflow interrupt only after 8.9 ms in active state with this clock setting.

In TinyOS the only code using the `sleep` instruction should be in the scheduler, which is shown in Algorithm 1. So one part of the accounting code is inserted just before the sleep instruction.

Algorithm 1 Instrumented scheduler of TinyOS

```
loop
  while ! queue.empty() do
    run_next_task()
  end while
  cpu_accounting()
  sleep()
  cpu_accounting()
end loop
```

But as the BtNode is usually woken up by some interrupt, there is typically more than one place to start the accounting. An accurate accounting would require the insertion of the accounting startup code at the beginning of every interrupt handler. For our proof-of-concept implementation we start accounting both in the `post` function and after the `sleep` instruction.

As an interrupt waking up the application has to return to the instruction following `sleep`, the accounting code which is placed there is used to start accounting for the synchronous half. The `post` instruction is used by functions in the asynchronous half of TinyOS (interrupt context) to transfer control to the synchronous half (`task`s). The insertion of accounting code here already starts accounting in the asynchronous half, thereby increasing the portion of accounted code. With only two insertions this makes sure that every operation in the synchronous half of TinyOS is accounted as well as part of the asynchronous half.

These two methods can be mixed, of course, resulting in more accurate values with less overhead than the first method. The method of instrumenting basic blocks could be restricted to a subset of instructions with non-average energy consumption, accounting only differences to the average consumption while the other method is responsible for accounting the average energy consumption for all instructions. For example, this subset could contain all instructions affecting memory (load and store on the ATMega128), because the values in Figure 4.1 suggest their power consumption to be higher than the average arithmetic operation. Basic blocks not containing any of these special instructions need not be instrumented, so the size and composition of this set has a direct influence on the accounting overhead. Compiler optimizations could reduce this overhead even further by aggregating the instrumentation of basic blocks.

When applied to a sample application that wakes up four times per second and performs some computations, the method of accounting the active clock cycles of the CPU increases the power consumption by 34.4 μW or 4.1%. This overhead depends on both the interrupt rate and the computation time, in short, the frequency of interrupt wake ups and sleep instructions. The code size is increased by 356 bytes and 6 bytes of memory are allocated for the counter. To overflow this counter, the BtNode would have to be continuously active for 442 days.

mode	power	current
`idle`	14.28 mW	4.8 mA
`power save`	0.79 μW	266 μA
`power down`	0.71 μW	239 μA

Figure 4.2: Power and current consumption in various sleep modes

4.1.2.2 Sleep Mode

For accounting how much energy the CPU spent in sleep mode, there is basically only one method. The microcontroller has to be aware of how long the sleep period actually was. One could think of using time information which is sometimes required in protocols for synchronizing sensor nodes, if already available.

But in order to cover all kinds of applications, there doesn't seem to be an alternative to an externally clocked timer. As an integrated part of the node, it does not have the latency associated with network protocols. And, by being partly external to the node, it allows the microprocessor to enter one of the deeper sleep modes in which nearly all internal clocks are inoperative. This is important because the deepest sleep mode possible while still using internally clocked timers is `idle`, the power consumption of which is shown by table 4.2 to be 14.28 mW, as opposed to the 0.79 μW possible in `power save` mode. In `power down` mode the external timer is disabled, so this mode is not suited for the accounting of idle time.

For implementation on the BtNode, timer/counter 0 of the ATMega128 was used because it is the only counter with the capability of being asynchronuously (i.e. externally) clocked. Insertion of the accounting code was done at the same locations as in the active CPU accounting method. The calculation of the amount of energy used in sleeping mode requires two values, the selected sleep mode and the length of time spent sleeping. While the first value can be obtained with two assembler instructions, the latter requires more instructions because it is very likely for the application to also make use of this timer and the accounting code should be transparent. So after recording the number of timer ticks, the accounting code has to check the prescaler settings and scale this number accordingly.

Together, active and idle accounting increase the power consumption of the above-mentioned test application by 39.1 μW or 4.7%. As before, the overhead depends on the frequency of transitions between active and sleep mode.

4.1.3 Peripheral Hardware

Energy accounting of peripheral hardware like the CC1000 can apply essentially two methods, already described while discussing active CPU accounting. The event-based method accounts hard-

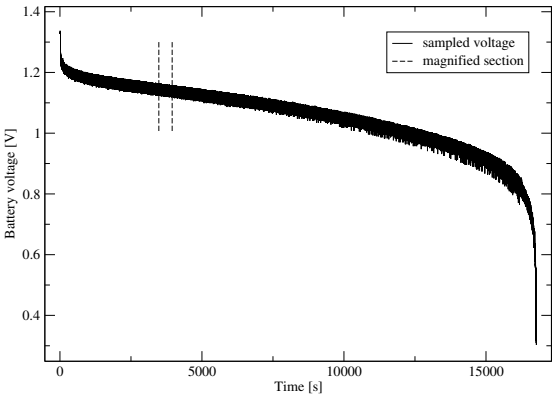

Figure 4.3: Discharge characteristics

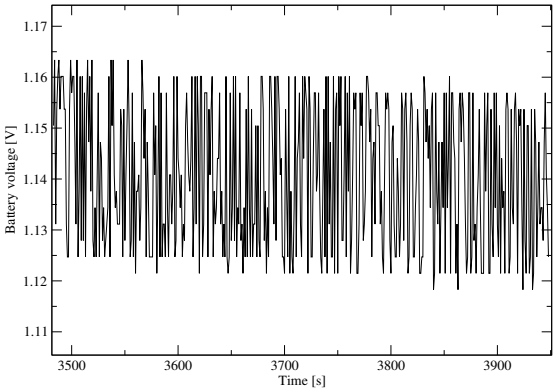

Figure 4.4: Close-up discharge

ware events, in the case of the CC1000 these would include the sending and receiving of packets. The other approach is time-based, i.e. the time spent in a state is accounted and multiplied with the power consumption associated with this state to get the energy.

The CC1000 is ideally suited for these methods because of its openness. The application has the ability to exercise fine-grained control over all relevant parts of the chip, thus providing the application with the data it needs. Details on the power consumption of the CC1000 in various modes can be found in AEON [10], for example.

