TinyOS
WSN applications

- Monitoring Environments
  - habitat monitoring, conservation biology, ...
  - Precision agriculture, land conservation, ...
  - built environment comfort & efficiency ...
  - alarms, security, surveillance, treaty verification

- Monitoring Structures and Things
  - structural response, condition-based maintenance
  - disaster management
  - urban terrain mapping & monitoring

- Interactive Environments
  - manufacturing, asset tracking, fleet & franchise
  - context aware computing, non-verbal communication

- Assistance
  - home/elder care
Common features

- Low-cost hardware
- Data sensing
- Wireless communication
- Data logging / storing / analysis
Sensor Hardware

Location Finding System
- Sensor
- ADC

Mobilizer
- Processor
- Memory
- Transceiver

Power Unit

ANTENNA
Internet of Things
TinyOS Programming and TOSSIM
(and Cooja)
Agenda

- Playing with TinyOS
  - Programming and components
  - Blink Application
- Using the Radio
  - RadioCountToLeds Application
- TinyOS SIMulator (TOSSIM)
  - General concepts
  - Send/ACK example
  - TOSSIM Live + Cooja
"TinyOS is an open-source operating system designed for wireless embedded sensor networks"

http://www.tinyos.net/
TinyOS Overview

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

- Scheduler + Graph of Components
  - constrained two-level scheduling model: tasks + events

- A component
  - specifies a set of interfaces by which it is connected to other components
  - provides a set of interfaces to others
  - uses a set of interfaces provided by others

- Interfaces are bi-directional
  - include commands and events

- Constrained Storage Model
  - frame per component, shared stack, no heap

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

### Clock
```plaintext
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

### Timer Component
```plaintext
Clock.nc
```
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
- Are preempted by events
- Able to perform operations beyond event context
- May call commands
- May signal events
- Not preempted by tasks

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

task void TskName()
{
  ...
}
```
Typical application use of tasks

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

```c
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);
}
```

- 128 Hz sampling rate
- Simple FIR filter
- Dynamic software tuning for centering the magnetometer signal (1208 bytes)
- Digital control of analog, not DSP
- ADC (196 bytes)
TinyOS Scheduling

- Tasks do computations
  - Non-preemptive FIFO scheduling
  - Bounded number of pending tasks

- Events handle concurrent dataflows
  - Interrupts trigger lowest level events
  - Events preempt tasks, tasks do not
  - Events can signal events, call commands, or post tasks
Playing with TinyOS

Components, Event Handler and Wiring
TinyOS is written in a C “dialect” *nesC*


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
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**

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

**Timer.nc**

```c
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 Application

- **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
#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();
    }
}
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;
}
BlinkC interfaces:
- Fired and Booted events
- Toggle command

BlinkAppC components:
- TimerN.Timer and MainC.Boot
- LedsC.Toggle
Blink Application - Demo

- TinyOS v2.1 already (and properly) installed
  (refer to [www.tinyos.net](http://www.tinyos.net) for installations guides)
- TinyOS Runs on different Platforms:
  - Linux (Red Hat, Ubuntu, Xubuntu, Debian)
  - Windows (through CygWin)
- Test applications are in tiny-os/apps
- How to compile it for MICAz platform
  - A look at the `makefile`
  - Compiling commands “make micaz”
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, TOSSIM and Cooja

- TOSSIM (TinyOS SIMulator) is the “official” simulator for TinyOS (we’ll see it later)
- Cooja is the Contiki WSN simulator
- Initially developed for Contiki, but it can run TinyOS applications too!
- Let’s run Blink on Cooja
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!
Using the Radio

Creating/Sending/Receiving/Manipulating Messages
Active Message

- event handler (am_id): “address”
- Sender
  - Name a handler
  - Request Transmission
  - Done - completion signal
- Receiver
  - The corresponding event handler invoked
General Idea

**SENDER**

1. Fill Message
2. Specify Destination
3. Pass Message to OS
4. Wait For Feedback

**RECEIVER**

1. OS Stores Message
2. OS Signal Message Reception
3. Pass Message to Applications
Message Buffer Abstraction

- In `tos/types/messages.h`

```c
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 are "implemented" by the specific link layer (radio module + platform)

- Data is "handled" by the application
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/).

