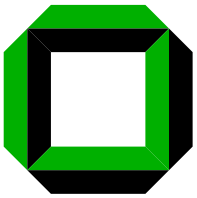

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Design and Implementation of Energy Containers in TinyOS

Sören Finster

Diploma Thesis

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Simon Kellner

July 14, 2008

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, July 14, 2008

Sören Finster

Abstract

With the advent of database interfaces to sensor nets, a demand for online energy accounting emerged. Information about the energy consumption of queries can be used to optimize them for energy efficiency or to bill the issuer of the query.
In this thesis, the concept of Resource Containers is adapted to the requirements of sensor networks. The designed and implemented system provides application developers with the possibility to assign a control flow to an energy container. The control flow is then tracked by the system and all emerging energy consumption is accumulated in the assigned energy container.
The online energy measurement system employed in this thesis is an early development version of a new system. The provided interface was not yet capable of differentiating energy consumption by hardware component. The developed energy container system therefore provides only limited accuracy. With later versions of the energy measurement system this shortcoming is easily addressable. The developed system meets the specified design goals: low memory consumption, small program memory footprint, portability and easy integration in existing applications.

Zusammenfassung

Mit dem Einsatz von Sensornetzen als datenbankartige Informationsquellen ist der Bedarf an Energieverbrauchsmessungen während des Betriebes entstanden. Informationen über den Energieverbrauch einer Abfrage können dazu benutzt werden, Abfragen energieeffizienter zu gestalten oder die verbrauchte Energie in Rechnung zu stellen.

In dieser Arbeit wird das Konzept der Ressource Container an die Bedürfnisse eines Einsatzes in Sensornetzen angepasst. Mit dem entworfenen und implementierten System können Anwendungsentwickler einem Kontrollfluss einen Energiecontainer zuweisen. Das System verfolgt dann diesen Kontrollfluss und fügt dessen Energieverbrauch dem zugewiesenen Container hinzu.

Die in dieser Arbeit verwendete frühe Version eines neuen Systems zur Energieverbrauchsmessung stellt noch nicht die nötigen Schnittstellen bereit um eine genaue Zuweisung der Energieverbrauchswerte von Hardwarekomponenten zu ermöglichen. Das hier entwickelte System bietet daher nur eine begrenzte Genauigkeit, die mit späteren Versionen des Messsystems deutlich verbessert werden kann.

Das hier entwickelte System erfüllt die aufgestellten Design-Ziele: geringer Speicherverbrauch, geringe Codegröße, Portierbarkeit und einfache Integration in bestehende Anwendungen.

Contents

1. Introduction

In today's sensor networks, energy is the most critical resource. A comparison of a standard battery pack to the node it supports shows that it not only outweighs the node by far but is also much bigger in its dimensions.

There are approaches to alternative energy supplies like solar power, but most of them cannot produce energy at a constant rate through day and night. There is always the need for an energy storage, which adds considerably to the size and weight of a sensor node.

If a single sensor node inside a sensor network runs out of energy, the whole network is at risk. A premature depleted node likely had a greater workload than other nodes in the network. If this higher load was crucial for the survival of the network – like being the only interconnect between two groups of sensors – the whole network could be rendered useless although most nodes have enough energy left to continue their service.

Network lifetime can be extended by the detection of energy consumption hot spots and the distribution of workload to less strained nodes. The detection of such hot spots in energy networks needs online energy accounting on sensor nodes.

1.1 Sensor Networks as Data Providers

Most sensor network applications are programmed for a very specific purpose. But there are projects which change the idea of a sensor network from a single purpose device to a multi-purpose data provider. This is the scenario of the ZeuS project [zeu] which provides the context for this diploma thesis.

A sensor network is seen as a cluster of nodes in which every node can participate by providing computing capabilities, sensing capabilities or routing capabilities. Most

often all nodes in the network can provide all those services and the decision which services of which node get used depends on the locality of the node.

Such a cluster is seen as an entity which accepts queries and returns the results. Queries are written in a special purpose query language which enables the distribution of queries not only in location but also in time. A query could, e.g. consist of one temperature measurement per second on all nodes for the duration of one minute. The desired result would then be the average temperature in the sensor network. Since not all nodes are necessarily involved in a query, and queries can also be distributed over time, it is possible to have several simultaneous queries in the network.

This model changes the purpose of single nodes. They no longer have exact predefined instructions but receive their instructions through queries.

The owner of such a sensor network has a big interest in the energy consumption of queries. Since energy is a valuable resource it is important to track its consumption for billing or optimizing a query. But since multiple queries can be active on a single node at the same time, the ability to annotate a query with its induced energy consumption becomes vital.

1.2 Energy Containers

In this thesis, an extension to TinyOS, the standard operating system in wireless sensor networks, is introduced. This extension enables tracking of control flow inside a sensor node. Energy consumption induced by a control flow can be assigned to it.

It uses a new operating system abstraction called energy containers which collect information about consumed energy. An application can create such a container and all further energy consumption is accumulated in it. The container contents can be queried and so, an application could include the energy consumption of the processing of a query in its response.

2. Related Work

Since the field of online energy accounting on sensor nodes is very new, there is only few related work. Therefore, most work referenced in this chapter is related to resource containers and online energy accounting on desktop hardware.

2.1 Resource Containers

Resource containers, which were intruduced in [BDM99] are an operating system abstraction which provides fine-grained resource management in server systems. In this concept, focus is set on fair sharing of available resources among server processes. The notion resources is understood as classical operating system resources like cpu cycles, disk activity or network usage.

A resource container in the spirit of this work is a logical entity which contains all system resources which are used by an application. Attached to a container are also attributes which are used for scheduling, like limitations to resource usage or quality of service parameters. Scheduling is not only changed to respect the attributes of resource containers but it is also suggested that cpu scheduling could happen with resource containers as schedulable entity.

Although resource containers are an important source of inspiration for energy containers, the concepts are very different. Resource containers rely on an operating system with strong abstractions of the CPU (e.g. threads) and well defined barriers between user space and kernel space. In sensor networks, operating systems are often not much more than a thin layer between hardware and user applications. Particularly abstraction of the CPU is not common in such operating systems.

2.2 Energy as Resource

In [Bel01] online energy accounting on commodity hardware was introduced. This made energy a first class operating system resource like cpu time or network usage. Information about energy consumption of specific threads is obtained by evaluating the contents of performance counters integrated in CPUs.

The inclusion of the newly available information in the resource container concept was suggested in this work and later implemented in [Wai03]. In this diploma thesis, resource containers in combination with online energy accounting are used to extend an operating system so that not only the consumed energy could be accounted to its source but also restrictions on the consumption of energy could be enforced. The enforcement employs throttling entities that exceed their limit.

Energy containers differ from this concept in the concentration on monitoring of consumed energy. Energy containers are neither used to manage energy resources nor to enforce limitations.

2.3 Online Energy Accounting on Sensor Nodes

The techniques for online energy accounting on server hardware cannot be used for online energy accounting on sensor nodes. The CPUs on sensor nodes are very basic and do not support performance counters. But due to the simplicity of sensor network hardware, other possibilities for online energy accounting emerge.

In [DOTH07] a first approach towards online energy accounting on sensor nodes is made. In this work, the energy estimation is based on the fact that hardware components are continously switched on and off. The time a hardware component spent online in conjunction with the average current draw of that component is used to estimate the consumed energy. This approach assumes that hardware components consume constant energy while they are switched on.

In [KB07] an accounting infrastructure for sensor nodes is introduced which considers hardware components with multiple states of different energy consumption characteristics. This work already outlines a connection to resource containers on sensor nodes. An idea which is further discussed in [Kel07] which provides the ideas realized in this thesis.

3. Background Information

In this chapter I will give a brief introduction of TinyOS, its component-based architecture and the used programming language. With a strong background in TinyOS version 2 it is safe to skip this chapter. However I recommend reading it for a better understanding of the design decisions later on. A thorough introduction to programming TinyOS can be found in [Lev06].

3.1 TinyOS

TinyOS is a popular open-source operating system for wireless embedded sensor networks. It includes an extensive library of components which enables developers to rapidly design and implement sensor network applications without the initial effort of writing device drivers or network stacks.

As its name implies TinyOS is a minimal operating system. The application in sensor networks enforces severe memory and code-size constraints which is why it bears very little resemblance with commodity operating systems. Since most sensor network hardware does not support advanced memory concepts like virtual memory, there is no distinction between kernel-space and user-space. TinyOS uses a component-based architecture to compensate for these missing features and hides this one address-space approach from the developer.

3.2 NesC

TinyOS is programmed in a dialect of C called nesC [GLvB+03]. It is focused on development of software for sensor networks.

NesC was custom-built for the development of TinyOS. The tight coupling between programming language and operating system prevents distinction between operating system and programming language features.

The nesC compiler produces one file with C code from several nesC source files. The C code is then compiled by a GCC[1] version for the used platform.

3.3 Component-based Architecture

The actual assembly of a sensor network node was the inspiration for TinyOS's component-based architecture. The basic idea was to abstract functionality into an independent unit and connect this unit as black box via well defined interfaces to other units, much like actual hardware chips on sensor nodes are wired to other chips onboard. The initial abstraction level of hardware chips soon evolved into a finer-grained abstraction. In todays TinyOS even single general purpose I/O-pins are represented as components.

The source code files of TinyOS and TinyOS-applications can be classified into four different categories: interfaces, modules, configurations and header files. A component is represented either by a module or by a configuration.

The source file naming convention changed from the last letter determining its type in TinyOS 1 to the last letter determining the nature of a file in TinyOS 2. A file ending on "P" is considered private and a file ending on "C" is considered public.

