Practical Course on Wireless Sensor Networks

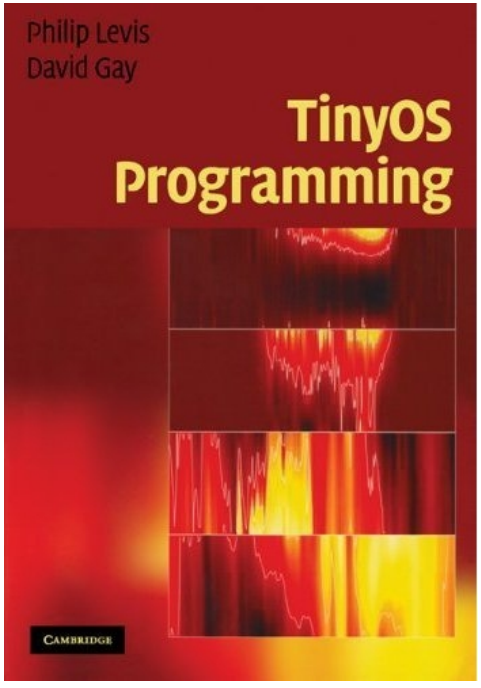
Dr. Henrik Brosenne
University of Goettingen
Institute of Computer Science

Winter 2012/2013

Table of Contents

References

TinyOS Homepage
`http://www.tinyos.net`



Philip Levis and David Gay,
TinyOS Programming,
Cambridge University Press,
March 2009.

References

For details on particular TinyOS subsystems consult the TinyOS Enhancement Proposals (TEPs), which detail the corresponding design considerations, interfaces, and components.

`http://docs.tinyos.net/tinywiki/index.php/TEPs`

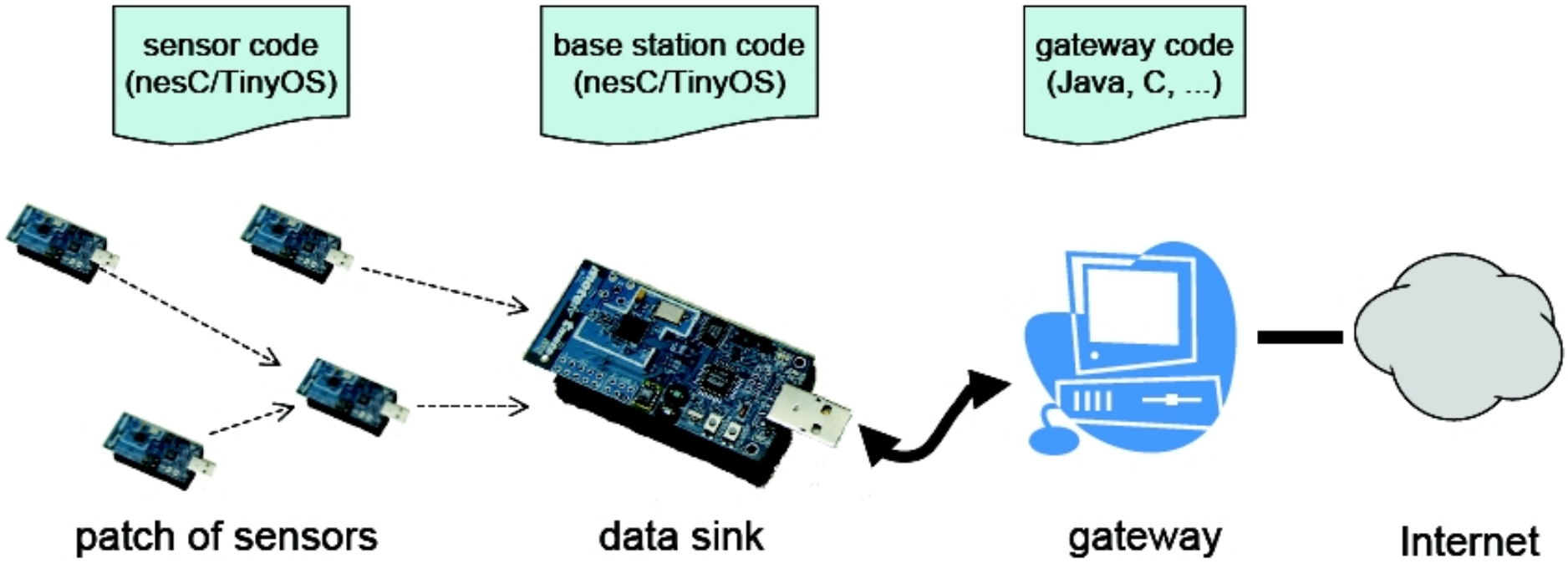
The TinyOS application programming interfaces (APIs) are online available, e.g. for TinyOS 2.1.0 and TelosB motes.

