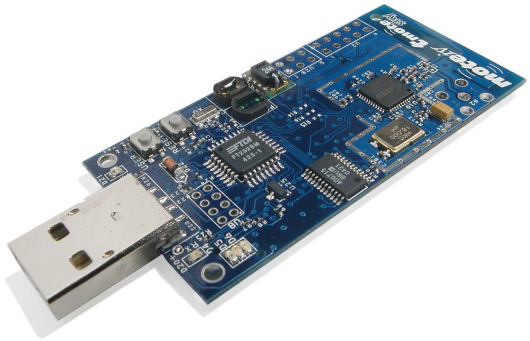
WSN Programming

Introduction

Olaf Landsiedel

Programming WSNs

- What do we need to write software for WSNs?
  (or: for any system, like your laptop, cell phone?)
  - Programming language
    - With compiler, etc.
  - OS / runtime libraries
    - Access to system resources
    - APIs: Communication, sensors, etc.



- Put Windows 7 on a sensor node?
  - Or Linux, Android, …
  - Use Java, Python, … as programming languages?
  - WSNs are embedded systems
    - Resource constraints

CPU	16bit, 8 MHz
RAM	10kB
ROM	48kB
Flash	1MB
Batteries	2xAA

Today's Topics

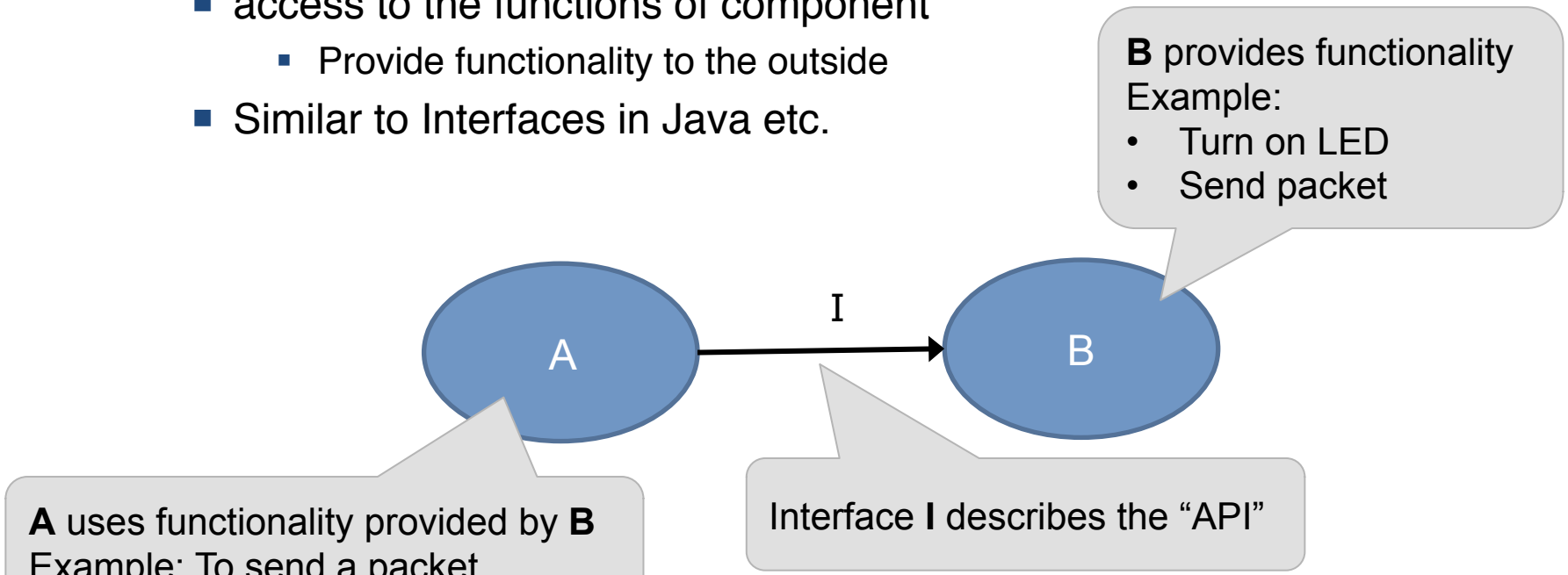
- NesC
  - Programming Language for WSNs
- TinyOS
  - WSN OS / runtime
  - Note: many other OS exist (Contiki, Mantis, LiteOS, Nano-RK, …)
    - TinyOS: widespread, especially in academia
- Examples
- Hands-on
  - Your turn ☺