Interfaces define the wiring possibilities between two components. They are completely independet from components and are defined in seperate files. TinyOS differentiates between two types of communication between components: commands and events. When a component declares to use a specific interface, it can call the defined commands and must respond to the defined events. On the other hand, if a component provides an interface, it has to implement commands and can signal events.

In listing 3.1 an example interface is given. A component which uses this interface can call the read command and has to implement the readDone function which is called with the result as argument when it is available.

This interruption of control flow is called a split-phase interface and plays an important role in the development of energy containers. Although both, commands and events, have return values and data can be transferred by them, this practise is discouraged and return values are mainly used for status indications.

[1] An open-source C compiler

```
interface Read {
  command error_t read();

  event void readDone(error_t result, val_t val);
}
```

Listing 3.1: A Simplified read interface

When a module is defined, used interfaces have to be listed in the module section of the file. The necessary function implementations are then defined in the implementation section. Every function is prefixed by the name of the interface it is defined in. If a component uses more than one interface of the same type, it can assign them different names in the module section and further on refer to them with their new name. In listing 3.2 an example module is given. It uses the interface *Boot*[2] to get an event when the sensor node finished its boot process.

Variables which are defined outside of event or command implementations, like **tmp**[3] in the example, are local to the component: they can be accessed from every function in that file but not from other components.

Configuration files are used to wire components together via their interfaces. In this example case, the component uses two interfaces and therefore two wirings have to appear in a corresponding configuration. A configuration file itself is also a component and therefore can use and provide interfaces which must be stated in the configuration section. The implementation section then specifies which components are used and how they get wired together.

A configuration file for the exemplary component given in listing 3.2 can be seen in listing 3.3. It neither uses nor provides interfaces which is why the configuration section is empty. It refers to three components: `ExampleP`[4], a component which provides the Boot interface called `MainC` and a component which provides the *Read* interface called `ReadImplementorC`. The two interfaces of `ExampleP` get wired to the corresponding interfaces of the other components via the `->` operator.

Configuration files instantiate components by referring to them. Normal components are singleton objects. If a configuration file refers to them at one point, all further appearance in configuration files is without effect since the component is already

[2]Interfaces are further on written in italic script

[3]Code excerpts are written in bold type

[4]Names of components are written in a typewriter font

```
module ExampleP {
  uses interface Read;
  uses interface Boot;
}

implementation {

  val_t tmp;

  event void Boot.booted() {
    call Read.read();
  }

  event void Read.readDone(error_t result, val_t val) {
    tmp = val;
  }
}
```

Listing 3.2: The ExampleP component

```
configuration ExampleC
{
}
implementation
{
  components MainC, ExampleP, ReadImplementorC;

  ExampleP.Boot -> MainC.Boot;
  ExampleP.Read -> ReadImplementorC.Read;
}
```

Listing 3.3: The ExampleC component

included for compilation. The wirings though are handled as usual and establish connections. Exceptions to this scheme are generic components which are covered in section 3.4.

Header files are C header files and mostly contain only preprocessor directives for constants or macros.

3.4 Generic Components

With TinyOS 2 the limitations of the component-based architecture with singleton components was addressed and an extension called generic components was introduced. This extension is comparable to classes in object-oriented programming languages. A generic component can be instantiated multiple times and every instance is independent of the other instances. This enabled the development of helper components like `BitVectorC` or `QueueC` which provide universally usable implementations of common problems.

The development of generic components differs only slightly from normal components. The only differences are the keyword **generic** before the **module** or **configuration** keyword and the possibility to pass arguments to the instantiation. A simple generic component can be seen in listing 3.4.

When a generic component is used in configuration files, its name is prepended with the **new** keyword. The newly generated component is only addressable in the configuration file which instantiated it. Although the nesC compiler just copies the source of the generic component and generates a normal component with a unique name, that name is not known prior to compilation.

This is of importance to the development of energy containers since often a generic component is subject to extension or modification. Then the problem is to make sure that added interfaces are accessible from outside the component. This involves the modification of the configuration file which instantiates the generic component.

3.5 Parameterized Interfaces

It is often necessary that multiple components wire to one interface of a specific component. This is, for example, the case with the component which implements the control for the onboard LEDs. When a component wants to switch a specific LED on or off, it uses the `LedsC` component. When multiple components want to do that, they all wire to `LedsC` and since the *Leds* interface only includes commands it is possible that they all share the same interface. But if the interface contained

```
generic module ToggleFlipflopC() {
  provides interface ToggleFlipflop;
}
implementation {
  uint8_t state = 0;

  command uint8_t ToggleFlipflop.state() {
    return state;
  }

  command uint8_t ToggleFlipflop.toggle() {
    if (state == 0) {
      state = 1;
      return 0;
    } else {
      state = 0;
      return 1;
    }
  }
}
```

Listing 3.4: Example of a simple generic component

```
module ParameterizedC {
  provides interface Read[uint8_t id];
}
implementation {
  command error_t Read.read[uint8_t id]() {
    val_t data = sample();

    signal Read.readDone[id](data);
  }
}
```

Listing 3.5: Example of a component providing a parameterized interface

events, `LedsC` could not signal a specific client. All events would get signalled to all components which are wired to the interface. Often, this is not desirable.

When a component should be able to signal specific clients it must provide an interface for each client. But when writing an application, the exact number of clients is seldom known, and the definition of a great number of interfaces is rather awkward. Therefore, nesC supports a concept called parameterized interfaces.

Instead of providing multiple interfaces, a component can provide one parameterized interface. Basically, this means that the same interface is provided multiple times and the selection of the correct one is possible through a parameter. A component providing a parameterized *Read* interface can be seen in listing 3.5.

An interface is parameterized by appending the parameter in brackets. This parameter is then used by configuration files to select a specific instance of the interface. The parameter is made available in function bodies as normal variable and can be used as parameter for the signaling of results.

One remaining problem is how a configuration file can decide which parameter it should use to wire a component to the interface. For this purpose nesC provides a special compile time function called **unique()**. It takes a string as argument and returns a unique integer for every invocation with the same string. Together with **uniqueCount()**, which returns the number of invocations of **unique()** calls for a specific string, these two functions enable the efficient use of parameterized interfaces.

Another problem which arises with the usage of parameterized interfaces is the signaling on unwired interfaces. Since the parameter to the interface can be calculated at runtime, it is possible that an application signals an event on a parameterized interface which is not wired to another component. To solve this problem, a component which provides a parameterized interface must define default handlers for its events. These default handlers are called when an event is signaled on an unwired parameterized interface.

3.6 Tasks

The execution of a command or an event handler cannot be interrupted from another command or event handler. Long running command bodies or event handlers thus delay further execution and should therefore be avoided. If long running code is required, TinyOS provides the concept of a task which can be used in this case.

Tasks are defined like commands but instead of the **command** keyword the **task** keyword is used. Inside the tasks body can be anything which can be in a normal command body. Particularly references to component variables or helper functions defined in the component.

The execution of a task is scheduled via a **post** call with the name of the task as argument. This hands the task over to the scheduler which will start it later. A task itself is not interruptible by another task or by normal commands or events. That is why moving a long running computation inside a task does not solve the problem on its own. The computation must be split up in smaller parts.

In listing 3.6 such a long running computation is wrapped in a task. It approximates the root of a function. The task consists of one step of the process. When the result is good enough, it is returned, otherwise, the task posts itself again and continues the calculation at a later point in time. The idea is that tasks can take turns on the CPU. This concept is comparable to cooperative scheduling.

Tasks have a second important function: the transition between asynchronous context and synchronous context.

3.7 Synchronous and Asynchronous Context

As mentioned earlier, tasks cannot get interrupted by other tasks or normal commands or event handlers. This is because they run in synchronous context, a concept of TinyOS to approach the problem of race conditions.

Code which is marked as asynchronous is always at risk of getting interrupted from other code or from itself. To prevent race conditions, nesC provides the **atomic**

```
task improveIteration() {
  tmp = function(left + (right-left) / 2);
  if (tmp > 0) {
    right = tmp;
  } else {
    left = tmp;
  }
  if (goodEnough(tmp)) {
        process(tmp);
  } else {
        post improveIteration();
  }
}
```

Listing 3.6: Example of a long running computation in a task

statement which guarantees that access to the variables in the enclosed code happen atomically. It does not guarantee that the atomic sections get not interrupted, it just promises, that the contained data operations cannot get flawed through race conditions.

All interrupts and low-level hardware events happen in asynchronous context. To prevent propagation of the risk of race conditions into synchronous code, asynchronous code is not allowed to call synchronous code. This forms a kind of one-way barrier since synchronous code is allowed to call asynchronous code. The only way to get through that barrier from the asynchronous side is the posting of a task. This operation is marked asynchronous and causes a later execution of synchronous code.

Therefore every low-level software which happens to need interrupts has to define a task which it uses to propagate data into synchronous context.

This concept enables a programming environment in which the developer can program easily without keeping race conditions in mind. But it also poses the risk of introducing long delays through excessive event handlers or commands. Most developers are used to preemptive schedulers, so this concept usually needs some time to get used to. To ease this, a thread concept which does not rely on cooperative scheduling is currently in development.

3.8 Abstraction Layers

TinyOS employs a hardware abstraction architecture to support different platforms and provide portable applications at the same time. It uses a three-layer design to gradually adapt the hardware capabilities to platform independent interfaces.

The lowest layer is formed by the Hardware Presentation Layer (HPL) which is tightly coupled to the hardware. HPL components just provide an abstraction on language level. They offer an interface which exposes all hardware capabilities. As HPL components are stateless they can do only a bare minimum of hardware interaction like clearing flags and copying single values.