In contrast to that, the Bluetooth chip hides its low-level functionality behind an interface. While this is good practice for developing robust software, it lacks the detailed information necessary for accounting energy usage. Both above-mentioned methods can be used, of course, but even a blending of both is not expected to provide accurate accounting information.

4.2 Battery Lifetime Estimation

No matter how accurate the estimation of power consumption may be, it is useless if the system has no information about the remaining capacity of the power source. Both pieces of data allow the node to compute the likelihood of achieving the desired lifetime and act accordingly.

As the BtNode is powered with standard AA-sized batteries, it can be easy to run such nodes using a variety of different battery types, especially in installations like building monitoring where the nodes can be located and their power source can therefore be renewed. Hence, an ideal method to estimate battery capacity would apply to many battery types while requiring as few manually supplied parameters as possible.

In Figure 4.3 the discharge characteristics of an already used alkaline battery are shown together with a more close-up view of the marked section in Figure 4.4. It is not immediately obvious which function the characteristics in Figure 4.3 conform to. The close-up view illustrates another problematic phenomenon, the erratic readings from the A/D converter of the BtNode. Both problems will be discussed in the course of the following sections.

4.2.1 Battery Model

An ideal way of estimating the remaining battery lifetime would be to have a generic battery model with few parameters which can easily be computed using the first batch of battery voltage samples. The remaining capacity of the battery would either be an explicit parameter of the model or could be deduced from the parameters. For a start, the focus was on finding an applicable model to the batteries in use.

One model which is close to the discharge characteristics of Figure 4.3 is the following. It assumes that inside the battery there is a large but limited amount of electrons which have to exit the battery, do their work in the circuits of some chip and arrive finally at some electron sink. For the model, only the way up to the battery surface is relevant, as the amount of electrons on the surface determines the battery voltage. Even the way back to the battery is disregarded. To exit the battery, these electrons have to travel through some liquid chemicals to the metal surface.

The battery is simulated by three capacitors, Q_0 for the electron heap, Q_1 for the transporting chemicals and Q_2 for the metal surface. Two functions which depend on the charge of two neighboring capacitors are used to compute the charges to be transported in one step, v_1 and v_2. Note that these functions are not based on laws of chemistry or electricity. They are selected for their simplicity.

$$
\begin{aligned}
v_1 = &\quad c_1 Q_0^{\frac{2}{3}} (M_1 - Q_1) \quad \text{(I)} \\
v_2 = &\quad c_2 (\tfrac{Q_1}{M_1} - \tfrac{Q_2}{M_2}) \quad \text{(II)} \\
dQ_0 = &\quad Q_0 - v_1 \\
dQ_1 = &\quad Q_1 + v_1 - v_2 \\
dQ_2 = &\quad Q_2 + v_2 - v_e
\end{aligned}
$$

The parameters M_1 and M_2 denote the maximum charge of capacitors 1 and 2, both being considerably smaller than the initial value of Q_0. All of these parameters, combined with the constants

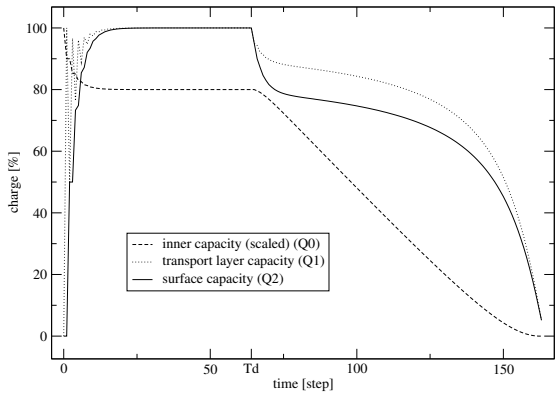

Figure 4.5: Discharge of the model battery

c_1 and c_2 which have some influence on the simulated reaction rate, provide the tuning knobs for fitting the simulated discharge characteristics to the measured values.

The exponent in equation (I) stems from the idea that the electron supplying chemicals are arranged in some 3-dimensional shape and disengage from this object when consumed. While Q_0 shrinks linearly with the volume, the reaction rate depends on the size of the surface, which can be computed as the squared cubic root of Q_0, multiplied by some constant factor, the value of which depends on the shape.

Once the parameters are fitted and stable, the model is updated by setting v_e to the capacity used by the system since the last update, calculating v_1 and v_2 and then adding and subtracting these values from the tree capacities accordingly. The remaining capacity can easily be computed as the sum of all three capacity variables.

The characteristics of the three capacities are shown in Figure 4.5. At first they are given time to stabilize, and from time T_d a constant current is drawn from the model battery. The constants used in creating this figure are: $M_1 = 100$, $M_2 = 100$, $c_1 = 0.01$, $c_2 = 50$ and $v_e = 10$. Apart from $Q_0 = 1000$, the initial values of the other capacities were set to zero.

The problems of this model begin at solving the differential equations to see the impact of all parameters on the simulated discharge characteristics curve. With information on the impact of those parameters it would be clear how to extract these parameters from a set of battery voltage samples. But each of the three differential equations depends on at least another one, on two in the case of Q_1.

$$dQ_0 = Q_0 - c_1 Q_0^{\frac{2}{3}} (M_1 - Q_1)$$

$$dQ_1 = Q_1 + c_1 Q_0^{\frac{2}{3}} (M_1 - Q_1) - c_2 \left(\frac{Q_1}{M_1} - \frac{Q_2}{M_2} \right)$$

$$dQ_2 = Q_2 + c_2 \left(\frac{Q_1}{M_1} - \frac{Q_2}{M_2} \right) - v_e$$

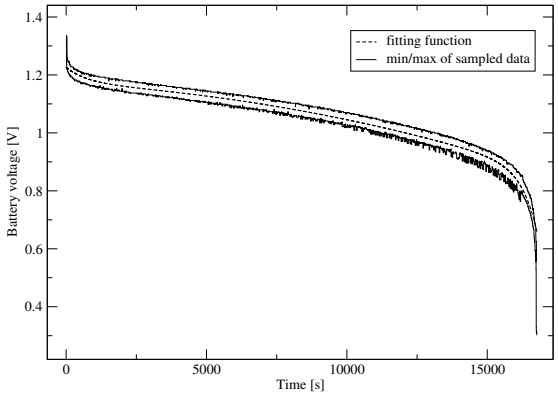

Figure 4.6: Fitting to discharge characteristics

After investing some time in solving this system of differential equations, it was decided to discontinue this approach and turn towards the less sophisticated, but more promising one of directly fitting a given function to the sampled voltage characteristics. Even if a solution were to be found, the computation of the parameters would still require a method of performing exactly this kind of operation on the node. We therefore study how to compute such a fitting operation on a sensor node.

