

Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Control Engineering

ITEM, TinyOS module for sensor networks

Bachelor Thesis

Author: Pavel Beneš

Supervisor: Ing. Jiří Trdlička

Thesis Due: July 2008

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Pavel Beneš**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **ITEM, TinyOS modul pro senzorové sítě**

Pokyny pro vypracování:


1. Seznamte se s problematikou senzorových sítí, nesC a operačního systému TinyOs.
2. Seznamte se s protokolem E-ASAP a modulem ITEM.
3. Upravte modul ITEM pro operační systém TinyOS 2.0
4. Identifikujte nedostatky modulu ITEM a implementujte novou verzi modulu.
5. Výsledky práce otestujte na jednoduché aplikaci.

Seznam odborné literatury:


Akimitsu Kanzaki , Takahiro Hara , Shojiro Nishio. An adaptive TDMA slot assignment protocol in ad hoc sensor networks. Proceedings of the 2005 ACM symposium on Applied computing, March 13-17, 2005, Santa Fe, New Mexico
TinyOS web pages: <http://www.tinyos.net/>
Inderjit Singh. Integrated TDMA E-ASAP Module (ITEM). FEL-CVUT, Prague, 2007

Vedoucí: Ing. Jiří Trdlička

Platnost zadání: do konce zimního semestru 2008/2009


prof. Ing. Michael Sebek, DrSc.
vedoucí katedry




doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 25. 2. 2008

Declaration

I hereby declare that I have written my bachelor thesis myself and used only the sources (literature, projects, SW etc.) listed in the enclosed references.

In Prague, July 9, 2008

A handwritten signature in cursive script, appearing to read 'Benes', is written above a horizontal dotted line.

Signature

Acknowledgements

I would like to thank the chief of my bachelor thesis Ing. Jiří Trdlička, for his patience, help and suggestions, which were very much appreciated.

I would like to special thank my family for constant support and encouragement during the course of my studies.

Abstrakt

Cílem této práce je úprava existujícího softwarového modulu ITEM pro operační systém TinyOS 2.0. Jde o modul pro bezdrátové senzorové sítě, který implementuje protokoly E-ASAP (Extended-Adaptive Slot Assignment) a TDMA (Time Division Multiple Access). Protokoly zajišťují, že modul je schopný reagovat na změny struktury sítě a starají se o zlepšení využití komunikačního kanálu. Původní modul, navržený pro operační systém TinyOS 1.x se skládá z několika menších modulů. Každý z těchto modulů má svoji specifickou funkci. Tyto moduly jsou na sobě nezávislé a proto je možné upravovat každý modul zvlášť. Nová verze modulu ITEM byla otestována na jednoduché aplikaci.

Abstract

The main aim of this thesis is modification of the existing ITEM software module for the TinyOS 2.0 operating system. It is a module for wireless sensor networks, which implements the E-ASAP (Extended-Adaptive Slot Assignment) and TDMA (Time Division Multiple Access) protocols. These protocols ensure that the module is able to react to network structure changes and improve channel utilization. The first version of the module, which was developed for the TinyOS 1.x operating system, consists of several smaller modules. Each of those modules has its own specific function. These modules are independent on each other and that is why it is possible to modify each module separately. The new version of the ITEM module was tested on a simple application.

Contents

1	Introduction.....	1
1.1	Aim of this thesis.....	1
1.2	Wireless sensor networks	1
2	Parts of the system	4
2.1	Hardware	4
2.1.1	Introduction.....	4
2.1.2	Tmote Sky module details	4
2.2	Software.....	6
2.2.1	Introduction.....	6
2.2.2	TinyOS operating system	6
2.2.3	NesC	7
2.2.4	Main differences between TinyOS 1.0 and TinyOS 2.0.....	10
3	ITEM Software module.....	14
3.1	Description and implemented changes.....	14
3.1.1	ITEM module.....	14
3.1.2	E-ASAP module	16
3.1.3	TDMA module.....	17
3.1.4	Comm module	18
3.1.5	TimeSync module.....	19
3.1.6	Data module.....	19
3.1.7	SimpleTime module.....	20
3.1.8	WatchDog module	21

3.1.9 Core module.....	21
3.2 Tests.....	22
3.2.1 Adding and removing nodes test	22
3.2.2 Data communication test	23
3.2.3 WatchDog module test	23
3.3 Actual inadequacies and possibility of their elimination	23
4 Conclusion	25
References.....	26
Contents of the CD-ROM.....	28

List of Figures

Figure 1.1 Typical multihop wireless sensor network architecture	2
Figure 2.1 TelosB module and block diagram.....	4
Figure 2.2 Tmote Sky module – top and bottom side view	6
Figure 2.3 An example of wired BlinkC module.....	9
Figure 2.4 An example of RealMainP module.	9
Figure 3.1 ITEM’s wiring.....	14
Figure 3.2 Module EASAP’s wiring.....	16
Figure 3.3 Module TDMA’s wiring	17
Figure 3.4 Module Comm’s wiring	18
Figure 3.5 Module TimeSync’s wiring.....	19
Figure 3.6 Module Data’s wiring.....	19
Figure 3.7 Module SimpleTime’s wiring	20
Figure 3.8 Module WatchDog’s wiring.....	21

List of Tables

Table 3.1 Dynamical changes in the network.....	22
---	----

Chapter 1

Introduction

1.1 Aim of this thesis

The ITEM software module was developed at the Czech Technical University in Prague in the Department of Control Engineering. The module used for wireless sensor networks utilizes TDMA (Time Division Multiple Access) and E-ASAP (Extended - Adaptive Slot Assignment Protocol) protocols. The main advantages are that the module is able to respond to changes in the network structure and improve channel utilization. The ITEM software module was designed for the 1.x version of the operating system TinyOS. TinyOS version 2.x has already been developed and it is slightly different from the previous version. The main aim of the research work described in this thesis is to make ITEM module work under the TinyOS operating system and to test its functionality.

1.2 Wireless sensor networks

Wireless sensor networks (WSNs) consist of spatially distributed autonomous wireless devices called nodes or motes. They are used for monitoring physical or environmental conditions (e.g. temperature, sound, vibrations, etc.) in various places. Like the most of the applications, wireless sensor networks were originally projected for military purposes. These days WSNs are being used in many civilian applications including environment monitoring, health conditions, home automation and traffic control. Monitoring an area is the most common application of WSNs. Individual nodes are deployed in places where a specific phenomenon is watched. These territories are often remote or hostile. For example, nodes can be arranged on a battlefield to detect any enemy movement. As soon as the sensor detects an expected event, it forwards information to the base station. There the information can be processed. Functions of WSNs and propagation of sensed dates vary according to the application.

Sensor networks usually form a wireless ad-hoc network [2], it means that each sensor node supports routing algorithm (from node to node, towards the base station).