The Hardware Adaption Layer (HAL) is the middle layer. In contrast to HPL components HAL components are allowed to keep state. They use the simple interfaces provided by HPL components to provide the best possible abstraction of the specific hardware. HAL components should provide the complete feature set of the hardware.

The top of the hadware abstraction architecture is formed by the Hardware Interface Layer (HIL). In this layer, platform dependent abstractions from the HAL are converted in platform independent abstractions. In this process hardware features can get lost and HIL components should only provide the typical hardware services needed in sensor network applications.

A more thorough discussion of the hardware abstraction architecture of TinyOS can be found in [HPH+].

4. The Problem of Energy Containers in TinyOS

In this thesis I will provide the design and an implementation of the energy container concept in TinyOS as discussed in [Kel07].

The energy container concept enables the application developer to precisely track energy consumption within a sensor node in terms of high level abstractions like query processing. As soon as a query arrives at a sensor node the energy container implementation can track energy consumption of all activities which are triggered on behalf of the query. After query processing is completed and the application is ready to send out results, a short call to the energy container implementation will provide a precise account of energy consumed during the whole query processing. Application developers are not responsible to track activities on the sensor node and assign them to queries. Once set up, the energy container implementation will follow the control flow of an application and relate future activities to the correct sources.

Energy containers rely on an accurate online energy accounting infrastructure. A very promising approach for TinyOS is described in [KB07].

As of now, measuring energy consumption of sensor networks needed additional hardware and was time-consuming and tedious. Hence most often only one sensor node was measured and used for hand-optimizing the energy consumption of an application. This approach prevented a large scale survey of power requirements of sensor networks and made it difficult to relate energy consumption to certain activities on a sensor node. Furthermore the collected data needed processing and interpretation and therefore did not provide immediate feedback. This resulted in slower development cycles.

With online energy accounting and energy containers it is possible to accurately relate energy consumption to activities within a sensor node and to get a bird's eye view on the energy condition of a whole sensor network. With immediate feedback on energy demands of queries, the energy container concept enables a fast development cycle for query design. The amount of available data also enables detection of interdependencies between nodes.

But also per-node application development benefits from Energy Containers. The accurate reports on energy consumption can reveal bugs and ineffecient implementations more easily since energy consumption of certain routines can be monitored independently from other activities on a sensor node.

The addition of online energy accounting and energy containers adds a valuable instrument to the development toolchain of sensor network applications.

4.1 Hardware specific Issues

The design and implementation in this thesis concentrates on a high-level implementation of energy containers which is portable to arbitrary platforms running TinyOS. The unavoidable platform specific decisions target the MICAz platform [Cro] which is one of the most common platforms for developing sensor network applications with TinyOS. The provided implementation is also done for the MICAz.

The hardware capabilities of this platform are limited by the processor which is an ATmega128L manufactured by ATMEL [ATM]. It is a RISC processor with four kilobytes of random access memory and 128 kilobytes of in-system flash program memory. Its clock frequency is software adjustable up to eight MHz.

The limitation of random access memory and the lack of memory virtualization create an environment in which the occupation of every single byte of random access memory must be well thought out. But also the restriction of program memory, although less severe, must be considered.

4.2 Operating System specific Issues

TinyOS is programmed in a dialect of C called nesC [GLvB+03] which was custombuild for the development of TinyOS. Many concepts of TinyOS are realized with special features of the nesC programming language which cannot be found in standard programming languages.

The tight coupling between language and operating system complicates understanding and extending the operating system.

A case in which the restriction to nesC causes difficulties in Energy Container development is with the anonymity of generic components which is outlined in section 3.4.

This thesis is about the extension of TinyOS with the energy container concept. Therefore, modifications of the fundamental concepts of the operating system are out of scope of this work.

5. Designing Energy Containers

The design process of energy containers for TinyOS crossed several abstraction layers from application programming interface down to device drivers. I start this chapter with a short overview of the goals I had in mind while working on the design. This is followed by a section about the general architecture of the designed energy container system. Then I will elaborate on the different design issues in a top-down approach starting with the application programming interface.

5.1 Design Goals

The design of energy containers in TinyOS is not only about the general issues to get a functional implementation but also about the usability for application developers and the maintainability for operating system developers.

General Issues

Since memory is spare on sensor nodes the implementation should be lightweight in memory consumption. Data structures should use as little memory as possible and unavoidable static memory allocations should be configurable via global parameters. Therefore the application developer can fine tune memory consumption with his deeper knowledge of the application.

Another big constraint in sensor nodes is computing power. Especially in time critical applications the limited speed of the main processor poses the risk of lateness. Additional code can prolong reaction times and make the development of such applications much harder. Since there are also calls to the energy container implementation from device drivers and interrupts the calls should have an upper limit in execution time.

When these design goals are reached the additional energy consumption through the inclusion of energy containers should already be at its minimum. Nevertheless it is important to keep an eye on the introduced energy overhead.

Application Developers View

To get not only a theoretical solution to this problem but also a practical one which proves itself in application it is important to provide easy access to the energy container infrastructure.

It should be possible to introduce the energy container concept into an existing project without any problems. The necessary changes to the codebase should be small.

Given that developers will want to test their applications without using energy containers but with the TinyOS installation which will be used later, the implementation should support the exclusion of energy containers and their overhead without changing the whole application.

OS Developers View

The implementation should be as platform independent as possible. Complete platform independence is not feasible since some low level interaction with device drivers is needed to monitor spontaneous energy consumption. Yet the higher level concepts should retain platform independence where possible to ease porting the implementation to new hardware platforms. Since other platforms may require a different approach towards energy measurement the implementation of energy containers should be independent from the underlying energy measurement component. Therefore it must not use implementation specific information but limit itself to the usage of the provided interface.

5.2 General Architecture

An implementation of an energy container concept fulfills two main functions. It tracks the control flow of the application to correctly correlate the energy consumption with its causes and it manages and accounts the records about the consumed energy.

To separate the development of those different tasks I chose to use a layered approach to the problem. The top layer is responsible for accounting and management and the lower layer implements control flow tracking. Because of easier portability the lower layer itself is divided in two layers which results in an overall three-layer architecture depicted in figure 5.1.

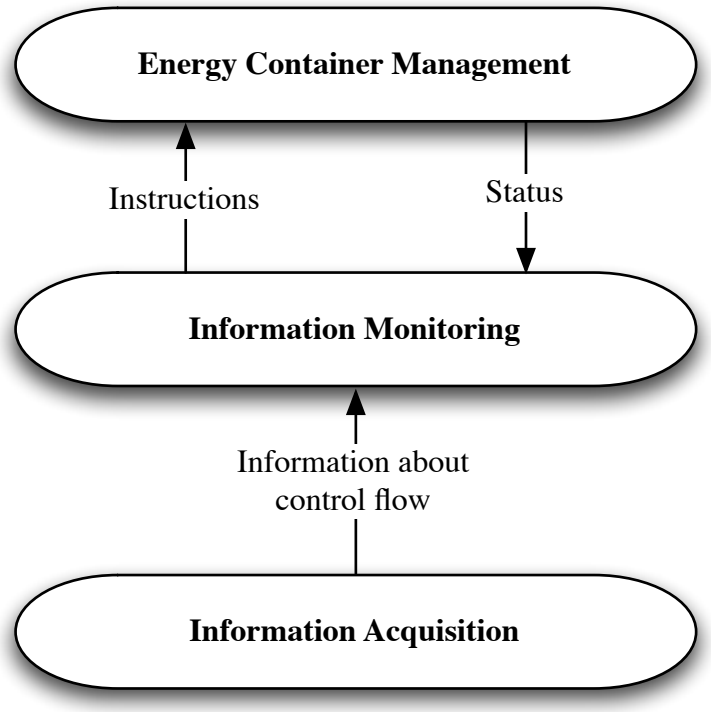

Figure 5.1: Architecture of an energy container system

To track the control flow of an application correctly it is necessary to get notified when the application is interrupted or interrupts itself and also to get notified when it will start processing again. This task is accomplished by the lowest layer of the architecture – the Information Acquisition Layer. Since the desired information is in most cases only available in the components which provide the non-linear behavior this layer is inside of theses components. This makes the Information Acquisition Layer the most hardware dependent layer and includes the modification of standard TinyOS components.

The split of the control flow tracking task in two layers allows to minimize the modifications to standard components. Since the Information Monitoring Layer is responsible for data interpretation, the Information Acquisition Layer only collects it. This results in the insertion of hooks inside of standard components which hand the required information out to the Information Monitoring Layer. Thus component specific modifications are kept at a minimum and should seldom exceed a few additional lines. Due to the minimal and non-intrusive nature of these modifications I call them probes.

The Information Monitoring Layer consists of several components of which each communicates with an installed probe via a specific interface. There must not necessarily be exactly one monitoring component per probe. It is possible to accumulate

several probes into one monitoring component, however, the very specific nature of the collected data suggest a similarly specific monitoring logic.

The monitoring components interpret the provided information and decide what needs to be saved for later usage and what can be safely discarded. Since this decision is based on the node-wide status of the energy container system, the monitoring components can query the energy container management for, e.g. the currently active energy container.

The Energy Container Management layer is mostly concerned with the high-level management of energy containers. It is not provided with information from the underlying layers but only accepts instructions about changes to the node-wide status.