4.2.2 Regression Fitting

The idea of this approach to battery lifetime estimation is basically the same as before. Only this time, the step of developing a model with differential equations is skipped. Instead, a function with the possibility of fitting to the discharge characteristics is selected beforehand, and the sensor node application is given the task of finding the actual parameters of this function.

For example, Figure 4.6 shows that it is possible to fit a function to the discharge characteristics of Figure 4.3. The parameters were computed manually, then given to the non-linear curve fitting algorithm of `xmgrace`, a program for plotting and analyzing data. The mentioned function is of the following form:

$$f(x) = a_6 e^{a_5 x} + a_4 e^{a_3 x} + a_2 x^2 + a_1 x + a_0$$

While the polynomial of degree 2 is responsible for fitting to the large middle part of the characteristics, the two exponential parts are needed for the outer data sections. Once all parameters are known, the sensor node can calculate the remaining lifetime of the batteries without any particular difficulty.

As this fitting is a potentially expensive operation, we investigated how to adapt simple fitting methods to functions which are most likely to fit to the sampled voltage characteristics. One of the most basic fitting methods is regression, which is usually used for linear functions.

4.2.2.1 Linear Regression

The idea of regression fitting is illustrated in the following paragraphs using a linear function. Suppose we have a set of n pairs of data (x_i, y_i) $(0 \leq i < n)$ which appear to be linearly dependent, and we therefore want to fit the function $f(x) = ax + b$ to these values.

The first step is to select an error function that describes the quality of the fitting. There are several functions which are suitable for this task, the most common being the *least squares* function, i.e. the sum of the squared differences between fitting and data, in this case:

$$E(a,b) = \sum_{i=0}^{n-1} (y_i - ax_i - b)^2$$

Now, fitting $f(x)$ to the data points means to find values for a and b where $E(a,b)$ reaches its minimum. To do that, the homogeneous system of partial derivatives with respect to the parameters a and b is solved:

$$\begin{aligned} \frac{\partial}{\partial a} E &= -2 \sum_{i=0}^{n-1} (y_i - ax_i - b) x_i \quad &= 0 \quad \text{(I)} \\ \frac{\partial}{\partial b} E &= -2 \sum_{i=0}^{n-1} (y_i - ax_i - b) \quad &= 0 \quad \text{(II)} \end{aligned}$$

$$\text{(II)} \Leftrightarrow \sum_{i=0}^{n-1} y_i - a \sum_{i=0}^{n-1} x_i - nb = 0 \Leftrightarrow b = \frac{1}{n} \left(\sum_{i=0}^{n-1} y_i - a \sum_{i=0}^{n-1} x_i \right)$$

$$\text{in (I):} \sum_{i=0}^{n-1} x_i y_i - a \sum_{i=0}^{n-1} x_i^2 - b \sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} x_i y_i - a \sum_{i=0}^{n-1} x_i^2 - \frac{1}{n} \sum_{i=0}^{n-1} x_i \cdot \sum_{i=0}^{n-1} y_i + \frac{a}{n} \left(\sum_{i=0}^{n-1} x_i \right)^2 = 0$$

$$\Leftrightarrow a = \frac{\sum_{i=0}^{n-1} x_i y_i - \frac{1}{n} \sum_{i=0}^{n-1} x_i \cdot \sum_{i=0}^{n-1} y_i}{\sum_{i=0}^{n-1} x_i^2 - \frac{1}{n} \left(\sum_{i=0}^{n-1} x_i \right)^2}$$

The required sums can be computed in one pass over the data, so potentially wasteful storage of data is not required. These formulas can be simplified further if x_i has additional properties like being equidistant. This would cause most of the sums containing x_i to be expressed in a simpler form.

4.2.2.2 The Quadratic Part

As already shown in Figure 4.6, a polynomial of degree 2 can be fitted to the middle section of the discharge characteristics with minor deviation. Compared to the fitting of a linear function, the computation is longer, but not more complex, which is the reason why it is skipped here.

The quadratic function used for fitting is of the form $f(x) = a_2x^2 + a_1x + a_0$. Solving the system of partial derivatives yields the following solution:

$$\begin{aligned} k_0 &= \sum x^2 - \tfrac{1}{n}\left(\sum x\right)^2 \\ k_1 &= \tfrac{1}{k_0}\left(\tfrac{1}{n}\sum x \sum x^2 - \sum x^3\right) \\ k_2 &= \tfrac{1}{k_0}\left(\sum xy - \tfrac{1}{n}\sum x \sum y\right) \end{aligned} \qquad \begin{aligned} c &= \tfrac{1}{n}\left(\sum y - a\sum x^2 - b\sum x\right) \\ b &= k_1 a + k_2 \end{aligned}$$

$$a = \frac{\sum x^2 y - k_2 \sum x^3 - \frac{1}{n}\sum x^2 \sum y + \frac{1}{n}k_2 \sum x \sum x^2}{\sum x^4 + k_1 \sum x^3 - \frac{1}{n}\left(\sum x^2\right)^2 - \frac{1}{n}k_1 \sum x \sum x^2}$$

Here, n is the number of data points (x_i, y_i). Each of the above x and y are actually x_i and y_i. Likewise, all $\sum$s should read $\sum_{i=0}^{n-1}$. These sub- and superscripts have been omitted for readability.

This method probably should not be applied to situations where the x_is are not accurate, since they are raised up to the fourth power during this computation, and with them any error. In contrast to the erratic A/D converter results on the BtNode, the time values constituting the x values can be measured quite accurately.

4.2.2.3 Exponential Terms

The real challenge, however, lies in fitting a function to the steeper ends of the discharge characteristics. As the start of discharging a battery has something in common with discharging a capacitor, it is natural to look at exponential functions as candidates for being fitted to this particular part of the voltage samples. Figure 4.6 already showed that these functions are suited for this operation.

