# TRANSPORT OF THE TINY-OS ENVIRONMENT TO THE SMART-ITS PLATFORM

A Thesis

by

FOTIS LOUKOS

Submitted to the Office of Graduate Studies of
University of Thessaly
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

September 2006

Major Subject: Wireless Sensor Networks

# TRANSPORT OF THE TINY-OS ENVIRONMENT TO THE SMART-ITS PLATFORM

A Thesis

by

FOTIS LOUKOS

Submitted to the Office of Graduate Studies of
University of Thessaly
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

| | |
|---|---|
| Chair of Committee, | Spyros Lalis |
| Committee Members, | Iordanis Koutsopoulos |
| | Georgios Stamoulis |

September 2006

Major Subject: Wireless Sensor Networks

# ABSTRACT

The smart-its are a low-cost platform made by the university of Karlsruhe used mainly for wireless sensor network applications. This thesis sentences the port of TinyOS to the smart-its platform. TinyOS is a wireless sensor network operating system originally written to work with the mica motes from the Berkeley University; since then a number of ports have been made. During the port a number of problems came up due to the fact that the smart-its use a completely different hardware from the other platforms TinyOS supports. All the problems together with the solutions are presented. Tests of the RF communications and their results are displayed.

To my grandmother, Theodosia

# ACKNOWLEDGMENTS

I would like to thank D. Syrivelis, M. Koutsoubelias and my supervisor S. Lalis for all the help and useful suggestions during this work at the writing of my thesis.

TABLE OF CONTENTS

CHAPTER Page

# LIST OF TABLES

# LIST OF FIGURES

FIGURE Page

# CHAPTER I

## INTRODUCTION

A Wireless Sensor Network is a mesh network of small sensor nodes communicating among themselves using RF communication and deployed in large scale to sense the physical world. The main characteristics of a wireless sensor network are

- Small-scale sensor nodes
- Limited power they can harvest or store
- Harsh environmental conditions
- Node failures
- Mobility of nodes
- Dynamic network topology
- Communication failures
- Heterogeneity of nodes
- Large scale of deployment
- Unattended operation

Wireless sensor networks have been used at many applications. Some of them are the following:

- Environmental control. Sensor networks can be used for early discovery of catastrophes like forest fires. A sensor node with a heat sensor can monitor the environment and signal events like high increase of temperature. Also, sensors

can be used to take measurements from not easily accessible places. A real life example is the volcano of Volcán Tungurahua in central Ecuador where a sensor network was deployed to monitor erruptions [13]. Furthermore, sensors have been used for waste management system monitoring [14].

- Health care. Monitoring systems based on sensor networks have been constructed [15][16]. As an example wireless sensors have been put in wriststraps to monitor for strokes or other weaknesses [17].

- Security monitoring. Sensors can be put inside rooms to monitor for movement. Another is example is security in cargo containers containing sensors [18].

There are many research issues on wireless sensor networks. We outline some of them below.

- Routing problems. Due to the nature of sensor networks a node can only communicate with a number of sensors which are close to it. This is because of the low transmit power. In order to communicate with a node which is out of reach a routing protocol must be used [19][20].

- Energy saving. The nodes should be very small and they are usually powered by a small battery with a limited energy. Most power consumption is due to rf communications. So, protocols must be created that have this in mind [21][22].

- Distributed data processing. Each sensor node gets it's own readings. There are many times when these data needs to be processed by more than one node. So new approaches on processing distributed sensor data in sensor networks must be found [23][24].

- Security. All messages exchanged between sensor nodes are broadcasted over the air. This can lead to important security issues when we want to transfer

"sensitive" data. Using another node a malicious attacker can intercept the messages and read them or even inject new messages. For these reasons a number of protocols for secure communications have been created [25][26].

In order to make programming a sensor node easier operating systems have been created. Many of them are designed to satisfy one of the above issues. Furthermore, the most used operating systems have been ported and work at many different architectures. A brief list of operating systems is the following:

- TinyOS[1].
- Contiki[2].
- ScatterWeb[3].

In chapter II the smart-its platform together with the hardware and software that supports it is presented. Then, in chapter III, the TinyOS operating system and it's concepts are analyzed. In chapter IV we document the process of porting the TinyOS to the smart-its platform together with the problem we faced. In chapter V an evaluation of the RF communications is presented.

[1]http://www.tinyos.net

[2]http://hstein.trix.net/contiki

[3]http://www.inf.fu-berlin.de/inst/ag-tech/projects/index.html

# CHAPTER II

# THE PARTICLE COMPUTER

## A. Particles

The particle computer is a prototyping platform created by the university of Karlsruhe. It is a low cost device, mainly used for the creation of wireless sensor networks. The hardware is the part created by TecO for the Smart-Its project [1]. It consists of a small sensor (the particle) whose main components are a microcontroller, a chip for wireless communications, external memory and a connector for an extra board. The most commonly used is a sensorboard which carries a custom set of sensors depending on the application (light, temperature, acceleration, etc).

All particles use a PIC microcontroller from Microchip. Also, there is an RF communications transceiver from RF Mononolithics, the TR1001. To get readings they connect to a sensorboard using a special type of connector, the CONAN connector.

As an example the hardware from the 2/10 particle is shown below.

- Processor: Microchip's PIC18F6720 at 20 MHz providing up to 10 MIPS operation. The internal memory is 128 kbytes of flash, 4 kbytes of RAM and 1 kbyte of EEPROM. Main features are 5 timer modules, $I^2C^{TM}$ and 3-wire $SPI^{TM}$ Master Synchronous Serial Port (MSSP), 2 addressable Universal Synchronous Asynchronous Receiver Transmitter (USART) and up to 12-channel 10-bit Analog-to-Digital Converter (A/D).

- External Memory: 32 kbytes EEPROM (Microchip's 24FC256).

- RF Communications: RFM TR1001, 125kbit bandwidth, 868.35 ISM band.

[1] http://smart-its.teco.edu/

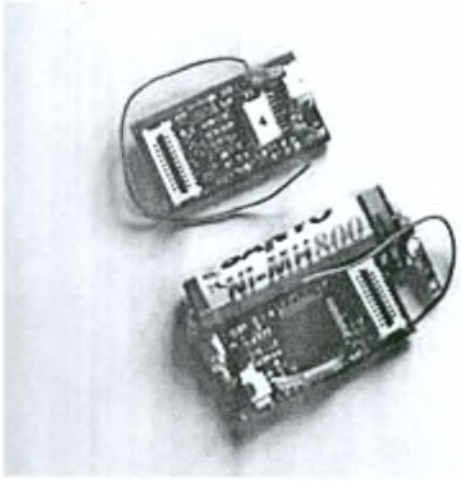


Fig. 1. The 2/10 particle

Fig. 2. The spart sensorboard

Fieldstrength regulation via software. Up to 30 meters inhouse range.

- Interface: CONAN connector with 21 pin multipurpose pins. Allows $I^2C^{TM}$, $SPI^{TM}$, serial communications (RS232), parallel bus, analog input lines and digital I/O lines.
- Real time clock (RTC).
- Two low power LEDS.
- Ballswitch to detect movement and connected to interrupt line for wake-up-on-movement.
- Power regulation for power in from 0.9-3.3 V. Supports AAA, 2xAAA, AA and lithium coin batteries. Also supports remaining energy measurement. Allows in circuit charging if rechargeable battery is used.
- Allows in circuit and over the air programming.
- Size: 33x17 mm without ballswitch and battery, 33x17x15 mm without ballswitch and with lithium coin cell and 45x27 mm with ballswitch and AAA battery.

More information about the devices and the hardware can be found at
http://particle.teco.edu/devices/index.html and
http://particle.teco.edu/documentation/content/particle.html.

## B. Sensorboards

Sensorboard are the boards that are connected to the particles that have the sensors which get the readings of various values.

There are two types of sensorboards. Both of them contain a number of different sensors for temperature, humidity, light, etc. The biggest difference is that one board contains it's own microcontroller, therefore it can process the sensor readings autonomously and needs the particle only for RF communications.

More information about the sensorboards can be found at
http://particle.teco.edu/documentation/content/ssimp.html and
http://particle.teco.edu/documentation/content/spart.html.

### C. Gateway hardware

The particles are able to communicate with each other in an ad-hoc way. No special backbone infrastructure is needed for this purpose.

Additionally, there are two special devices that enable communication with other computers, the XBridge and the USB Bridge.

The XBridge allows any device which supports the UDP/IP protocol to access every particle in the network. The USB Bridge connects to a computer and allows this computer to access the particles in the network.

For more information about bridges you can visit
http://particle.teco.edu/documentation/content/bridge.html.

### D. The AwareCon runtime

Along with the hardware there is also a software part. It comes as the AwareCon protocol stack.

AwareCon is a strict time synchronized TDMA system with a frame duration of 13ms. It is designed so that when a particle powers up it synchronizes as fast as possible with the rest of the particles and stays synchronized at anytime. Further-

more, it can achieve a high speed of communication when it transmits and receives all data. Due to the channel access control it implements it can avoid collisions and full throughput can be achieved even with hundrends of particles running.

Since there is no multithreading implemented, every 13ms the program running at the particle is stopped and the AwareCon stack takes over. The particle exchanges all data with the rest of the network and the program continues and the point where it stopped. The program continues to run until the next timeslot where it stops again. If there are data to exchange the warranted application time is a little more than 4.5ms but if there are no data then the application time is more than 11ms.

The data are transmitted as packets consisting of three layers:

- The RF layer for synchronization, channel coding etc.
- The LL (link layer) layer for access control, data encoding and error checking (CRC16 algorithm)
- The ACL layer as abstract user interface and data representation

The payload data have a maximum size of 64 bytes. They are organized in units called tuples. Each tuple consists of 2 bytes which represent the data type, one byte that holds the length of the packet and the data bytes. These tuples are concatenated and then placed inside the payload buffer for the ACL layer. The first tuple has a special meaning since it's data type represents the subject of the packet. Every time a packet is sent it is broadcasted to all particles in the network. These have a subscriber list which holds a maximum of seven subjects they are interested into. If the packet's subject is not in this list then the packet is dropped. Otherwise the program is notified that a new packet has arrived which can be processed.

E. Creating an application

There are a number of functions that are provided by the AwareCon protocol stack. The most important categories of functions are the following:

- Preparing and transmitting packets.

  Functions that allow you to create a new packet, append tuples into it and finally send it. You can also check the remaining number of bytes in the packet.

- Status and flow control.

  Functions to check if the communications stack is transmitting a packet and if the last packet transmission was successful.

- Subscription to subjects.

  Functions to subscribe or unsubscribe to certain subjects.

- Receive data.

  Functions to check if a new packet has arrived and get access to it's contents.