`http://www.tinyos.net/tinyos-2.1.0/doc/nesdoc/telosb/`

Table of Contents

Sensor Network Architecture

A typical sensor network architecture.



Patches of ultra-low power sensors, running nesC/TinyOS, communicate to gateway nodes through data sinks. These gateways connect to the larger Internet.

Since energy consumption determines sensor node lifetime, sensor nodes, commonly referred to as **motes**, tend to have very limited computational and communication resources.

Table of Contents

TinyOS

TinyOS is a lightweight operating system specifically designed for low-power wireless sensors, it is designed for the small, low-power microcontrollers motes have.

TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, TinyOS's structure makes it reasonably easy to port to new platforms. TinyOS applications and systems, as well as the OS itself, are written in the **nesC** language.

The C dialect nesC has features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions.

TinyOS

TinyOS provides things to make writing systems and applications easier.

- A component model, which defines how you write small, reusable pieces of code and compose them into larger abstractions.
- A set of important services and abstractions, such as sensing, communication, storage, and timers.
- A concurrent execution model, which defines how components interleave their computations as well as how interrupt and non-interrupt code interact.
- Application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

TinyOS

The component model is grounded in nesC.

It allows to write pieces of reusable code which explicitly declare their dependencies.

For example, a generic user button component that tells you when a button is pressed sits on top of an interrupt handler.

The component model allows the button implementation to be independent of which interrupt that is (e.g., so it can be used on many different hardware platforms) without requiring complex callbacks or magic function naming conventions.

TinyOS

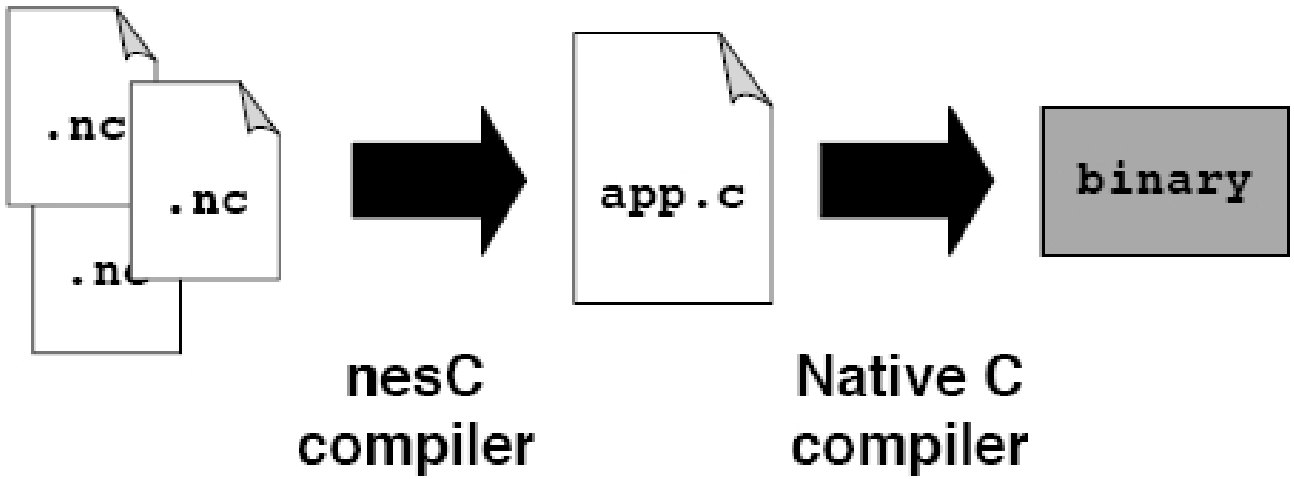
The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM.