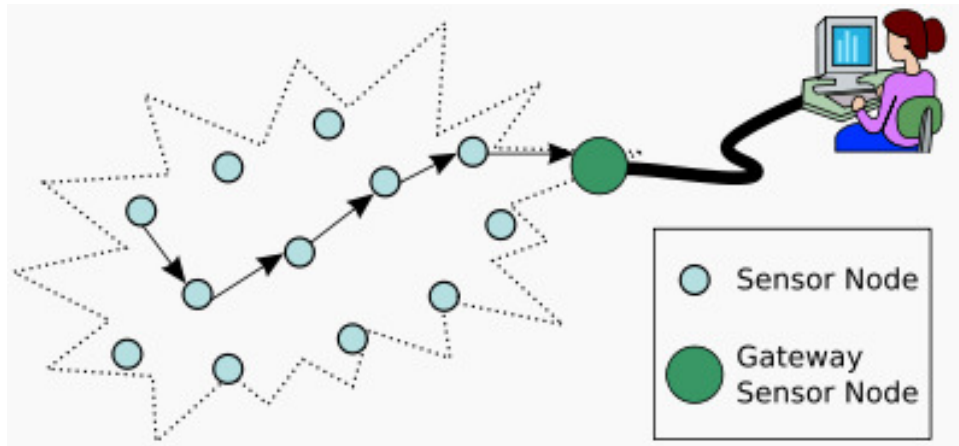


Figure 1.1 Typical multihop wireless sensor network architecture

Each sensor node consists of a processing unit with limited computational power and limited memory, sensors, a communication device (usually radio transceivers) and a power source (usually a battery). The base station forms a gateway between a user and sensor nodes and usually has much more computational, energy and communication resources. The most frequent requirement is to produce tiny sensors with low costs. Size, complexity and price of individual nodes depend on the desired application. An overview of the currently used sensor platforms, components, technologies and related topics is available in the SNM – the Sensor Network Museumtm [3].

WSNs characteristics are:

lifetime maximization

robustness and fault tolerance

self-configuration

security

mobility.

Operating systems for wireless sensor network nodes are typically less complex than general-purpose operating systems. The reasons are special requirements of sensor network

applications and resource constraints in sensor network hardware platforms. For example, applications for WSN usually aren't interactive in the same way as applications for PCs. That's why the operating system doesn't need to contain support of user interfaces. Mechanisms such as virtual memory and memory mapping hardware support are, because of constraints in terms of memory, unnecessary or impossible to implement. Probably the first operating system designed for wireless sensor networks is TinyOS, which is based on event-driven programming model instead of a multithreading model. This model consists of a event handlers and tasks with run to completion-semantics. When an event occurs (e.g. incoming data packet or reading sensor), TinyOS calls corresponding event handler to handle the event.

Programs for TinyOS are written in a special programming language, called *nesC*, which is an extension of the C programming language.

Chapter 2

Parts of the system

2.1 Hardware

2.1.1 Introduction

Devices for wireless sensor networks are mainly prototypes. It is due to requirements for sensor nodes. Requirements differ from applications. A list of commercial sensor nodes is available at 0. For the design of the ITEM software module were used the TelosB and Tmote Sky modules. Both modules are almost identical. The Tmote Sky module will be described in more detail in the next part.



Figure 2.1 TelosB module and block diagram

2.1.2 Tmote Sky module details

Tmote Sky is an ultra low power wireless module for use in sensor networks, monitoring applications and rapid application prototyping. By using industry standards such as USB and IEEE 802.15.4, integrating humidity, temperature, and light sensors and providing flexible interconnection with peripherals Tmote Sky enables a wide range of mesh network applications. The main components of a sensor node are a microcontroller, a radio transceiver, a power source, an external memory and one or more sensors.

The low power operation of the Tmote Sky module is due to the ultra low power Texas Instruments MSP430 F1611 microcontroller featuring 10kB of RAM, 48kB of flash and 128B of information storage. This 16-bit RISC processor features extremely low active and sleep current consumption that permits Tmote Sky to run for years on a single pair of AA batteries. The MSP430 has an internal digitally controlled oscillator (DCO) that may operate up to 8MHz. The DCO may be turned on from sleep mode in 6 μ s, however 292ns is typical at room temperature. When the DCO is off, MSP430 operates off an eternal 32768Hz watch crystal. The features of MSP430 F1611 are presented in detail in the Texas Instruments MSP430x1xx Family User's Guide [6].

Tmote Sky features the Chipcon CC2420 radio for wireless communications. CC2420 is an IEEE 802.15.4 compliant radio. CC2420 provides reliable wireless communication. It is controlled by the TI MSP430 microcontroller through the SPI port and series of digital I/O lines and interrupts. The radio may be shut off for low power duty cycled operation by the microcontroller. The CC2420 has programmable output power and provides a digital received signal strength indicator (RSSI). Features and usage of CC2420 is available in Chipcon's datasheet [7].

Tmote Sky is powered by two AA batteries. When plugged into the USB port, it receives power from the computer. Operating range is 2.1 to 3.6V DC. When a microcontroller flash or an external flash are programmed, the voltage must be at least 2.7V.

Tmote sky uses the ST M25P80 40MHz serial code flash for external data and a code storage. The flash holds 1024kB of data and is decomposed into 16 segments, each 64kB in size. The flash shares SPI communication lines with the CC2420 transceiver. Care must be taken when reading or writing to the flash so that it is interleaved with radio communication, typically implemented as a software arbitration protocol for the SPI bus on the microcontroller.

A variety of sensors may be used. Tmote Sky has connections for two photodiodes and a humidity/temperature sensor. Additional devices such as analog sensors, LCD displays and digital peripherals may be connected to one of two expansion connectors.