5.3 Application Programming Interface

The central component of the energy container implementation is the `ECManagerC` component. By referencing this component in a configuration file the application developer causes the inclusion of the complete energy container infrastructure. `ECManagerC` provides the interface listed in listing 5.1.

```
interface ECManagement {
  command ec_id startMonitoring();
  command void startMonitoringOn(ec_id id);
  command energy_t getCurrentContainer();
  command ec_id getCurrentContainerId();
  command energy_t getContainer(ec_id id);
  command void switchToContainer(ec_id id);
  command void attachToContainer(ec_id id);
  command void stopMonitoring(ec_id id);
  command void emptyContainer(ec_id id);
}
```

Listing 5.1: ECManagement interface definition

It is notable that the interface to the *ECManagement* interface only includes commands and therefore is not split-phase like most TinyOS interfaces. This is necessary since the expected usage of energy containers in applications is reduced to two calls which need immediate results:

The first call should set up the environment to track the control flow and account the consumed energy to an energy container. The second call is made when the

application reaches a state where the main work is done. This call returns the amount of energy accounted on the current energy container. This energy bill can then be sent back to the principal along with the results of the query. The commands in question are **startMonitoring()** and **getCurrentContainer()**. After that, the used container can be discarded by calling **stopMonitoring()** or reused through the **startMonitoringOn()** command.

This approach is very easy and unobtrusive when the requests return immediately but would get difficult if a split-phase interface was used. There would not be a significant performance penalty in using a split-phase interface but application development would be much harder.

The additional commands in *ECManagement* provide applications with a more sophisticated use of the energy container concept. Although an application could use those commands to work more sophisticated with energy containers the two aforementioned commands should suffice for most areas of application. They relieve the application developer completely from caring about energy containers and their data structures and enable him to concentrate on application development.

5.4 Energy Container Management

The management of energy containers is done completely inside of one component. This component offers the application programming interface and also provides the interface needed from the underlying Information Monitoring Layer. Although the monitoring layer consists of multiple components, the management just offers one generalized interface to them.

The management component is responsible for the allocation and maintenance of energy containers. It would be possible to allocate energy containers dynamically but in general the number of needed energy containers should be closely estimatable by the application developer. In this design the energy container management allocates static memory for a fixed number of energy containers. How much energy containers are allocated is configurable via a global constant.

In addition to the general purpose energy containers, the management component also maintains a root container which measures the complete energy used by the sensor node. This enables some rough estimations about battery life and sensor network durability.

5.5 Information Monitoring

The Information Monitoring Layer contains most of the logic involved in control flow tracking. When an application is about to hand off its control of the processor

it usually takes measures to get the control back at later point in time. These measures are for example the programming of a timer or the posting of a task. The components in the Information Monitoring Layer get notified about those measures and must make sure that they can reestablish the currently active energy container when control is handed back to the application.

In most cases components which notify the monitoring components provide an identifier with which the monitoring components can associate the currently active energy container. When the application is about to get restarted, the monitoring components get notified again with the same identifier. So they can look up this identifier and set the corresponding energy container active.

For performance reasons the needed memory for the identifier / energy container relation is statically allocated where possible. This saves the burden of dynamic memory management and since the number of identifiers is available at compile time in most cases, no memory is unnecessarily allocated.

5.6 Information Acquisition

The information about the control flow is collected via small probes which get inserted in standard TinyOS components or reside in special proxy components if the insertion in standard components is problematic. This can happen if the components in question are hardware dependent and hardware independence is possible without much overhead or if the modification of standard components would get too complicated and difficult to maintain.

6. Implementation

In contrast to the design chapter, the implementation chapter is structured in a bottom-up approach.

At first I will explain an implementation technique which allows easy conditional integration of probes into the build process and the interface of the energy measurement system.

I continue with the implementation of probes in standard TinyOS components and the corresponding monitoring components. Since the implementation is very specific to the component in which the probe is inserted, I will provide three distinctive approaches. This covers the timer subsystem, the scheduler and device drivers.

After that the implementation of the energy container management layer is covered.

6.1 Conditional Integration of Probes

The overhead introduced into standard TinyOS components by the insertion of probes should be negligible and even not existent when the energy container infrastructure is not used by the application. This would be possible through conditional compilation which would require modifications to the build system and the usage of command-line options.

Instead of that, I chose to use an implementation technique which relies on the optimization capabilities of GCC. This means that the code of the probes always gets translated by the nesC compiler but can be optimized away by GCC when the probes are not used.

In listing 6.8 on page 48 the interface between a probe and the corresponding monitoring component can be seen. This interface is provided by the probe and uses only

events to deliver information. As with parameterized interfaces it is also possible to define default event handlers with normal interfaces. By defining those default handlers for every event the probe can deliver, it is no longer necessary to wire the monitoring component to the probe. As can be seen in listing 6.7 (page 47) all default events are empty. In the resulting C source, the nesC compiler still inserts calls to the interface functions but since they have no body, GCC optimizes them away completely. After optimization by GCC there is no overhead in keeping the probes in the standard components.

6.2 Interfacing the Energy Measurement System

The energy measurement system used in this thesis is an early development version of the system proposed in [KB07]. It offers the function **sampleComp()** which returns the energy spent since the last call to this function.

The used version of the energy measurement system implements this function only in synchronous context. This shortcoming is mitigated in later versions of the system.

6.3 Information Acquisition and Management in the Scheduler

In TinyOS, tasks are the main environment for application specific code. For most applications it is not necessary to implement routines which execute in asynchronous context.

Tasks are always defined in context of a component and they can only be posted from inside of the component they are defined in. These limitations result from how tasks are implemented in TinyOS:

When a component defines a task via the **task** keyword the provided name is used to extend the list of interfaces of the component with a *TaskBasic* interface with that name. The task's body is used for the body of the event handler for the **runTask** event. The newly added *TaskBasic* interface is then wired to `TinySchedulerC` using a **unique()** call. Every occurrence of **post** in the component is then exchanged with a call to the **postTask** command of the corresponding *TaskBasic* interface. This transformation can be seen in listing 6.2 which is the equivalent to the normal task definition in listing 6.1.

The implementation of tasks differs significantly from common approaches. It is optimized for performance and low memory consumption. To reach these goals it sacrifices most of the common features of task concepts. There is no representation

```
task improveIteration() {
  tmp = function(left + (right-left) / 2);
  if (tmp > 0) {
    right = tmp;
  } else {
    left = tmp;
  }
  if (goodEnough(tmp)) {
      process(tmp);
  } else {
      post improveIteration();
  }
}
```

Listing 6.1: Task definition before transformation

```
module { uses interface TaskBasic as improveIteration }
...
event void improveIteration.runTask() {
  tmp = function(left + (right-left) / 2);
  if (tmp > 0) {
    right = tmp;
  } else {
    left = tmp;
  }
  if (goodEnough(tmp)) {
    process(tmp);
  } else {
    call improveIteration.postTask();
  }
}
```

Listing 6.2: Task definition after transformation

of a task other than the id of the parameterized interface to which the tasks body is connected. Tasks do not get interrupted by other tasks or the scheduler. They have to give up control of the CPU by themselves and repost themselves if there is more work to be done.

The duty of the scheduler is to record calls to **postTask** using the parameter of the interface and then signal a **runTask** event on the same interface when the task is at the head of the queue. This can be done with a minimum of memory usage because the scheduler knows in advance how many tasks can be posted. The memory consumption of the standard TinyOS scheduler is only one byte per task and two bytes constant.

The scheduler acts as gateway between asynchronous and synchronous code, therefore **postTask** is defined as **async**. In its body, synchronous commands cannot be called. The **postTask** command uses an atomic section to insert the new task into the queue. The task is then called by the taskloop which is in synchronous context.

It is important to note that the scheduler can run out of ready tasks and then puts the CPU in a sleep mode. This event must be propagated to the management component since the accounting for the energy consumption during sleep modes must be handled separately.

Extending the standard scheduler with probes is straightforward but there are two specific features which must be handled. The scheduler does not only start tasks on its own behalf but also has to provide the command **runNextTask(bool sleep)** which is mainly used while initializing the mote or during simulation on a desktop computer. If there is a next task, this command always runs it and returns **TRUE**. If there is no task in the queue the **sleep** parameter determines if the scheduler should wait until a task is available or return **FALSE**. Since this command is used during the boot process and it is not possible to initialize the energy container system before **runNextTask()** is called for the first time, it cannot be monitored. If it was monitored, the calls made during the initialization process would operate on uninitialized memory.

The omission of this command is no problem since it is not used anymore during normal operation.

The second feature is the asynchronous characteristic of **postTask**. Since this command is defined as asynchronous it is not possible to leave asynchronous context without posting a task. The consequences are that all work which must be done at the posting of a task has to be asynchronous. This propagates up to the monitoring component and from there, up to the energy container management component.

```
interface SchedulerMonitor {
  event void startingTask(uint8_t id);
  event void goingIdle();

  async event void queueingTask(uint8_t id);
}
```

Listing 6.3: The SchedulerMonitor interface

```
command void Scheduler.taskLoop() {
  for (;;) {
    uint8_t nextTask;
    atomic {
      while ((nextTask = popTask()) == NO_TASK) {
        signal SchedulerMonitor.goingIdle();
        call McuSleep.sleep();
      }
    }
    signal SchedulerMonitor.startingTask(nextTask);
    signal TaskBasic.runTask[nextTask]();
  }
}
```

Listing 6.4: The modified scheduler loop

Since only the currently active energy container id is needed to establish a correct relation, only the function to fetch this value has to be made asynchronous.

The interface between the probe and its monitoring component is called *SchedulerMonitor* and can be seen in listing 6.3.

An excerpt of the modified scheduler taskloop can be seen in listing 6.4. Only two lines with signals to the *SchedulerMonitor* interface were added.

The monitoring component for the task probe uses the same technique the scheduler uses to reduce its memory consumption. The maximum number of tasks in the system is acquired through the use of **uniqueCount()** and then used to allocate an

array of the correct size. The index of this array reflects the id of a task while the contents at this index reflect the associated energy container id.

In listing 6.5 the source code of the monitoring component can be seen.