TinyOS and NesC

Component Model

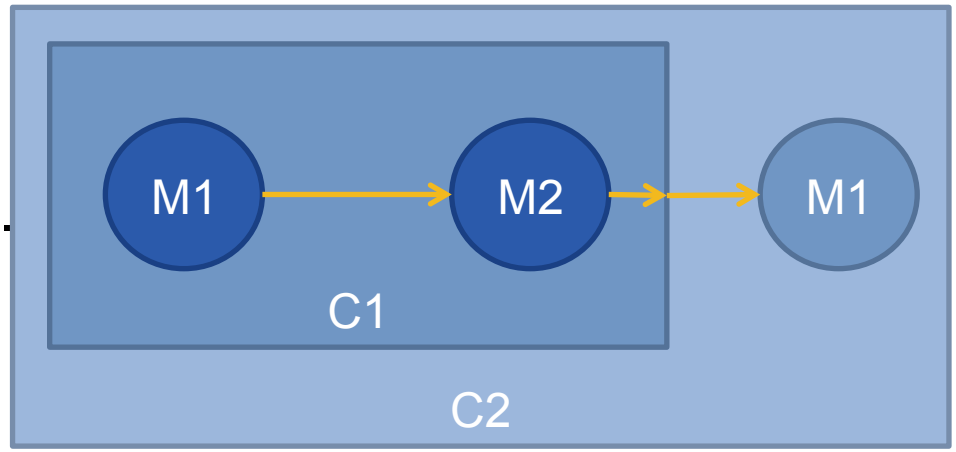
TinyOS Components

- TinyOS and its applications are written in nesC
  - C dialect with extra features: mainly Components
- Basic unit of nesC code is a Component
- **Components** connect via **Interfaces**
  - Connections called “wiring”
  - Interfaces:
    - access to the functions of component
      - Provide functionality to the outside
    - Similar to Interfaces in Java etc.



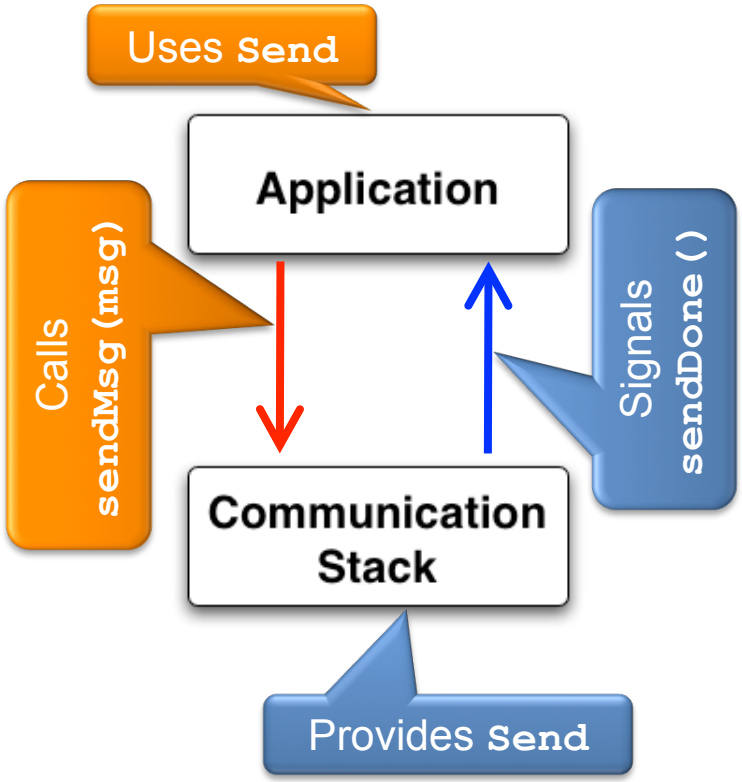
TinyOS Components

- Two types of components
  - Modules (M) :
    - Base type
    - Contains functionality / implementations: “The real code”
  - Configurations (C):
    - Contain multiple sub-components
      - Modules or Configurations: Nesting
    - Describe connections
      - Which module uses the function
    - Parameterize sub-components:
      - Options, etc.
  - Very similar to classes in Java etc.
    - But static
    - No instantiation at runtime (new “new()”)