Every I/O call in TinyOS is **split-phase** rather than block until completion, **a request returns immediately** and the **caller gets a callback** when the I/O completes.

Since the stack is not tied up waiting for I/O calls to complete, TinyOS only needs **one stack**, and has **no threads**.

TinyOS supports tasks, which are lightweight deferred procedure calls. Any component can post a task, which TinyOS will run at some later time.

The nesC compilation model



The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates **a mote binary**.

The mote binary consist of the whole operation system (TinyOS) for the mote type (z.B. iris) including all compoments and interfaces for the hardware (e.g. sensor, timer, etc.) as well as all applications.

Table of Contents

The simplest TinyOS program

There is some value in knowing the simplest code that can be compiled without errors.

The C equivalent to the example is nearly as follows.

```
int main () {
  return 0;
}
```

Step 1

Create a **components** file `SimpleAppC.nc`, preferable in a new directory `Simple`.

This **configuration** connect the involved components.

```
configuration SimpleAppC { }
implementation{
  components SimpleC, MainC;
  SimpleC.Boot -> MainC.Boot;
}
```

There are two components in this program, the new component SimpleC and the main component MainC.

MainC provides the **interface** Boot and thereby **signals** the `Boot.booted` **event** which essentially is the entry point of the application.

Step 2

Create the **component** file `SimpleC.nc`.

This **module** contains the implementation code of the component SimpleC.

```
module SimpleC {
  uses interface Boot;
}
implementation{
  event void Boot.booted() {
    //The entry point of the program
  }
}
```

Step 3

Now you need to create a `Makefile` so that the compiler can compile it.

```
COMPONENT=SimpleAppC
include $(MAKERULES)
```

Put the name of the top level configuration file in the `COMPONENT` field.

Start commpilation with `make`, which should work successfully because in the sensor lab the environment is set properly.

Compile for installing on an iris mote.

```
> make iris
```

Compile for the event simulator for TinyOS sensor networks (TOSSIM), the only platform TOSSIM supports are micaz motes, currently.

```
> make micaz sim
```

Step 4

Install the binary on a mote or run it with the simulator.

Table of Contents

Components

Components are the building blocks of nesC programs. A nesC program is a collection of components.

Every component is in its own source file, and there is a 1-to-1 mapping between component and source file names. For example, the file `LedsC.nc` contains the nesC code for the component LedsC, while the component PowerupC can be found in the file `PowerupC.nc`.

Components in nesC reside in a **global namespace**, there is only one PowerupC definition, and so the nesC compiler loads only one file named `PowerupC.nc`.

Component Signatures

Every component has a **signature**, which describes the functions it needs to call as well as the functions that others can call on it.

Modules and **configurations** are the both kinds of components, they differ in their **implementation** sections.

- Modules are components that implement and call functions.

  The implementation sections consist of nesC **code** that looks like C, the code declares variables and functions, calls functions, and compiles to assembly code.

- Configurations connect components into larger abstractions.

  The implementation sections consist of nesC **wiring code**, which connects components together.
  Configurations are the major difference between nesC and C (and other C derivatives).

A component declares its signature with **interfaces**, which are sets of functions for a complete service or abstraction.

Component Signatures

All components have two code blocks. The first block describes its signature, and the second block describes its implementation.

Module

```
module PowerupC {
  // signature
}
implementation {
  // implementation
}
```

Configuration

```
configuration LedsC {
  // signature
}
implementation {
  // implementation
}
```

Signature blocks in modules and configurations have the same syntax, but the syntax of the implementation sections is complete different.

Component Signatures

Interfaces define a set of related functions for a service or abstraction. How to write interfaces (interface files) is described later.

Example

There is a Leds interface for controlling node LEDs, a Boot interface for being notified when a node has booted, and an Init interface for initializing a component's state.

A component signature declares whether it **provides** or **uses** an interface. A single component can both provide and use interfaces.

Example

A component that needs to turn a node's LEDs on and off uses the Leds interface, while the component that implements the functions that turns them on and off provides the Leds interface.

Component Signatures

Example

Signatures of the module PowerupC and the configuration LedsC.

```
module PowerupC {
  uses interface Boot;
  uses interface Leds;
}
```

PowerupC is a module that turns on a node LED when the system boots.