With the information gained from such a fitting the exponential curve can be filtered out of future voltage samples, assuming that it is possible to achieve a fitting before the influence of the exponential function diminishes anyway. More importantly, the computed fitting parameters could be used to distinguish between several types of batteries, or to predict the parameters of the second exponential curve. Here, the point of "becoming a significant term", i.e. the point where the curve diverts observably from its previously slightly polynomial course towards a steep decline, is of particular interest.

As it is desirable to get the fitting parameters as early as possible while the polynomial character of the middle part becomes apparent only after a significant time has passed, a combination of

exponential and linear function was selected as fitting function:

$$f(x) = a_0 e^{a_1 x_i} + a_2 x_i + a_3$$

The regression method does not readily apply to exponential curves because the resulting sums of $\exp(\cdot)$ tend to make the equations hard to solve. To work around this problem, the following conditions are necessary:

- the total number of sample points (x_i, y_i) is $3n$, i.e. a multiple of 3
- these samples are equidistant, i.e. $x_i = x_0 + i \cdot step$

Together, these conditions make it possible to use n equidistant triplets of points to get an error function without exponential terms. If the samples were of the form

$$y_i = a_0 e^{a_1 x_i} + a_2 x_i + a_3$$

the following condition would hold:

$$\frac{y_{i+n} - a_2 x_{i+n} - a_3}{y_i - a_2 x_i - a_3} = \frac{y_i - a_2 x_i - a_3}{y_{i-n} - a_2 x_{i-n} - a_3}$$

This can be converted to an error function:

$$E = \sum_{i=n}^{2n-1} (y_{i+n} - a_2 x_{i+n} - a_3)(y_{i-n} - a_2 x_{i-n} - a_3) - (y_i - a_2 x_i - a_3)^2$$

Using the same steps as in section 4.2.2.1, we obtain solutions with regard to a_2 and a_3, the linear part of the fitting function, displayed below. Note that each of the $\sum$s is the sum over all triplets, that is $\sum_{i=n}^{2n-1}$.

$$\begin{aligned}
b_i &= y_{i+n}x_{i-n} + y_{i-n}x_{i+n} - 2x_i y_i & a_3 &= k_1 a_2^2 + k_2 a_2 + k_3 && \text{(I)}\\
c_i &= y_{i+n} + y_{i-n} - 2y_i & t_3 &= \textstyle\sum 2n^2 \left(n^2 + c_i k_1\right) &&\\
d_i &= y_{i+n}y_{i-n} - y_i^2 & t_2 &= \textstyle\sum \left(n^2 b_i + b_i c_i k_1 + 2n^2 b_i + 2n^2 c_i k_2\right) &&\\
k_1 &= \tfrac{1}{\sum c_i^2}\left(-n^2 \textstyle\sum c_i\right) & t_1 &= \textstyle\sum \left(b_i^2 + b_i c_i k_2 + 2n^2 c_i k_3 - 2n^2 d_i\right) &&\\
k_2 &= \tfrac{1}{\sum c_i^2}\left(-\textstyle\sum b_i c_i\right) & t_0 &= \textstyle\sum \left(b_i c_i k_3 - b_i d_i\right) &&\\
k_3 &= \tfrac{1}{\sum c_i^2}\left(\textstyle\sum c_i d_i\right) & 0 &= t_3 a_2^3 + t_2 a_2^2 + t_1 a_2 + t_0 && \text{(II)}
\end{aligned}$$

Note that (II) can have up to 3 different solutions for a_2, and in practice it often will. So a fitting algorithm has to compute the other coefficients a_0, a_1 and a_3 threefold and select the most accurate

fitting solution. When a_2 and a_3 are known, a_0 is basically computed as $a_0 = y_0 - a_3$ while a_1 requires a little more effort:

$$e^{ma_1} = \sum_{i=0}^{m-1} \frac{y_i - a_2 x_i - a_3}{y_{i+m} - a_2 x_{i+m} - a_3} \quad \left(\text{with } m = \left\lfloor \frac{3}{2} n \right\rfloor \right)$$

The use of this method is problematic. For one, at least some of the data must be stored to the very end of the computation to make a decision on the most accurate fitting. While it should be no problem on the BtNode to store a few hundred or thousand samples of the battery voltage, this may be the case for other platforms.

This method works best if the data samples contain the beginning of the linear section. The time which is necessary to collect all this data is another disadvantage. The typical power consumption of a sensor node is already quite low, so it can take a long time for the battery to reach the linear section of its discharge characteristics. Assuming a specified lifetime of 3 years, this data collection period could take up to 2 months.

Yet another problem of this method is its complexity: For a one-pass implementation, 8 different sums have to be updated for each A/D converter result. To compute a_2, a polynomial of degree 3 has to be solved, and for each of its solutions the corresponding fitting has to be checked against the stored data. Furthermore, most of these variables must be floating point or multi-precision types to accommodate very small values like a_1 and a_2. The `avr-libc` implements floating point operations, but this only reduces development time, neither code size nor runtime.

On the BtNode only half of the battery voltage is connected to the A/D converter. Combined with the fact that operation quickly ceases once each battery produces less than 0.8 V, this reduces the range of the measured values from the maximum 1024 of the A/D converter to about 250. Furthermore, these values are either not very accurate or the battery voltage changes very quickly, since they seem to oscillate between 4 adjacent values at any given time. One would have to smooth the graph in order to apply this method despite of all its disadvantages, but this increases code size, runtime and power consumption even more.

Possibly the most serious fault of this method is its “brittleness”: When given data samples that already conform to a combined exponential and linear function it produces the desired result. However, moving the samples by a random offset considerably affects the parameters of the fitting to the point where the term “fitting” is no longer appropriate. This can be seen when comparing the figures 4.7 and 4.8 where both above-mentioned data samples and the resulting fitting are plotted. The random offset added to the y-axis part of these data points in Figure 4.7 was $\pm 0.125\%$ and $\pm 0.25\%$ in Figure 4.8. In comparison, the relative error on the BtNode of the measured battery voltage samples is about $\pm 0.4\%$. The range of data points on the y-axis was chosen to resemble battery voltage samples from the A/D converter of the BtNode.