TinyOS Components

- **Components**
  - *use* or *provide* interfaces
  - *Use*: component uses func. of others
  - *Provide*: provide func. to others
- **Interfaces** contain
  - The API
  - Contain *commands* and *events*



Function calls	**Commands**	**Events**
Use	Call Command ↓	Implement Event Handler ↑
Provide	Implement Command Body ↓	Signal Event ↑

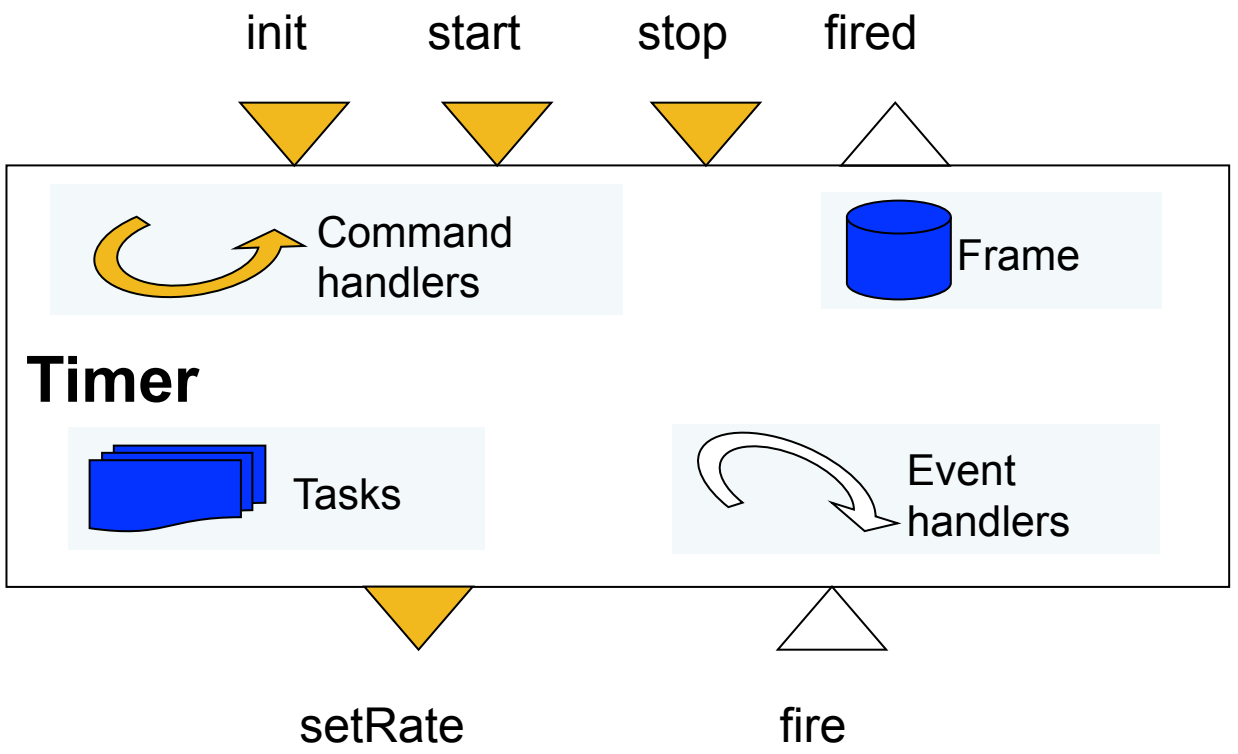
TinyOS Modules

- Specification
  - List of interfaces the component
    - Provides
      - Functionality it provides to others
    - Uses
      - Functionality of others it uses
  - Alias interfaces as new name
- Implementation
  - Commands of provided interfaces
  - Events of used interfaces

```
module FooM {
  // Specification
  provides {
    interface Foo;
  }
  uses {
    interface Poo as PooFoo;
    interface Boo;
  }
}
//Implementation
implementation {
  // Command handlers
  command result_t Foo.comm{
      …
  }

  //Event handlers
  event void Boo.event{
      …
  }
}
```