But, in order to use all these functions your application must conform to certain rules. First of all, your application should be aware that it will be stopped every 13ms. There is no warranty about the time remaining for the application. It can be 11ms if there was no packet reception/transmission or 4.5 if there where data that had to be processed. There are functions that allow the application to wait until the next timeslot or check how many time is left until the next interrupt. Furthermore, there are a number of timers that are reserved by the AwareCon protocol stack. In order to fulfil all these requirements your application should not use interrupts for handling various events e.g. analog to digital conversions since code inside an interrupt handler cannot be stopped by another interrupt i.e. the AwareCon protocol stack timer

interrupt. This imposes an asynchronous model of operation so the application cannot take full advantage of the time that remains in each timeslot.

There are a number of examples together with documentation that give general guidelines for creating an application at the particles homepage[2].

[2]http://particle.teco.edu/software/particlesw/index.html

# CHAPTER III

## THE TINYOS OPERATING SYSTEM

The TinyOS is an open-source operating system[1] designed for wireless sensor networks. It was initially designed by the university of Berkeley and for the Berkeley/Crossbow motes. It uses an event-driven model which allows better power management control. The programming language it is written in is nesC (with the lower level components written in assembly) which allows a component architecture that minimizes the code size as required by the severe memory constrains of sensor network nodes.

### A. NesC

The NesC 1.1 programming language, as specified in [4], is used mainly for programming for the TinyOS operating system. It is based on C [5] but has some extensions in order to support an event driven model.

The main concepts of NesC are the interfaces and the components. Interfaces can be seen as the specifications for the behaviour of components. The components are the main parts of a program. There are two types of components, modules and configurations. Modules contain the actual code for implementing interface specifications. Configurations contain only wiring specifications for putting together several modules in order to create a final component.

[1] All sources can be found at http://www.tinyos.net.

1. Interfaces

A component may use certain interfaces and provide some others. Interfaces have a bidirectional nature. Each interface defines two sets of named functions, the commands and the events. Events correspond to notifications that must/can be handled by components. Commands correspond to functions that can be invoked from within components.

A sample interface for a generic timer is the following. It has two commands, `start` and `stop`, and one event, `fired`.

```
/**
 * This interface provides a generic timer that can be used to
 * generate events at regular intervals.
 */
interface Timer {
  /* Start the timer. */
  command result_t start(char type, uint32_t interval);
  /* Stop the timer, preventing it from firing again. */
  command result_t stop();
  /* The signal generated by the timer when it fires. */
  event result_t fired();
}
```

Each interface provider must implement a certain set of commands defined in an interface. The commands may be executed either synchronously or asynchronously, e.g. by disabling interrupts. The main usage of commands is the initialisation of components and performing complex interactions that must be non-blocking, e.g. start and stop timers, prepare sensors or send data over the network.

An example of a command is the following. It belongs to the *TimerM* module and is called *stop()*. It's usage is to stop a timer by clearing the appropriate bits in a state variable.

```
command result_t Timer.stop[uint8_t id]() {
  if (id>=NUM_TIMERS) return FAIL;
  if (mState&(0x1L<<id)) { // if the timer is running
    atomic mState &= ~(0x1L<<id);
    if (!mState) {
      setIntervalFlag = 1;
    }
    return SUCCESS;
  }
  return FAIL; //timer not running
}
```

Events are functions that are called when signalled from another part of a program. When an event is defined in an interface all components that use this interface must implement the event handler. As an exception, an event handler may not be implemented if the component that provides this interface defines a default event handler. The main usage of events is to report the completion of complex commands, e.g. send data over the network, send data over the serial line, e.t.c., and to be tied on interrupts e.g. fire event when a clock goes off, detection of data from a sensor, e.t.c.

An event used at the *TimerM* module is the following. It is signalled every time a timer interrupt fires and handles it appropriately.

```
async event result_t Clock.fire() {
```

```
  atomic {
    /* DCM: Once we've posted HandleFire(), don't post it again until
     * the original one is handled.  This prevents the task queue
     * from getting flooded when mInterval is small. */
    if (interval_outstanding == 0)
      post HandleFire();
    else
      dbg(DBG_ERROR, "Don't post handle fire, we're not ready\n");
    /* DCM: Keep track of the interval since the last interrupt */
     interval_outstanding += call Clock.getInterval() + 1;
  }
  return SUCCESS;
}
```

## 2. Components

A component can be either a module or a configuration. Each component provides or uses an interface, so it must implement all commands specified in the interfaces it provides and all events specified in the interfaces it uses. The commands "call down" from the application towards the hardware components and events "call up" from hardware components towards the application. Finally, all components used by an application are wired together to form the final program.

### a. Modules

Modules contain the main code for the implementation of the desired functionality. The language that is used is C99 as specified in [6]. Inside a module we can have

- Standard C declarations and definitions.
- Task declarations and definitions.
- Implementations of commands.
- Implementations of events.

All the definitions belong to the module's component implementation scope. The definitions of commands and events must match the specifications of the interfaces the module provides and uses. Furthermore, a module can call any of it's commands and signal any of it's events.

Inside a module there can be one or more task definitions. Tasks are functions that are added in a task queue and are executed one after another. This allows a model of cooperative multithreading that can be achieved by splitting a big computation into many tasks. Each task must post the next task which is added at the end of the queue so all other intermediate tasks can execute. By using this mechanism tasks, commands and events can be executed "concurrently". A task doesn't take an argument and can't return a value so interaction must be done using external variables.

A special type of statements that exist inside a module are atomic statements. They are defined inside `atomic {}` blocks and are executed "as-if" no other computation occurred simultaneously. The way TinyOS works, a block of code can be interrupted by an external interrupt. If this block is defined as atomic then interrupts are disabled and the block is executed without any interruptions.

Furthermore, inside the code we can see the `call`, `signal` and `post` statements which are used for calling commands, signalling events and posting new tasks and are the major extentions of the nesC language as far as statements are concerned.

The structure of a module can be seen at figure 3. The first section is a general description where the interfaces that are provided and the ones that are used are listed. In the second section we have the implementation of the commands and events for these interfaces. Apart from the commands and events, tasks together with helper functions and global variables can also be defined. These helper functions and the global variables are visible only inside the module and so they can used only by the tasks, the commands and events of this module and no other. Global variables are useful when data needs to be passed to a task. According to NesC, tasks can take no arguments, so all data must be stored to a global variable and then the task can read them.

An example of a real module can be seen at appendix B, page 55. It is the *HPLClock* component, ported for the smartits platform. It provides two interfaces, *StdControl* and *Clock*, and uses one, *PIC18F6720Interrupt*. Inside the module there is the C code that implements all the commands for the *StdControl* and *Clock* interfaces and the C code that implements the events of the *PIC18F6720Interrupt* interface. Also, there is another helper function, *setInterval*, that is neither a command nor an event, but is used by some commands and events.

b. Configurations

Configurations are wiring specifications for a collection of components used to implement a new one. Every configuration uses a set of included components and a set of wiring specifications.

Wiring is used to connect interfaces, commands and events together. Every specification element has some endpoints. Using wiring you can link the endpoints of a set of elements. The wiring can only link compatible sets of endpoints.

An example configuration is shown below. It is the *ClockC* configuration which

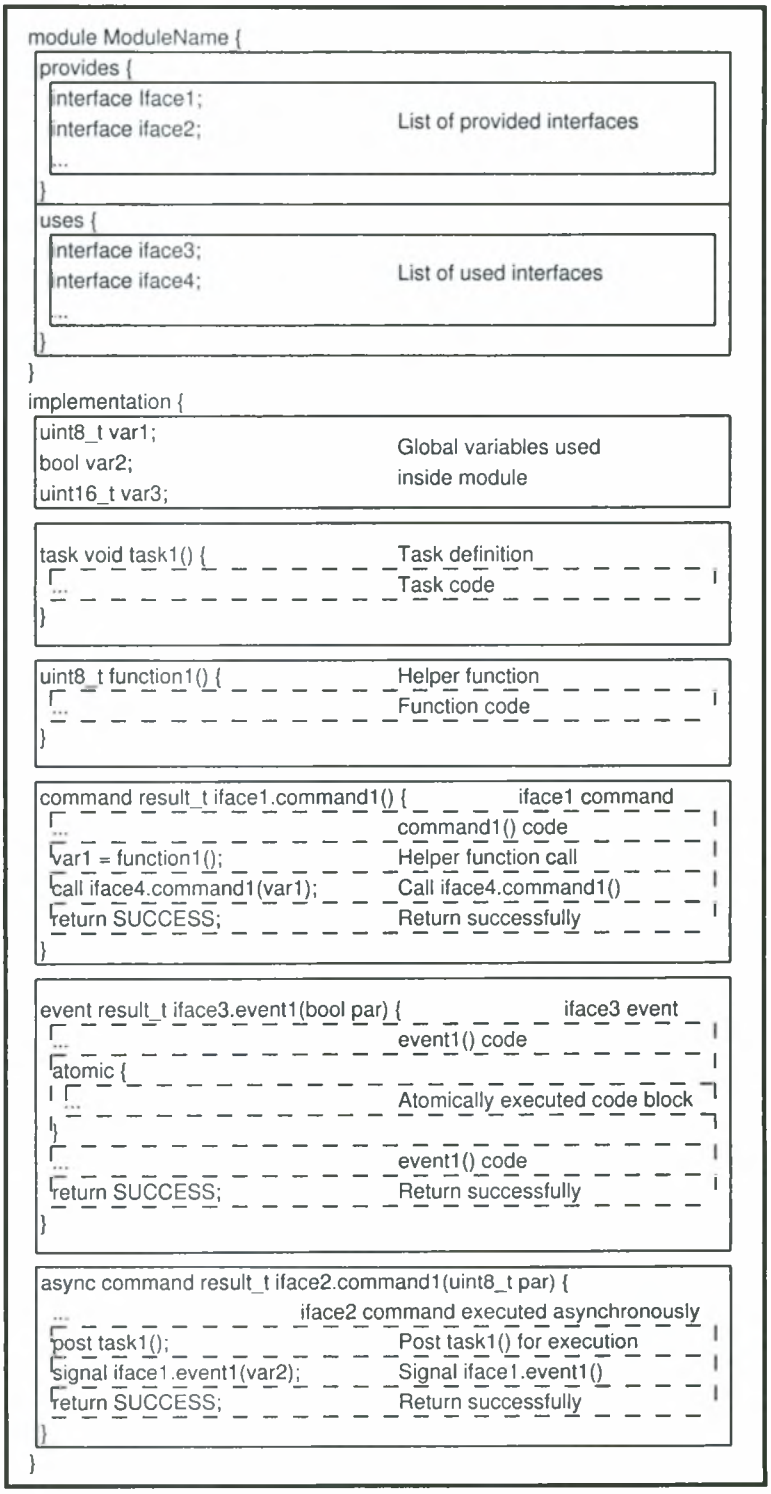


Fig. 3. Module structure

belongs to the platform specific code and it is used by the generic TinyOS code. It defines two components, *HPLClock* and *PIC18F6720InterruptC*. These are then wired together to finally provide two interfaces, *Clock* and *StdControl*.

```
configuration ClockC {
  provides {
    interface Clock;
    interface StdControl;
  }
}
implementation
{
  components HPLClock, PIC18F6720InterruptC;

  StdControl = HPLClock;
  Clock = HPLClock;

  HPLClock.TIMER1_Overflow -> PIC18F6720InterruptC.TIMER1_Overflow;
}
```