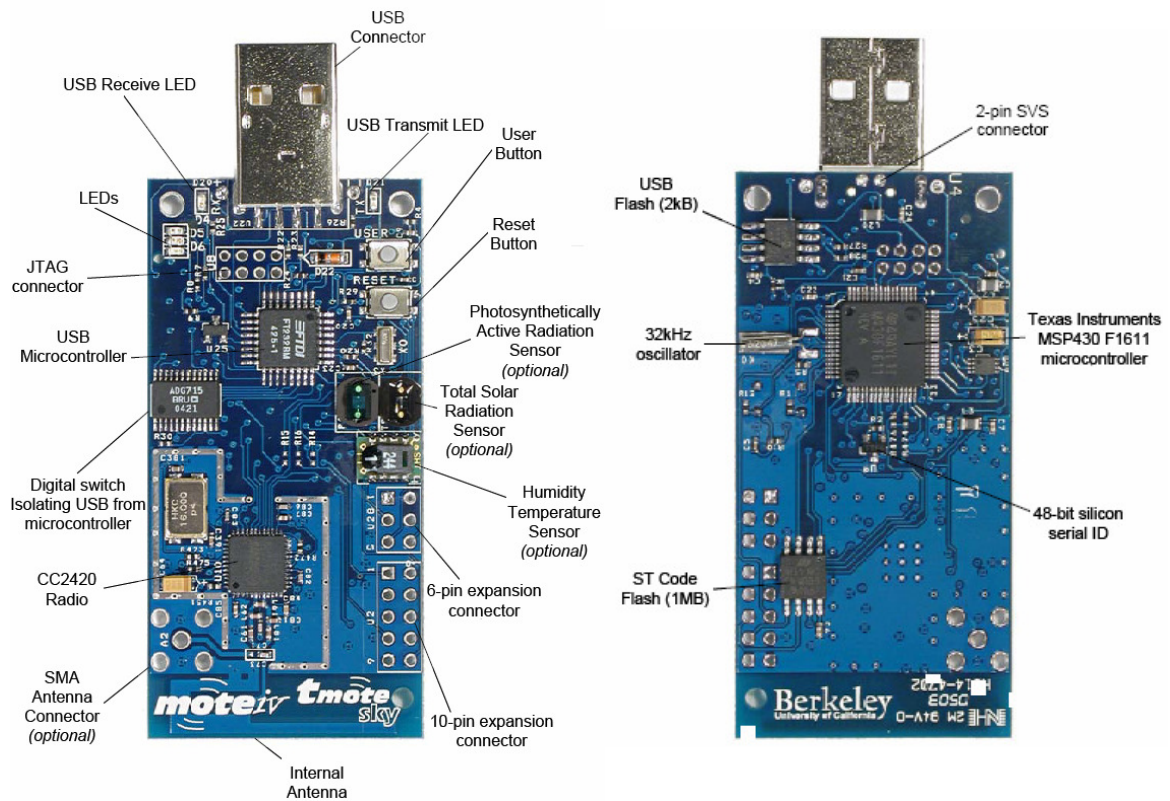


Figure 2.2 Tmote Sky module – top and bottom side view

2.2 Software

2.2.1 Introduction

The Wireless sensor networks hardware is very similar to the traditional embedded systems. It means that operating systems such as *eCos* or *uC/OS* could be used. These operating systems are often designed with real-time properties while operating systems for sensor networks often do not have real-time support. The Tiny OS operating system, which is specially designed for wireless sensor networks, will be described in the next part of this chapter. Another operating systems allowing programming in C are *Contiki*, *SOS*, *MANTIS*, *Btnut* and *Nano-RK*.

2.2.2 TinyOS operating system

TinyOS is an operating system specifically designed for network embedded systems. Its programming model is customized for event-driven applications.

The main important features that impressed nesC's design were:

component-based architecture

simple event-based concurrency model

split-phase operations.

TinyOS provides a set of reusable system components. An application connects individual components using a wiring specification that is independent of components it uses.

Tasks and *events* are two sources of the concurrency. Tasks are deferred computation mechanism. They run to completion and do not preempt each other. A component can post the task. The post operations return immediately. A computation is postponed and scheduled by the scheduler later. A component can use tasks when timing requirements are not strict. Individual tasks must be short to ensure low task execution latency. Long operations should be separate across multiple tasks. Events also run to completion, but can preempt the execution of a task or another event.

TinyOS operations are non-blocking because tasks are executed non-preemptively. All operations with long latency are split-phase (operation request and completion are separate functions). Commands are typically requests to execute an operation. If the operation is split-phase, the command returns immediately and completion will be signaled with an event. Non-split-phase operations do not have completion events.

2.2.3 NesC

NesC supports a programming model that integrates environment reactivity, concurrency and communication. NesC simplifies the application development, reduce the code size and eliminate many sources of potential bugs by performing whole-program optimization and compile-time data race detection.

Every mote runs only one application at a time. Mote applications are deeply tied to hardware. These approaches bear three important properties. First, all resources are known statically. Second, applications are built of a set of reusable system components connected with application-specific code. Third, a hardware/software boundary varies depending on the application and hardware platform.

NesC programming language meets the main requirements for sensor networks. These requirements are:

motes are fundamentally event-driven, reacting to changes in the environment rather than driven by the interactive or batch processing,

motes have very restricted physical resources due to the goals of small size, low price and low power consumption,

motes can fail, but we must enable a long lifetime. An important target is to reduce run-time errors. There is no recovery mechanism except for automatic reboot.

NesC applications are built of components with well-defined bidirectional interfaces. NesC defines a concurrency model based on tasks and events and detects data races at compile-time. C programming language provides little help in writing safe code or in structuring applications. NesC addresses safely through reduced expressive power and structure through components. NesC programs are subject to whole-program analysis (for safety) and optimization (for performance). Therefore we do not consider separate compilation in nesC's design. There is no dynamic memory allocation and the call-graph is fully known at the compile-time. These restrictions make the whole program analysis and optimization significantly simpler and more accurate. NesC is based on the concept of components and directly supports the TinyOS event-based concurrency model.

NesC applications are formed of components. These components provide and use interfaces. These interfaces are the only point of the access to the component. An interface generally models any service and it is specified by an interface type.

Interfaces in nesC are bidirectional. They contain commands and events. A provider of an interface implements commands, while a user of an interface implements events.

In nesC programming language there are two types of components: modules and configurations. Modules provide an application code and implement one or more interfaces. The body of module is written in a C-like code with straightforward extensions. A command or event f in interface i is named $i.f$. A command is like a regular function called with a key word *call*, similarly an event is called with prefix *signal*. A definition of commands or events is prefixed with *command* or *event*. An interface of a component can be wired zero, one or

more times. Configurations bind components together. Every nesC application is described by a top-level configuration. An example of wired components is in Figure 2.3 and Figure 2.4. These graphs are generated by a feature for generating documentation called *nesdoc*. In the nesdoc diagram, a single box is a module and a double box is a configuration. Dashed board denotes, that a component is generic (a component can has more instances). Lines denote wirings and shaded ovals denote interfaces that a component provides or uses.

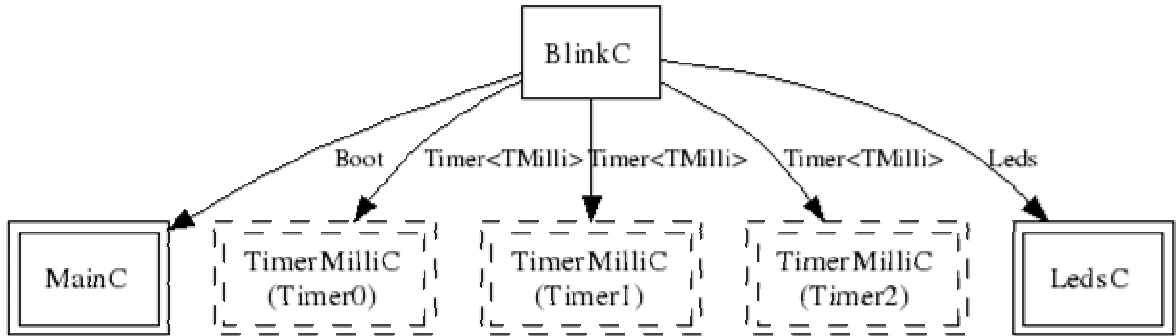


Figure 2.3 An example of wired BlinkC module.