TinyOS Modules



TinyOS and NesC

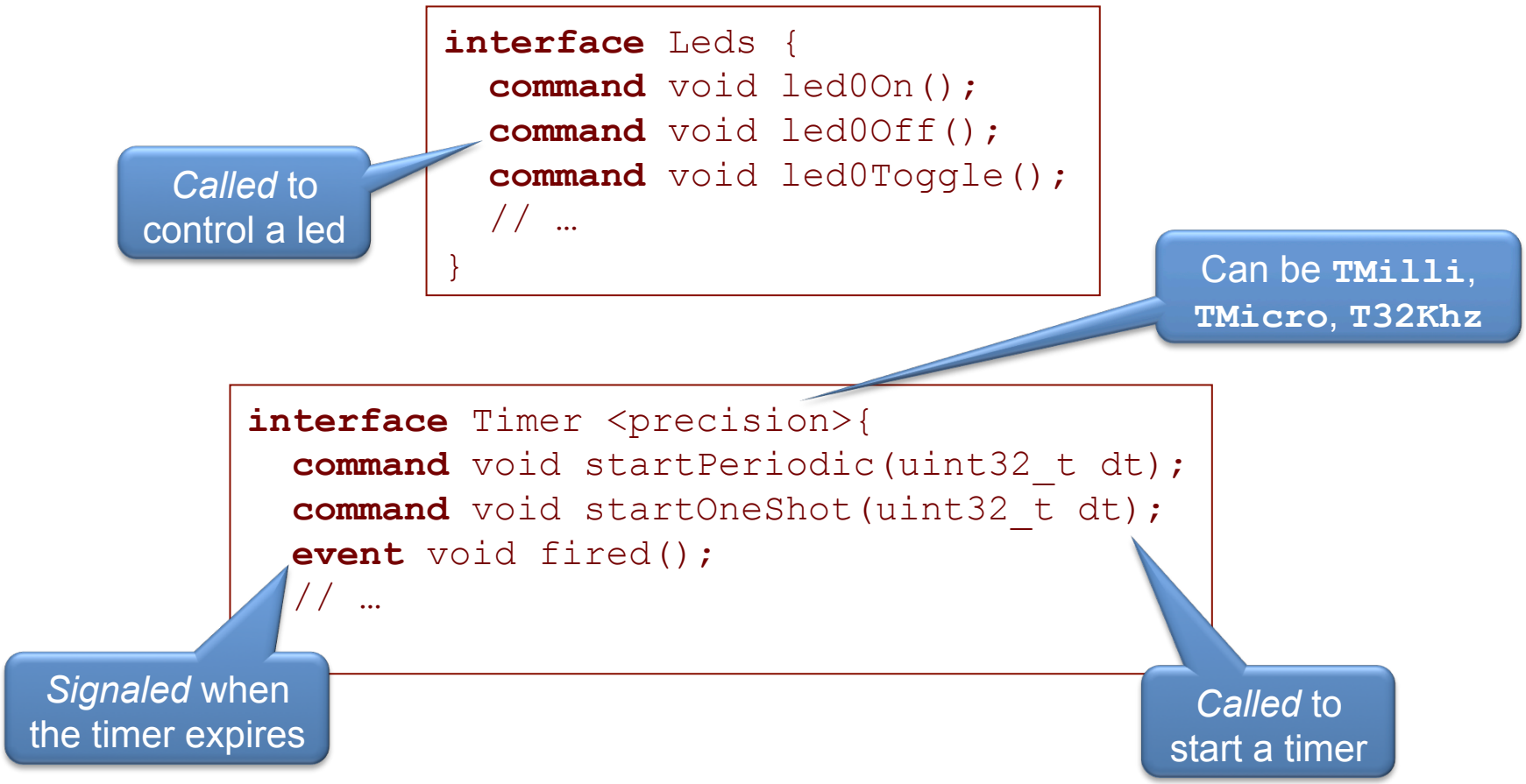
Example: Hello World

“Hello World”

- No screen, we cannot print “Hello World”…
- …so, let them periodically blink the LEDs!