3. Flow of control inside a NesC program

Every NesC program is a configuration. It uses the *Main* component which is the equivalent of the main() function in C (actually internally it is translated to the main() function). This component uses the *StdControl* interface which has three commands that must be implemented, *init()*, *start()* and *stop()*. When a program starts, the *Main* component calls first *init()* and then *start()*. The user must implement these

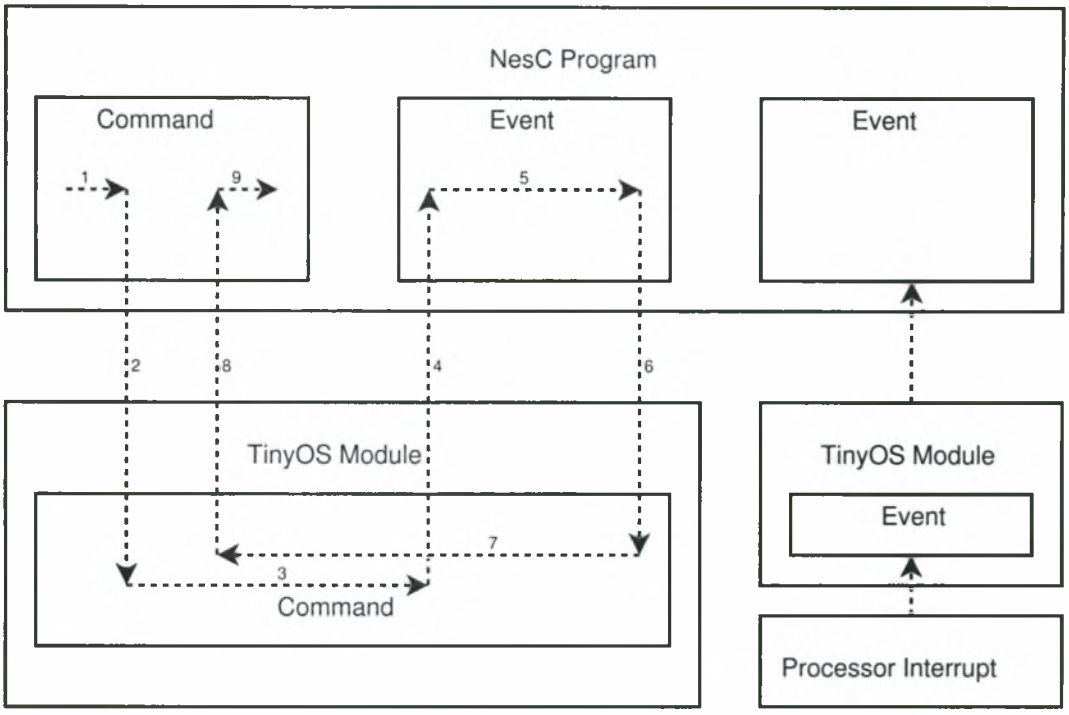


Fig. 4. Flow of control in TinyOS

commands which form the main program. Inside, these commands may call other commands from TinyOS or from other user written modules if the program is split into many modules. Some of the commands it calls may signal some events in the program so the program continues inside the event code until the event returns. Another way to change the flow of control is when an interrupt from the processor occurs. This can signal an event inside TinyOS which then usually signals an event in the user program.

A diagram showing the flow of control can be seen at figure 4. Each step is described below:

1. The program executes a command. When a program is started it begins execution inside two special commands, the *init()* at first and then the *start()* commands of the main component that implements the *StdControl* interface.

2. The program calls a command inside a TinyOS module it uses.

3. Code starts executing inside the command.

4. The command signals an event of the NesC program.

5. The code of the event handler is executed.

6. Execution finishes and control returns to the module's command.

7. The rest of the code is executed.

8. Execution finishes and control returns to the application's command.

9. The rest of the code is executed.

Apart from these steps there is another way to execute an event. At any time an interrupt from the processor may occur. Then, the event handler of a TinyOS module takes over which starts execution of code. Inside this event handler, an event inside the user's application can be signalled. This starts execution until it finishes. Then control returns to the module's event handler and when it finishes execution continues at the point where it stopped before the interrupt.

## B. Structure of TinyOS

TinyOS is a collection of interfaces, system code, libraries and modules for every platform and sensorboard.

The system code is some general code that is used mainly by other system components. It is not platform-depended so the code is the same for every platform. Examples of system code is the random number generator *RandomLFSR*, the timer *TimerM* and the CRC calculator *CrcC*.

Libraries are bigger "packages" of code that are used by user programs. They usually consist of a collection of code that provides interfaces which are directly wired and used by the user. An example are the counters. There are a number of counters which can take an integer from a source and output it to a destination, e.g. *IntToLeds* which takes an integer and outputs it to the leds and *RfmToInt* which takes an integer over RF. A simple program could just wire together these two components provided by the counters library and output and integer taken using the RF communications to the leds.

The modules for the various platforms and sensorboards consist of the actual C code that handles all low-level operations. Each platform and sensorboard has it's own hardware which is different and must be handled in a special way by the operating system. This code is usually plain C code or even assembly in some special occasions when very low-level operations must be done.

Finally, there is a special build system, some utilities and java libraries that help in the development of a program. These are not the core of the operating system but rather a collection of tools to help the programmer.

There are a number of different interfaces. Most of them are platform independent but there are a number of low level interfaces that must be implemented differently for each platform. A brief list of the interfaces is provided in figure 5. At the left there are the generic interfaces common for all platforms and at the right there are the platform specific interfaces. They are matched in groups whose interfaces are related. The distinction between platform dependent and platform independent may sometimes change according to the nature of the platform.

The most important interfaces that are used in sensor network applications are briefly presented below.

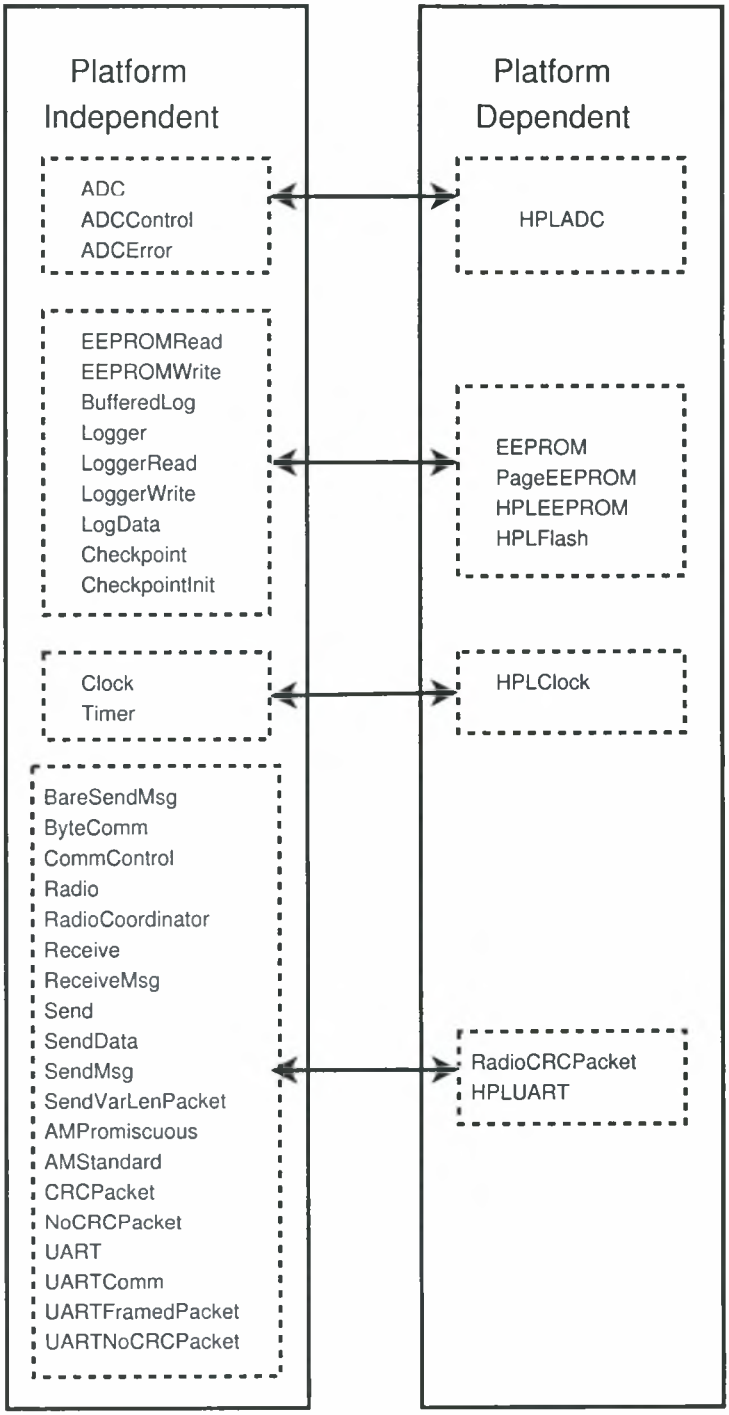


Fig. 5. Interfaces overview

1. Analog To Digital Converter

The analog to digital converter interface is mainly used to collect data from the various sensors. A lot of sensors return data in an analog form and these must be converted to a number that can be later used. This is mainly done using analog to digital converters. In order to interface with these converters a number of interfaces is used. The main one is `ADC`. It contains two commands and one event. The first command, `getData()` starts a conversion from a specific analog to digital converter. When this conversion ends the `dataReady()` event is signalled. The second command is `getContinuousData()`. This starts a continuous conversion. It begins with the first conversion. When this ends, the `dataReady()` event is signalled and another one starts. This way, by implementing the `dataReady()` event you can have a continuous sampling of the data from the various external sensors.

2. Clock and Timers

The clock and timer interfaces are used in order to schedule events for a specific time. The `Timer` interface is the one that is used the most. It depends on the `Clock` interface and schedules an event to be signalled after a specific amount of time. It contains the `start()` and `stop()` commands and the `fired()` event. Every time you call `start` you specify the type of the timer and the interval. If the timer is of the type `TIMER_ONE_SHOT` then after interval binary milliseconds[2] the `fired()` event is signalled and the timer is stopped. If the type is `TIMER_REPEAT` then every interval binary milliseconds the `fired()` event is signalled. In order for the timer to stop you must call the `stop()` command.