The implementation of the **goingIdle** event is of specific interest. It ensures that no energy container gets charged for idle energy consumption. Before changing the active energy container, this function checks if another monitoring component set a flag to prevent interference with hardware activity. This problem is further addressed in sections 6.5 and 6.6.

6.4 Information Acquisition and Management in the Timer Subsystem

Apart from tasks, timers are the second most used source of control flow interruption.

Timers are usually programmed via a call to the **startPeriodic** or **startOneShot** command. Both take as parameter a time interval the timer waits until it fires. By using the timer interface, an application has to implement the **fired** event which gets called when the timer fires. This call happens in a synchronous context since the underlying timer implementation posts a task which signals the event. Therefore it is not possible to interrupt a **fired** event with another one. This is especially important for periodic timers since this means that a time-demanding handling of the **fired** event could postpone the next **fired** event. This leads to a periodic timer with an effective cycle period of the time it takes to handle the **fired** event. Since the rearrangement of resource container contexts can only take place between the hardware signal and the event handling by the application, it is desirable to keep that overhead as low as possible.

From a higher perspective the task concept bears resemblance with the timer concept. Both defer code execution until later, although timers provide the possibility to exactly define the point in time while the task concept just guarantees that it will happen. A call to **postTask** is equivalent to **startOneShot** with the difference that **startOneShot** accepts a parameter which specifies the time it should wait until it signals a **fired** event.

Currently all platforms of TinyOS use a single hardware timer and virtualize it to support arbitrary numbers of timers. The interface presented to the application developer consists of a generic component (`TimerMilliC`) which just wires the provided timer interface via a **unique()** call to the parameterized timer interface of the underlying `TimerMilliP` component. `TimerMilliP` takes care of initialization and wiring of `HilTimerMilliC` which is, as the name suggests, in the hardware interface layer

```
#include <energy_container.h>
module SchedulerMonitorP {
  provides interface Init;
  uses interface ECControl;
  uses interface SchedulerMonitor;
}
implementation {
  uint8_t task_ec[uniqueCount("TinySchedulerC.TaskBasic")];
  command error_t Init.init() {
    uint8_t size = uniqueCount("TinySchedulerC.TaskBasic");
    for (uint8_t i=0; i < size; i++) {
      task_ec[i] = NO_CONTAINER;
    }
    return SUCCESS;
  }

  event void
  SchedulerMonitor.startingTask(uint8_t id) {
    call ECControl.setActiveEC(task_ec[id]);
  }

  async event void
  SchedulerMonitor.queueingTask(uint8_t id) {
    atomic {
      task_ec[id] = call ECControl.getActiveEC();
    }
  }

  event void
  SchedulerMonitor.goingIdle() {
    if (! ECControl.hwActivity())
      call ECControl.setActiveEC(NO_CONTAINER);
  }
}
```

Listing 6.5: Implementation of the task monitoring component

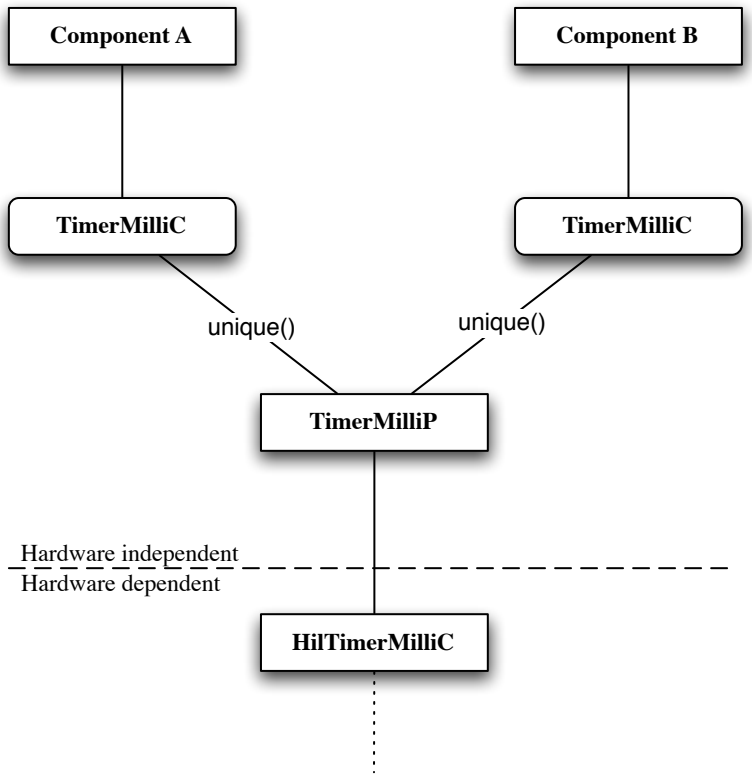

Figure 6.1: Architecture of the timer subsystem

of the hardware abstraction architecture. Therefore it is not platform independent. However, below `HilTimerMilliC`, all current platforms use the same component, provided by TinyOS, to virtualize the single hardware timer: `VirtualizeTimerC`. Although this component is used from a platform dependent component, it is platform independent. This component is the actual target for timer related commands and the source of all timer related events. It is the topmost component in the timer architecture hierarchy which actually contains an implementation in nesC.

An illustration of this dependency tree can be seen in figure 6.1. In this and future illustrations, generic components are shown as boxes with rounded corners.

This `VirtualizeTimerC` component would be the right place to insert a probe for monitoring timer activity. Commands and events have their endpoints in it and it would be easy to add a signal in the implementations of **startPeriodic** and **startOneShot**. Signaling out a firing event to the monitoring component could be accomplished by altering the task which gets posted when a timer fires. This implementation would function correctly for all current platforms of TinyOS and inflict a minimum in overhead. But if a developer of a new platform chooses to not use the `VirtualizeTimerC` component to virtualize the hardware timers, the resource container architecture would be broken. Platform developers are free in this decision since `HilTimerMilliC` and `BusyWaitMicroC` are the only components which must

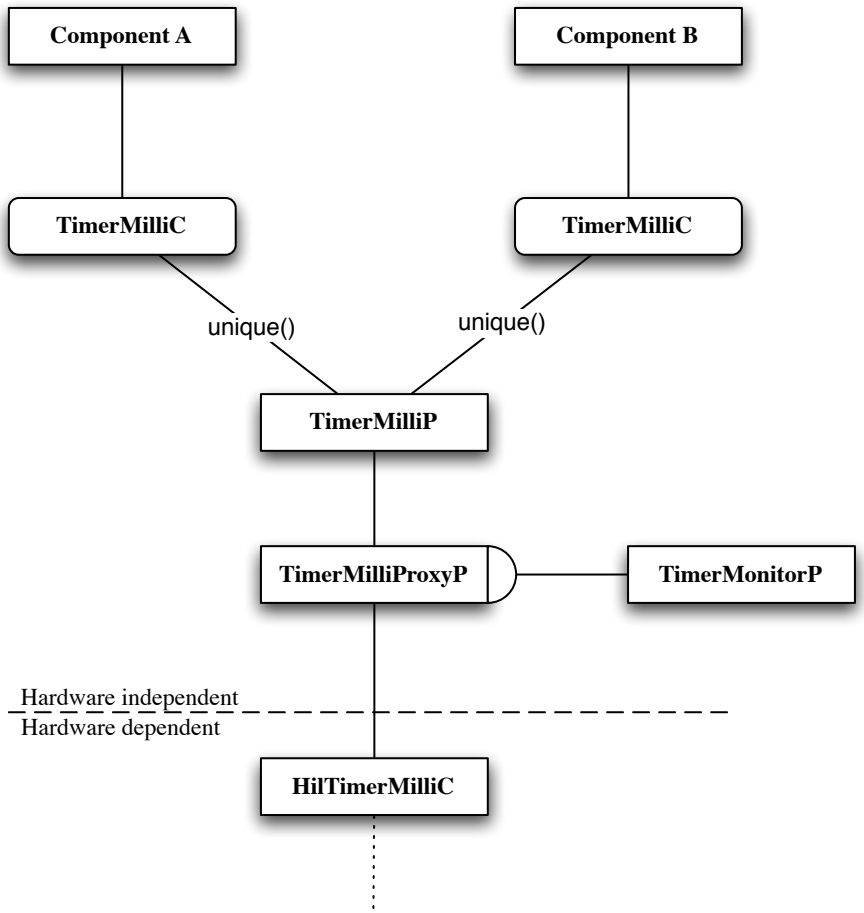

Figure 6.2: Architecture of the timer subsystem with proxy component

be implemented by every platform. How they are implemented is not dictated by TinyOS.

Therefore, probes must be installed on a higher, platform independent level. Since all other components in the timer architecture only wire to underlying components, there is no spot to install probes in a hardware independent way without adding components. The solution to this is a proxy component which fully implements a parameterized timer interface and gets placed between `TimerMilliP` and `HilTimerMilliC`. This concept is illustrated in figure 6.2.

This involves a change in `TimerMilliP` which now instantiates not only `HilTimerMilliC` but also `TimerMilliProxyP`. It keeps initializing `HilTimerMilliC` but now wires its provided parameterized timer interface to `TimerMilliProxyP` and the parameterized timer interface used by the proxy component to `HilTimerMilliC`. Thus `TimerMilliProxyP` acts as a transparent proxy and just adds signals which get sent out through the *TimerMonitor* interface (cf. listing 6.6, 6.7 and 6.8).

```
configuration TimerMilliP {
  provides interface Timer<TMilli>
    as TimerMilli[uint8_t id];
}
implementation {
  components HilTimerMilliC, TimerMilliProxyP, MainC;
  MainC.SoftwareInit -> HilTimerMilliC;
  TimerMilli = TimerMilliProxyP;
  TimerMilliProxyP.SubTimerMilli
    -> HilTimerMilliC.TimerMilli;
}
```

Listing 6.6: TimerMilliP instantiates and wires TimerMilliProxyP

The important events in the timer subsystem are the programming and the firing of a timer. It is not necessary to distinguish if a programmed timer is a one-shot or periodic timer. Also the stop command is not needed to be signaled out since no reaction to this event is required. A binding which was established during the programming of the timer would just remain unused and would get overwritten when the timer is programmed again.