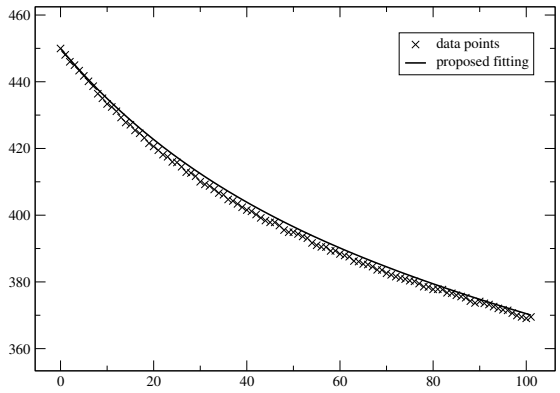

Figure 4.7: Fitting data points with little random offsets

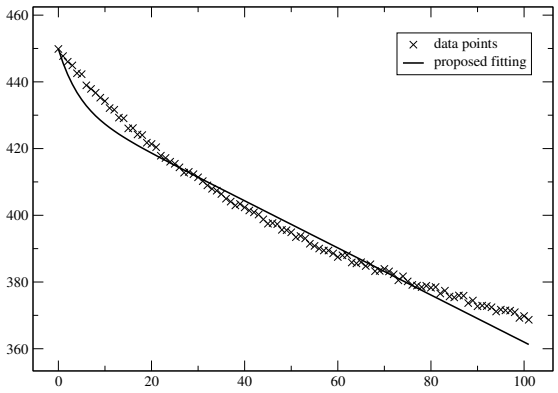

Figure 4.8: Proposed fitting of data points with a slightly larger random offset

4.2.3 Other Possibilities

Although all battery lifetime estimation methods mentioned so far try to predict the characteristics of the one battery type used, they are only necessary if several different battery types are allowed to power sensor nodes with the same software. This power source variability affects the BtNode in particular because AA-sized batteries are based on a wide variety of chemical reactions, let alone differences between producers and different battery models from one manufacturer.

However, we assume that in most sensor networks it is possible to impose a limit on the used battery types. When only few types of batteries are allowed to power a project's sensor nodes, tables describing the batteries' characteristics can be stored on the nodes, thus eliminating the need for expensive fitting operations.

The advantage of this approach is that the addition of environmental parameters, particularly the temperature, can be handled without radically changing the method. The number of pages would increase significantly, but this could be countered by compressing these tables. Trying to integrate temperature into the above-mentioned approaches would complicate the differential equations or the fitting operation further, possibly rendering their implementation impractical on a sensor node.

As an alternative to storing large tables of data on sensor nodes, special batteries with approximately linear discharge characteristics can be used. Although more expensive, this would eliminate the need for costly algorithms and could thus save even more energy.

Chapter 5

Reconfiguration

In some systems power consumption information may be sufficient on its own, for example in application evaluation before deployment, if some hardware characteristics were difficult to simulate. This allows application developers to tailor their code to certain goals before deployment, e.g. to get as much sensor readings as possible out of the node while still guaranteeing a specified lifetime. But there are already other tools to get this information before deployment, e.g. using simulation together with an energy model of the sensor node.

Therefore the focus of this chapter lies on methods to alter an application at runtime. The idea is that the node has some parameters like the desired lifetime or a power consumption limit, and can invoke various means, some of them quite radical, to fulfill these requirements. These methods are discussed in the following sections with regard to their requirements, flexibility and overhead both in code size and energy.

5.1 Parameter Adjustment

It is hard to imagine an application without any parameter arbitrarily selected by the developer from a range of reasonable possibilities. Examples of such parameters in typical applications include values for timeouts, size of small buffers and so on. On a sensor node, such parameters can have a direct influence on energy consumption and thus can be used to control it.

Most of the time a sensor node will be sleeping to prolong the battery lifetime. Usually this involves some kind of sleep period, which qualifies as such a parameter. Of course, the sensor node could only wake up at some external event, eliminating the need for this kind of parameter.

The other functionality of a sensor node, communication, also offers some parameters. The data aggregated by a sensor node is usually sent to other nodes. Here, the amount of collected

data needed to justify the power consumption of communication is an important parameter that represents a trade-off between potential data loss and wasted energy.

Of course, this is a very naïve view of the communication system. The power consumption of the radio hardware makes it necessary to arrange communication windows with the neighboring nodes, in which the node can send and receive. In this system it is more sensible for a node to arrange for more infrequent communication windows, requiring a potentially complex decision algorithm.

Both types of parameters can be seen in an exemplary application (Algorithm 2). The parameter *sleep_period* defines the normal duty cycle, *buffer_length* determines the frequency of normal sending operations.

Algorithm 2 Exemplary application with parameters

```
buffer.setlength(buffer_length)
loop
  val ← read_sensor()
  buffer.add(val)
  if val ≥ threshold or buffer.full() then
    send_packet()
  end if
  sleep(sleep_period)
end loop
```

Were it not for the parameter called *threshold*, the application's behavior would be deterministic and therefore the calculation of the duty cycle and energy consumption would be straightforward. The inclusion of the threshold parameter introduces a probabilistic element p which denotes the probability of the sensor value exceeding the *threshold*:

$$P_{average} = \frac{E_{sense} + \frac{1+p\left(n_{buf}-1\right)}{n_{buf}} E_{send} + t_{sleep} P_{sleep}}{t_{sense} + \frac{1+p\left(n_{buf}-1\right)}{n_{buf}} t_{send} + t_{sleep}}$$

The term "probabilistic" here must be dealt with care. With certain sources of the sensor data, once a sensor value exceeds the threshold there is a high probability that subsequent readings do as well. So p must be assigned a value in order to calculate the expected power consumption before deployment, but environmental conditions can rapidly invalidate this assumption. Once an application's power consumption reaches a level resulting in a premature end of operation, it can adjust some of the parameters to remain consistent with the targeted lifetime.