[2]One binary millisecond equals $\frac{1}{1024}$ seconds.

3. EEPROM and Flash memory access

Another interface that is important is the EEPROM and Flash memory access. It is used for storing various data in a non-volatile memory so they can be later processed. There are two interfaces for reading the EEPROM or Flash memory, `EEPROMRead` and `EEPROMWrite`. The `EEPROMRead` interface contains one command, `read()` that is used to start reading from a specific location in the memory. When the read is complete, the `readDone()` event is signalled which means that the data are already read. The `EEPROMWrite` interface is a little bit more complicated. It contains three commands, `startWrite()`, `write()` and `endWrite()`. `startWrite()` is called before any write. After calling it you can start calling the `write()` command to begin writting. When a write is completed the `writeDone()` event is signalled. When there's nothing else to be written `endWrite()` must be done. When it finishes it signals `endWriteDone()` and the write procedure is finished.

4. RF communications

One of the most important parts are the RF communications. The model presented below is the first one that was created and belongs to the mica platform.

The basic communications stack diagram is presented in figure 6.

The four top levels, from `Application` to `RadioCRCPacket`, are generic and are common to all radio stacks supported by TinyOS. From the `MicaHighSpeedRadioM` level and moving downwards all levels are specific to the mica platform. Using the `GenericComm` interface a user can send packets with custom content to other wireless sensors. Also, a lot of libraries use this interface, e.g. `IntToRfm` which sends an integer using the rf. The `GenericComm` interface provides the parameterised interfaces `SendMsg[uint8_t id]` and `ReceiveMsg[uint8_t id]` that are mainly used for

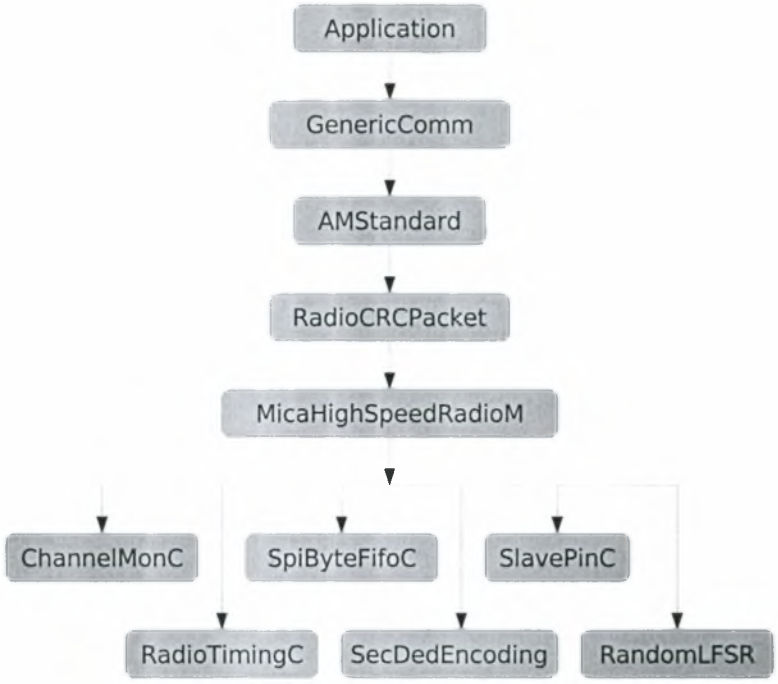


Fig. 6. Mica Communications Stack Diagram (interfaces)

communications. Interface `SendMsg` provides one command, `send()` and one event, `sendDone()`. The `send()` command is used to send a packet. After sending it the `sendDone()` event is signalled. The `ReceiveMsg` interface just defines the `receive()` event which is signalled every time a packet is received.

All packets have a common structure which is defined in a special file. The structure is the following

```
typedef struct TOS_Msg
{
  uint16_t addr;
  uint8_t  type;
  uint8_t  group;
  uint8_t  length;
  int8_t   data[TOSH_DATA_LENGTH];
```

```
  uint16_t crc;
} TOS_Msg;
```

The first field is the destination address. Each wireless sensor has it's own address and a packet can be send only to this sensor by putting the appropriate value at this field. Furthermore, there is a special address, `TOS_BCAST_ADDR`, which is the broadcast address, and all packets that have this destination address are delivered to all sensors.

The second field is the type of the packet. This is defined by the user and depends on the application. It is used in order to discriminate different types of packets and process only the ones needed.

The third field is the group. Again this is a characteristic of the packet. There is a default group for all packets, `TOSH_DEFAULT_AM_GROUP`.

The next field is the length of the data contained inside a packet. This can have the maximum value of `TOSH_DATA_LENGTH` which by default is equal to 29. Together with the other 7 bytes of the packet we have a maximum packet size of 36 bytes. It should be noted here that as we're going to see, this is not the number of bytes transmitted over the air but the number of bytes in the packet.

After the length comes the real data. They are at most `TOSH_DATA_LENGTH`.

Finally, the crc comes. It is used in order to check if the data received are valid.

This is a general description of the upper level of the RF communications stack. As we move to a lower level there are a number platform specific components. Each platform uses it's own combination of components which should be analyzed separately when studying the platform's RF communications hardware.

# CHAPTER IV

## PORTING TINYOS TO THE SMART-ITS PLATFORM

### A. Major differences between berkeley motes and smart-its

There are a number of important differences between the berkeley motes, the wireless sensors TinyOS was originally designed for, and the smart-its. These are presented at Table I.

| | Smart-its | Berkeley motes |
|---|---|---|
| Microcontroller | Microchip's PIC 18F6720 | Atmel's ATmega103 |
| RF Communications | RF Monolithics' TR1001 connected through the SPI and UART bus | RF Monolithics' TR1000 connected through the SPI bus |

Table I. Berkeley motes and smart-its major differences

### B. The microcontroller

The smart-its use the PIC 18F6720 by microchip, an 8-bit RISC microcontroller. In contrast, berkeley motes use atmel's ATmega103 from the AVR family, another 8-bit microcontroller with a RISC core. The two microcontrollers have a different number of peripherals and a different way to handle them.

White studying the microcontroller architecture, we uncovered the following problems:

- The real time clock handling is different.
- The analog to digital converter has a different way to get analog value, store them and has a different number of analog input lines.

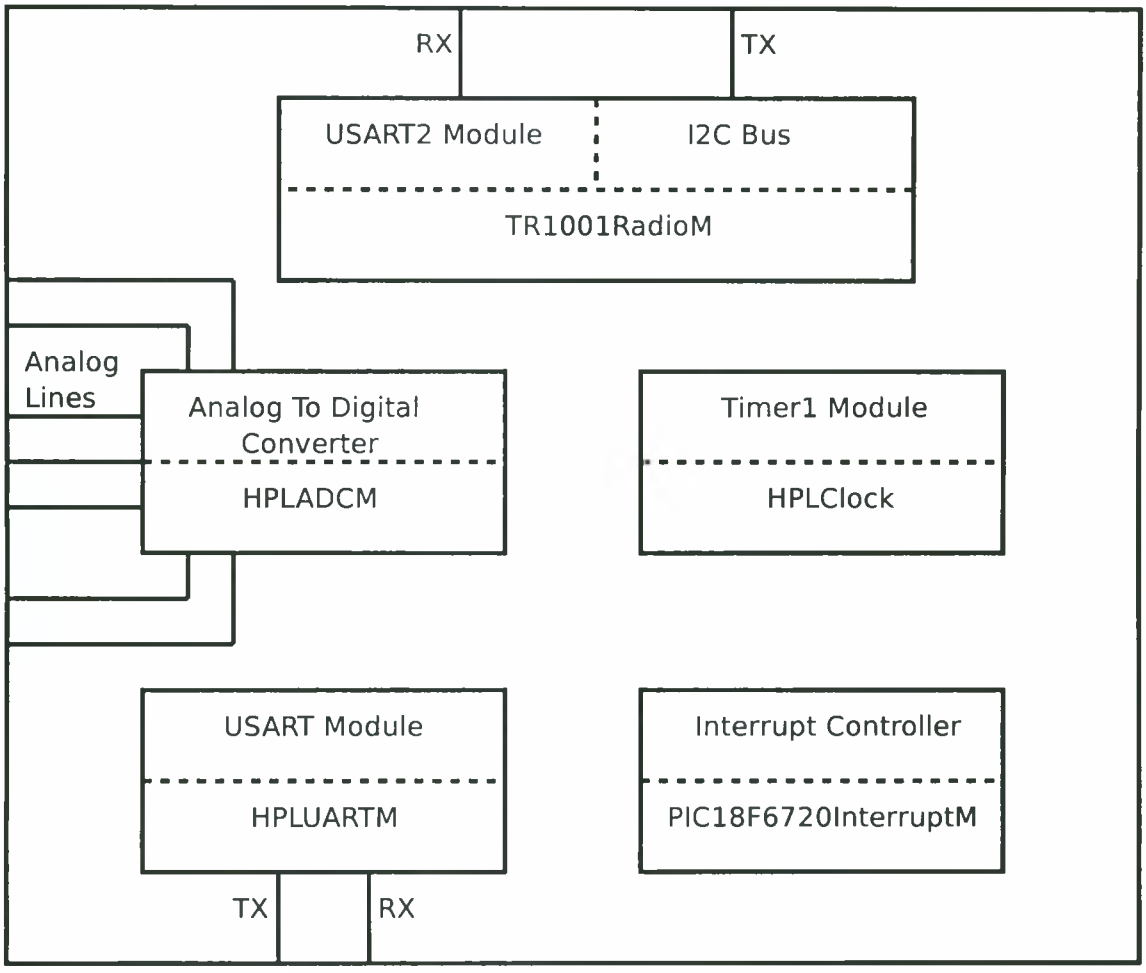


Fig. 7. Basic microcontroller architecture and modules

- There is a UART module on the microcontroller but the serial board uses software emulation.
- Interrupt handling is done in a total different way by the interrupt controller.
- The RF chip and the connection to the microcontroller is different.

The basic architecture together with the modules written for the port are presented in figure 7.

The first four hardware modules are presented below. For the RF communica-

tions part a special section has been written.

1. Real time clock

The hardware that supports the real time clock is a 32 kHz externally connected crystal. In order to use it the Timer1 components of the microcontroller is used. Timers can be used as counters and as clocks. They have a register associated with them that counts with a specific rate and each time an interval passes an interrupt occurs. One big difference between the AVR and the PIC family as far as timers are concerned is that the AVRs count from 0 to a specific number and then the interrupt occurs so that you know an interval has passed and the PICs start from a specific number and count until 0xFFFF. Because TinyOS was originally written for the mica motes it uses the AVR representation. So, there are always two representations kept, the PIC one which is used internally and the TinyOS one which is the one the programmer uses. Each time a function that involves setting or getting the clock is called one on-the-fly conversion happens that is transparent to the programmer. Finally, the values for the hardware scaling and intervals for some standard times (msec, sec, etc) had to be computed.