The changes to the TinyOS code base for installing the probes in the timer architecture boil down to altering the `TimerMilliP` configuration and adding the `TimerMilliProxyP` component.

The very simple interface to the monitoring component suggests an equally simple implementation of that component. For each timer which gets programmed, the monitoring component has to remember the active energy container at that instant. Later, when the timer fires, it has to switch back to that energy container. Since no other synchronous activity can get interrupted, the management component needs not to save the current energy container before it switches to the correct one.

The memory needs for saving the correlations between timers and energy containers can be estimated at compile time. Every timer which gets instantiated via the `TimerMilliC` component wires to a parameterized interface via a **unique()** call to the string stored in the compiler macro **UQ_TIMER_MILLI**. As can be seen in listing 6.9 the relation is saved via an array of bytes which has an entry for every instantiated timer.

```
module TimerMilliProxyP {
  provides interface Timer<TMilli>
    as TimerMilli[uint8_t id];
  uses interface Timer<TMilli>
    as SubTimerMilli[uint8_t id];
  provides interface TimerMonitor;
}

implementation {
  command void
  TimerMilli.startPeriodic[uint8_t id](uint32_t dt) {
        signal TimerMonitor.startedTimer(id);
        call SubTimerMilli.startPeriodic[id](dt);
  }
  event void
  SubTimerMilli.fired[uint8_t id]() {
        signal TimerMonitor.firingTimer(id);
        signal TimerMilli.fired[id]();
  }
  default event void
  TimerMonitor.startedTimer(uint8_t id) {}
  default event void
  TimerMonitor.firingTimer(uint8_t id) {}

  // ...
}
```

Listing 6.7: Excerpt of the implementation of TimerMilliProxyP

```
interface TimerMonitor {
  event void startedTimer(uint8_t id);
  event void firingTimer(uint8_t id);
}
```

Listing 6.8: The interface provided by the timer probe

```
#include <energy_container.h>
module TimerMonitorP {
  provides interface Init;
  uses interface ECControl;
  uses interface TimerMonitor;
}
implementation {
  uint8_t timer_rc[uniqueCount(UQ_TIMER_MILLI)];

  command error_t Init.init() {
    uint8_t size = uniqueCount(UQ_TIMER_MILLI);
    for (uint8_t i = 0; i < size; i++) {
      timer_rc[i] = NO_CONTAINER;
    }
    return SUCCESS;
  }

  event void TimerMonitor.startedTimer(uint8_t id) {
        timer_rc[id] = call ECControl.getActiveEC();
  }
  event void TimerMonitor.firingTimer(uint8_t id) {
        call ECControl.setActiveEC(timer_rc[id]);
  }
}
```

Listing 6.9: Implementation of the timer monitoring component

Coverage of Energy Consumption

It is important to note that by monitoring the task system through the energy container system, all synchronously executed code is covered. The inclusion of the timer subsystem, which is more or less a device driver, covers the most important source of control flow interruption. Since synchronous code accounts for most of an applications code basis this means that the CPU-bound energy consumption of an application is accurately accounted.

6.5 Information Acquisition and Management in Device Drivers

The accuracy of probes in device drivers is limited to non-concurrent hardware activity due to the interface which is used between energy container management and energy measurement. It is currently not possible to get detailed information about the energy consumption of a specific hardware component. This is problematic when multiple components consume energy at the same time. This can happen, e.g., when a message is sent over the radio while a light sensor samples data. The energy container implementation can neither know how much energy was used by the radio chip nor how much was used by the light sensor. Only the sum of both is available. Therefore it is not possible to account energy consumption arising from concurrent hardware activities accurately to their origin.

Hardware activity can be classified in two groups. Processor-induced hardware activity and processor-independent hardware activity.

As the name suggests, processor-independent activity can occur without initiation by the processor. The independent hardware can notify the processor of the activity, but it is also possible, that the processor is completely unaware of the ongoing hardware activity. An example for processor-independent hardware activity is the wireless chip, an essential part of any wireless sensor node. Wireless chips listen on a channel and react if they detect a message for the node.

Processor-induced activity is characterized by a tight coupling of external hardware activity with processor activity. The processor initiates hardware activity and it happens promptly after that. The temperature sensor is an example for processor-induced activity. The thermistor is connected to the analog/digital converters of the processor. A reading of the sensor promptly causes a short hardware activity.

Processor induced hardware activity can be monitored by the current energy container system. As an example of the probe insertion process, an implementation for the temperature sensor of the basicsb sensor board driver is outlined below.

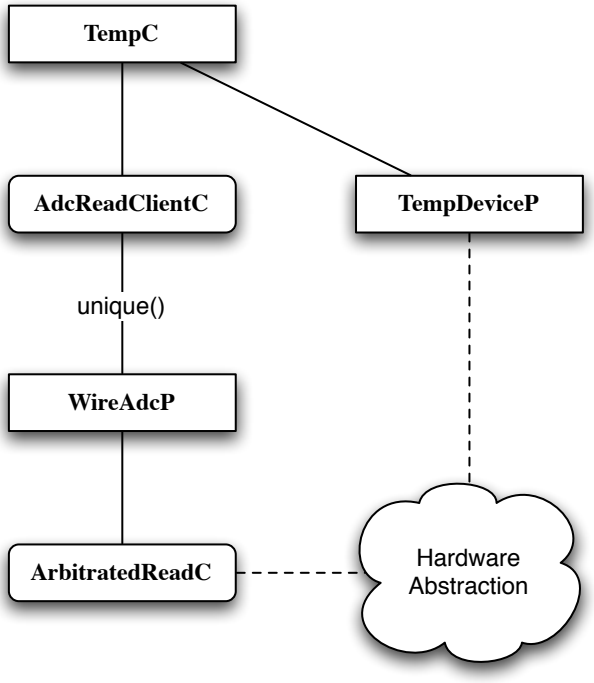

Figure 6.3: An excerpt of the dependency tree of TempC

Accounting Processor Induced Hardware Activity

For a reliable probe it is necessary to find the component which fits best for probe insertion. This process is complicated by the heavy usage of configuration files in TinyOS. Since probes need real implementations and cannot get inserted into configurations, it is often necessary to track down several layers of abstraction until a component is found which satifies the requirements.

Examining `TempC` yields that it offers a *Read* interface which gets wired to a new instance of `AdcReadClientC`. It also references a component called `TempDeviceP` which offers the hardware abstraction of the sensor and wires two interfaces of `AdcReadClientC` to it. Since the **read** command should be intercepted it is reasonable to follow the path of the call. `AdcReadClientC` wires the *Read* interface to the parameterized *Read* interface of `WireAdcP` via a **unique()** call. A look into `WireAdcP` shows that it hands *Read* over to a new instance of `ArbitratedReadC` which actually contains an implementation of the *Read* interface. An illustration of these dependencies can be seen in figure 6.3.

`ArbitratedReadC` implements parts of the resource arbitration protocol [KLG+] which allows multiple software components shared access to a hardware resource.

This component is a very good candidate for probe insertion since this component knows when its managed resource is requested, granted and released. These events

```
command error_t Read.read[uint8_t client]() {
  signal ReadMonitor.request(client);
  return call Resource.request[client]();
}
event void Resource.granted[uint8_t client]() {
  signal ReadMonitor.granted(client);
  call Service.read[client]();
}
event void Service.readDone[uint8_t client]
                  (error_t result, width_t data) {
  signal ReadMonitor.readDone(client);
  call Resource.release[client]();
  signal ReadMonitor.returning(client);
  signal Read.readDone[client](result, data);
}
```

Listing 6.10: Excerpts from ArbitratedReadC

are important for the probe, since at the request of the resource, a binding between the request and the currently active energy container has to be established. When the resource is granted to the client later on, the energy container which was active during the request has to be reactivated since now the actual access to the hardware will happen.

Excerpts from a modified `ArbitratedReadC` can be seen in listing 6.10. The **readDone** event is of particular interest. There are two signals out in this event because the **release** call to the resource causes a posting of a task in a lower layer arbiter. By signaling the monitoring component that this will happen it can take countermeasures to prevent that the resulting task is associated with the current energy container. After the resource is released, the correct energy container has to be reactivated since now the processing of the acquired data is triggered in the application.

With the insertion of the probe into `AdcReadClientC`, the problem of how the probe can be wired to a corresponding management component arises. Since `ArbitratedReadC` is a generic component, it is not possible to refer to it outside of the instantiating configuration file. The configuration file in question is in this case `WireAdcP`. A wiring of the instance of `ArbitratedReadC` to its monitoring compo-

```
configuration WireAdcP {
  provides {
    interface Read<uint16_t>[uint8_t client];
    interface ReadNow<uint16_t>[uint8_t client];
    interface ReadMonitor;
  }
  uses {
    interface Atm128AdcConfig[uint8_t client];
    interface Resource[uint8_t client];
  }
}
implementation {
  components Atm128AdcC, AdcP,
    new ArbitratedReadC(uint16_t) as ArbitrateRead;
  Read = ArbitrateRead;
  ReadNow = AdcP;
  ReadMonitor=ArbitrateRead;
  Resource = ArbitrateRead.Resource;
  Atm128AdcConfig = AdcP;
  ArbitrateRead.Service -> AdcP.Read;
  AdcP.Atm128AdcSingle -> Atm128AdcC;
}
```

Listing 6.11: The modified WireAdcP component

nent by `WireAdcP` violates the design goals of the implementation technique specified in section 6.1. It requires that probes do not wire to their management component but only offer an interface for them. This technique demands that probes can be installed only in normal, not-generic, components because they are statically addressable. In this specific case it is possible to use `WireAdcP` for this purpose. After modification, `WireAdcP` offers the probe interface and wires it to the newly instantiated `ArbitratedReadC`. This can be seen in listing 6.11.

