Politecnico di Milano

Advanced Network Technologies Laboratory

Internet of Things

Hands on activities

https://politecnicomilano.webex.com/meet/edoardo.longo

TinyOS

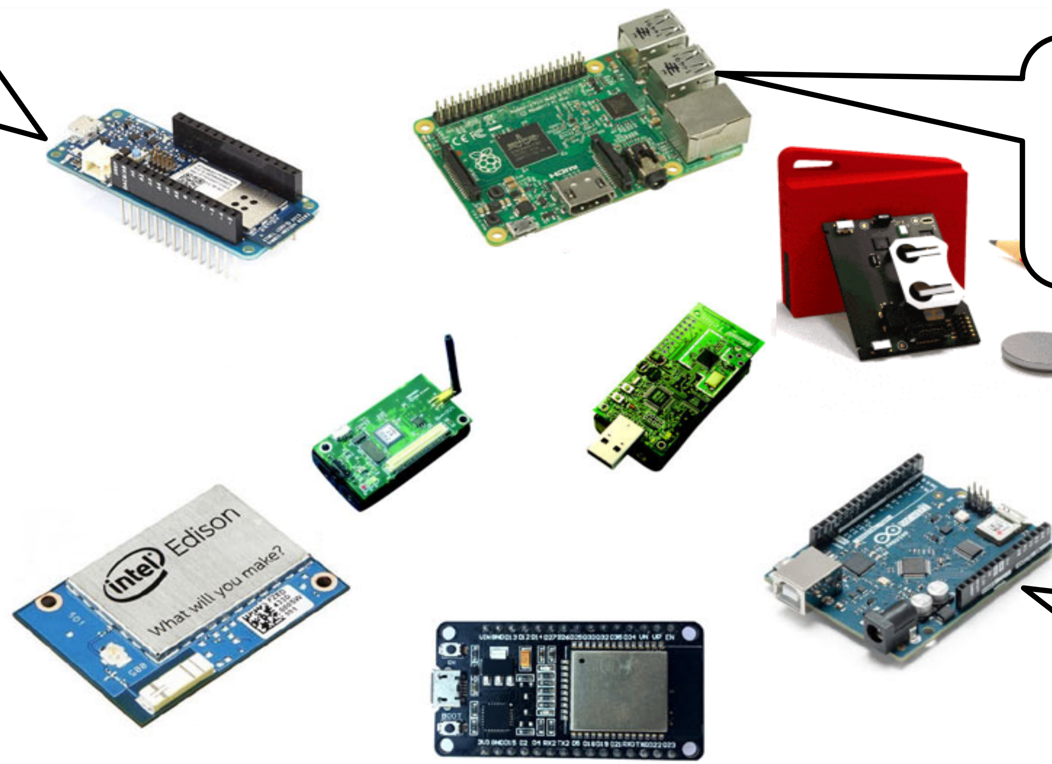
"TinyOS is an *open-source operating system* designed for *wireless embedded sensor networks*"

http://www.tinyos.net/

Hardware

Hardware

MKR-1000
Flash Memory:
256KB
SRAM: 32KB
Clock Speed: 48MHz
Wi-Fi



RASPBERRY PI
Flash Memory: microSD
RAM: 1GB
Clock Speed: 1.2GHz
(Wi-Fi / Bluetooth) / Eth