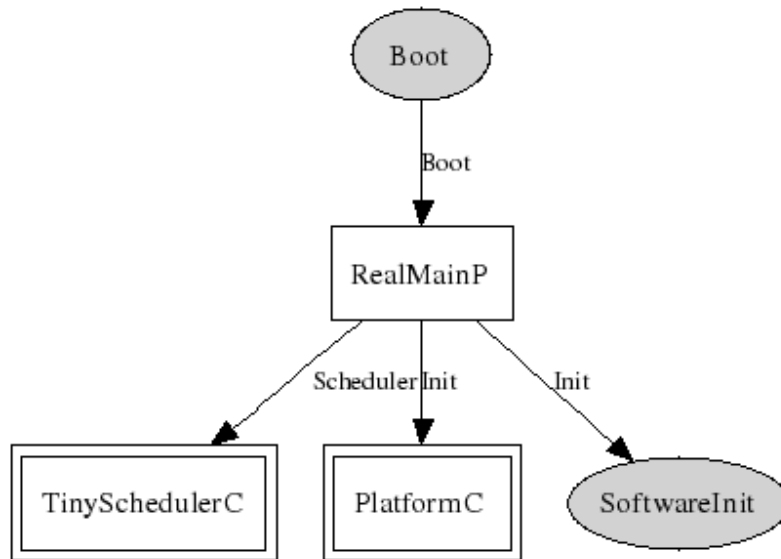


Figure 2.4 An example of RealMainP module.

Data races occur due to concurrent updates to the shared state. In order to prevent them, a compiler must understand the concurrency model and determine the target of every update. In TinyOS code a scheduled task runs either asynchronously (in response to an interrupt) or synchronously. A code reachable from at least one interrupt handler is

an asynchronous code. A synchronous code is atomic with respect to other synchronous codes. Any update to the shared state from the asynchronous code or any update to shared state from the synchronous code that is also updated from the asynchronous code is a potential race condition. A programmer has two options to keep the atomicity. He or she can either convert all of the sharing code to tasks or use atomic sections to update the shared state.

Events may be signaled directly or indirectly by an interrupt, which makes them the asynchronous code. To handle this concurrency, nesC provides atomic sections and tasks. A task may be posted. Posted tasks are executed by the TinyOS scheduler when the processor is idle.

Atomic sections are currently implemented by disabling and enabling interrupts. However leaving interrupts disabled for a long time delay interrupt handling, which makes the system less responsive. If a variable x is accessed by the asynchronous code, then any access of x outside of an atomic statement is a compile-time error.

The nesC programming language meet unique requirements of programming networked embeded systems. The success of the component model is shown by the way in which components are used in the TinyOS code. Applications are small and make use of large number of reusable components. Moreover nesC's component model makes it possible to pick and choose which parts of the operating system are included with each application.

2.2.4 Main differences between TinyOS 1.0 and TinyOS 2.0

When TinyOS 1.x was designed and implemented, many aspects were not foreseen. The structure and interfaces that TinyOS 1.x defines have several fundamental limitations. Therefore a new version of operating system TinyOS 2.0 was developed. TinyOS 2.0 is a redesign and a reimplementaion of TinyOS 1.x but it is not backwards compatible with TinyOS 1.x. The code written for the latter will not compile for the former. Minimalization of the difficulty of upgrading code is an important aspect of TinyOS 2.0. This part describes some of the ways in which TinyOS 2.0 departs from TinyOS 1.x. Details can be found in the TEPs (TinyOS Enhancement Proposals) documentation 0.

Hardware abstractions in TinyOS 2.0, called HAA (Hardware Abstraction Architecture), follow the three-level abstraction hierarchy. The bottom layer, called HPL

(Hardware Presentation Layer) is a thin software layer on the top of the raw hardware. The HPL presents hardware such as IO pins or registers as a nesC interfaces and it usually has no variables. HPL components have prefix `Hpl`, followed by the name of the chip (e.g. `HplCC2420`). On the top of the HPL is the middle layer called HAL (Hardware Abstraction Layer). The HAL provides higher-level abstractions that are easier to use than the HPL but still provides the full functionality of the underlying hardware. The HAL components have prefix of the chip name (e.g. `CC2420`). On the top of the HAL is upper layer called HIL (Hardware Independent Layer). The HIL provides abstractions that are hardware independent. This generalization means that the HIL usually does not provide the functionality that the HAL can. The HIL components have no naming prefix, because they represent abstractions that an application can use and safely compile on multiple platforms.

The TinyOS 2.0 scheduler has non-preemptive FIFO policy such as TinyOS 1.x, but tasks in TinyOS 2.0 operate differently than in TinyOS 1.x. In TinyOS 1.x, all tasks share a task queue and a component can post a task multiple times. If the task queue is full, the post operation fails. If a task signals completion of a split-phase operation and post fails, the component might block forever, waiting for the completion event. In TinyOS 2.0, every task has its own slot in a task queue and each task can be posted only once. If a task has already been posted, next post fails. If a component needs to post a task multiple times, it can set an internal variable that (after the task executes) report itself.

The boot sequence in TinyOS 2.0 is different from the boot sequence in TinyOS 1.x. The `StdControl` interface from TinyOS 1.x has been split into `Init` and `StdControl` interfaces. The `StdControl` interface has only start and stop commands. In TinyOS 1.x, wiring components to the boot sequence would cause them to be powered up and started at the boot. In TinyOS 2.0, the boot sequence only initializes components. When it has completed initializing the scheduler, hardware and software, the boot sequence signals the `Boot.booted` event. This event can be handled by a top-level application component to start services accordingly.

Many basic TinyOS services are now virtualized. Program instanties a service component that provides the needed interface rather than wire to a component with

parametrized interfaces. This service component does all of the wiring underneath automatically, reducing wiring mistakes and simplifying use of the abstraction.