The modification of `WireAdcP` must be examined thoroughly since it is a system component which can be used in several places. Indeed, the probe implementation for the temperature sensor does not only cover this sensor but also implements effective covering of all sensors which use an instance of `AdcReadClientC` to arbitrate

their **read** calls. This includes the photodiode on the same sensor board but also the voltage sensor on the mica2 platform and several other sensors. This shows the elegance of this approach since a complete line of sensors is covered with a rather low effort.

Currently, monitoring components for device drivers must prevent the task monitoring component from switching to **NO_CONTAINER** when the CPU is put into sleep mode. External hardware can still consume energy and information about this would be lost otherwise. For that purpose the energy container management component provides a flag which is used to signal other monitoring components that a hardware device is expected to consume energy and therefore the currently active energy container should persist. Otherwise, the monitoring component for the probe implemented above is comparable to other monitoring components.

This procedure will be rendered obsolete through the tighter integration of device driver probes into the energy measurement system proposed in section 8.1.

The inclusion of a probe in a generic component can complicate the implementation of the corresponding monitoring component. In this case, only `WireAdcP` instantiates `ArbitratedReadC` which simplifies the process. Since the wirings to the parameterized interface of `WireAdcP` use a call to **unique()**, the parameter to this call can be used with **uniqueCount()** to get the maximum number of clients. Therefore, the memory consumption of the monitoring component is static.

As already stated, accounting the energy consumption of the wireless chip is currently not possible. However, the energy spent in the protocol stack of TinyOS is accountable with the current interface.

Particularly, the energy which is consumed in the short period between a message arrives in the stack and is delivered to the application is of interest. Since this energy consumption happens without advance knowledge it is called spontaneous energy consumption.

Accounting Spontaneous Energy Consumption

Most of the protocol stack of TinyOS is kept in synchronous context. Due to that, the sending of a message is already covered. When an application prepares a message and hands it over to the protocol stack, all further computation is accounted to the applications energy container.

The energy consumption of the protocol stack while receiving a message is more complicated. Since the receiver of a message is not known when the message is handed over from the hardware, the energy spent while processing the raw data

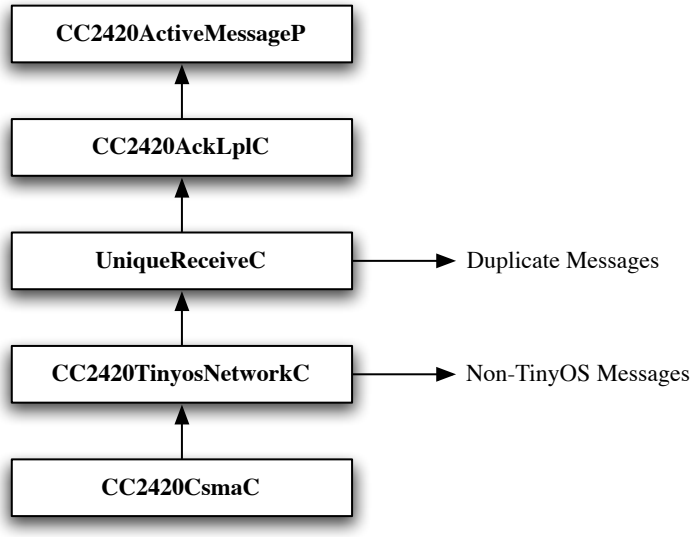

Figure 6.4: The receive part of the protocol stack of TinyOS

cannot be accounted to an energy container. However, it is possible to account the energy consumed by the protocol stack to a new energy container which can be claimed by an application, later on.

All further processing of the message is then accounted to the new energy container. This includes the **receive** event which is triggered when the message is delivered. If an application is already monitoring its energy consumption, and the energy which is consumed by receiving the message should be added to an already existing energy container, it can call the **attachToContainer(ec_id id)** command with the already existing energy container as argument.

If an application wants to start monitoring from the reception of a message, no action is required. All further control flow will be accounted as regular.

Since a new energy container is generated with each received message, an application must take care of them. If a container is not attached or used later on, **stopMonitoring** must be called to prevent the energy container management system from running out of energy containers.

For the insertion of the probe, the protcol stack for the receiving of a message must be examined. It is shown in figure 6.4.

The topmost component which handles messages in synchronous context is a component which decides if received data is TinyOS network compatible or if it originated from other sources than TinyOS (`CC2420TinyosNetworkP`). The signal of the probe is inserted before the message is passed up in the protocol stack. Due to the simplicity of this modification, a source code listing is omitted.

```
module NetworkMonitorP {
  uses interface ECControl;
  uses interface NetworkMonitor;
}
implementation {
  event void NetworkMonitor.receive() {
    call ECControl.startMonitoring();
  }

  event void NetworkMonitor.discard() {
    ec_id activeEC = call ECControl.getActiveEC();
    call ECControl.stopMonitoring(activeEC);
  }
}
```

Listing 6.12: The monitoring component for the protocol stack

Another important point in the protocol stack is `UniqueReceiveC` which discards duplicate messages. If applications are interested in duplicate messages, they can wire to a special *DuplicateReceive* interface. Only if this interface is unwired, the monitoring should stop. Therefore, a probe is inserted into the default implementation of this interface.

The monitoring component for this driver differs from the other monitoring components since its duty is not to save an energy container/control flow relation but to always set up a new container when the probe in `CC2420TinyosNetworkP` signals a new packet. If the probe in `UniqueReceiveP` signals that the message was discarded, the monitoring is stopped. Its source code can be seen in listing 6.12.

6.6 Energy Container Management

From an application developers view the energy container system is represented by the `ECManagerC` configuration file which provides the *ECManagement* interface as API and *ECControl* for monitoring components. `ECManagerC` can be seen in listing 6.13.

`ECManagerC` uses a `PoolC` component to statically allocate a fixed number of structures provided by the energy measurement system. This structure represents an

```
#include <energy.h>
#include <energy_container.h>

configuration ECManagerC {
  provides interface ECManagement;
}
implementation {
  components MainC, ECManagerP;
  components new PoolC(struct energy, MAX_CONTAINERS)
                    as Pool;
  MainC.SoftwareInit -> ECManagerP;
  ECManagerP.ECPool -> Pool;

  components TimerMonitorC, SchedulerMonitorC,
             AdcMonitorC, NetworkMonitorC;
  MainC.SoftwareInit -> TimerMonitorC;
  MainC.SoftwareInit -> SchedulerMonitorC;
  MainC.SoftwareInit -> AdcMonitorC;
  TimerMonitorC.ECControl -> ECManagerP;
  SchedulerMonitorC.ECControl -> ECManagerP;
  AdcMonitorC.ECControl -> ECManagerP;
  NetworkMonitorC.ECControl -> ECManagerP;

  components EnergyOpsC, Atm128EnergyInfoC;
  ECManagerP.EOps -> EnergyOpsC;
  ECManagerP.EInfo -> Atm128EnergyInfoC;

  ECManagement = ECManagerP.ECM;
}
```

Listing 6.13: The main configuration file for the EC-System

```
module ECManagerP {
  provides interface ECManagement as ECM;
  provides interface ECControl as ECC;
  provides interface Init;
  uses interface Pool<struct energy> as ECPool;
  uses interface EnergyInfo as EInfo;
  uses interface EnergyOps as EOps;
}
```

Listing 6.14: Interfaces of ECManagerP

amount of energy and can be handled through the use of functions provided by the *EnergyOps* interface. The energy measurement system contains a component (`EnergyOpsC`) which provides this interface.

`ECManagerC` also sets up all monitoring components and wires them to `ECManagerP`. This enables the application developer to include the complete energy container system with the inclusion of `ECManagerC`.

Due to the size of `ECManagerP`, it cannot be printed in its full length. Discussed parts are printed as excerpts.

To shorten the source code of `ECManagerP` some interfaces are renamed. As can be seen in listing 6.14, *ECManagement* is abbreviated to *ECM*, *ECControl* to *ECC*, *EnergyInfo* to *EInfo* and *EnergyOps* to *EOps*.

`ECManagerP` uses an array of pointers to energy structures to represent energy containers. The id of an energy container is the position of the corresponding pointer to an energy struct in this array. This array is initialized with zeros and gets filled when an application starts monitoring. There are two static allocations of energy structs inside `ECManagerP`, **rootC** and **tempC**. The first one represents the root container of the node and contains all energy used by the node. The second one is used in several functions as a scratch container.

The initialization process of `ECManagerP` can be seen in listing 6.15. The last operation in this command is a **sampleComp** call to the energy measurement system to reset energy accounting. Its result is discarded because this is the point where accounting to the root container starts.