ARDUINO UNO
Flash memory: 32KB
SRAM: 2KB
Clock Speed: 16MHz
Ethernet

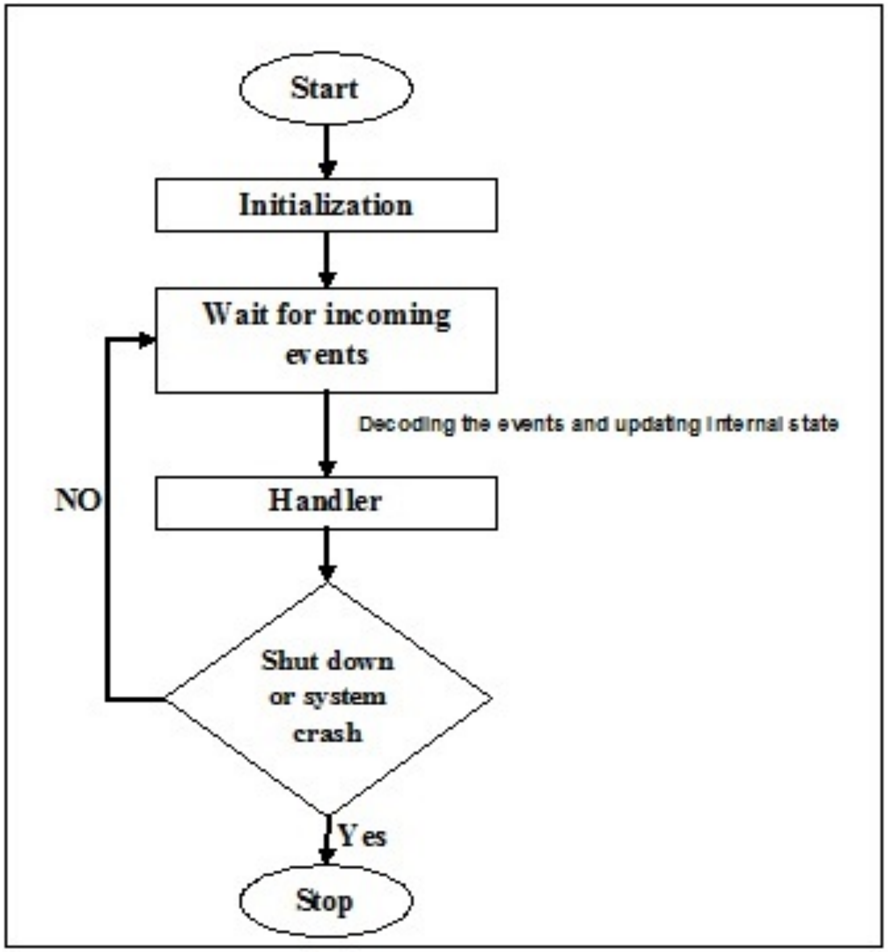
Hardware

TELOSB
Flash Memory: 48KB
SRAM: 10KB
Clock Speed: 8 MHz
802.15.4

MICAZ
Flash Memory:
128KB
SRAM: 4KB
Clock Speed: 7 MHz
802.15.4

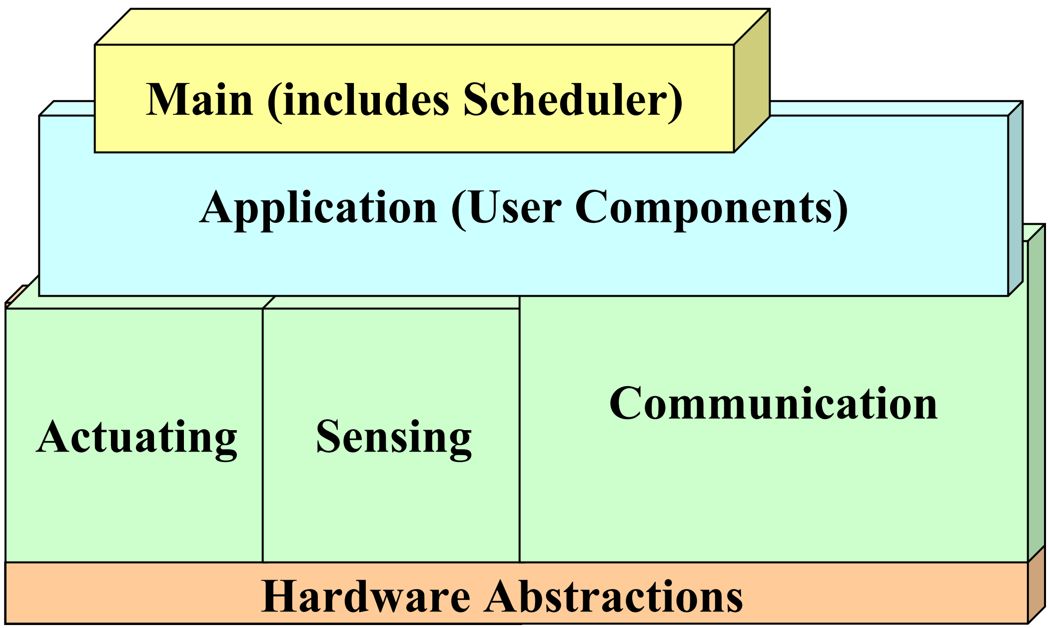
TinyOS Overview

- Event-driven architecture
  - OS operations are triggered by hardware interrupt (asynchronous management)



TinyOS Overview

- Event-driven architecture
  - OS operations are triggered by hardware interrupt (asynchronous management)
- Single shared stack -> no dynamic allocation
- No kernel/user space differentiation



TinyOS “Ingredients”

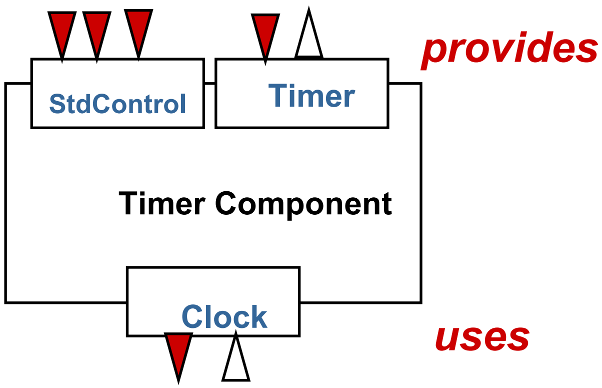
- TinyOS is not an OS in traditional sense
- Provides a programming framework to build application-specific OS instances
- Programming Framework made of:
  - Scheduler (always there)
  - Components
  - Interfaces