- Ingredients:
  - a timer to trigger the periodic blinking
    - TOS provides a **`Timer`** library
  - a way to turn the LEDs on and off
    - TOS provides a **`LedsC`** component to control the LEDs

“Hello World”


interface Leds {
command void led0On();
command void led0Off();
command void led0Toggle();
// …
}
Called to control a led
interface Timer <precision>{
command void startPeriodic(uint32_t dt);
command void startOneShot(uint32_t dt);
event void fired();
// …
}
Can be TMilli, TMicro, T32Khz
Signaled when the timer expires
Called to start a timer

“Hello World”

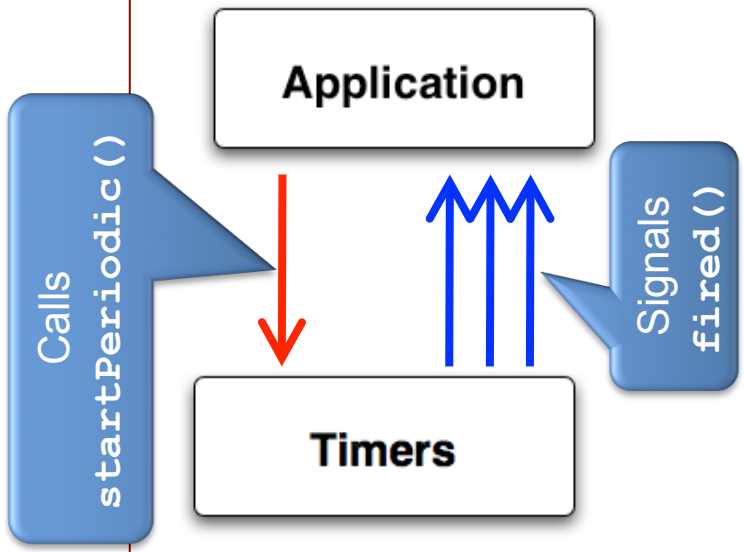
Timer with millisecond precision

```
module BlinkC {
  uses interface Timer<TMilli> as BlinkTimer;
  uses interface Leds;
  uses interface Boot;
}

implementation {
  event void Boot.booted() {
    call BlinkTimer.startPeriodic(1000);
  }

  event void BlinkTimer.fired() {
    call Leds.led0Toggle();
  }
}
```

Signaled when hw ready



“Hello World”

```
configuration BlinkAppC {}

implementation {
  components MainC,
             BlinkC,
             LedsC;
  components new TimerMilliC() as Timer0;

  BlinkC -> MainC.Boot;
  BlinkC.BlinkTimer -> Timer0;
  BlinkC.Leds -> LedsC;
}
```