Timers are one of the most critical abstractions a mote OS can provide and so TinyOS 2.0 expands accuracy and form that timers take. A component can use 23kHz as well as millisecond granularity timers and the timer system may provide one or two high-precision timers that fire asynchronously. The timer is a good example of virtualization in TinyOS 2.0. In TinyOS 1.x, a program instantiates a timer by wiring to TimerC:

```
components App, TimerC;

App.Timer -> TimerC.Timer[unique("Timer")];
```

In TinyOS 2.0 a program instantiates a timer:

```
components App, new TimerMilliC();

App.Timer -> TimerMilliC;
```

In TinyOS 2.0, the message buffer type is `message_t` and it is a buffer that is large enough to hold a packet from any of the node's communication interfaces. Components cannot reference structures' field. An access to these fields is through interfaces. Send interfaces distinguish the addressing mode of communication abstractions. An active message communication has the `AMSend` interface (sending packets require an AM destination address) but broadcasting and data collection tree abstractions have the address-free `Send` interface. `TOS_UART_ADDRESS` no longer exists but a component can wire to the `SerialActiveMessageC` component, which provides active message communication over the serial port.

A return code type `resut_t` (whose values are: a non-zero value `SUCCESS` or a zero-value `FAIL`) is replaced by `error_t` (whose values are `SUCCESS`, `FAIL`, `EBUSY` and `ECANCEL`) in TinyOS 2.0. Interface commands and events define which error codes they may return and why.

In TinyOS 2.0 there are two parts of power management: the power state of the microcontroller and the power state of devices. The former is computed in a chip-specific

manner by examining which devices and interrupt sources are active. The latter is handled through resources arbiters.

TinyOS 2.0 represent the next step of the TinyOS development. It uses user experiences over the past few years. The developers hope that TinyOS 2.0 lead to simpler and easier application development. TinyOS is still under active development.

Chapter 3

ITEM Software module

3.1 Description and implemented changes

3.1.1 ITEM module

The ITEM software module was originally designed for the operating system TinyOS 1.x (ITEM 1). The new version of TinyOS 2.0 is not backward compatible, so ITEM 1 cannot be under TinyOS 2.0 compiled. The main goal of this thesis is to make the second version of ITEM (ITEM 2) that is compatible with TinyOS 2.0.

ITEM consists of several modules. All modules are implemented as independent components. Only one component (called Core) wires other components together.

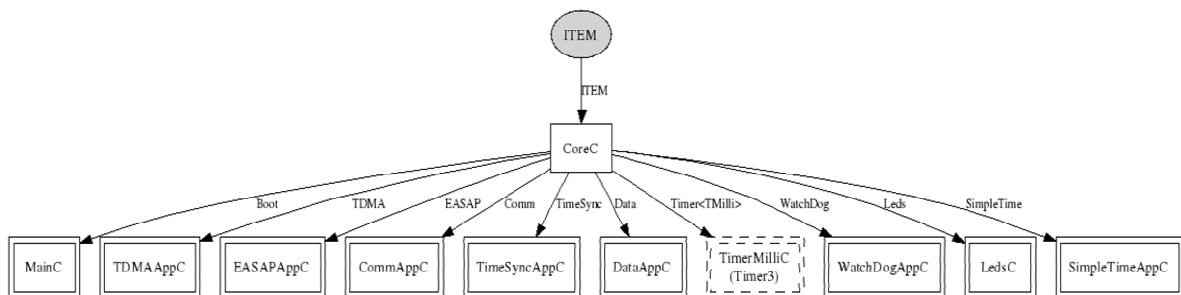


Figure 3.1 ITEM's wiring

Each component consists of four or five files. These files are: a configuration (e.g. DataAppC.nc), a module (e.g. DataC.nc), a header file (e.g. Data.h) and an interface (e.g. Data.nc). Some components have an internal interface (e.g. DataImp.nc) for internal implementation.

There are many changes that are needed to make during porting a code written for TinyOS 1.x to TinyOS 2.0. Changes, that were made to make ITEM module compatible with TinyOS 2.0, will be described further in the text.

SUCCESS is a non-zero error code in TinyOS 1.x and FAIL is zero. In TinyOS 2.0, SUCCESS is equal to a zero error code and other error codes are non-zero. So if any program contains any block, similarly:

```
if (call any_command()) {
    // SUCCESS!: do this...
}
```

should be changed to make sure that calls test the result for equality with SUCCESS:

```
if (call any_command() == SUCCESS) {
    // SUCCESS!: do this...
}
```

There are some easy string substitutions. The `result_t` data type from TinyOS 1.x is replaced by the `error_t` data type in TinyOS 2.0.

`TOS_LOCAL_ADDRESS` no longer exists. There is now a distinction between the local node's ID (`TOS_NODE_ID`) and the active message address. The active message address of a communication interface can be obtained through the `AMPacket.localAddress()` command and it can be changed at runtime while `TOS_NODE_ID` is bound at compile-time. By default, node's ID and its AM address are the same. `TOS_BCAST_ADDR` (the address for broadcast) is replaced by `AM_BROADCAST_ADDR`. `TOS_UART_ADDRESS` (the address for serial communication) no longer exists. When we want to send data to the serial port, we must use the `SerialAMSenderC` component.

Labels for leds such as red, green and yellow are replaced by `led0`, `led1` and `led2`. That is why calls such as `call Leds.greenToggle()` need to be replaced by calls `Leds.led1Toggle()`.

Changes are also related to timers. The timer must be created as a new instance with a keyword `new` (e.g. `components new TimerMilliC`). The interface `Timer` is replaced by `Timer<TMilli>`. Functions for launching the timer, called with `call Timer.start(TIMER_ONE_SHOT, x)` or `call Timer.start(TIMER_REPEAT, x)` where `x` is a time to fire, are implemented as `call Timer<TMilli>.startOneShot(x)` or

call `Timer<TMilli>.startPeriodic(x)` in TinyOS 2.0. In TinyOS 1.x, there is the `result_t` return type from the `Timer.fired()` event, while in TinyOS 2.0, the return type from this event is `void`.

The `init()` and `start()/stop()` methods in the `StdControl` interface must be implemented in TinyOS 1.x to initialize and start or stop a component. In TinyOS 2.x these methods are separated. The `StdControl` interface contains only `start()` and `stop()` methods while the `init()` method is in the `Init` interface. These methods needn't to be implemented in TinyOS 2.0 and a component may be started with a signaled event `booted()` in the `Boot` interface connected to the `MainC` component.

Individual modules and their specific changes are described further.