TinyOS Concepts

- Interfaces are bi-directional
  include *commands* and *events*

<u>Timer.nc</u>

```
interface Timer {
.          command void startPeriodic(uint32_t dt)
           command void startOneShotAt(uint32_t t0, uint32_t dt)
           command void stop()
           ….
           event void fired();
}
```



- Scheduler: *tasks + events*
- A component
  - Module
  - Configuration: wire module to other components

Event implementation

- Event is independent of FIFO scheduler
- Lowest level events are supported directly by Hardware interrupt
- Software events propagate from lower level to upper level through function call

TASKS

- Provide concurrency internal to a component
  - longer running operations
  - background processing
- Are interrupted by events
- May call commands
- May signal events
- Not preempted by tasks

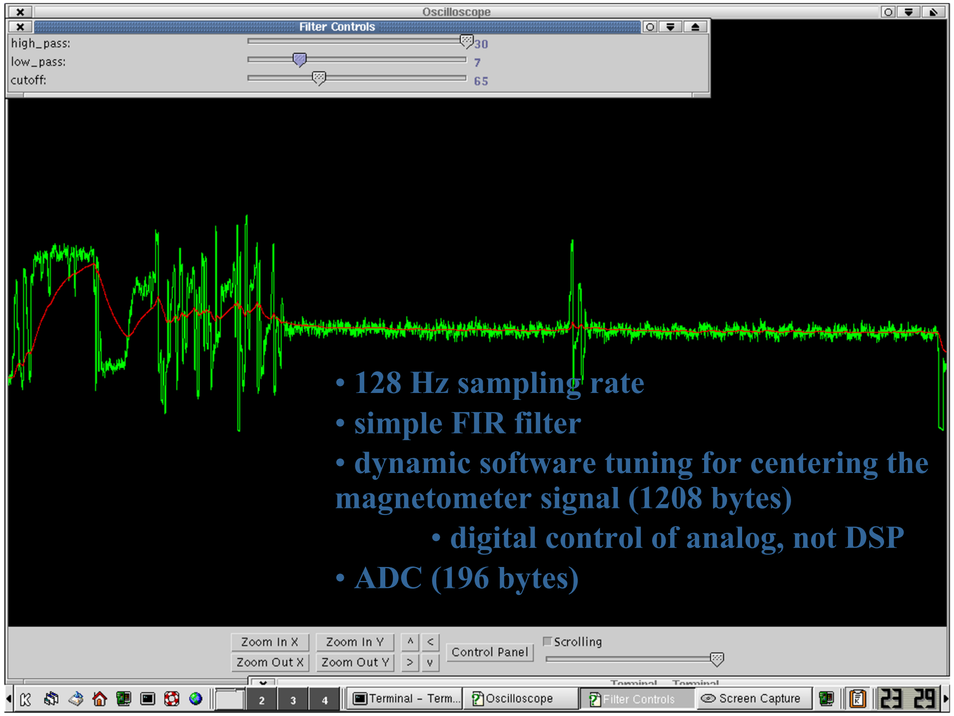
```
{
...
post TskName();
...
}
```

```
task void TskName()
{
...
}
```

Typical application use of tasks

- Event driven data acquisition
- Schedule task to do computational portion

```
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
}
task void processData() {
    int16_t i, sum=0;
    for (i=0; i < maxdata; i++)
        sum += (rdata[i] >> 7);
    display(sum >> shiftdata);
}
```



Tasks - Examples

- Transmit packet
  - Send command schedules task to calculate CRC
  - Task initiated byte-level data pump
  - Events keep the pump flowing
- Receive packet
  - Receive event schedules task to check CRC
  - Task signals packet ready if OK
- Byte-level TX/RX
  - Task scheduled to encode/decode each complete byte
  - Must take less time that byte data transfer

Politecnico di Milano

Advanced Network Technologies Laboratory

Internet of Things

TinyOS Programming and Cooja

TinyOS

Agenda

- Playing with TinyOS
  - Programming and components
  - Blink Application
  - Print on Cooja
  - Cooja and Node-RED