No interfaces used or provided

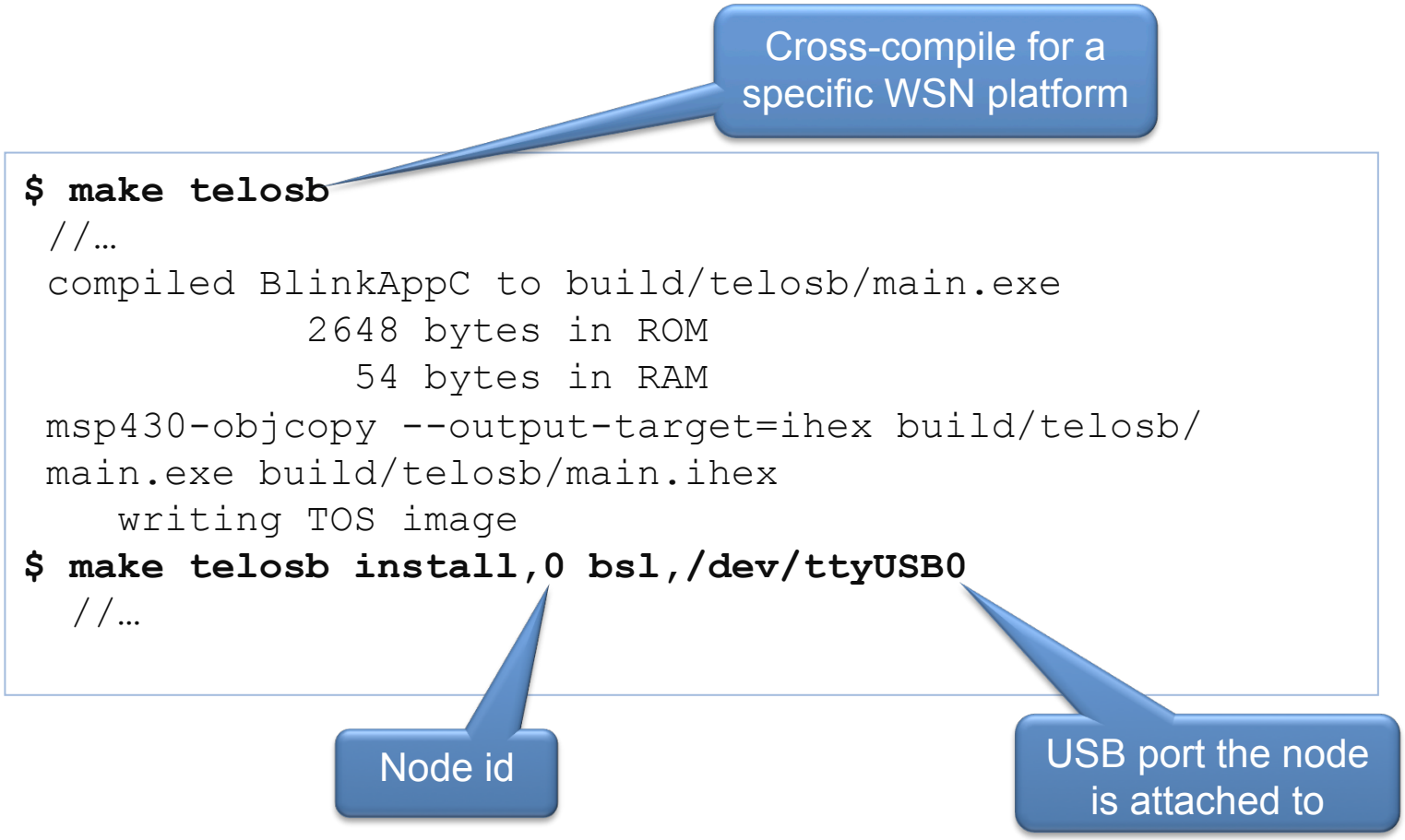
Provides **Boot**

TimerMilliC is a *generic* component: can be instantiated multiple times!

Wirings

- Instantiation is done at compile-time
  - very different from OO programming

“Hello World”



- Use “motelist” command to determine USB port

Programming (cont.)

TinyOS and NesC

Asynchronous Programming Model

```
event void Boot.booted() {
  call BlinkTimer.startPeriodic(1000);
}

event void BlinkTimer.fired() {
  call Leds.led0Toggle();
}
```

```
void main(){
  while(){
    sleep(1);
    printf('hello');
  }
}
```

- TinyOS:
  - Asynchronous programing (split phase)
  - No blocking calls
    - Not synchronous
  - Result:
    - Memory efficient concurrency
      - If execution is neither in booted nor in fired, it will “return” to the OS
      - No context switch required
    - Users must avoid endless loops etc.

Tasks

- TinyOS has a single stack:
  - Long-running computation can reduce responsiveness
- Tasks: mechanism to defer computation
  - Tells TinyOS “do this later”
- Tasks run until completion
  - Tasks do not interrupt each other
  - TinyOS scheduler runs them one by one in the order of their post
  - Keep them short!
- Interrupts run on stack, can post tasks
  - Interrupts can interrupt tasks
    - Interrupts: low level hardware events
  - Interrupts can interrupt each other
    - See interrupt priority

```
//…

task void compute(){
  //do something;
}

event void Timer.fired()
{
  post compute()
}

//…
```

TOSSIM

- TinyOS mote SIMulator
  - Allows nesC code to run on a standard machine
    - Linux etc.
  - Includes models for
    - Communication
    - Sensing
    - …
  - Compiles directly from TinyOS source
    - Run the same code in simulator and sensor node
    - Good for testing before deployment
- See TinyOS website for details

Code Examples

- TinyOS has an “apps” folder
  - Many good examples

- Other sources of information
  - TinyOS Book
    - TinyOS Programming by Philip Levis, David Gay
    - In part: csl.stanford.edu/~pal/pubs/tinyos-programming.pdf
  - TinyOS web book
    - csl.stanford.edu/~pal/pubs/tos-programming-web.pdf
  - TinyOS tutorial:
    - IPSN 2009 (enl.usc.edu/talks/cache/tinyos-ipsn2009.ppt)
  - TinyOS website: http://www.tinyos.net/
  - TinyOS mailing list (and its archives)
    - On the TinyOS website

Programming (cont.)