2. Analog to digital converter

The PIC 18F6720 microcontroller has a 10-bit analog to digital converter module which offers 12 analog inputs. The smartits however use only 6 of them. In order to have a successful conversion a number of timings had to be taken care of. There had to be a minimal interval between two conversions and between the time the analog to digital converter module is powered on and the time it starts converting. In order to avoid the second problem we could power on the analog to digital converter and never power it off but this would be very power consuming. So, delay routines had

to be written. In order to achieve maximum accuracy all the code was written in assembly.

3. Serial connection

The PIC 18F6720 microcontroller has two addressable universal synchronous asynchronous receiver transmitter (USART) modules. The second one is used for RF communications so that leaves only the first one free. Unfortunately, the 1/82 serial board doesn't use these pins but two other digital pins so a UART had to be emulated using software. The code needed as much accuracy as possible, especially if working with high speeds is desired so it was written in assembly. The speed that it uses is the standard 57600 bps with 8 data bits, no parity and 1 stop bit (8N1). There are two main problems introduced here. The first one is that since the communications are done using software emulation all that time the microcontroller can do nothing else. As it can be seen this is very time consuming but there is no other way to achieve UART communications. The AwareCon protocol and the runtime TecO provides use a software emulation too. The second problem is that we have no way telling if there are incoming data. When using the USART module there is a special interrupt when a byte is received but since an emulation is used this is impossible. The TecO runtime has a receive function but it is used synchronously which can't happen at TinyOS since it a completely asynchronous environment.

4. Interrupt handler

The PIC microcontroller has an interrupt vector which is a standard address where the program jumps when an interrupt occurs. In contrast the ATmega103 microcontroller has a different interrupt vector for every interrupt. TinyOS was designed so that each component can handle an interrupt by it's own which is accomplished by declaring

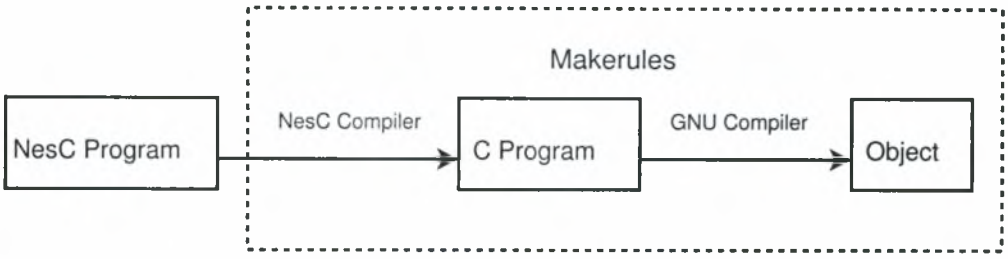


Fig. 8. Compile procedure for the mica platform

some functions as interrupt handlers and leaving the compiler do the final placement. In order to accomplish something like this a special interrupt handler had to be written which checks what interrupt has occurred and jumps to the appropriate function each time it is called.

## C. The compiler

TinyOS was written to work with the gnu toolchain and the avr-gcc compiler but there is no gnu toolchain for the PIC microcontroller. The usual compile procedure for the mica motes is presented at figure 8. The makerules are the series of commands needed to compile a program written inside a special file used by the "make" utility for the build process.

So the code must be transformed in order to compile using another C compiler which supports the PIC architecture. There are a number of different compilers but the one chosen is C18 by microchip. It was chosen because it is a full-featured ANSI compliant compiler and there is a free student version.

The problem that came up is the incompatibility of the C code produced by the nesC compiler with the C18 compiler. There are a number of keywords and extensions that are used by avr-gcc and they don't exist in C18. So, a converter script had to be written. It should remove all extensions that don't exist in C18 and convert all

those who use another name.

Another issue is that there are some processor specific configuration bits, the fuses. They are used in order to change the behaviour of the processor and set some special features, like the watchdog timer and the oscillator selection. In order to program them you must use some preprocessor directives. Again, using the convert script, these are added to form the final C program.

Some special directives are also needed for the interrupt handler. As it was said a special interrupt handler was written that jumps to the appropriate function. In order to declare a function as an interrupt handler a number of special directives have to be used which are not supported by the nesC compiler. So the interrupt handler wasn't written inside the main module of the nesC code but was put inside the final program by the converter.

Apart from the C code problems there was the assembly problem too. As it was said before some parts had to be written in PIC assembly. Since nesC uses the avr-gcc compiler there is no way to "understand" and put inline PIC assembly code. So all assembly was written externally and compiled into libraries. If an assembly function was used by the final program, after compiling it had to be linked against the corresponding library. So there had to be a way to understand which libraries are needed each time so that we link only against them to make the final binary as small as possible. So another script was created that checks the functions that are used and creates a final linker script for the compiler.

A sample run of the script that creates the final linker script for a program that uses serial communications but doesn't use exact timers is presented at figure 9.

In order to create the converter the perl scripting language was chosen. It is very good for searching and replacing text and it has support for regular expressions. The final program can be seen at Appendix A, page 50. Since the library searching

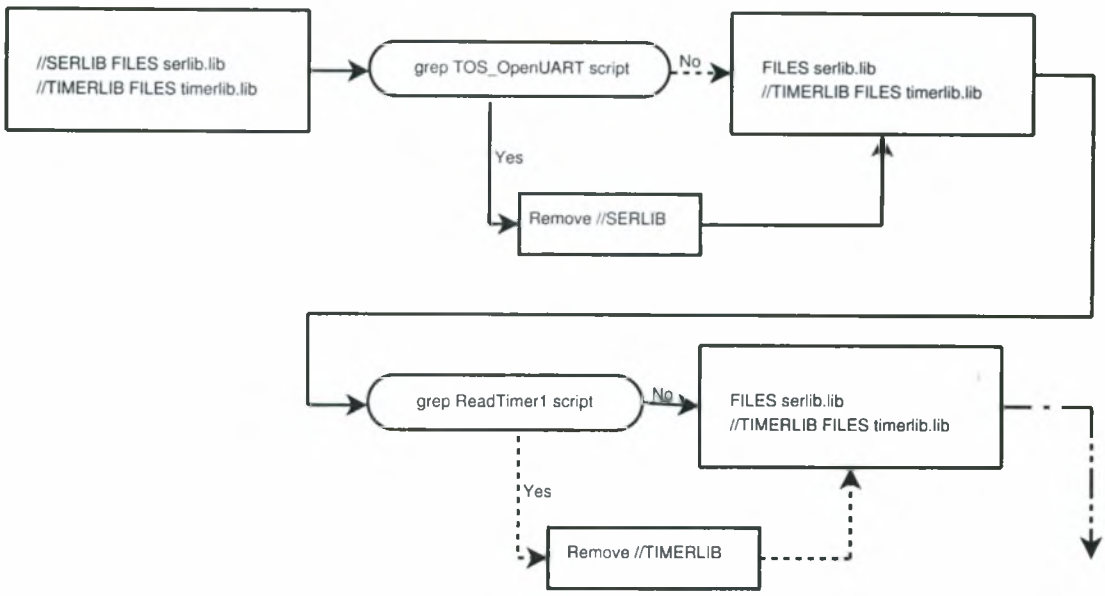


Fig. 9. Creation of a linker script

script needs to do a lot less work that could be easily done with the "grep" and "sed" utilities, it was written as a shell script.

To finish all these a special make rule was written that is used each time a program for the smartits platform is produced. Instead of compiling the nesC program and giving the final binary file, a complete MPLAB project is produced. MPLAB is an IDE made by microchip which is fully compatible with the C18 compiler.

So, the final compile procedure for the smartits port can be seen at figure 10.

## D. The RF communications

Both the smart-its and the mica motes use an RF chip from the ASH transceivers series by RF Monolithics in On-Off Keying (OOK) mode. This type of modulation represents digital data as the presence or absence of a carrier wave. The biggest difference is that at the mica motes the AVR microcontroller is connected to the RF chip using the SPI bus and at the smart-its the output from the microcontroller to

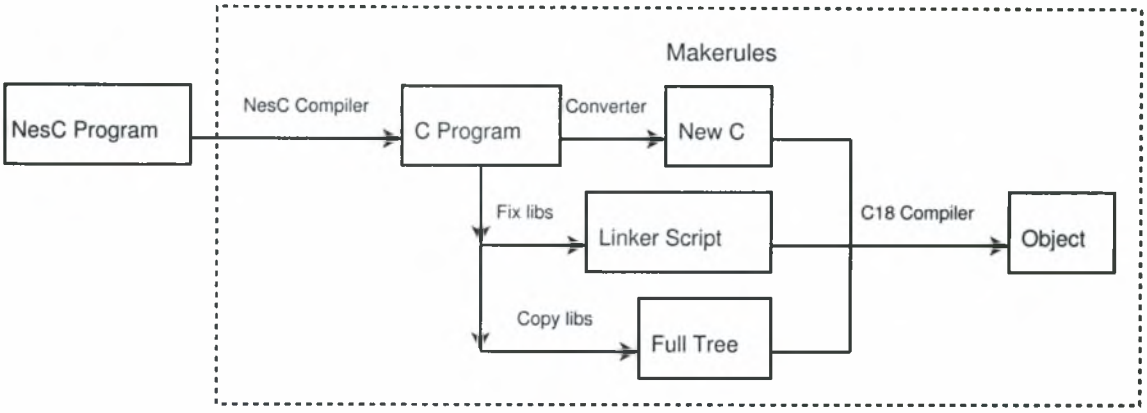


Fig. 10. Compile procedure for the smartits platform

the RF chip is done using the SPI bus and the input using the UART bus.

The biggest difference between the SPI and the UART bus is that the first one has a clock line and so it is synchronous and the second one has no clock and is asynchronous. The UART bus transmission is presented in figure 11. Both ends must have already agreed at a common speed. The transmit line is normally high at the sender which means that the receive line is also high at the receiver. When transmission is about to begin then a start bit is transmitted which is a 0. After this start bit, the data bits follow. Finally, an end bit which is a 1 is transmitted and the voltage remains high until the next start bit is sent. In contrast the SPI bus doesn't have to use a start and a stop bit. The SPI bus transmission can be seen in figure 12. Apart from the transmit line there is the clock line. Each time the clock line voltage goes high the next bit is transmitted by the sender. The receiver is able to understand a new bit is coming by watching the clock line. So to transmit a byte only 8 bits are sent.

This leads to the following problem. In order for the receiver to understand a byte comming over the UART bus it must get 10 bits which is not possible since the transmission is done over the SPI bus which sends only 8 bits.