- Using the Radio
  - RadioCountToLeds Application

Programming TinyOS

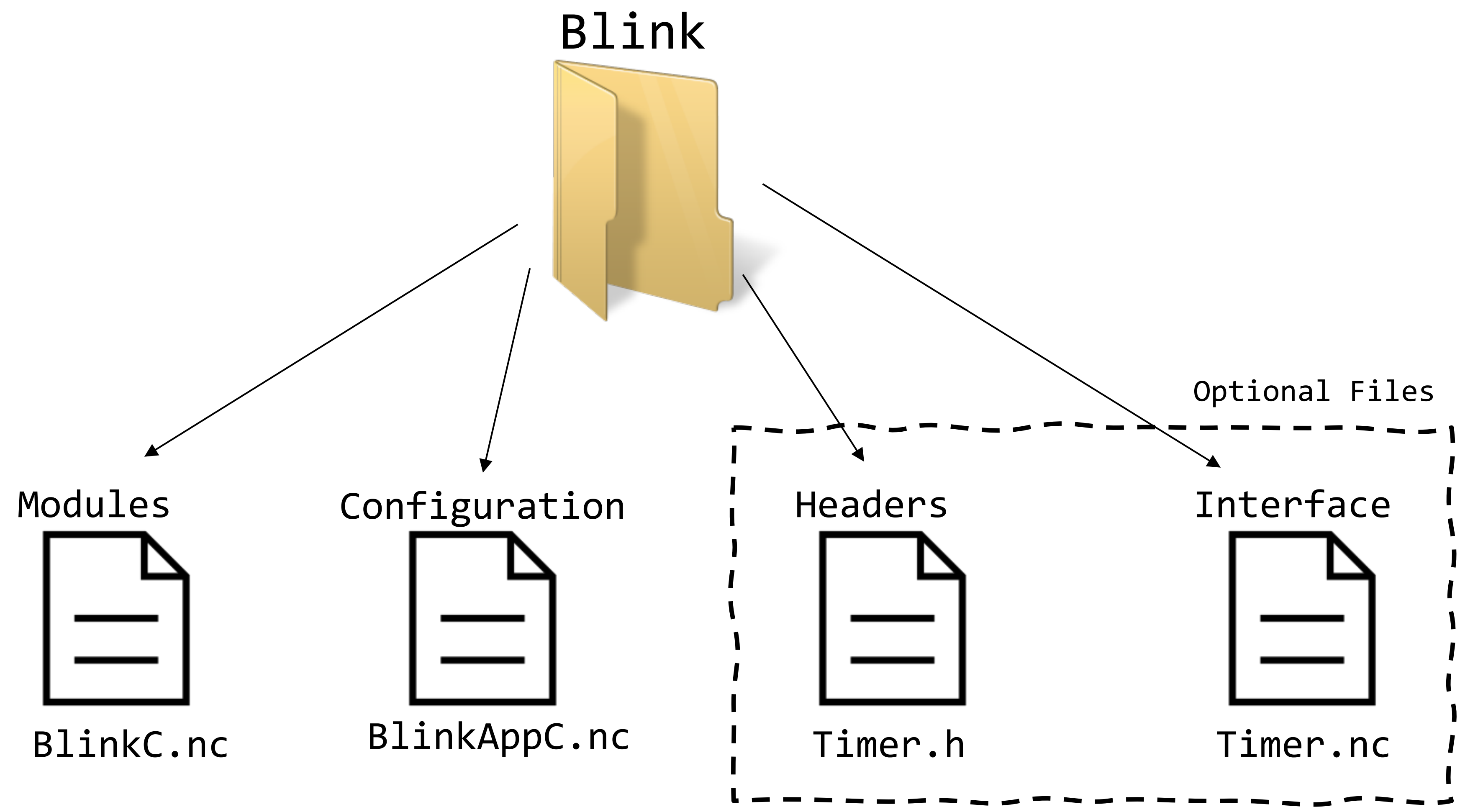
- TinyOS is written in a C "dialect" *nesC*

http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf

- Provides syntax for TinyOS concurrency and storage model
  - commands, events, tasks
  - local frame variable
  - static memory allocation
  - no function pointers

- Applications:
  - just additional components composed with the OS components

Folder structure



Components

- Modules (*BlinkC.nc* files)
  - provide code that implements one or more interfaces and internal behavior
- Configuration (*BlinkAppC.nc* files)
  - link together components to yield new component
- Headers (*Timer.h* files)
  - Define parameters
- *Interface* (*Timer.nc* files)
  - logically related set of commands and events

StdControl.nc

```
interface StdControl {
command result_t init();
command result_t start();
command result_t stop();
}
```

Timer.nc

```
interface Timer {
 command result_t start(char type,uint32_t
interval);
 command result_t stop();
 event result_t fired();}
}
```

Applications in TinyOS

- Configurations *(BlinkAppC.nc)*:
  - Used to configure applications
  - Used to wire components through interfaces

- Modules *(BlinkC.nc)*:
  - Used to implement components, call commands, events, and tasks.

Example: Blink

- Operation: the application keeps three timers at 1 [Hz], 2 [Hz] and 4 [Hz], upon timer expiration a LED is toggled.