<table>
<thead>
<tr>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMPacket</td>
<td>Manipulate Packets</td>
</tr>
<tr>
<td>Packet</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>Send Packets</td>
</tr>
<tr>
<td>AMSend</td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Receive Packets</td>
</tr>
</tbody>
</table>
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

```c
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:

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

- Define some variables:

```c
implementation {
    message_t packet;
    bool locked; ...
}
```
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) {
}
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;
        }
    }
}
Implement Events of Used Interfaces

```c
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
  ```c
  uses interface Receive;
  ```

- We need to implement an event Receive handler
  ```c
  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
  ```c
  implementation {
    ... components new AMReceiverC(AM_RADIO_COUNT_MSG); ... } 
  implementation {
    ... App.Receive -> AMReceiverC; ... }
  ```
Configuration File

```java
implementation {

... 

components ActiveMessageC;
components new AMSenderC(AM_RADIO_COUNT_MSG);

...

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

}
```
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?
TinyOS SIMulator

Simulate a Wireless Sensor Network
Motivations

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

- Check correctness of application behavior
- Sensors are hard to debug!
  - “... prepare to a painful experience”
    - [Tinyos authors’ own words]
Simulation Pros and Cons

- Advantages
  - Study system in controlled environment
  - Observe interactions difficult to capture live
  - Helps in design improvement
  - Cost effective alternative

- Disadvantages
  - May not represent accurate real-world results
  - Depends on modeling assumptions
Existing tools

- Not very suitable for WSN
  - NS2, NS3, OMNet… Too fine grained / too coarse

- Do not address the complex interactions
  - Among motes (application level)
  - Within motes (network stack)

- Need to handle 2 parallel, different implementations!
General concepts

- TOSSIM is a discrete event simulator
- It uses the same code that you use to program the sensors
- There are two programming interfaces supported: Python and C++
Key requirements

- **Scalability**
  - Large deployments \((10^3\text{ motes})\)

- **Completeness**
  - Cover as many interactions as possible
  - Simulate complete applications

- **Fidelity**
  - Capture real-world interactions
  - Reveal unexpected behavior

- **Bridging**
  - Between algorithm and implementation
Configuring a Network

- It’s easy to simulate large networks
- You have to specify a network topology
- The default TOSSIM radio model is signal-strength based
Network Topology

- You can create network topologies in terms of channel gain
  - source, the destination and the gain (for example “1 2 -54.0”)
Radio Channel

- You need to feed a noise trace
- The directory `tos/lib/tossim/noise` contains some sample models
- On reception, SNR is evaluated for each bit of the message
Debugging Statements

- The output is configurable by channels
  - `<ch>` identifies the output channel
  - `<text>` text debugging and application variables

- `dbg(<ch>,<text>) → DEBUG(ID):<text>`
- `dbg_clear → <text>`
- `dbgerror → ERROR(ID):<text>`

- A channel can have multiple outputs
How to Run TOSSIM

- To compile TOSSIM you pass the sim option to make: “make micaz sim”

- You can either write a script and just tell Python to run it, or you can compile a CPP executable file
Send/ACK example
Many applications require to collect data on a pc/gateway connected to the Internet

Typically, the sink is connected to a PC via serial communication

It is possible to simulate the serial link in different ways:
- TossimLIVE
- Cooja
TossimLIVE

- Tossim can be used to communicate through a virtual serial port
- Useful for writing user applications
- Server side application can be written in Java / C / C++

- Example: TossimLIVE with Java interface (look at the README!)
TinyOS, Cooja and NodeRed

- TossimLIVE is good, but working in the terminal is not that nice
- We saw that Cooja can simulate TinyOS application in a nice graphical environment
- Cooja can also provide a virtual serial connection, just as TossimLIVE
- The (TCP) serial connection can also be used to communicate with Node-RED (see provided example)