One of the central functions of `ECManagerP` is **account()**. It is defined as a utility function because it is used in multiple situations. It is shown in listing 6.16.

```
command error_t Init.init() {
  for (int i = 0; i < MAX_CONTAINERS; i++) {
    containers[i] = 0;
  }
  activeEC = NO_CONTAINER;
  hwActivity = 0;
  call EOps.reset(&rootC);
  call EInfo.sampleComp(&tempC);
  return SUCCESS;
}
```

Listing 6.15: The initialization procedure for ECManagerP

```
void account() {
  call EInfo.sampleComp(&tempC);
  if (activeRC != NO_CONTAINER) {
    call EOps.add(containers[activeRC],
                  containers[activeRC],
                  &tempC);
  }
  call EOps.add(&rootC, &rootC, &tempC);
}
```

Listing 6.16: The account utility function in ECManagerP

```
ec_id prepareNewContainer() {
  ec_id free_index = findFreeIndex();
  containers[free_index] = call ECPool.get();
  call EOps.reset(containers[free_index]);
  return free_index;
}
```

Listing 6.17: The prepareNewContainer utility function in ECManagerP

In **account()**, **tempC** is used to obtain the energy consumption since the last call. If an energy container is currently active, this energy consumption is added to it. In any case, the energy consumption is added to the global variable **rootC** which represents the energy consumption of the whole node.

The definition of **account()** as utility function simplifies important commands like **setActiveEC()** which is reduced to a call to **account()** and the setting of **activeEC** to its new value.

Due to similarities in the *ECManagement* and *ECControl* interfaces, there is some duplicated code in `ECManagerP`. The implementation of **setActiveEC()** for the *ECControl* interface is the same as **switchToContainer()** in *ECManagement*. Although this could be avoided through the use of another utility function, the brevity of those commands renders that unnecessary.

When an application or a monitoring component calls the **startMonitoring** command, a new energy container is prepared via the utility function **prepareNewContainer** (listing 6.17). Before this new container is set as the active energy container, a call to **account** is necessary to restart sampling.

The **stopMonitoring** command checks if the specified container is the currently active container and, if this is the case, sets the active container to **NO_CONTAINER**. After that it returns the energy struct into the pool and zeroes its array entry. A call to **account** is not necessary since, if the stopped container was the active energy container, the pending energy consumption is lost anyway. If the stopped container was not the active container, there is no influence to the energy accounting which is currently in progress.

The **switchToContainer** command is very simple in its implementation. It just calls **account** and then changes the active energy container to the specified one.

7. Evaluation and Discussion

In this chapter, the energy container system is evaluated with regard to random access memory consumption and computational overhead. The following evaluation is based on source code. Due to time constraints, only very few tests were possible. Their resulst are listed in the summary below. After that the implementation is discussed with regard to the design goals.

7.1 Overhead Estimation

Through the use of the implementation technique described in section 6.1, an unused energy container system does neither consume any memory nor does it have any computational overhead. The following evaluation discusses the memory consumption and the computational overhead of an active energy container system. The values exclude the energy measurement system.

Energy Container Management

Memory consumption of the energy container management component is mostly due to the number of pre-allocated energy containers which is adjustable via a header file. For each pre-allocated energy container, the energy container management component needs a pointer to it. The size of a pointer on the MICAz platform is two bytes. The `PoolC` component needs a pointer array to the actual structs, too. This adds up to an additional memory consumption of four bytes per pre-allocated energy container.

Independent from the number of pre-allocated energy containers, the energy container management component allocates two static energy containers and two bytes in variables. The `PoolC` component adds another two bytes to that.

The size of an energy struct is currently four bytes. This adds up to an overall memory consumption of 12 bytes constant and 8 bytes per pre-allocated energy container.

The computational overhead of the energy management component is very low. All operations on energy containers should not consume more than a few cycles and due to the expected small number of pre-allocated energy containers, procedures with a linear overhead in the number of energy containers should also have very little impact.

Monitoring Components

Memory consumption of monitoring components is linear with the number of relations they have to store. In case of the task monitoring component this is the maximum number of tasks in the system. For the timer monitoring component this is the number of timers used by an application. The monitoring component for spontaneous energy in the network stack is an exception to this. It does not use any memory.

All others use one byte per relation.

Since the introduced monitoring components only look up a value in a static array, the computational overhead is negligible.

Probes

Due to their simple design, probes neither consume memory nor introduce computational overhead.

Summary

An energy container system does not add significantly to the memory consumption of an application.

This statement is backed by memory consumption information provided by the nesC compiler. A small example application was written to examine the memory consumption of the energy container system. All discussed monitorable concepts were used (timers, tasks, temperature sensor and wireless network). The number if pre-allocated energy containers was set to five.

After extension with energy containers, the application consumed an additional 164 bytes of RAM and 1,990 bytes of program memory. This accounts for approximately 4% of the available RAM and 1.5% of the available program memory. These numbers include the memory consumption of the energy measurement system.

The energy container system alone – without energy measurement system – uses 74 bytes of RAM (1.8%) and 574 bytes of program memory (0.4%).

The computational overhead introduced by the energy container system is also negligible. But together with the computational overhead of the underlying energy measurement system and the fact that overhead is introduced in crucial system functions, this overhead can have effects on time-critical applications. The investigation of the extents of those effects is left for future work.

7.2 Discussion

This discussion is based on the design goals proposed in chapter 5.

The requirement of a lightweight implementation was met. The memory consumption of the energy container system is very low and for the most significant part adjustable via global configuration parameters.

Calls to the energy container system do not induce big computational overhead. However, in future work measurements should be made to prove that.

To extend an application with energy container support, very few changes to the source code are needed. Application developers only have to include one component which provides an easy to use interface.

With the exception of probes in device drivers, the energy container system is platform independent. It is also independent from the underlying energy measurement system although a tighter coupling is needed to monitor hardware activity (see section 8.1).

8. Future Work

Apart from the work which can be done with a richer interface to the energy measurement system, a suggestion for extending the energy container system is provided.

8.1 Tighter Coupling with the Energy Measurement System

The limitations of the energy container system described in this thesis are due to the limited interface to the energy measurement system. Future versions of the energy measurement system will provide a richer interface and therefore allow a tighter integration with the energy container system.

With support by the energy measurement system for queries about the energy consumption of specific hardware, accurate accounting is possible.

Monitoring components could use the knowledge about the energy consumption of the hardware they are monitoring to account that consumption to the container which was active when the hardware activity was induced. Even when that container is not active while the hardware activity happens.

Only small changes to the existing system would be necessary to accomplish this task. Foremost, the monitoring components need access to the energy measurement system and more privileges in energy container handling.

8.2 Energy Container Hierarchy

In this thesis, energy containers are not structured in a hierarchy. The root container is completely independent from other containers and specially treated by the energy container system.

The introduction of a hierarchy would not only remove this special treatment but also provide application developers with fine-grained measurement possibilites. An application developer could use long running energy containers to get additional information besides the direct, per-query information.

With great efforts from the application developer, the current implementation is capable of a simulation of an energy container hierarchy. However, an integration of this principle in the system could be useful.

9. Conclusions

In this thesis I described how to design and implement an energy container system for the TinyOS operating system for wireless sensor nodes. The design decisions were driven by the limitations of the target hardware platform and the goal of a system with a high usability.

A control flow in an application can be associated with an energy container. All energy consumption induced from that control flow is accounted to its energy container.

The design is based on a three-layered approach. The tasks of control flow tracking, the processing of data about control flow and the management of energy containers are partitioned in three layers.

Control flow is tracked via probes which are inserted in control flow interrupting components. The data sampled by those probes is sent to monitoring components which register the interruption and save the current energy container environment. On continuation of the control flow, the environment is reestablished and further action is accounted to the correct energy container.

Energy consumption values are provided by an early development version of an online energy measurement system for sensor nodes. Unfortunately, the interface between energy measurement and energy container system was not sufficient to accurately account all energy consumption on the node. Only accurate accounting of the main processor and some sensors on the board is available.

However, when a more extensive interface to the energy measurement system is available, only small changes to the energy container system are necessary to completely cover the energy consumption of the whole node.

Evaluation of the implemented system showed that it meets the requirements. It features low memory consumption and easy integration in existing applications.

Bibliography

[ATM] ATMEL. *ATmega128L Datasheet*. Published at http://www.atmel.com/dyn/resources/prod_documents/2467S.PDF.

[BDM99] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. *Proceedings of the 1999 USENIX/ACM Symposium on Operating System Design and Implementation*, pages 45–58, 1999.

[Bel01] F. Bellosa. The case for event-driven energy accounting. *Department of Computer Science, University of Erlangen TR-I4-01-07*, 2001.

[Cro] Crossbow Technology. *MICAz Datasheet*. Published at http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[DOTH07] Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 28–32, New York, NY, USA, 2007. ACM.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[HPH+] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, David Culler, and David Gay. Tinyos extension proposal 2: Hardware abstraction architecture. Published at http://tinyos.net/.

[KB07] Simon Kellner and Frank Bellosa. Energy accounting support in tinyos. In *GI/ITG KuVS Fachgespräch Systemsoftware und Energiebewusste Systeme*, pages 17–20. Fakultät für Informatik, Universität Karlsruhe (TH), October 2007.

[Kel07] Simon Kellner. Wip: Energy container for database-oriented sensor networks. In *6. Fachgespräch Sensornetzwerke*, number 2007-11 in AIB, pages 5–7. GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme", Distributed Systems Group, RWTH Aachen University, July 2007.

[KLG+] Kevin Klues, Philip Levis, David Gay, David Culler, and Vlado Handziski. Tinyos extension proposal 108: Resource arbitration. Published at http://tinyos.net/.

[Lev06] Philip Levis. *TinyOS Programming*. 2006. http://csl.stanford.edu/~pal/.

[Wai03] M. Waitz. Accounting and control of power consumption in energy-aware operating systems. Master's thesis, University of Erlangen-Nürnberg, 2003.

[zeu] Zeus, zuverlässige informationsbereitstellung in energiebewussten ubiquitären systemen. Website: http://www.zeus-bw-fit.de.