3.1.2 E-ASAP module

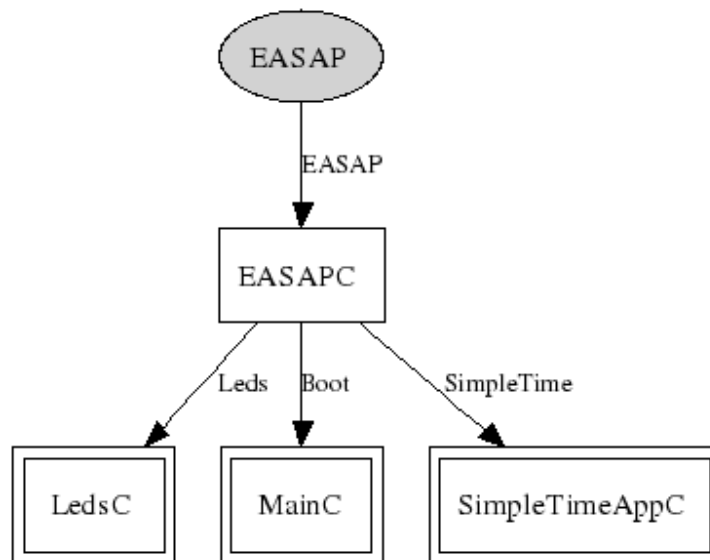


Figure 3.2 Module EASAP's wiring

The EASAP module implements Extended Adaptive Slot Assignment Protocol (E-ASAP) described in the article [9]. A use of this protocol improves channel utilization and prevents the excessive increase of unassigned slots by minimizing each node's frame length. Each node holds detailed information about nodes in its contention area such as node's ID, assigned slots and frame length and a time stamp. According to this information a new node can take a slot and adjust the frame length. When a node leaves the network, other nodes change the information appropriately (they remove inactive node's slots and minimize

the frame length when possible). Detection of conflicts and their solutions, described in the article are not implemented accordingly.

The first version of this module uses the TimeUtilC component for time calculations. This component is not in TinyOS 2.0 included. That is why a new module called SimpleTime was created. Other applied changes have already been mentioned.

3.1.3 TDMA module

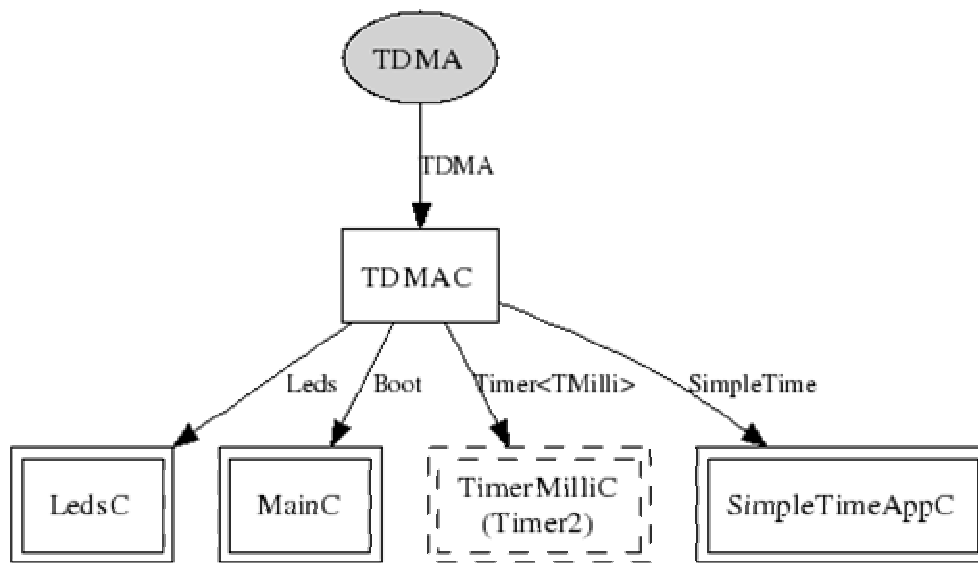


Figure 3.3 Module TDMA's wiring

The TDMA module adapts TDMA (Time Division Multiple Access) method. This is a method for the channel access for a shared medium. Slots, assigned by a fixed interval, are repeated within a frame. A frame can be increased or decreased by a request whenever appropriate.

The first version of this module uses interfaces such as Time, TimeSet and TimeUtil for time calculations and manipulations connected to the SimpleTime and TimeUtilC components. In TinyOS 2.0, these parts are missing but created SimpleTime module with corresponding functions replaces them. Other applied changes have already been mentioned.

3.1.4 Comm module

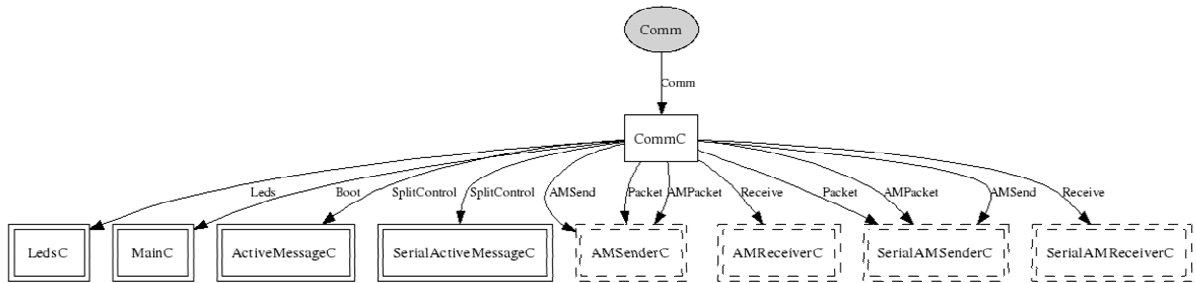


Figure 3.4 Module Comm's wiring

The Comm module is responsible for radio and serial communication. The module is able to send a message via radio to other nodes or via serial to computer. When a message is received, corresponding event is signaled.

This module contains the most extensive changes. A radio or a serial communication needs to be started manually using call `SplitControl.start()`. Components `ActiveMessageC` or `SerialActiveMessageC` signal `SplitControl.startDone(error_t error)` event with `SUCCESS` error code when the device is ready. The `SendMsg` and `ReceiveMsg` interfaces, used in TinyOS 1.x, are replaced by the `AMSend` and `Receive` interfaces. The `GenericComm` component no longer exists. In TinyOS 2.0 it is replaced by the `AMSenderC` and `AMReceiverC` components for the radio transmission and the `SerialAMSenderC` and `SerialAMReceiverC` components for the serial transmission.

In TinyOS 2.0, there is a new approach to data packets. The `TOS_Msg` data type structure is replaced by the `message_t` data type structure. Packets are now an abstract data type. A direct access to the `message_t` structure is not possible. Interfaces such as `Packet` and `AMPacket` are used for this purpose.

3.1.5 TimeSync module

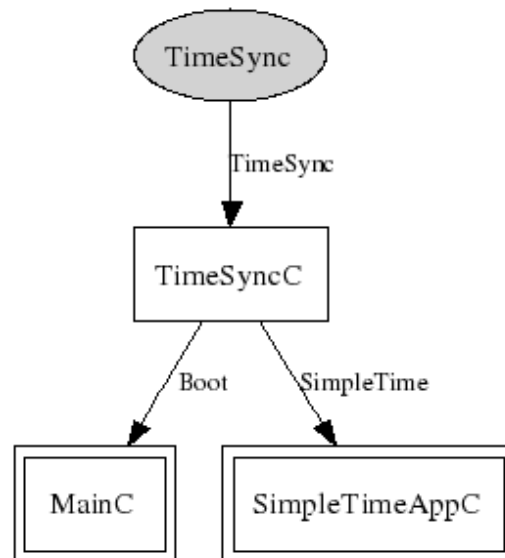


Figure 3.5 Module TimeSync's wiring

The TimeSync module averages the time difference between two given time arguments. It is used for time synchronization.

