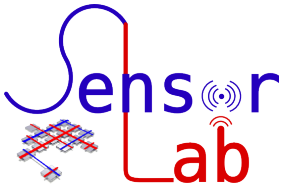
Faculty of Mathematics and Computer Science

– Georg-August University Göttingen –

Telematics Group

Lab 1 - Getting in Touch

Sensorlab



Contents

1 Introduction

Wireless Sensor Networks (WSN) are networks of distributed sensor nodes communicating wirelessly, with networks based around base stations or as ad-hoc-networks. Together they monitor the environment. It's possible to measure various conditions, such as temperature, air pressure, acceleration and sound. The field of WSNs is still rather new, but WSNs are now widely in use observing industrial processes, traffic and other environments. This lab is going to introduce you to sensor networks and sensor network programming.

Further Information

- Each of the lab exercises is graded (50 points for each lab). 30% of your final grade depends on the grading of the lab exercises.
- Please send your finished lab reports to: *benjamin.leiding@stud.uni-goettingen.de*
- To login on the sensorlab website (`http://user.informatik.uni-goettingen.de/~sensorlab/`) and download the lab documents, quick reference, etc., use the following login data:

  Username: `course`
  Password: See handout
- **Hint**: Always read all the way through the whole lab document once before you start working on assignments. Sometimes unclear things are explained right after the page break.

Useful links and literature

- An overview of our hardware (boards, etc.): `https://user.informatik.uni-goettingen.de/~sensorlab/Hardware.php`
- Getting started with TinyOS `http://tinyos.stanford.edu/tinyos-wiki/index.php/Getting_Started_with_TinyOS`
- TinyOS API documentation `http://www.btnode.ethz.ch/static_docs/tinyos-2.x/nesdoc/iris/`

- Datasheet for IRIS sensor motes `https://doc.informatik.uni-goettingen.de/wiki/images/3/35/IRIS_Datasheet.pdf`

Question 1.1 (6 Points)
Find out where wireless sensor networks are recently used. Give at least three examples.

1.1 Sensor Node

The sensor node is the component doing the monitoring itself. It gathers data from its surroundings and is, in many cases, able to preprocess it before sending it via wireless communications or storing it to an internal log. The core component of sensor nodes (or motes) are usually microcontroller. A radio transceiver, power source as well as analog-to-digital converters are usually also available. As hinted by the name, it is usually possible to attach additional circuit boards containing various sensors to a sensor mote.

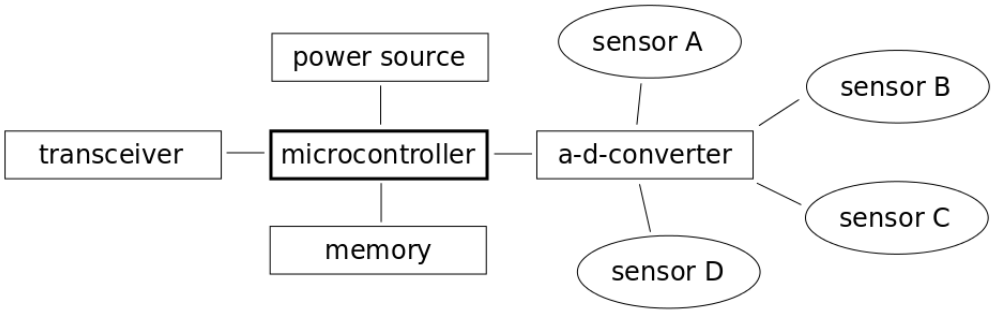


Figure 1: Concept of a sensor node

The microcontroller is not only the core component but also connecting all other components with each other. Microcontroller are mainly used because of their low cost, interoperability and low power consumption.

To communicate with each other and/or a base station, WSNs have a transceiver operating mostly in the *ISM band* (license-free). The transmission can use radio frequency, optical communication or Infrared.

The (external) memory can contain to kinds of data: application data and (gathered) data stored for the purpose of analysis.

The cost of wireless communication is a high power consumption for receiving and transmitting data. Far less power is needed to process the data acquired. Regularly, simple batteries are used as power sources (both rechargeable and not rechargeable). Also, high charge capacitors are a possible energy supply.