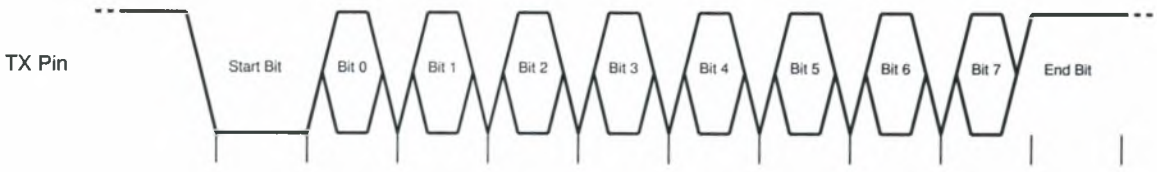


Fig. 11. UART bus transmission

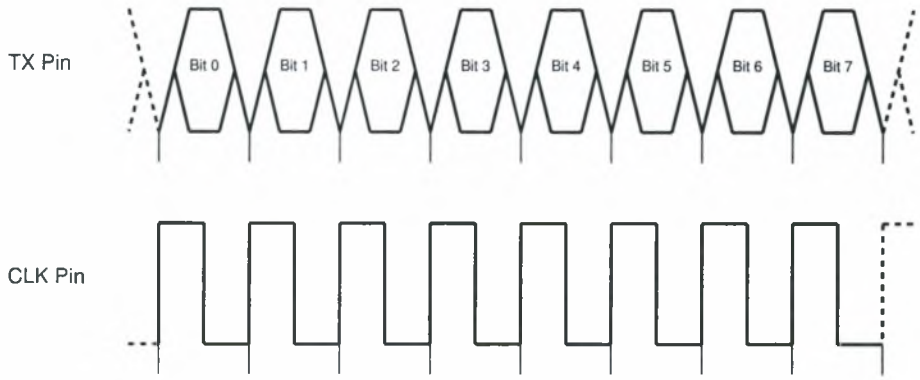


Fig. 12. SPI bus transmission

Also, another problem is that the line cannot be high all the time as needed by the UART protocol. In order to do so there must be a carrier wave all the time which is not possible. After some tests it was discovered that the PIC would detect a start bit by only checking for a falling edge at the receive line. So, a 1 must prepended before the start bit. When going from the first bit which is a 1 to the second bit, the 0, the processor understands the falling edge and begins the reception. However, this introduces yet another bit of overhead (a total of 3 to send a byte).

Another problem that came up concerned the final stop bit. If a 1 was sent at the end then when the transmitter stopped sending the processor would see a falling edge and believe a new byte is going to arrive. Again, after testing, it was discovered that the PIC ignores the value of the stop bit. Since it only detects a falling edge it doesn't care whether there was a high voltage or not. It uses the internal baud generator to get the first 8 bits and then waits again for the next falling edge. So, by

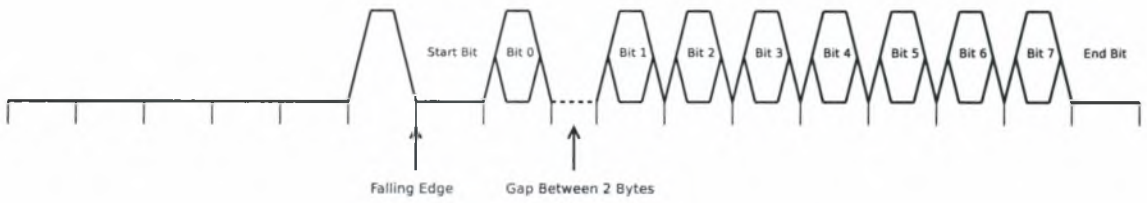


Fig. 13. Data received at the UART

changing the stop bit from a 1 to a 0 fully compatibility with the UART protocol is lost but the problem is fixed.

But again the original problem still exists, a single byte consisting of 8 bits can only be transmitted using the SPI bus and the receiver at the other end must get 11 bits in the UART bus input line. In order to solve this problem the following approach was used. Two bytes one after another are sent with the same speed and no delay between them (there is a very small delay because an assembly instruction is needed to start sending the next byte). The first one consists of five 0, one 1 to get the high voltage, a 0 in order to get the falling edge, which is the start bit, and the least significant bit of the byte to be transmitted. The second byte is the second least significant bit to the most significant bit and finally a 0 which is the stop bit we use (not UART protocol compliant). Something that needs to be noted is that the order of the bits needs to be changed because SPI transmits the most significant bit first and the UART expect the least significant bit as the first one.

The final output is presented at figure 13.

As it can be seen there is a little delay between the first and the second data bit received by the UART. This is due to the fact that they belong to two different bytes transmitted over the SPI bus as it was explained before. In order to make this delay as small as possible the transmission code was written in assembly.

After the low level hardware part was completed the MAC protocol should be

written. The decision was to use the same MAC protocol as the mica motes. After some tests the higher speed that could be used without having problems because of the low processing power was 2.5 Kbps[1]. By using the AwareCon protocol stack a much greater speed can be achieved but this happens every 13 msec, so effectively the speed is much lower. The smartits are originally designed to work with the AwareCon protocol so all external components connected to the TR1001 are tuned to work with this high speed. So the MAC protocol had to be changed in order to train the TR1001 a little more. Also some other values at the preamble had to be changed for the same reason.

The mica MAC protocol uses the Sec/Ded byte level encoding which allows single error correction and double error detection. The fallback is that encoding one byte produces three bytes. Another encoding that provides a basic error detection is the manchester encoding. It doesn't support error correction but one byte is encoded to two bytes, which gives a 33% gain. This could save some bandwidth but after some tests it was clear that error correction was needed and so Sec/Ded encoding was chosen.

[1]Mica motes use a much higher speed but they have an AVR processor with 20 MIPS processing power and smartits a PIC with 10 MIPS. Furthermore, smartits need more processing power because of the many things that need to be emulated using software.

# CHAPTER V

# EVALUATION OF RF COMMUNICATIONS

## A. Packet Loss

RF communications is probably the most important part in an operating system for wireless sensor networks. What is more interesting in communications is the final packet loss. Packet loss depends on many factors.

### 1. Transmission rate

The transmission rate is one of the most important factors. The increase of the packet loss occurs due to the fact that the receiver needs to read more bytes. Even if the packet contains a small amount of data the receiver still reads the preamble for the training, the start of packet symbols, the header and the trailing bytes after the data. All these need to be processed later. Even if the transmitter also has to transmit a high number of bytes, it transmits them in a synchronous manner and doesn't need to handle interrupts so it has no work load problem. Furthermore, after each packet an acknowledge is sent which takes an amount of time too. That means that the bigger the transmission rate is, the higher the packet loss.

### 2. Packet size

The packet size can also influence the packet loss. The bigger the packet size the more bytes are sent and need to be received and processed at the receiver. It should be noted here that each byte is encoded to three bytes in order to have the error correction and detection. That means that by just using a packet that is one byte bigger the receiver needs to receive three more bytes and decode them which is a very

time consuming process.

3. External work load

Apart from the work load for receiving and decoding packets there is the external work load. Receiving data from the analog to digital sensors or using the clock is one type of external work load. But the bigger problem is when using emulation for various external components such as the serial UART. This type of external work load is the heavier and the application needs to be designed in a way that it avoids as much as possible using those components.

4. Distance and environment

Distance and environment play a very important role too. The sensors transmit using a very low transmission power. That means the bigger the distance the most likely the packets are lost over the air. This is a problem that doesn't depend on the operating system but on the hardware design of the sensors. Also, the packet loss rate may also increase when there is a high radio interference from the environment. External factors such as other transmitters operating at the same frequency may cause such an interference.

B. Measurements

A number of measurements have been taken in order to see the packet loss when changing the values of the various variables.

The testing environment included two sensors, the transmitter and the receiver. They were at a very close distance in order to minimize the distance and environment factors since the measurements are for the processing power of the sensors.

The transmitter submitted a packet with a variable size at a different number of rates at each execution. It contained a counter which was incremented when a transmission occurred and this counter was sent as the first byte of the packet.

On the other end, the receiver had another counter. This counter was incremented every time a packet was received. When this counter reached number 30, which means 30 packets were received, it sent the first byte of the current packet to a computer using the serial port. Then, the counter was reset again to 0.

A program at the computer took the number and subtracted from it the one it took last time from the receiver. This is the number of packets the transmitter sent from which the receiver took only 30. Using these numbers the packet loss was calculated.

All of the following measurements have been taken with a very close distance between the two sensors in order to see the behaviour that depends on them and not distance.

In figure 14 we can see how the packet loss changes when we change the transmission rate using a one byte packet.

As it can be seen the packet loss is very low and isn't a monotonic function of the transmission rate. There are points where we have no packet loss at all. This can be explained because a one byte packet is very small and can be easily handled by the receiver. The packet loss we have at some points is due to some external factors such as radio interference.

In order to have a better understanding the same measurements were taken using a five byte packet. The results can be seen at figure 15.

What can be seen here is that the packet loss is an increasing function of the transmission rate. A five byte packet is acceptable when one packet per second is transmitted but when the transmission rate increases to two packets per second then