Not all parameters are suitable for this kind of adjustment. A common choice, for example, would be the length of the sleep period. This is certainly a good choice for a single node, but

inconsiderate changes to this parameter could result in more energy consumption rather than less. For example, the sleep period could affect the synchronization of the nodes, and a node looking for communication partners when all other nodes are asleep could thus easily consume more energy than the prolonged sleep period theoretically would have saved. Therefore, changing the threshold parameter would be more sensible here.

This exemplary application is simple when compared to real world specimen. But as applications grow, so do their number of parameters, and in turn the difficulty to use these parameters for saving energy. So this method might not be applicable to complex applications.

The leverage of of application parameters on power consumption heavily depends on the application in question. The overhead of parameter modification can also vary widely. Changing a parameter can be as simple as writing a new value to memory or as complex as negotiating new communication parameters with the network. For this reason, we cannot give useful numbers on the overhead of this method.

5.2 Application Upload

An application update method which has already been under extensive research is the method of distributing a new application over the sensor network and cause the nodes to switch to this new version. Different variants of this concept have been proposed. For example, there are at least two implementations for TinyOS, called Xnp and Deluge, both providing services for distributing and activating complete images of new applications.

Other variants try to reduce the amount of code that has to be transferred for each update. The operating system SOS does this by introducing indirections on inter-module calls, so that only changed modules have to be uploaded. Maté goes one step further. It provides a virtual machine on the nodes, with additional primitives for reading sensors as well as for sending and receiving packets. Applications written for this machine are therefore quite small. An update of these applications can consist of as few as 24 bytes.

For this work, we wanted to make use of the BtNode's most prominent feature, the Bluetooth controller. It relieves the developer of implementing error detection and data streams. In addition, an ordinary PC can be used to directly upload the new application. The implemented, upload supporting application waits for a peer to connect, stores the data it receives to the external RAM, and then proceeds to program this code and reset the node.

The code for reprogramming the flash memory of the ATMega128 on the BtNode has to be located inside a special memory area. This can be done by modified linker scripts. Since this area is rather small, updates to this memory section can be more complex than to the main part of program memory. For this reason, the implementation used in this work never updates this memory

section and new applications are not linked against this code. Instead, new applications have to be supplied with the symbol position of the self-programming function. This procedure slightly reduces the overhead, as this function does not have to be uploaded again.

Transfer of control to the new application could be done by simply jumping to the start address of the new code. But doing so would require the old application to shut down in a clean manner, i.e. all subsystems should be set to "off", because the new application might not be aware of some parts of the hardware, which then would continue to consume energy.

Although this could have been done in our applications, it was decided to transfer control by resetting the node. One way of doing this is to intentionally overflow the watchdog timer. This ensures that all hardware on the BtNode is set to its initial state. Sensors might have to be reset or turned off explicitly, depending on the sensor board. Note that the contents of the RAM are not affected by the reset, except for the part reserved for global variables, which are initialized by the startup code of the new application.

There is considerable overhead using this method. First of all, the Bluetooth stack shows room for improvement. When waiting for a connection, the Bluetooth chip consumes more than 75 mW, and the transfer time of a 16.5 KB application takes 18.9 s. Second, according to [2], programming a 256 byte page of flash memory takes 20 ms and uses 1–2 mJ, resulting in about 100 mJ being consumed in 1.32 seconds. After the update, no further overhead occurs.

The code size of the residual function implementing the self-programming is 1274 bytes, the application code responsible for storing the new application to external RAM uses 964 bytes of flash memory. Parts of these 964 bytes may be already required by the application for storing sensor data, so sometimes this may not count as overhead. To summarize, this approach offers very high flexibility while implying a heavy overhead.

5.3 Application Switching

The next method of altering an application also allows radical changes. The idea is to provide the sensor node with two or more applications stored in ROM. On the BtNode, the 128 KB of flash memory allow to store at least two large applications. If the predicted lifetime of the default application is lower than expected, the application can initiate a switch to the other application. This harsh measure could be useful in dynamic sensor networks if a node is out of communication range and wants to stop collecting data and save the data obtained so far.

Although the default application could adapt to a lower power consumption limit, application switching offers more possibilities like changing hard-coded control flow to a simpler variant. For example, if the default application uses one timer for more than one purpose, e.g. delivering events at different rates to several software modules, the event generation code could grow to some com-

plexity, like keeping lists of scheduled events sorted by time. If the second application merely uses the timer for one event, the code for the event generation can shrink to one function call.

The switching itself can be done in several different ways, depending on the applications' requirements and behavior.

5.3.1 Applications Without Interrupts

If none of the applications needs interrupts, it would be sufficient for application switching to jump to the startup code of the desired application. This switching operation would have the lowest cost in all aspects when compared to the other methods.

However, it it also very impractical. In order to avoid using interrupts, a node has to be active the whole time. Even if it could sleep, the sensors and radio chips would have to be polled regularly, and the microprocessor would be active at least until the sensor or radio chip respond. Also, with operating systems such as TinyOS it would be hard to write a complete application without using interrupts, because the NesC language (in which TinyOS is written) with its `command`s and `event`s is clearly designed for an interrupt aware application.

This strategy can be salvaged in situations where exactly one application uses interrupts. Then a switch from the interrupt-aware application would clear the global interrupt flag, and vice-versa.

This variant of application switching is mentioned only for the sake of completeness. Applications using no interrupts are not really suitable for sensor nodes, as the use of interrupts enables applications to enter the energy preserving sleep modes.

5.3.2 Interrupt Table Modification

In more realistic cases where two or more applications on the node answer to interrupts some more work has to be done to enable switching between these applications. One idea is to use the self-programming code of the application upload method to overwrite the interrupt table.

In our hardware, this means to overwrite the first 256 byte page of program memory. This first page consists of a jump instruction to the initialization code, a table of jump instructions to all interrupt handlers and the beginning of the initialization code which ends with calling `main`. So by overwriting this page, a reset starts the newly selected application.

The overhead of this variant is quite small. As only one page of flash memory is written, this takes 20 ms and 1–2 mJ, using the above-mentioned values. The increase in code size of two small applications, each allowing to switch to the other one, is 2722 bytes. Apart from the work at switching, no additional overhead is encountered. The flash memory on the ATMega128 is specified to support at least 100,000 writing operations per page. This should be sufficient for most applications.