Hands-On

Your Tasks

- 1. System test
  - Install Blink on your sensor node
    - See apps/Blink
    - Compile, install (see prev. slides, see tutorial from TA)

- 2. Main task:
  - Program Anti-Theft App for your sensor node

Your Main Task

- Goal: write an anti-theft device.
  - Detect when someone steels your sensor node
- Two parts:
  - Detecting theft.
    - Assume: thieves put the motes in their pockets.
    - So, a “dark” mote is a stolen mote.
    - Every N ms check if light sensor is below some threshold
  - Reporting theft.
    - Assume: bright flashing lights deter thieves.
    - Theft reporting algorithm: light the **red LED** for a little while!
- What you will use
  - Basic components, interfaces, wiring
  - Essential system interfaces for startup, timing, sensor sampling
- Start from Blink
  - In apps/Blink
  - See apps/RadioSensoToLeds on sensor use
    - Replace “DemoSensorC” with “HamamatsuS10871ParC” or “HamamatsuS10871TsrC”

The Basics – Let's Get Started

```
module AntiTheftC {
  uses interface Boot;
  uses interface Timer<TMilli> as Check;
  uses interface Read<uint16_t>;
}
implementation {
  event void Boot.booted() {
    call Check.startPeriodic(1000);
  }
  event void Check.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t ok, u
    if (ok == SUCCESS && val < 200)
      theftLed();
  }
}
```

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

```
interface Timer<tag> {
  command void startOneShot(uint32_t per
  command void startPeriodic(uint32_t peri
  event void fired();
}
```

Components start with a *signature* specifying
- the interfaces *provided* by the component
- the interfaces *used* by the component

A module is a component implemented in C
- with functions implementing commands and events
- and extensions to call commands, events

The Basics – Split-Phase Ops

```
module AntiTheftC {
  uses interface Boot;
  uses interface Timer<TMilli> as Check;
  uses interface Read<uint16_t>;
}
implementation {
  event void Boot.booted() {
    call Check.startPeriodic(1000);
  }
  event void Check.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t ok, uint16_t val) {
    if (ok == SUCCESS && val < 200)
      theftLed();
  }
}
```

In TinyOS, all long-running operations are split-phase:

- A command starts the op: read
- An event signals op completion: readDone

```
interface Read<val_t> {
  command error_t read();
  event void readDone(error_t ok, val_t val);
}
```

The Basics – Split-Phase Ops

```
module AntiTheftC {
  uses interface Boot;
  uses interface Timer<TMilli> as Check;
  uses interface Read<uint16_t>;
}
implementation {
  event void Boot.booted() {
    call Check.startPeriodic(1000);
  }
  event void Check.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t ok, uint16_t val) {
    if (ok == SUCCESS && val < 50)
      theftLed();
  }
}
```

In TinyOS, all long-running operations are split-phase:

- A command starts the op: read
- An event signals op completion: readDone

Errors are signalled using the error_t type, typically

- Commands only allow one outstanding request
- Events report any problems occurring in the op

```
interface Read<val_t> {
  command error_t read();
  event void readDone(error_t ok, val_t val);
}
```

The Basics – Configurations

```
generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
implementation { ... }
```

```
configuration AntiTheftAppC
implementation
{
  components AntiTheftC, MainC, LedsC;

  AntiTheftC.Boot -> MainC.Boot;
  AntiTheftC.Leds -> LedsC;

  components new TimerMilliC() as MyTimer;
  AntiTheftC.Check -> MyTimer;

  components  new HamamatsuS10871ParC();
  AntiTheftC.Read -> HamamatsuS10871ParC;
}
```

```
generic configuration PhotoC() {
  provides interface Read;
}
implementation { ... }
```

A configuration is a component built out of other components.
It *wires* “used” to “provided” interfaces.
It can instantiate *generic* components
It can itself provide and use interfaces

Thanks! Questions?

- Based on slides from
  - Phil Levis, David Gay, David Culler, Luca Mottola, Hamad Alizai, and many others