This module has no specific changes.

3.1.6 Data module

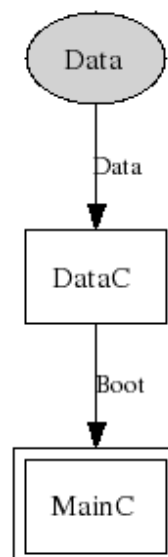


Figure 3.6 Module Data's wiring

The Data module is in charge of data storage. It has two queues that hold the data. These queues (for receiving and transmitting) are organized in FIFO order. If any of queues is full, new data are discarded.

Changes that were implemented were: easy string substitutions (mentioned earlier) and a way of module's initialization.

3.1.7 SimpleTime module

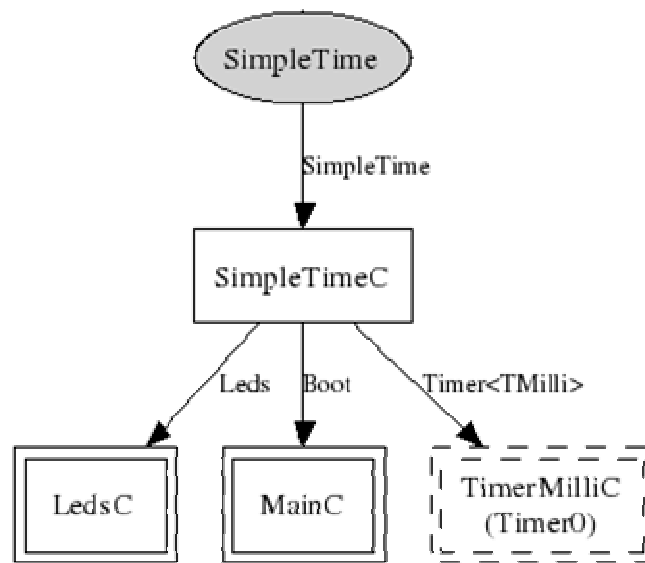


Figure 3.7 Module SimpleTime's wiring

In TinyOS 1.x, components such as TimeUtilC or SimpleTimeC allow time manipulations and calculations. These components are missing in TinyOS 2.0, so a new module called SimpleTime was created as compensation. This module generates local time. The local time is a 64-bit long number. A precision of the timer, which increments the local time, is 32 milliseconds (according to the implementation in TinyOS 1.x). Furthermore, there are functions for time manipulations such as a time creation, a time addition/subtraction or a comparison of two times.

3.1.8 WatchDog module

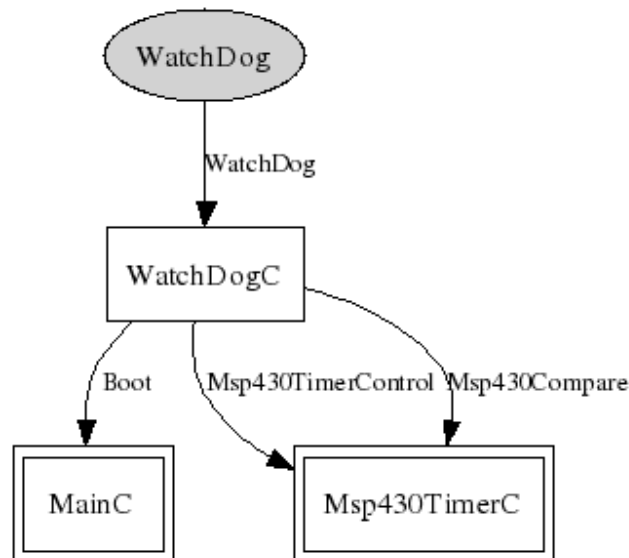


Figure 3.8 Module WatchDog's wiring

The main function of this module called WatchDog is to reset the system if it locks itself up due to an error. The WatchDog module can be enabled or disabled. This module uses the MSP430TimerC component (component from the middle layer of Hardware Abstraction Architecture), so it only works for the MSP430 architecture in this implementation.

Changes in this module include a way of module's initialization and some string substitutions mentioned earlier.

3.1.9 Core module

The Core module has the main function of all individual modules. This module combines all the other modules in ITEM and makes them work together.

In this module, there is one significant modification. There is a timer that delays own slot handling (because of node's local time deviation). In the first version, this timer is started from TDMA.sigNewSlot(tSlot slot) event but it fires as soon as the event finishes instead of required time expiration. Now, the timer is started from a task that is posted and it fires duly. Other applied changes are: module's initialization and mentioned string substitutions.

3.2 Tests

The new created version of the ITEM software module, that is compatible with the TinyOS 2.0 operating system, was tested. Performed tests verify: dynamical changes in the network (adding and removing nodes), data communication and a test of the WatchDog module.

3.2.1 Adding and removing nodes test

When a node powers up, it listens if any other node transmits an information packet. As soon as adjusted time (two-time frame length) elapses, the node takes a free slot (zero slot is reserved) according to received information packets and informs other nodes about the changes. Other nodes change stored information about nodes in their contention area appropriately (mark the slot and increase the frame length if needed). If a node is first in the network, it must be started by a user button manually. Then, after some time, the node takes a slot.

If a node is removed from the network, other nodes don't receive its information packet and they recognize that its slot can be released. The frame length is reduced if possible.

Dynamical changes are shown in Table 3.1 which contains the performed action and node information (its slot and frame length).

ACTION	NODE ID				
	1 slot / frame	2 slot / frame	3 slot / frame	4 slot / frame	5 slot / frame
node 1 launched	1 / 4	-	-	-	-
added 2	1 / 4	2 / 4	-	-	-
added 3	1 / 4	2 / 4	3 / 4	-	-
added 4	1 / 8	2 / 8	3 / 8	4 / 8	-
added 5	1 / 8	2 / 8	3 / 8	4 / 8	5 / 8
removed 4	1 / 8	2 / 8	3 / 8	-	5 / 8
removed 5	1 / 4	2 / 4	3 / 4	-	-
removed 2	1 / 4	-	3 / 4	-	-
added 5	1 / 4	-	3 / 4	-	2 / 4
added 2	1 / 8	4 / 8	3 / 8	-	2 / 8
removed 3	1 / 8	4 / 8	-	-	2 / 8
added 4	1 / 8	4 / 8	-	3 / 8	2 / 8
removed 2	1 / 4	-	-	3 / 4	2 / 4
removed 5	1 / 4	-	-	3 / 4	-
removed 4	1 / 4	-	-	-	-

Table 3.1 Dynamical changes in the network

The table shows, that nodes network works correctly. A detailed node information packet's extract can be found in the Test1.txt file in /ITEM2/tests/ directory on the enclosed CD.