- Application Files:
  - BlinkAppC.nc, configuration
  - BlinkC.nc, module

Blink Application - Demo

- Test applications are in tiny-os/apps
- Compile the code for real-motes platforms
  - Open the terminal and move to the code folder (cd tiny-os/apps/Blink
  - Compiling commands:
    - **`make micaz`**

      or
    - **`make telosb`**
  - **`Telosb and Micaz are two different type of motes`**
    - Look at the *makefile* for more information

BlinkC.nc

```
#include "Timer.h"

module BlinkC {
    uses interface Leds;
    uses interface Boot;
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Timer<TMilli> as Timer2;
}
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(); }
}
```

Used Interfaces

Timer Initialization

Events of timer expiry

Note:
- NO interfaces provided to other components
- NO commands defined
- NO tasks needed

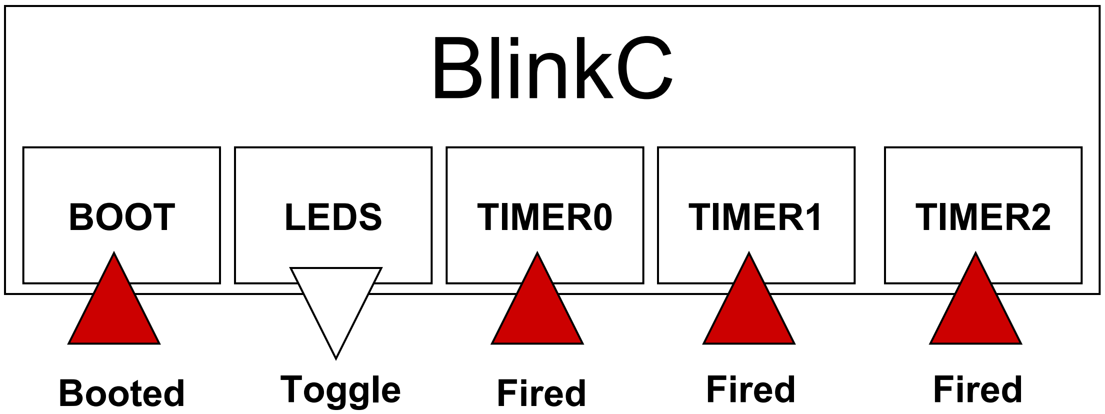
BlinkAppC.nc

```
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;
}
```

List of components implementing Blink applications

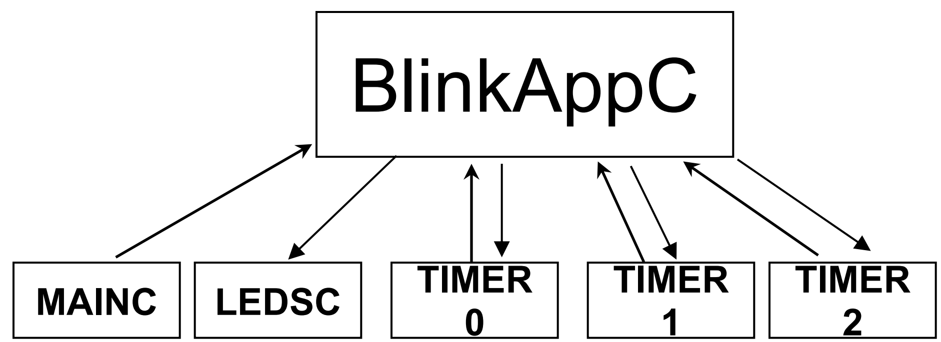
Components Wiring

Blink – Interfaces, Events and Commands



BlinkC interfaces:

- Fired and Booted events
- Toggle command



BlinkAppC components:

- TimerN.Timer and MainC.Boot
- LedsC.Toggle

Documentation

- The best way to understand which module must be used is check the documentation

- Documentation path in the VM:

`tiny-os/main/doc/nes/telosb/index.html`

Politecnico di Milano

Advanced Network Technologies Laboratory

WSN Simulation

Using Cooja

WSN simulation: why?

- WSN require large scale deployment
- Located in inaccessible places
- Apps are deployed only once during network lifetime
- Little room to re-deploy on errors

TinyOS Blink on Cooja

- Compile blink for Telosb
  - make telosb
- Open Cooja and create a new simulation
- Create a new Sky mote
- Select the main.exe file as firmware (located in the Blink build/telosb directory) and create the mote
- Watch the leds blink!