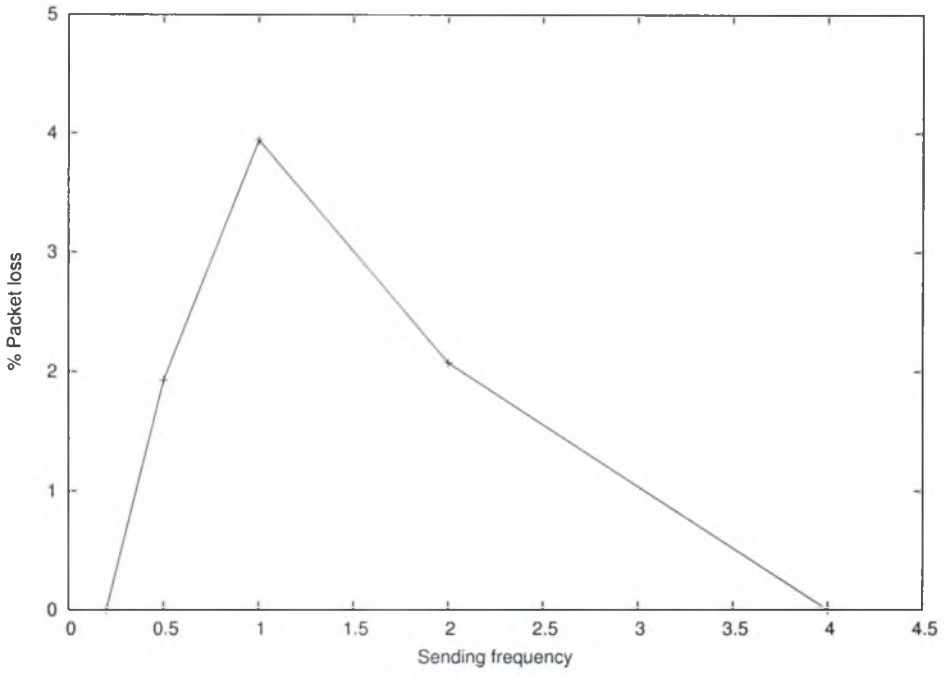


Fig. 14. Packet loss vs Transmission rate for one byte packets

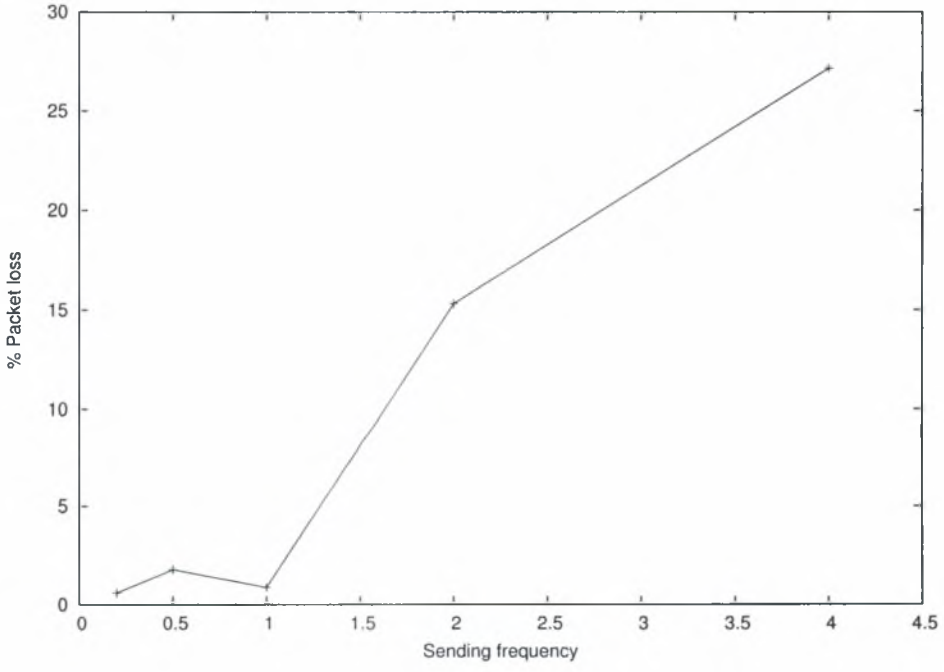


Fig. 15. Packet loss vs Transmission rate for five byte packets

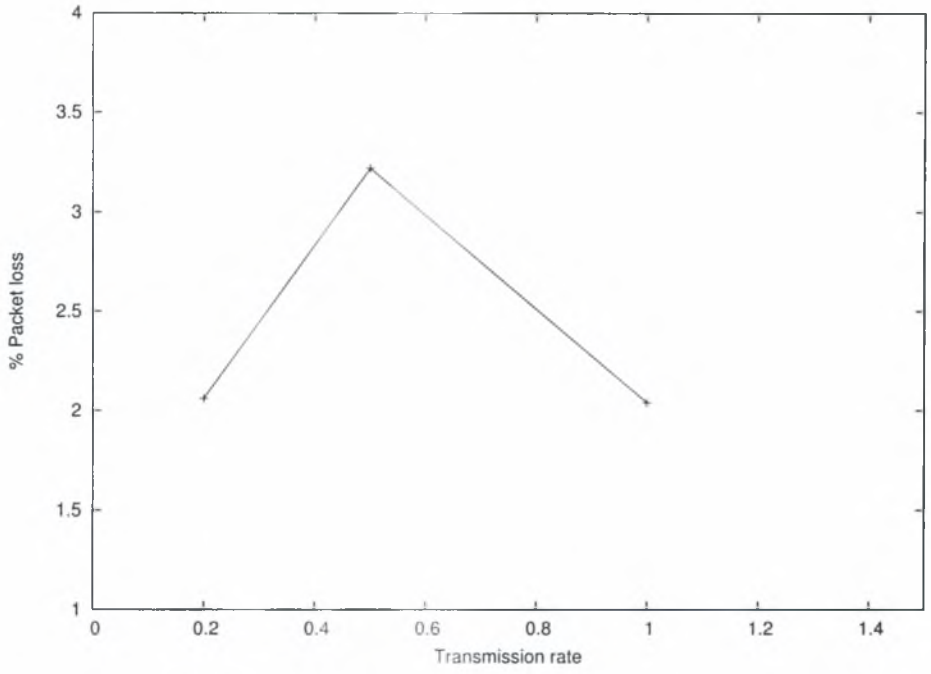


Fig. 16. Packet loss vs Transmission rate for twenty byte packets

limitations from the hardware start to come up. The microcontroller can't handle receiving so many bytes at this rate and starts dropping them. At four packets per second it is clear that the problems continues and it is much bigger.

The final measurements where made using a twenty byte packet. The results can be seen at figure 16.

Values are presented only for one packet per second, one packet per two seconds and one packet per five seconds. Measurements were made for two packets per second and four packets per second but the results were too big to display. At the highest frequency there was a packet loss of about 90%.

As a conclusion, it can be seen that for the maximum performance a program that needs a high transmission rate must send small packets (around five bytes is a good value). In case the transmission rate decreases the packet can have any size with a very small packet loss.

1. Changing the distance

The distance between the sensors is very important. If the sensors have a distance of a few meters then the packet loss can increase and reach almost 100%. The RF chip's capability of distinguishing between a 0 and an 1 becomes very limited and so unrecoverable errors are introduced.

This is due to the fact that the sensors are designed to work at a different, much higher speed. The AwareCon stack uses the speed of 125kbps to send all packets. So the hardware is adjusted to work better with this speed. Since it is a completely synchronous environment there exists no problem with such a high speed once the sensors are synchronized. Every 13ms the AwareCon stack "knows" it's going to receive a new packet so it starts reading. But TinyOS has a different philosophy. It is an asynchronous environment and it must keep a complete state machine because a packet can arrive at any time. Since the processing power is limited, working with such high speeds and making such computations leaves no time for doing other work.

2. Changing the work load on the transmitter

As part of the tests a load was introduced to the transmitter. Instead of transmitting a dummy packet, it took the readings from a sensor and transmitted them. The results were the same as above and this is due to the asynchronous nature of TinyOS. At the test program the transmission occurred when a reading was ready. So, the transmission could be delayed due to the analog to digital conversion but this happened while the sensor was waiting to send the command to start a new conversion. The time a conversion takes to complete is too small comparing to the time we wait to start a new conversion so there is no problem introduced. So, for carefully designed applications which take full advantage of the features of TinyOS a small increase of

the work load has no effects on the packet loss.

3. Changing the work load on the receiver

There are many ways to introduce load to the receiver. There can be load due to data processing by the program or due to some internal kernel command. Having a heavy load due to a user program demanding a lot of cpu has no impact on the receiver. When a new packet arrives an interrupt is thrown so the user program stops it's processing and the microcontroller is free to receive the packet. Then, when it finishes, operation can continue from the point it stopped in the user program. On the other hand, an internal kernel command that must be executed asynchronously can have a big impact when receiving a packet. An asynchronous command disables interrupts when it starts and re-enables them when it returns to the caller. If a new packet arrives during this period then it is discarded. In order to avoid this problem the parts that are executed asynchronously should be as small as possible. An example of an asynchronous operation is UART transmission. Because of the design of the particles software uart emulation must be used. In order to achieve the baud that we need there are some strict delays we follow. An interrupt while transmitting would make the sender desynchronize and change transmission baud. So, if a packet arrives when the program transmits a byte over the UART then this packet is lost.

C. AwareCon and TinyOS

One of the early thoughts was compatibility between the AwareCon and the TinyOS stacks. Unfortunately, this is not possible because of the big differences between the two operating systems. The AwareCon stack needs to exchange data every 13ms and

leaves the rest of the time for the user application. On the other hand TinyOS has no such limitations. If communications are not needed by the user application no data exchange is made. Also, due to the asynchronous nature of TinyOS we can't be sure that every 13ms the microcontroller will be able to handle the RF communications. For example, at this time the interrupt handler may be handling an interrupt used by the *HPLClock* module for the real time clock and so it won't handle the RF communications correctly. This results in loss of synchronization with the rest of the network. After noting all these differences it was clear that a new stack should be used and compatibility with AwareCon wasn't possible.

# CHAPTER VI

## CONCLUSION

One of the major contributions of this thesis is the analysis of the smart-its platform and identification of its shortcomings when it comes to running a full-fledged sensor operating system. As it was shown a working port of TinyOS is made even if there are many limitations by the hardware. Further tests should be made and real world applications should be tested. The high packet loss when the sensors are not close to each other may be discouraging but with careful design of the application and with the use of routing protocols we can overcome this problem.

Future work includes porting TinyOS to the new generation of smart-its. Almost all upcoming particles use a much more advanced RF chip which requires less processing power from the microcontroller and allows much higher speeds. The rest of the hardware and especially the microcontroller is the same so not much work has to be done there.

# REFERENCES

[1] A. Hac, *Wireless sensor network designs.* John Wiley and Sons, 2003.

[2] N. Lee, P. Levis, and J. Hill, *Mica High Speed Radio Stack.* September 2002.

[3] P. Levis, and N. Lee, *TOSSIM: A Simulator for TinyOS Networks.* September 2003.

[4] D. Gay, P. Levis, D. Culler, and E. Brewer, *nesC 1.1 Language Reference Manual.* May 2003.

[5] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[6] International Standard ISO/IEC 9899, *Programming Languages - C.* Second Edition, December 1999.

[7] Microchip Technology Incorporated, *PIC18F6520/8520/6620/8620/6720/8720 Data Sheet.* 2004.

[8] Maxim Integrated Products, *Maxim 5160/5161 Low-Power Digital Potentiometers.* 2001.

[9] RF Monolithics, *ASH Transceiver Designer's Guide.* May 2004.

[10] RF Monolithics, *ASH Transceiver Software Designer's Guide.* May 2004.

[11] Universitaet Karlsruhe, Telecooperation Office, *Particle Base System Source Code.* 2005.

[12] TinyOS Alliance, *TinyOS 1.1 Source Code.* 2006.

[13] Geoff Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh, *Monitoring Volcanic Eruptions with a Wireless Sensor Network.* EWSN'05.

[14] Safwan Al-omari, Weisong Shi, and Carol J. Miller, *SESAME: SEnsor System Accessing and Monitoring Environment.* November 2004

[15] Konrad Lorincz, David Malan, Thaddeus R. F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoff Mainland, Steve Moulton, and Matt Welsh, *Sensor Networks for Emergency Response: Challenges and Opportunities.* October - December 2004.