5.3.3 Interrupt Dispatcher

It is also possible to avoid writing to flash memory when switching applications. To this end, code for dispatching interrupts has to be inserted. Doing this transparently for the application requires a few implementation tricks.

First of all, our implementation substitutes all `jump` instructions to various interrupt handlers in the interrupt table by `call` instructions to the dispatching function, to provide the dispatcher information about which interrupt was called. After saving the used registers, the dispatching function calculates the address of the original `jump` instruction to the interrupt handler in the saved interrupt table of the selected application and replaces the return address on the stack with it. The number of the selected application is stored in RAM, so all an application has to do in order to switch is to change this byte and reset the microcontroller.

The code size is increased by 334 bytes if this method is applied to the two above-mentioned switching applications. The dispatching code itself causes a delay of 56 clock cycles for each interrupt. This method has the lowest switching overhead of all presented reconfiguration methods. The major part of the overhead should occur during the normal runtime of an application. However, the overhead on our test application was within the measurement error of our DAQ (Data AQuisition) hardware.

Chapter 6

Future Work

Given the attention that parts of the methods presented in this work already have received, it is not surprising that only few directions for future research remain.

The energy models used for offline energy accounting are fairly advanced and accurate, so porting them to online energy accounting seems like a task of overcoming mere technicalities. The method of energy accounting implemented for this work could be improved with regard to its accuracy by starting the accounting at the beginning of each interrupt. What could be of more interest, though, is the development of an energy model for a Bluetooth chip and the means to control its power consumption.

Methods to accurately predict the remaining battery lifetime online comprise the one part of this thesis with a large spectrum of directions for future research. The development of an improved battery model should be possible if the chemistry inside the battery is incorporated. More advanced methods of non-linear curve fitting such as the Levenberg-Marquardt algorithm [11, 12] could be evaluated for the use on sensor nodes. This kind of method probably has the most benefit.

In contrast to that, the area of reconfiguration does not seem to offer much space for future work. One possibility would consist in the development of a system for automatically adjusting selected application parameters when given a description of their impact on the application. This could eliminate the need for application updates for the one reason of changing a set of parameters.

Chapter 7

Conclusion

This work shows that adaptive power management on sensor nodes is possible under certain circumstances. Most of the presented methods have been implemented and found to work.

The implementation of the time-based energy accounting method for active and idle states of the microprocessor should provide values sufficiently accurate for most applications. If more accurate accounting information is required, the additional event-based method offers a trade-off between overhead and accuracy. For this method, compiler support for instrumenting basic blocks before optimization seems to be the best approach.

The estimation of the batteries' remaining lifetime proved to be a challenge not easily mastered. Battery models quickly tend to get too complex, and use of the simple method of regression to find a direct approximation also failed. This was due to the brittleness of the regression method when applied to exponential curves. A method working for every kind of battery would be to store its explicit discharge characteristics on the node. Because of their size, this would restrict the number of allowed battery types, which may not be a problem for productive use.

For software reconfiguration, a variety of working methods with different requirements have been presented. Adaptation of important parameters is one method which is too application dependent to give useful information on its overhead. A method which has got a lot of attention in research is to distribute a new application over the network and have the nodes update themselves. Because of the large overhead, several variants have been proposed by others to keep the size of distributed code to a minimum while still being energy efficient. Another method of switching between multiple pre-installed application variants has been investigated, differing only in the way the interrupt table is updated. One method makes use of the self-programming capabilities of the microprocessor while the other uses the more dynamic approach of an interrupt dispatcher. So these two approaches offer a trade-off between switching and runtime overhead.

Bibliography

[1] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, September 2003.

[2] Hui Dai, Michael Neufeld, and Richard Han. Elf: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

[3] Adam Dunkels, Bjrn Grnvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, November 2004.

[4] Eidgenössische Technische Hochschule Zürich (ETH). Btnode - a distributed environment for prototyping ad hoc networks. Web page. Visited 2005-12-06, http://www.btnode.ethz.ch.

[5] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. Sos: A dynamic operating system for sensor nodes. In *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, June 2005.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) 2000*, November 2000.

[7] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[8] Crossbow Technology Inc. Mica2. Web page. Visited 2005-12-06, http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[9] Crossbow Technology Inc. *Mote In-Network Programming User Reference*, 2003.

[10] Olaf Landsiedel, Klaus Wehrle, and Stefan Götz. Accurate prediction of power consumption in sensor networks. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005.

[11] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, July 1944.

[12] D. Marquardt. An algorithm for the least-squares estimation of nonlinear parameters. *SIAM Journal of Applied Mathematics*, 11(2):431–441, June 1963.

[13] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner-Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

Kurzzusammenfassung

In den letzten Jahren ist aufgrund des technischen Fortschritts die neue Geräteklasse der Sensorknoten entstanden. Davon existieren mittlerweile viele unterschiedliche Varianten, die jedoch alle folgenden primären Eigenschaften aufweisen: Neben einem programmierbaren Mikrocontroller ist mindestens eine Form der drahtlosen Kommunikation vorhanden sowie Sensoren oder zumindest Anschlüsse für diese. Gespeist werden Geräte dieser Art üblicherweise mit einem Batteriepack. Sensorknoten werden beispielsweise in der Biologie oder im Hoch-/Tiefbau verwendet, um in einem größeren Gebiet über einen langen Zeitraum hinweg Messungen aufzunehmen, weiterzuleiten und zeitnah zur Verfügung zu stellen. Die dabei auftretenden Probleme im Routingbereich waren schon Gegenstand intensiver Forschung.

Die Laufzeit eines Sensorknotenprojektes reicht typischerweise von mehreren Monaten bis hin zu einigen Jahren. Die Tatsache, dass in Projekten mit Tausenden dieser Knoten ein Aufladen bzw. ein Auswechseln der Stromversorgung teilweise unmöglich ist, stellt harte Anforderungen an die Designer solcher Systeme, sowohl in Bezug auf Hard- als auch Software.

Eine Anwendung muss deshalb auf den gewünschten Energieverbrauch hin zugeschnitten werden, der sich aus der spezifizierten Laufzeit ergibt. Das geschieht heutzutage mit Hilfe von Anwendungssimulationen. Deren Aktivitätsaufzeichnungen ergeben in Kombination mit einem detaillierten Energiemodell des verwendeten Sensorknotens ein genaues Bild über den zu erwartenden Energieverbrauch. Dies funktioniert mit den verwendeten deterministischen Anwendungen relativ zuverlässig.