Politecnico di Milano

Advanced **N**etwork **T**echnologies **Lab**oratory

Using the Radio

Creating/Sending/Receiving/Manipulating Messages

General Idea

SENDER

Fill Message → Specify Destination → Pass Message to OS → Wait For FeedBack

RECEIVER

OS Stores Message → OS signal Msg Reception → Pass Message to Applications → Read Data

Message Buffer Abstraction

- In tos/types/messages.h

```
typedef nx_struct message_t {
   nx_uint8_t header[sizeof(message_header_t)];
   nx_uint8_t data[TOSH_DATA_LENGTH];
   nx_uint8_t footer[sizeof(message_header_t)];
   nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

- Header, footer, metadata: already *implemented* by the specific link layer
- Data: *handled* by the application/developer

Message.h

```
#ifndef FOO_H
#define FOO_H

typedef nx_struct FooMsg {
        nx_uint16_t field1;
        nx_uint16_t field2;
        ….
        nx_uint16_t field_N;
    } FooMsg;

enum { FOO_OPTIONAL_CONSTANTS = 0 };

#endif
```

Header and Metadata

```
typedef nx_struct cc2420_header_t {
    nxle_uint8_t length;
    nxle_uint16_t fcf;
    nxle_uint8_t dsn;
    nxle_uint16_t destpan;
    nxle_uint16_t dest;
    nxle_uint16_t src;
    nxle_uint8_t network; // optionally included with 6LowPAN
    layer
    nxle_uint8_t type;
} cc2420_header_t;

typedef nx_struct cc2420_metadata_t {
    nx_uint8_t tx_power;
    nx_uint8_t rssi;
    nx_uint8_t lqi;
    nx_bool crc;
    nx_bool ack;
    nx_uint16_t time;
} cc2420_metadata_t;
```

Interfaces

- Components above the basic data-link layer MUST always access packet fields through interfaces (in /tos/interfaces/).

- Messages interfaces:
  - AMPacket: manipulate packets
  - AMSend
  - Receive
  - PacketAcknowledgements (Acks)

Sender Component

- AMSenderC

```
generic configuration AMSenderC(am_id_t id)
{
   provides {
      interface AMSend;
      interface Packet;
      interface AMPacket;
      interface PacketAcknowledgements as Acks;
   }
}
```

Receiver Component

- AMReceiverC

```
generic configuration AMReceiverC(am_id_t id)
{
   provides{
      interface Receive;
      interface Packet;
      interface AMPacket;
   }
}
```

Example 2 - RadioCountToLeds

- Create an application that counts over a timer and broadcast the counter in a wireless packet.

What do we need?

- Header File: to define message structure (RadioCountToLeds.h)
- Module component: to implement interfaces (RadioCountToLedsC.nc)
- Configuration component: to define the program graph, and the relationship among components (RadioCountToLedsAppC.nc)

Message Structure

- Message structure in RadioCountToLeds.h file

```
typedef nx_struct radio_count_msg_t {
   nx_uint16_t counter;  //counter value
} radio_count_msg_t;

enum {
   AM_RADIO_COUNT_MSG = 6, TIMER_PERIOD_MILLI = 250
};
```

Module Component

1. Specify the interfaces to be used
2. Define support variables
3. Initialize and start the radio
4. Implement the core of the application
5. Implement all the events of the used interfaces

Module Component

- Define the interfaces to be used:

```
module RadioCountToLedsC
{
    uses interface Packet;
    uses interface AMSend;
    uses interface Receive;
    uses interface SplitControl as AMControl;
}
```

Packet Manipulation Interfaces

Control interface

- Define some variables:

```
implementation {
    message_t packet;
    bool locked; ...
}
```

Local Variables

Initialize and Start the Radio

```
event void Boot.booted() {
    call AMControl.start();
    }


event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
        call MilliTimer.startPeriodic(TIMER_PERIOD_MILLI
    ); }
    else {
        call AMControl.start(); }
    }

event void AMControl.stopDone(error_t err) { }
```

Events to report Interface Operation

Implement the Application Logic

```
event void MilliTimer.fired() {
    ...
    if (!locked) {
    radio_count_msg_t* rcm = (radio_count_msg_t*)call
    Packet.getPayload(&packet,
    sizeof(radio_count_msg_t));

    rcm->counter = counter;

    if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
    sizeof(radio_count_msg_t)) == SUCCESS) {
        locked= TRUE; }
    }
}
```

Creates and Set Packet

Send Packet

Implement Events of Used Interfaces

```
event void AMSend.sendDone(message_t* msg, error_t error
{
   if (&packet == msg) {
      loked = FALSE;
   }
}
```

Must implement the events referred to all the interfaces of used components.

And What About Receiving?

- We need a Receive interface

```
uses interface Receive;
```

- We need to implement an event Receive handler

```
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t
    len) {
    if (len == sizeof(radio_count_msg_t)) {
        radio_count_msg_t* rcm= (radio_count_msg_t*)payload;
        call Leds.set(rcm->counter);
    }
    return msg;
}
```

- We need to modify the configuration component

```
implementation {
... components new AMReceiverC(AM_RADIO_COUNT_MSG); ... }
implementation {
... App.Receive -> AMReceiverC; ... }
```

Configuration File

```
implementation {

...
components ActiveMessageC;
components new AMSenderC(AM_RADIO_COUNT_MSG);

...

