Power Management in Reflex

André Sieber, Karsten Walther, Stefan Nürnberger, Jörg Nolte
Distributed Systems/Operating Systems group, TU Cottbus
{as, kwalther, snuernbe, jon}@informatik.tu-cottbus.de

Abstract

Energy consumption is a crucial factor for the lifetime of many embedded systems, especially wireless sensor networks. Most modern microcontrollers provide various low power sleep modes. Utilizing them can lead to great energy savings. In this paper we present an approach for power management in embedded systems, based on the event-driven operating system REFLEX. The implicit power management is mostly hardware independent, lightweight and chooses efficiently the optimal power saving mode of the microprocessor automatically.

1 Introduction

Typical sensornet applications such as environmental monitoring demand that sensor nodes should work for month or even years with a single battery. Thus, saving energy is essential to archive this goal. Even in embedded systems with external power supply, saved energy helps to reduce costs. Modern microcontrollers provide the programmer with fine grained control over components and sleep modes. To utilize these features the operating system should at least be able to provide the application with power saving mechanisms. It would be even better to do this implicitly without any need of control from the programmer.

Since most microcontrollers support a variety of sleep modes with different energy footprints, the selection of the mode can have an intense effect on the lifetime of battery powered devices. If the decision was wrong, the energy savings could be marginal or, even worse, events can get lost.

2 Related Work

The most common operating system for wireless sensor nodes is TinyOS[1], it features a wide range of software modules and runs on various platforms. As it is an event-based operating system, the scheduler of TinyOS puts the controller to sleep if the `task-queue` is empty and thus no work has to be done. TinyOS is capable of computing the deepest possible sleep mode autonomously. To do so, all components that change the state of the hardware and might influence the deepest possible sleep mode have to call the `McuPowerState.update()` function, resulting in its recomputation.

The computation is done by reading all device registers. Because the update operation is executed atomically it can produce a significant overhead [2]. Additionally, the function is hardware dependent and must be reimplemented for every platform. The `PowerOverride.lowestState()` function makes it possible for higher level components to influence the chosen sleep mode.

Other event-driven sensor node operating systems like SOS[3] leave the power management to the programmer and do not implement any deepest sleep mode computation. Contiki[4] can not take advantages from any sleep mode because of its polling methodology for interrupt handlers.

In thread based operating systems for deeply embedded systems it is more challenging to determine the possibility of going into a certain sleep mode. The scheduler has to determine if all threads are blocked for some reason (e.g. waiting on I/O) or are idle. Mantis [7] has support for sleep modes. It provides a `mos_thread_sleep()` function, similar to the UNIX sleep(). Every thread has to call it with the desired duration of the sleep period. Mantis only distinguishes between an idle sleep mode and a deeper sleep mode. The first is used if the system waits on IO, the second if all threads have called the `mos_thread_sleep()` function.

3 Power Management in Reflex

REFLEX (**R**eal-time **E**vent **FL**ow **EX**ecutive) is an operating system implemented in C++, targeting deeply embedded systems and wireless sensor nodes. It is based on the so called event flow model presented in [5]. Initial source of all activity in the system are interrupts. In REFLEX so called activities are schedulable entities, which are triggered if something was posted to their associated event buffers.

The power management in REFLEX divides two abstractions, a system view and a user view. The system view is responsible for the determination of the deepest possible sleep mode. The user view provides the programmer with two instruments, namely groups and modes, to ease the handling of all hard and software components.

3.1 System View

The system provides the class `EnergyManageAble`, each class derived from it is concerned by the power management.

Every instance of a component has a variable which specifies the deepest possible sleep mode that may be used when it is active. The power manager contains a table with counters for every available sleep mode of the microcontroller used. If a component is enabled it signals its deepest possible sleep mode to the power manager by increasing the corresponding counter in the sleep mode table. If a component is disabled, the counter of its sleep mode is decreased. If no

event is pending, the scheduler calls the `powerDown()` function. This power manager function iterates the sleep mode table starting at the lightest mode. The first value different from zero is the deepest possible sleep mode. In contrast to TinyOS, it is not necessary to evaluate the complete machine state, which makes the changing of the lowest possible sleep mode very lightweight.