It uses the Boot interface for notification of system boot and the Leds interface for turning on a LED.

```
configuration LedsC {
  provides interface Leds;
}
```

LedsC is a configuration which provides the abstraction of three LEDs that can be controlled through the Leds interface.

Component Signatures

Example

The signature for the configuration MainC.

```
configuration MainC {
  provides interface Boot;
  uses interface Init;
}
```

MainC is a configuration which implements the boot sequence of a node.

It provides the Boot interface so other components, such as PowerupC, can be notified when a node has fully booted. MainC uses the Init interface so it can initialize software as needed before finishing the boot sequence.

If PowerupC had state that needed initialization before the system boots, it might provide the Init interface.

Table of Contents

Interfaces

Interfaces describe a functional relationship between two or more different components.

The role a component plays in this relationship depends on whether it **provides** or **uses** the interface.

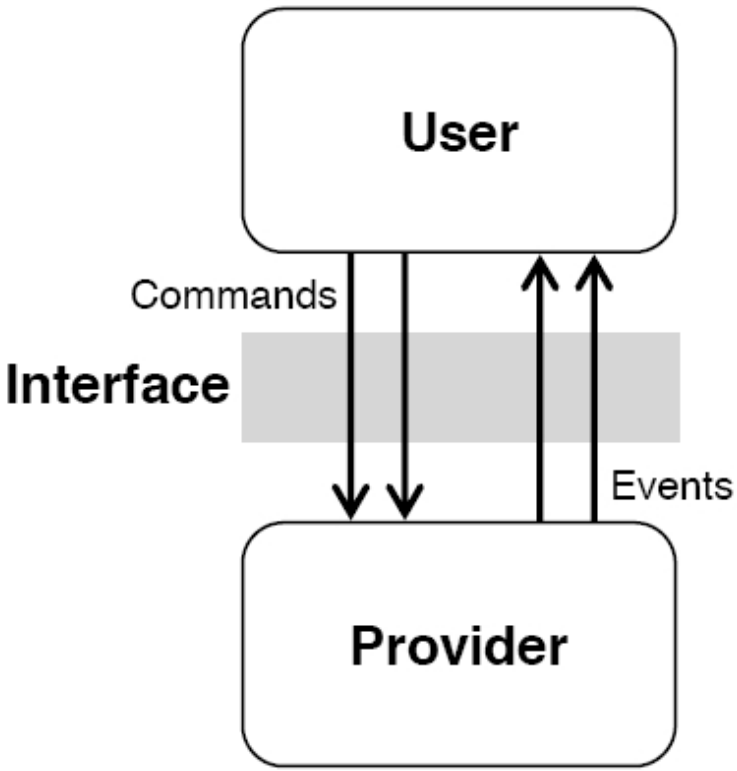
Like components, interfaces have a 1-to-1 mapping between names and files: the file `Leds.nc` contains the interface Leds while the file `Boot.nc` contains the interface Boot.

Just as with components, interfaces are in a global namespace.

Interfaces

Whether a function is a command or event determines which side of an interface, a user or a provider, implements the function and which side can call it.

Users can call commands and providers can signal events. Conversely, users must implement events and providers must implement commands.



Interfaces

Syntactically interfaces are quite different from components. They have a single block, the interface declaration.

An interface declaration has one or more functions in it. Interfaces have two kinds of functions **commands** and **events**.

Example

Init and Boot are two simple interfaces, each of which has a single function. Init has a single command, while Boot has a single event.

```
interface Init {
  command error_t init();
}
```

```
interface Boot {
  event void booted();
}
```

Interfaces

Example

The Leds interface has three LEDs, mostly for historical reasons. They are named led0, led1, and led2.

The Leds interface allows to turn LEDs on, to turn them off and toggle them.

```
interface Leds {
  command void led0On();
  command void led0Off();
  command void led0Toggle();

  command void led1On();
  command void led1Off();
  command void led1Toggle();

  command void led2On();
  command void led2Off();
  command void led3Toggle();
  // ...
}
```

Table of Contents

Module implementations

Module implementations are written in nesC.

At the level of single statements, nesC is almost identical to C. It differs mainly in how code calls functions and variable scoping.

As a component must implement **commands** for interfaces it provides and **events** for interfaces it uses, a module must define these functions.