App.Packet -> AMSenderC;
App.AMSend -> AMSenderC;
App.AMControl -> ActiveMessageC;
...

}
```

RadioCountToLeds - Demo

- Let’s have a look at the files
- Let’s see how it works
- Let’s try to turn off a device

- Can you do that in Cooja?

Printf

TestPrintfC

```
#include"printf.h"
module TestPrintfC {
  uses {
  interface Boot;
  interface Timer<TMilli>;
  }
}
implementation {
  event void Boot.booted() {
    call Timer.startPeriodic(1000);
  }

  event void Timer.fired() {
    printf("Hi I am writing to you from my TinyOS
application!!\n");
    printfflush();
    }
}
```

TestPrintfAppC

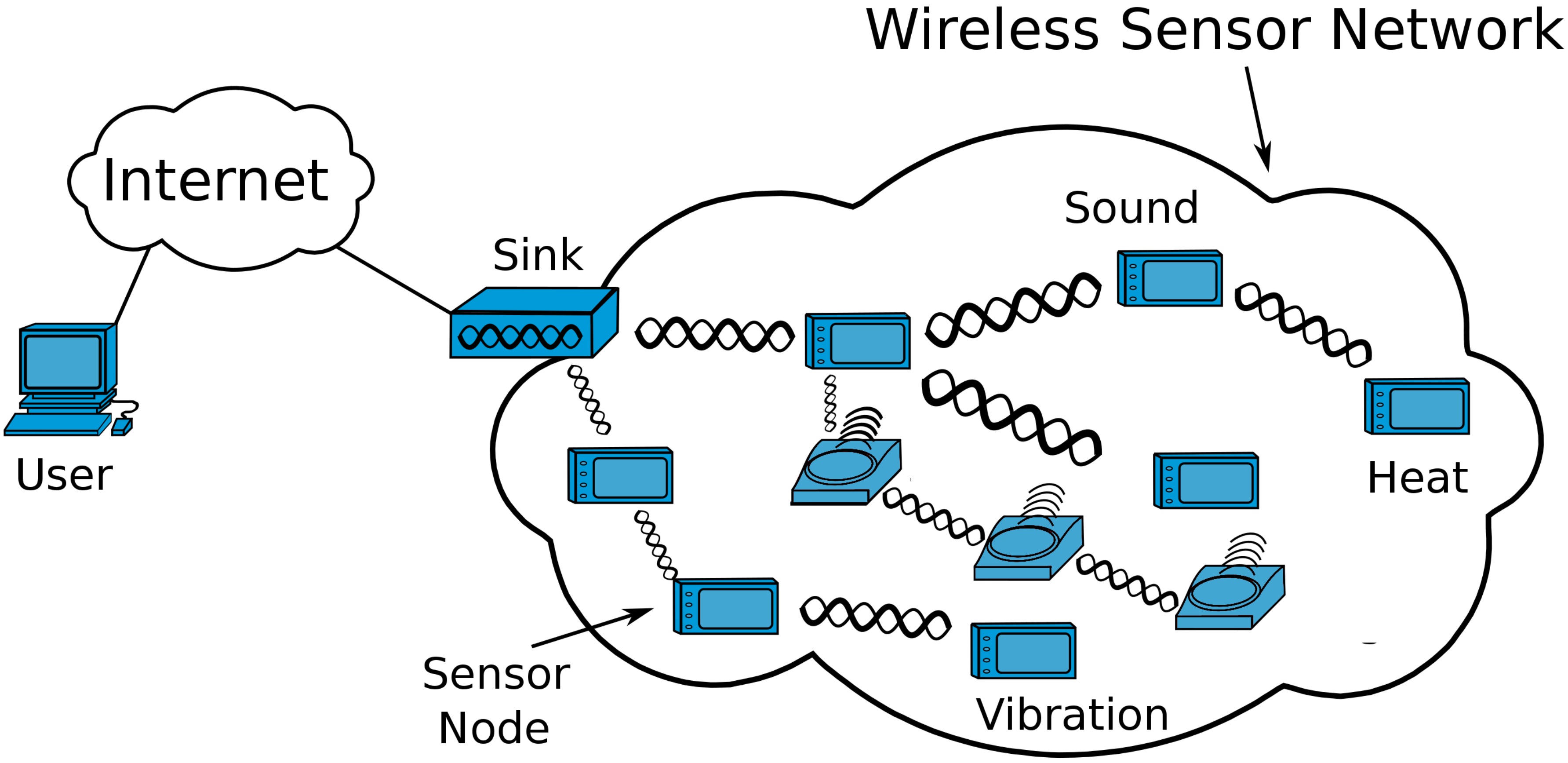
```
#define NEW_PRINTF_SEMANTICS
#include "printf.h"

configuration TestPrintfAppC{
}

implementation {
  components MainC, TestPrintfC;
  components new TimerMilliC();
  components SerialPrintfC;
  components SerialStartC;