The sensors themselves are basically analog devices able to measure certain properties. They may have their own power supply or (mostly) make use of the battery feeding the microcontroller. In general, one distinguishes between three types of sensors:

- Passive
- Active
- Narrow-beam

Passive sensors do not interfere with their environment and do not need an external power supply to sustain their functionality whereas active sensors are invasively scanning sensors needing extra power to do this. Narrow-beam sensors are in between passive and active sensors and do not scan actively like a radar sensor but rather emit a narrow beam like sensors in elevator doors.

1.2 Operating Systems for Sensor Nodes: TinyOS

Question 1.2 (6 Points)
What are the requirements of an operating system for wireless sensor nodes? List and explain them.

All requirements for wireless sensor nodes are matched by *TinyOS*. *TinyOS* is an open-source operating system especially used for wireless sensor devices. It is virtually a standard for sensor networks. The main development takes places at the university of Berkley; an international alliance enhances the features of *TinyOS*, collaborating with industrial organizations.

TinyOS is built using *nesC*, a *C* dialect. To compile a first program, we must understand how *nesC* works.

1.2.1 nesC

Though *nesC* is a *C* dialect it differs completely in certain matters. A *nesC* implementation always consists of two parts. While the first one, a *module*,

contains the executable logic of the program, the second one specifies the connection of the module's components to *TinyOS* and is called a *configuration*.

In the *module* the interfaces of the application are defined and their implementation is given. The correspondence to functions in *C* are components in *nesC*. That is why instead of functions calling each other, components call other components, triggered by events.

The *configuration* wires the interfaces in the application to interfaces given by *TinyOS*. To understand it better, look at the example application *Blink*. It can be found in `/opt/tinyos-2.1.1/apps/Blink`. The two important files are `BlinkAppC.nc` and `BlinkC.nc`. You can additionally take a look at: `http://tinyos.stanford.edu/tinyos-wiki/index.php/The_simplest_TinyOS_program`
It should also be noted, that nesC follows an event based programming model.

Question 1.3 (3 Points)
Explain the relevant parts of the Blink application. What is happening in which order and why?

Further Information on nesC

- The nesC Language: A Holistic Approach to Networked Embedded Systems `http://nescc.sourceforge.net/papers/nesc-pldi-2003.pdf`
- The nesC v1.1 Language Reference `http://nescc.sourceforge.net/papers/nesc-ref.pdf`

1.3 Compiling and installing an application

First, copy the application folder to your home directory and move into the folder by running:

```
cp -a /opt/tinyos-2.1.2/apps/Blink $HOME/
```

followed by:

```
cd $HOME/Blink
```

Issue the command `make iris` to compile the application for the *iris* platform. The output you get will be similar to the following for the Blink application (do **not** type this in, the build system does it for you!):

```
mkdir -p build/iris
    compiling BlinkAppC to a iris binary
ncc -o build/iris/main.exe -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -target
     =iris -fnesc-cfile=build/iris/app.c -board=mts400 -DDEFINED_TOS_AM_GROUP=0x22 -
     -param max-inline-insns-single=100000 -DIDENT_APPNAME=\"BlinkAppC\" -
     DIDENT_USERNAME=\"sl-pc\" -DIDENT_HOSTNAME=\"SensorLabPC03\" -DIDENT_USERHASH=0
     x31cb7cbbL -DIDENT_TIMESTAMP=0x4dc14044L -DIDENT_UIDHASH=0x9222291eL -fnesc-
     dump=wiring -fnesc-dump='interfaces(!abstract())' -fnesc-dump='referenced(
     interfacedefs, components)' -fnesc-dumpfile=build/iris/wiring-check.xml
     BlinkAppC.nc -lm
    compiled BlinkAppC to build/iris/main.exe
           2268 bytes in ROM
             51 bytes in RAM
avr-objcopy --output-target=srec build/iris/main.exe build/iris/main.srec
avr-objcopy --output-target=ihex build/iris/main.exe build/iris/main.ihex
    writing TOS image
```

As the above output shows, the build finished successfully.
Having compiled the application it can now be flashed on one of the motes. Connect the USB programming board to the computer and attach the IRIS mote to the programming board. **Make sure the mote is switched off and no batteries are inside. If batteries are inside and the power is switched on, the hardware can become damaged when connecting the mote to the PC, so be careful!**
The next step is to plug the sensor mote in the programming board and find out, where the mote is connected to the computer. Issue the command `ls /dev/ttyUSB*`. It outputs you all connected USB devices. As you can see, the mote is probably connected to `/dev/ttyUSB0`. To install the program on the mote, enter the following into your terminal (do **not** enter "NodeID" as is, **adjust it** according to the text in the following!):

```
make iris reinstall.NodeID mib520,/dev/ttyUSB0
```