Module implementations

Example

```
module PowerupC {
  uses interface Boot;
  uses interface Leds;
}
implementation {
  event void Boot.booted() {
    call Leds.led0On();
  }
}
```

As PowerupC uses the Boot interface, and Boot has a single event, booted, PowerupC must provide an implementation.

```
event void Boot.booted() { ...}.
```

When the provider of Boot signals this event, PowerupC turns on LED 0 of the Leds interface. Modules invoke commands with the **call** keyword and invoke events with the **signal** keyword.

Because of the call and signal keywords the terms **call commands** and **signal events** are used often.

Module implementations

Example

```
module PowerupToggleC {
  uses interface Boot;
  uses interface Leds;
}
implementation {
  event void Boot.booted() {
    while (1) {
      call Leds.led0Toggle();
      call Leds.led1Toggle();
      call Leds.led2Toggle();
    }
  }
}
```

The module PowerupToggleC will cause system boot to enter an infinite loop that toggles all three LEDs.

Configuration implementations

For PowerupC and PowerupToggleC to do anything, we must specify which components providing Boot and Leds we want to connect these modules to. This is the role of the configurations.

Modules allocate state and implement executable logic. However, like all components, they can only name functions and variables within their local namespaces, i.e., those defined in their signature and implementation.

For one module to be able to call another, we have to map a set of names (generally interfaces) in one component's signature to a set of names in another component signature.

In nesC, connecting two components in this way is called **wiring**. Configurations contain these wirings in the implementation section.

Configuration implementations

Example

The PowerupAppC and PowerupToggleAppC configuration, connects the module to the components that provide the Boot and Leds interfaces.

```
configuration PowerupAppC {}
implementation {
  components MainC, LedsC, PowerupC;
  PowerupC.Boot -> MainC.Boot;
  PowerupC.Leds -> LedsC.Leds;
}
```

```
configuration PowerupToggleAppC {}
implementation {
  components MainC, LedsC, PowerupToggleC;
  PowerupToggleC.Boot -> MainC.Boot;
  PowerupToggleC.Leds -> LedsC.Leds;
}
```

Now the examples are ready to be compiled, installed and tested.

Table of Contents

Naming Conventions

The names of components end in the letters C or P. Components use these two letters in order to clearly distinguish them from interfaces.

- The C stands for *component*.
  Components whose names end in C are abstractions that other components can use freely, (an externally usable abstraction).
  Once the signature for an externally usable component is written , changing it is very hard. Any number of other components might depend on it, and changing it will cause compilation errors.
- The P stands for *private*.
  Components whose names end in P should not be used directly, as they are generally an internal part of a complex system (an internal implementation).
  Because an internal implementation is only wired to by higher-level configurations within that software abstraction, their signatures are much more flexible.

Naming Conventions

Example

Changing the signature of LedsC (the basic TinyOS LEDs abstraction) would break almost all TinyOS code.

An internal change to LedsP (the implementation of the standard 3 LED mote abstraction) and changing its wiring in LedsC should not be apparent to the user.

Coding Conventions

TinyOS follows a couple of coding conventions, following two important.

Split-phase commands must never directly signal their callback. In a split-phase operation the request that initiates an operation completes immediately. Actual completion of the operation is signaled by a separate callback.

The relation between most TinyOS components is hierarchical. Application components use interfaces provided by system services, which themselves use interfaces provided by lower-level services, and so on down to the raw hardware.

Outlook

Example

This example is the TinyOS solution to toggle the leds periodical using the **generic interface** Timer and the configuration TimerMilliC (the virtualized millisecond timer abstraction) belonging to `Timer<TMilli>`.

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation {
  // ...
}
```

Outlook

Example

```
module BlinkC {
  // ...
}
implementation {
  event void Boot.booted() {
    call Timer0.startPeriodic(250);
    call Timer1.startPeriodic(500);
    call Timer2.startPeriodic(1000);
  }
  event void Timer0.fired() {
    call Leds.led0Toggle();
  }
  event void Timer1.fired() {
    call Leds.led1Toggle();
  }
  event void Timer2.fired() {
    call Leds.led2Toggle();
  }
}
```

Outlook

Example

```
configuration BlinkAppC { }
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC.Boot -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;

  BlinkC.Leds -> LedsC.Leds;
}
```

End

Questions?