The sleep mode counter table is the only hardware specific part of the power manager, since it can have a different size depending on the microcontroller used.

Since the initial source for activity are the interrupts, they define the deepest possible sleep mode. In general there are two types of interrupts, primary and secondary. The first are caused by external events, the second are a result of software events. E.g. the TX interrupt of a serial connection has only to be active when a send operation is in progress, if it is finished, the driver of the serial connection can deactivate the interrupt and possibly change the deepest possible sleep mode. Thus the power management approach is implicit for secondary interrupts, because the drivers know when a interrupt has to be enabled.

Primary interrupts can not be deactivated implicitly. Their state is determined by the current stage of the application. Sampling of sensors and sleeping for a given time needs different interrupts at different times. The decision which interrupt has to be active must be decided by the application programmer.

3.2 User View

At startup, each manageable object is registered with the power management and assigned to one or more programmer defined groups .

During operation, groups can be independently activated and deactivated. This allows to easily activate or deactivate any number of objects with only one method call. If a manageable object is member of multiple groups it is only deactivated when all of these groups are deactivated.

Modes are defined to switch easily between active groups and thus utilizing different hardware configurations. This makes it possible to divide the execution of an application into different phases, while ensuring that the hardware components are active when demanded. The programmer is responsible for changing the modes. For example a timer driven module can be used for mode changes.

4 Evaluation

We compared our approach with TinyOS 2.0.2 using a simple sense and send application utilizing the implicit power management of both systems running on a TMoteSky[6]. It samples the SHT11 temperature and humidity sensor of the TMoteSky every two seconds, aggregates these values four times and sends them over the serial connection at the beginning of the fifth phase. The results are shown in figure 1 and table 1.

For the given application the results show that the power management of REFLEX is more effective than that of TinyOS. The REFLEX application consumes about 38% respectively 51% less energy, depending on the voltage. The main reason is that the power consumption during sleep is
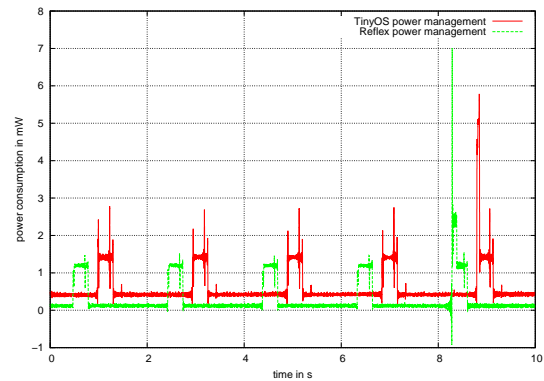

Figure 1. Power consumption of TinyOS and REFLEX **at 1MHz and 3V**

significantly higher in TinyOS than in REFLEX. This is because the serial connection was only used unidirectional, which was accounted for by Reflex, so the driver module could be deactivated when there was no data to transmit. The difference in power consumption comes mainly from the baudrate generator running all the time in the TinyOS application. At the beginning of the send operation there is a higher spike in the reflex curve due to the necessary startup of the baudrate generator.

	Reflex	TinyOS
TMoteSky 2.2V @ 1MHz	0.193	0.316
TMoteSky 3V @ 1MHz	0.281	0.575

Table 1. Average power consumption of TinyOS and RE-FLEX **in mW per second**

Acknowledgments

5 References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In the *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[2] R. Szewczyk, P. Levis, M. Turon, L. Nachman, P. Buonadonna, V. Handziski. TinyOS Microcontroller Power Management Documentation TEP112. Webpage http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html

[3] C.-C. Han, R. S. Ram Kumar, E. Kohler, and M. Srivastava. A dynamic operatingsystem for sensor nodes. *In Proc. of the 3rd international conference on Mobile systems, applications, and services MobiSys*, 2005.

[4] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and exible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, 2004.

[5] K. Walther and J. Nolte. A flexible scheduling framework for deeply embedded systems. In *In Proc. of 4th IEEE International Symposium on Embedded Computing*, 2007.

[6] TMote Sky Datasheet, Moteiv Corperation, Webpage http://www.sentilla.com/moteivtransition.html, 2006.

[7] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. In *ACM/Kluwer Mobile Networks Applications (MONET), Special Issue on Wireless Sensor Networks*, 2005,