  TestPrintfC.Boot -> MainC;
  TestPrintfC.Timer -> TimerMilliC;
}
```

Printf

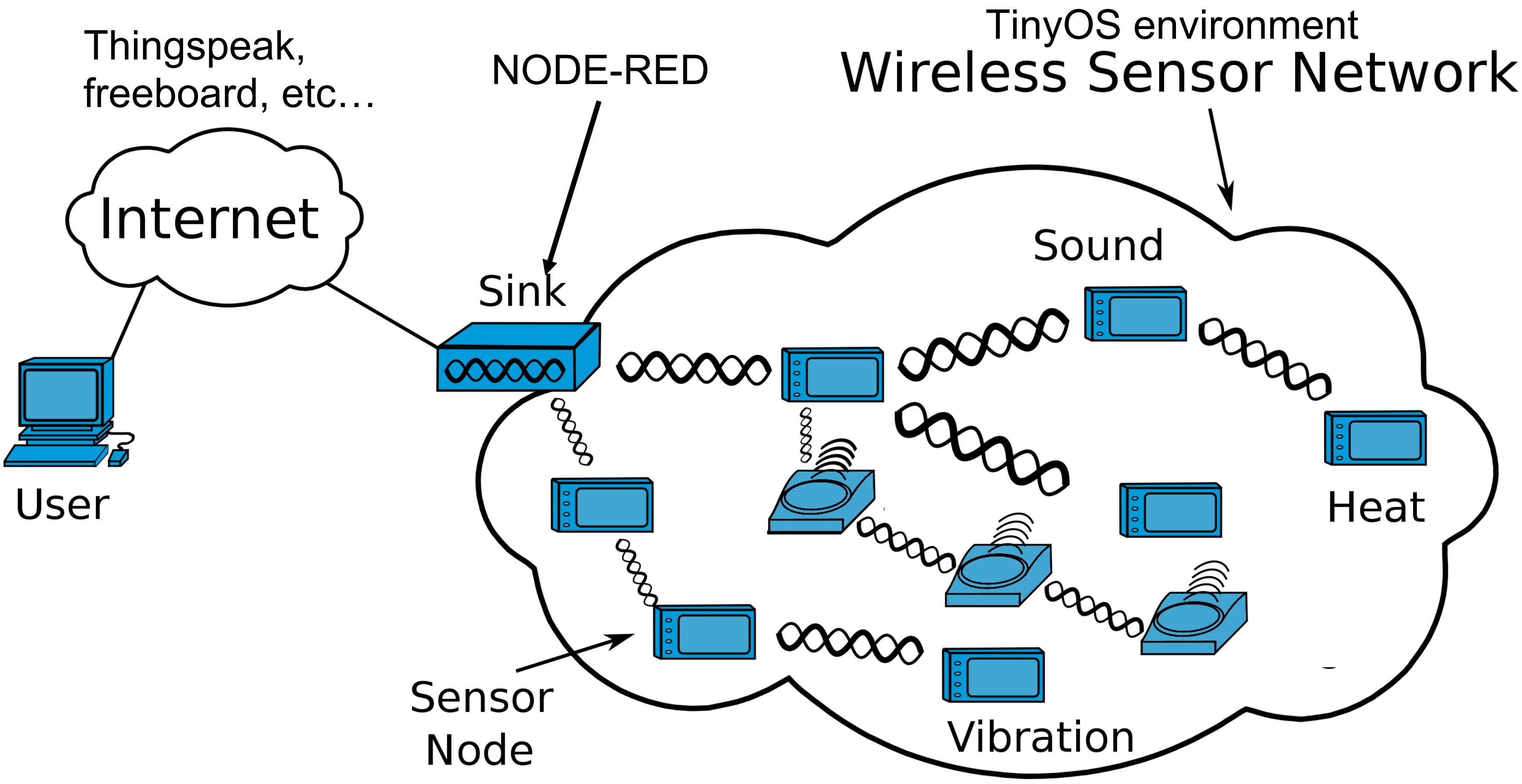
- Add the following line to the Makefile to add the library

```
CFLAGS += -I$(TOSDIR)/lib/printf
```

Architecture



Architecture



TinyOS and Node-red

- Goal: read tinyos msgs on node-red
- Example: tinyos-main/apps/tutorials/Printf
- Printf is used to make debug visible on cooja
- Let’s simulate it with Cooja

Cooja and NodeRED

- Cooja can be attached to NodeRED using the serial monitor
- Data coming from a WSN can be thus used as starting point for high-level, web-based applications
- Let’s see an example…

Cooja and NodeRED

- Cooja:
  1. Start the serial socket (server) on node

- Node-red
1. Use the tcp input block to collect the messages coming from cooja
2. Set port and hostname correctly!
3. Parse as Stream of String
4. Read the message with a debug node