Here `NodeID` has to be replaced with the ID of the node. For this example, try 0, 1 or another small number (the address is a 16bit unsigned integer, with 0xffff being reserved as the broadcast address).
The resulting output should be similar to this (just given for reference):

```
tos-set-symbols build/iris/main.srec build/iris/main.srec.out-1 TOS_NODE_ID=1
     ActiveMessageAddressC__addr=1
Could not find data section in build/iris/main.exe, aborting.
Could not find symbol ActiveMessageAddressC__addr in build/iris/main.exe, ignoring
     symbol.
Could not find symbol TOS_NODE_ID in build/iris/main.exe, ignoring symbol.
   installing iris binary using mib510
avrdude -cmib510 -P/dev/ttyUSB0 -U hfuse:w:0xd9:m -pm1281 -U efuse:w:0xff:m -C/etc/
     avrdude/avrdude.conf -U flash:w:build/iris/main.srec.out-1:a

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e9704
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
```

```
        To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "0xd9"
avrdude: writing hfuse (1 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 1 bytes of hfuse written
avrdude: verifying hfuse memory against 0xd9:
avrdude: load data hfuse data from input file 0xd9:
avrdude: input file 0xd9 contains 1 bytes
avrdude: reading on-chip hfuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of hfuse verified
avrdude: reading input file "0xff"
avrdude: writing efuse (1 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 1 bytes of efuse written
avrdude: verifying efuse memory against 0xff:
avrdude: load data efuse data from input file 0xff:
avrdude: input file 0xff contains 1 bytes
avrdude: reading on-chip efuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of efuse verified
avrdude: reading input file "build/iris/main.srec.out-1"
avrdude: input file build/iris/main.srec.out-1 auto detected as Motorola S-Record
avrdude: writing flash (2268 bytes):

Writing | ################################################## | 100% 0.54s

avrdude: 2268 bytes of flash written
avrdude: verifying flash memory against build/iris/main.srec.out-1:
avrdude: load data flash data from input file build/iris/main.srec.out-1:
avrdude: input file build/iris/main.srec.out-1 auto detected as Motorola S-Record
avrdude: input file build/iris/main.srec.out-1 contains 2268 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.29s

avrdude: verifying ...
avrdude: 2268 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done. Thank you.

rm -f build/iris/main.exe.out-1 build/iris/main.srec.out-1
```

Done! You should now see that the LEDs of the sensor mote are blinking at different frequencies.

Hint: If you would like to rebuild your application and reflash it in a single step, you can do so by using `install` instead of `reinstall`.

1.4 Your first own application

Assignment 1 (35 Points)
Write your first own *TinyOS* application. What it should be able to do is:

- Boot the mote and enable the radio.
- Once the radio is enabled, switch `led3` on.
- After some time disable the radio.
- Once the radio is disabled, switch the LED off again and switch on `led1`.

Try writing the application from scratch.

Following this paragraph, you will find a list of interfaces that will be useful for your application you can read and refer back to as needed.

```
interface Boot{
  /* Signaled when OS booted */
  event void booted();
}

interface Leds{
  command void led0On();
  command void led0Off();
}

interface Timer<tag>{
  command void startOneShot(uint32_t period);
  command void startPeriodic(uint32_t period);
  /* Signaled when the timer is fired */
  event void fired();
}

interface SplitControl{
  command error_t start();
  /* Signaled when the radio has finished starting or an error occured */
  event void startDone(error_t ok);

  command error_t stop();
  /* Signaled when the radio has finished stopping or an error occured */
  event void stopDone(error_t ok);
}
```

Reminder 1: A `command` is an exported function that can be called using the `call` keyword. An `event` is signaled by a component and has to be implemented by the module using the component. All `command` calls are split-phase/asynchronous, that means that they return right away and do not block. Once the operations triggered by the `command` have finished, a corresponding `event` will be signaled.

Reminder 2: When using nesC, all variables have to be defined at the beginning of each new block (e.g beginning of a function)!