3.2.2 Data communication test

This test verifies data communication. All nodes have a counter. The node with ID 1 increments the counter and sends a data packet via radio. This packet contains a node ID and a counter value. Other nodes receive this packet. If a node, that received a packet, has one higher ID, it increments own counter, add its ID and a counter value to the data packet, and sends a data packet via radio. Therefore a node with ID 5 holds information about nodes 1, 2, 3 and 4 counters.

A detailed node information packet's extract from this test can be found in the Test2.txt file in /ITEM2/tests/ directory on the enclosed CD. This test ran almost 40 hours and there were no node collapses.

3.2.3 WatchDog module test

The last test tries the function of the WatchDog module. Its main function is to reset the system if it locks itself up due to an error.

For this test, the same application as described in part 3.2.2 was used. There is only one minor modification. When node's counter reaches the desired value, an infinite while cycle starts and node's communication freezes. The WatchDog restarts a node after some time.

A detailed node information packet's extract from this test can be found in the Test3.txt file in /ITEM2/tests/ directory on the enclosed CD.

3.3 Actual inadequacies and possibility of their elimination

The ITEM software module is suitable to wireless sensor network requirements, but it contains small inadequacies. These inadequacies had already occurred already in the first version of the ITEM module. This part contains a description of these inadequacies and a sketch of their elimination.

Possible slot conflicts are not resolved. Devices must be connected to the network in turn. When two or more nodes are connected to the network simultaneously, they can take the same slot. Then, if nodes transmit data, collisions occur. A possible solution is: When a collision is detected, node slots are released. Nodes take a new slot with a random time delay.

The second limitation is that the target address of a transmitted packet cannot be entered. Data are transmitted to all nodes in the network. A modification of the Comm module can remove this limitation.

There is no full channel utilization because only one message per slot can be transmitted. The solution is following: to measure the remaining time in the slot and if there is enough time, the next message can be transmitted.

Last detected inadequacy concerns removing of inactive nodes and a frame length reduction. When a node is removed and the number of nodes is accordant with frame reduction, but any node has slot from the upper half of the frame, the frame reduction does not occur. For example: nodes have slots 1, 2, 5 and 6 and the frame length is 8. We remove the node with slot 2, but no reduction of the frame length occurs although frame length 4 is possible. A solution could be checking a number of nodes when any node is removed. If reduction of the frame length is possible, nodes with slot from the upper half of the frame releases the slot and takes new slot from the bottom half of the frame. A reduction of the frame is now possible.

Removing of these inadequacies improves quality of the ITEM module, but it is above the range of this thesis.

Chapter 4

Conclusion

The main goal of this thesis is to make the existing version of the ITEM software module functional under the TinyOS 2.0 operating system. Applications written for TinyOS 1.x cannot be compiled under TinyOS 2.0, because TinyOS 2.0 is not backward compatible with TinyOS 1.x.

The first objective is to acquire knowledge about sensor networks, the TinyOS 1.x and TinyOS 2.0 operating systems and the nesC programming language. Further, it was necessary to familiarize oneself with E-ASAP (Extended-Adaptive Slot Assignment Protocol) and TDMA (Time Division Multiple Access) protocols. After that, modifications of the first version could start.

The ITEM software module consists of several smaller modules with specific functions. These modules are independent on each other (only the Core module wires all modules together). Therefore it was possible to modify each module separately. Each module requires a special modification (except common changes e.g. string substitutions and a way of the module initialization). The most important changes were implemented in the Comm module. A lot of changes were also implemented in the EASAP and Core modules.

All problems that occurred during modifications were removed successfully. One of the problems was to test correct functionality. There are only two ways of testing: switching of three LEDs on the TelosB/Tmote Sky module or reading transmitted data on PC. A content of the data packet was shown in PC by the Java listen tool described in 0.

Tests in part 3.2 confirm that the new version of the ITEM software module works correctly. The module contains inadequacies that can be removed or improved. Despite all inadequacies, the module can now be successfully used in sensor networks applications.

References

- [1] Wikipedia (2008), Wireless sensor network.
http://en.wikipedia.org/wiki/Sensor_network
- [2] Wikipedia (2008), Wireless ad-hoc network.
en.wikipedia.org/wiki/Wireless_ad-hoc_network
- [3] The Sensor Network Museumtm.
www.btnode.ethz.ch/Projects/SensorNetworkMuseum
- [4] Wikipedia (2008), Sensor node.
en.wikipedia.org/wiki/Sensor_node
- [5] Moteiv, Tmote Sky Datasheet.
www.btnode.ethz.ch/pub/uploads/Projects/tmote_sky_datasheet.pdf
- [6] Texas Instruments, MSP430F1611 Datasheet and MSP430x1xx Family User's Guide.
ti.com/msp430
- [7] Texas Instruments/Chipcon, CC2420 Datasheet.
<http://focus.ti.com/docs/prod/folders/print/cc2420.html>
- [8] TinyOS, TinyOS 2.0 Documentation.
www.tinyos.net/tinyos-2.x/doc/
- [9] Akimitsu Kanzaki, Takahiro Hara, Shojiro Nishio. An Adaptive TDMA Slot Assignment Protocol in Ad Hoc Sensor Networks. Proceedings of the 2005 ACM symposium on Applied computing, March 13-17, 2005, Santa Fe, New Mexico
- [10] Inderjit Singh, Software implementation design for ITEM.
http://rtime.felk.cvut.cz/~trdlj1/lib/exe/fetch.php?id=item&cache=cache&media=item:item_implementation_design.pdf

- [11] Inderjit Singh, Real-time Object Tracking with Sireless Sensor Networks, Diploma Thesis, August 2007.
<http://epubl.ltu.se/1653-0187/2007/059/LTU-PB-EX-07059-SE.pdf>
- [12] TinyOS, Mote-PC serial communication and Serial Forwarder.
www.tinyos.net/tinyos-2.x/doc/html/tutorial/lesson4.html
- [13] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, David Culler.
The nesC Language: A Holistic Approach to Networked Embedded Systems.
nesc.sourceforge.net
- [14] Inderjit Singh. Integrated TDMA E-ASAP Module (ITEM). FEL-CVUT, Prague 2007

Contents of the CD-ROM

The included CD-ROM contains:

- /Documentation/ : Bachelor thesis in PDF format.
- /ITEM1/src : Source codes of the first version of ITEM (for TinyOS 1.x).
- /ITEM2/src : Source codes of the new version of ITEM (for TinyOS 2.0).
- /ITEM2/doc : A documentation of the new version of ITEM
- /ITEM2/tests/ : Extracts from tests