Dieser Ansatz hat natürlich seine Grenzen. Beispielsweise kommt es desöfteren vor, dass bereits nach wenigen Wochen nach Aussetzen der Sensorknoten einige Parameter geändert werden müssen. Wird die auf den Knoten laufende Anwendung mehrmals geändert, ist der Prozess der Anpassung an den gewünschten Energieverbrauch ebenfalls mehrmals zu durchlaufen, was diese Methode schnell mühevoll macht. Außerdem sind nichtdeterministische Anwendungen durchaus vorstellbar, z.B. wenn Umwelteinflüsse in Form von Messwerten den Ablauf der Anwendung beeinflussen.

Für ähnliche Situationen gibt es im PC-Bereich adaptives Powermanagement, das es dem Benutzer erlaubt, die Leistungsaufnahme seines Systems nach Belieben zu drosseln. In dieser Arbeit

wird am Beispiel der BtNodes untersucht, inwieweit sich diese Ziele auf Sensorknoten verwirklichen lassen. Dazu ist es notwendig, sowohl den Energieverbrauch als auch den -vorrat zu kennen, und auf zu kleine Extrapolationsergebnisse der Laufzeit oder auf geänderte Anforderungen reagieren zu können. Dementsprechend gliedert sich diese Arbeit in zwei Hauptteile, den Strategien zur Energieabschätzung und Methoden der Anwendungsmodifikation.

Energieverbrauch und -vorrat

Die vorgestellten Methoden zur Bestimmung des Energieverbrauchs orientieren sich an bereits existierenden und für Simulationen benutzten Energiemodellen. Im Unterschied zu diesen wird jedoch der Energieverbrauch erst während der Laufzeit abgeschätzt.

Zur Abschätzung der Restkapazität des Batteriepacks werden mehrere Ansätze vorgestellt. Zuerst wird versucht, ein einfaches Modell der verwendeten Batterien zu erstellen. Das Modell, dessen Entladungskurve Ähnlichkeiten mit der der Batterien aufweist, ist jedoch schon zu kompliziert, um in kurzer Zeit gelöst zu werden.

Ein weitaus pragmatischer Ansatz ist, eine vorgegebene Funktion direkt an die Entladungskurve anzugleichen, um dann aufgrund dieser Funktion die Restkapazität abzuschätzen. Dabei ist zu berücksichtigen, dass der hierfür verwendete Algorithmus auf den Sensorknoten laufen muss und deswegen kein zu hoher Aufwand betrieben werden darf. Aus diesem Grund basiert die vorgestellte Methode auf dem relativ einfachen Verfahren der Regressionsrechnung. Um einfache Funktionen wie beispielsweise lineare oder quadratische Funktionen an passende Teile der Entladungskurve anzugleichen, ist diese Methode durchaus geeignet. Andere Teile der Entladungskurve jedoch können durch Regression nur schwer angenähert werden, wie der Versuch zeigt, eine Exponentialfunktion mit zusätzlicher linearer Verschiebung an den Beginn der Kurve anzupassen. Der dafür benötigte Rechenaufwand ist zwar beträchtlich, aber die Nichteignung wird erst deutlich durch die inhärente „Zerbrechlichkeit“, d.h. die Methode liefert schon bei Daten, die mit deutlich weniger Fehlern als die Daten von der BtNode behaftet sind, keine sinnvollen Parameter mehr.

Eine Abschätzung der Restlaufzeit ist dennoch möglich, durch Verwendung von Spezialbatterien mit fast linearer Entladungskurve oder durch eine Beschränkung auf einen oder wenige Batteriemodelle, deren Entladungseigenschaften dann in Form expliziter Graphen auf einem Sensorknoten gespeichert werden können.

Rekonfiguration

Zum anderen großen Bereich der Arbeit, der Software Rekonfiguration auf Sensorknoten, werden ebenfalls mehrere Verfahren vorgestellt. Es besteht beispielsweise die Möglichkeit, entscheidende Parameter der Applikation zu ändern. Diese Modifikation kann, wie heutzutage verbreitet, manuell

berechnet und dem Sensornetz eingespeist, oder aber vom Knoten selbst durchgeführt werden. Für Aufwandsabschätzungen können wegen der hochgradigen Abhängigkeit zur verwendeten Applikation keine sinnvollen Werte angegeben werden.

Ein anderes Verfahren, das bereits erfolgreich eingesetzt wird, ist, eine neue Applikation über das Sensornetz zu verbreiten und die Knoten durch Überschreiben des Programmcodes zu veranlassen, auf die neue Anwendung zu wechseln. Hier existieren einige unterschiedliche Varianten, angefangen von komplett übertragenen Applikationen bis hin zu Interpretersprachen, die es erlauben, einen Applikationswechsel durch Austauschen weniger Bytes zu vollziehen.

Eine bisher wenig beachtete Methode besteht darin, auf einem Sensorknoten von Anfang an mehrere Applikationen unterzubringen, und bei Bedarf unter diesen umzuschalten. Hierzu muss vor allem darauf geachtet werden, die Interrupttabelle anzupassen. Dafür bieten sich zwei Möglichkeiten an: Entweder wird die Tabelle komplett überschrieben oder durch einen „Verteiler“ersetzt, der die ankommenden Interrupts an die gerade aktive Applikation weiterreicht. Bei ersterer Variante fällt nur beim Überschreiben zusätzlicher Aufwand an, während bei letzterer ein sehr viel geringerer Aufwand bei jeder Interruptbehandlung geleistet werden muss.

Ausblick

Verbesserungen der in dieser Arbeit vorgestellten Methoden werden vor allem im Bereich der Batterieabschätzung erwartet. Hier könnte die Konzentration auf Methoden helfen, die auf den ersten Blick aufwendiger aussehen, sich aber besser für die Anwendung auf Sensorknoten eignen. Eine wichtige Voraussetzung hierfür besteht jedoch in der Möglichkeit, genaue Messwerte der Batterien zu erhalten.