[16] Victor Shnayder, Bor-rong Chen, Konrad Lorincz, Thaddeus R. F. Fulford-Jones, and Matt Welsh, *Sensor Networks for Medical Care.* Harvard University Technical Report TR-08-05, April 2005.

[17] David Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton, *CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care.* International Workshop on Wearable and Implantable Body Sensor Networks, April 2004.

[18] C. Guo and A. Fano, *Cargo Container Security using Ad Hoc Sensor Networks.* IPSN/SPOTS 2005.

[19] Philip Levis at al. *Ad-Hoc Routing Component Architecture.* February 2003.

[20] Alec Woo, Terence Tong, and David Culler, *Taming the Underlying Challenges of Reliable Multhop Routing in Sensor Networks.* SenSys 2003.

[21] Wei Ye, John Heidemann and Deborah Estrin, *An Energy-Efficient MAC Protocol for Wireless Sensor Networks.* INFOCOM 2002.

[22] Wendi Rabiner Heinzelman, Anantha Chandrakasan and Hari Balakrishnan, *Energy-efficient Communication Protocols for Wireless Microsensor Networks.* HICSS 2000.

[23] F. Zhao, J. Liu, J. Liu, L. Guibas, and J. Reich, *Collaborative Signal and Information Processing: An Information Directed Approach.* Proc. IEEE, 2003.

[24] M. Chu, H. Haussecker, F. Zhao, *Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks.* Int'l J. High Performance Computing Applications, 16(3):90-110, Fall 2002.

[25] Chris Karlof, Naveen Sastry, and David Wagner, *TinySec: A Link Layer Security Architecture for Wireless Sensor Networks.* SenSys 2004.

[26] Prasanth Ganesan, Ramnath Venugopalan, Pushkin Peddabachagari, Alexander Dean, Frank Mueller, and Mihail Sichitiu, *Analyzing and Modeling Encryption Overhead for Sensor Network Nodes.* WSNA 2003.

# APPENDIX A

## NESC COMPILER C OUTPUT TO MPLAB C CONVERTER

```
#!/usr/bin/perl

$source = $ARGV[0];
$output = $ARGV[1];

if ($source eq $output) {
        die("perl: source and output cannot be the same");
}

open(SOURCE,$source) or
  die("perl: Could not open source-file: $source");
open(OUTPUT,">$output") or
  die ("perl: Could not open output-file: $output");

print OUTPUT "#include <p18f6720.h>\n";
print OUTPUT "#define _CONFIG 1\n";
print OUTPUT "#pragma config OSC = HS\n";
print OUTPUT "#pragma config OSCS = OFF\n";
print OUTPUT "#pragma config PWRT = ON\n";
print OUTPUT "#pragma config BOR = OFF\n";
print OUTPUT "#pragma config BORV = 25\n";
print OUTPUT "#pragma config WDT = OFF\n";
```

```
print OUTPUT "#pragma config WDTPS = 128\n";
print OUTPUT "#pragma config CCP2MUX = ON\n";
print OUTPUT "#pragma config STVR = ON\n";
print OUTPUT "#pragma config LVP = OFF\n";
print OUTPUT "#pragma config DEBUG = OFF\n";

while (<SOURCE>) {
  s/^\# [0-9]+/\/\//;
  s/^\#line [0-9]+/\/\//;
  s/\$/_/g;
  s/__inline//;
  s/inline//;
  s/__attribute\(\(packed\)\)//;
  s/__attribute__\(\(packed\)\)//;
  s/ __nesc_atomic_t __nesc_atomic/
    auto  __nesc_atomic_t __nesc_atomic/;


  if(/long long/) {
    s/(.*)long[ ]+long(.*)/\1long\2/;
  }

  if(/progmem/) {
    $_ = '//' .$_;
  }
```

```
if(/static const prog_uchar/) {
  $_ = '/*' .$_;
}

if(/TRUE \} \}\;/) {
  $line= $_;
  chomp $line;
  $_ = $line."*/\n";
}

s/int asm_nop\;//;

s/asm_nop = 1\;/_asm nop _endasm\n/;

s/int asm_sleep\;//;

s/asm_sleep = 1\;/_asm sleep _endasm\n/;

s/int asm_TBLWT\;//;

s/asm_TBLWT = 1\;/_asm TBLWT _endasm\n/;

s/int asm_clrwdt\;//;

s/asm_clrwdt = 1\;/_asm clrwdt _endasm\n/;
```

```
if(/^\/\//) {
  $_ = "";
}


if(/^static[ ]*\n$/) {
  $_ = "static ";
}


s/__attribute\(\(interrupt\)\)//;

if(/void    InterruptHandler\(void\)[ ]*\n/) {
  $_ = "\#pragma code InterruptVector = 0x08\n";
  $_ = $_ . "void InterruptVector (void)\n  ";
  $_ = $_ . "\{\n    _asm GOTO InterruptHandler _endasm\n  \}\n";
  $_ = $_ . "\#pragma code\n\#pragma interruptlow ";
  $_ = $_ . "InterruptHandler\nvoid InterruptHandler (void)\n";
}


s/int8_t crc8/rom int8_t crc8/;


s/uint16_t crc16/rom uint16_t crc16/;
```

```
  if(/int[ ][\w]+bits_/) {
    $_="";
  }

  s/([0-9A-z]+bits)_/ $1./g;

  if(/int[ ][\w]+_register/) {
    $_="";
  }

  s/([0-9A-z]+)_register/ $1/g;


  unless (/^\n/) {
    print OUTPUT $_;
  }
}

close(OUTPUT);
close(SOURCE);
```

# APPENDIX B

## HARDWARE PRESENTATION LAYER FOR THE REAL TIME CLOCK

```
/*
 * Clock handling.
 * Fotis Loukos <fotisl@inf.uth.gr>
 * Check Clock.h for some default values.
 */

module HPLClock {
  provides {
    interface StdControl;
    interface Clock;
  }

  uses {
    interface PIC18F6720Interrupt as TIMER1_Overflow;
  }
}
implementation
{
  bool set_flag;
  uint8_t mscale, nextScale;
  uint16_t minterval, picinterval, nextInterval ;
```

```
void setInterval(uint16_t interval) {
  uint32_t inttmp;
  uint16_t passed;

  /* Tinyos representation */
  minterval = interval;
  /* PIC representation, clocks before reaching 0xFFFF */
  picinterval = 0xFFFF - interval + 1;

  passed = TOS_ReadTimer1();
  inttmp = picinterval + passed;

  if(inttmp > 0xFFFF)
    TOS_WriteTimer1(0xFFFF);
  else
    TOS_WriteTimer1((uint16_t) inttmp);

  return;
}

command result_t StdControl.init() {
  atomic {
    set_flag = FALSE;
    mscale = DEFAULT_SCALE;
    minterval = DEFAULT_INTERVAL;
  }
```

```
  T1CONbits_TMR1ON = 1;                /* Timer on */
  T1CONbits_TMR1CS = 1;                /* External clock */
  T1CONbits_T1SYNC = 1;                /* Do not synch ext clk inp */
  T1CONbits_T1OSCEN = 1;               /* Oscillator enabled */
  T1CONbits_RD16 = 0;                  /* 8 bit values */

  /* Scaling */
  T1CONbits_T1CKPS0 = mscale & 1;
  T1CONbits_T1CKPS1 = mscale >> 1;

  TOS_WriteTimer1(0);

  return SUCCESS;
}

command result_t StdControl.start() {
  uint8_t mi;
  uint16_t ms;

  atomic {
    mi = minterval;
    ms = mscale;
  }

  call Clock.setRate(mi, ms);
```

```
  T1CONbits_TMR1ON = 1;
  return SUCCESS;
}

command result_t StdControl.stop() {
  uint16_t mi;
  atomic {
    mi = minterval;
  }

  call Clock.setRate(mi, 0);
  T1CONbits_TMR1ON = 0;

  return SUCCESS;
}

async command uint16_t Clock.getInterval() {
  uint16_t in;

  atomic {
    in = minterval;
  }

  return in;
}
```

```
async command uint8_t Clock.getScale() {
  uint8_t ms;

  atomic {
    ms = mscale;
  }

  return ms;
}

async command void Clock.setNextInterval(uint16_t value) {
  atomic {
    nextInterval = value;
    set_flag = 1;
  }
}

async command void Clock.setNextScale(uint8_t scale) {
  atomic {
    nextScale = scale;
    set_flag = 1;
  }
}

async command void Clock.setInterval(uint16_t value) {
  PIE1bits_TMR1IE = 0;
```

```
    atomic setInterval(value);

    PIE1bits_TMR1IE = 1;

    return;
  }

  async command result_t Clock.setIntervalAndScale(uint16_t interval,
    uint8_t scale) {

    PIE1bits_TMR1IE = 0;

    atomic {
      mscale = scale & 3;
      T1CONbits_T1CKPS0 = mscale & 1;
      T1CONbits_T1CKPS1 = mscale >> 1;
      setInterval(interval);
    }

    PIE1bits_TMR1IE = 1;

    return SUCCESS;
  }

  async command uint16_t Clock.readCounter() {
```

```
  uint16_t val;

  val = TOS_ReadTimer1();

  if(!val)
    return val;
  return(0xFFFF - val + 1);
}

async command void Clock.setCounter(uint16_t n) {
  /* You're smart enough to pass a value > 0 */
  TOS_WriteTimer1(0xFFFF - n + 1);
}

async command void Clock.intDisable() {
  PIE1bits_TMR1IE = 0x0;
}

async command void Clock.intEnable() {
  PIE1bits_TMR1IE = 0x1;
}

async command result_t  Clock.setRate(uint16_t interval,
  uint8_t scale) {

  uint32_t inttmp;
```

```
    uint16_t passed;

    PIE1bits_TMR1IE = 0;

    scale &= 0x3;
    T1CONbits_T1CKPS0 = scale & 1;
    T1CONbits_T1CKPS1 = scale >> 1;

    passed = TOS_ReadTimer1();
    inttmp = 0xFFFF - interval + 1 + passed;

    if(inttmp > 0xFFFF)
      TOS_WriteTimer1(0xFFFF);
    else
      TOS_WriteTimer1((uint16_t) inttmp);

    PIE1bits_TMR1IE = 1;

    return SUCCESS;
  }

  async event result_t TIMER1_Overflow.fired() {
    atomic {
      if(set_flag) {
        mscale = nextScale;
        T1CONbits_T1CKPS0 = mscale & 1;
```

```
        T1CONbits_T1CKPS1 = mscale >> 1;
        setInterval(nextInterval);
        set_flag=0;
      }
    }

    signal Clock.fire();

    return SUCCESS;
  }

  default async event result_t Clock.fire() { return SUCCESS; }
}
```