Outline

- Sensors and Actuators
- Mote hardware
- Mote software
Ideal sensor system architecture

Physical signal at source

Signal Channel

Physical signal at transducer

Transducer

Analog Signal Processing

A/D data Conversion

Digital signal processing

Output signal
Non-ideal operation

- Analog interface noise
- Physical signal at source
- Noise signals
  - Noise Channel(s)
  - Transducer input-referred noise
- Crosstalk
- Quantization noise

Signal Channel

Physical signal at transducer

Transducer

Analog Signal Processing

A/D data Conversion

Digital signal processing

Output signal
Environmental Sensors

- **Temperature sensors**
  - Temperature-sensitive resistors: *thermistors*
    - Calibration curve

- **Humidity sensors**
  - Measure changes in dielectric constant

- **Pressure sensors**
  - Transform pressure to observable displacement
    - Mercury or water columns
    - Aneroid
    - MEMS
Motion and force sensors

- Translate sensor package motion or force to electronically detectable signals

- Acceleration sensors
  - Used in seismic measuring stations, airbags, manufacturing process control, and inertial navigation systems
    - Mass supported by spring

- Piezoresistive strain/force transducers
  - Translate force to resistance variation

- Piezoelectric transducers
  - Translate force to electric charge and potential difference
Sensing application model

- Most sensing applications have a similar operating mode
  - Sense
  - Process
  - Transmit
  - Sleep

- Focus on energy efficiency
  - Wakeup quickly
  - Finish application tasks promptly
  - Consume as little energy as possible during sleep
Power Consumption

- **Challenges**
  - Energy to wirelessly transport bits is ~constant
    - Shannon, Maxwell
  - Fundamental limit on ADC speed*resolution/power
  - No Moore’s law for battery technology
    - ~ 5%/year capacity improvement

- **How is power consumed**
  - CPU
  - Radio
  - Sensors
Source of Power: Batteries

- When in operation the **electrochemical cell** essentially **discharges** its chemical energy in favor of electric energy. If the cell is connected via an external circuit from the cathode to the anode, electrons flow from the oxidized anode and are received by the cathode, which is subsequently reduced. The electric circuit is completed by cations and anions, within the electrolyte, which flow to the cathode and anode, respectively.

![Cell in Operation (Discharge)]
Battery Characteristics

- Important characteristics:
  - Energy density (Wh/liter) and specific energy (Wh/kg)
  - Open-circuit voltage, operating voltage
  - Cut-off voltage (at which considered discharged)
  - Shelf life (leakage)
  - Cycle life (rechargeable)

- The above are decided by “system chemistry”
  - Advances in materials and packaging have resulted in significant changes in older systems
  - New systems
    - Primary and secondary (rechargeable) Li
    - Secondary zinc-air, Ni-metal hydride
Modeling the Battery Behavior

- Theoretical capacity of battery is decided by the amount of the active material in the cell
  - Batteries often modeled as buckets of constant energy
- In reality, delivered or nominal capacity depends on how the battery is discharged
  - Discharge rate (load current)
  - Discharge profile and duty cycle
  - Operating voltage and power level drained
Battery Capacity

- Current in “C” rating: load current normalized to battery’s capacity
  - e.g. a discharge current of 1C for a capacity of 500 mA-hrs is 500 mA

*from [Powers95]*
Energy Consumption in Wireless Sensor Nodes

- Processing
  - excluding low-level processing for radio, sensors, actuators
- Radio
- Sensors
- Actuators
- Power supply
Power Consumption in CMOS Digital logic

\[ P = A \cdot C \cdot V^2 \cdot f + A \cdot I_{sw} \cdot V \cdot f + I_{leak} \cdot V \]

where

- \( A \) = activity factor (probability of 0 \( \rightarrow \) 1 transition)
- \( C \) = total chip capacitance
- \( V \) = total voltage swing, usually near the power supply voltage
- \( f \) = clock frequency
- \( I_{sw} \) = short circuit current when logic level changes
- \( I_{leak} \) = leakage current in diodes and transistors

Typical power consumption numbers (MSP430 @ 8 MHz)

- Active: 2 mA
- Sleep: 1 \( \mu \) A
Wireless communication subsystem consists of three components with substantially different characteristics.

Their relative importance depends on the transmission power of the radio.
Examples

The RF energy increases with transmission range.

The electronics energy for transmit and receive are typically comparable.

Telos $E_{tx} = 129\text{nJ/bit}, E_{rx} = 144\text{nJ/bit}$
Energy/bit ÷ Energy/op large even for short ranges!

<table>
<thead>
<tr>
<th>Mote-class Node</th>
<th>Transmit</th>
<th>720 nJ/bit</th>
<th>Processor</th>
<th>4 nJ/op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive</td>
<td>110 nJ/bit</td>
<td>~ 200 ops/bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Energy breakdown for acoustic

- Encode
- Decode
- Receive
- Transmit

Energy breakdown for image

- Encode
- Decode
- Transmit
- Receive
## Mote Evolution

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>AT90LS8535</td>
<td>ATmega163</td>
<td>ATmega128</td>
<td>TI MSP430</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program memory (KB)</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM (KB)</td>
<td>0.5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Power (mW)</td>
<td>15</td>
<td>15</td>
<td>8</td>
<td>33</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleep Power (μW)</td>
<td>45</td>
<td>45</td>
<td>75</td>
<td>75</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wakeup Time (μs)</td>
<td>1000</td>
<td>36</td>
<td>180</td>
<td>180</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonvolatile storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip</td>
<td>24LC256</td>
<td>AT45DB041B</td>
<td>ST M24M01S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection type</td>
<td>I²C</td>
<td>SPI</td>
<td>I²C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (KB)</td>
<td>32</td>
<td>512</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio</td>
<td>TR1000</td>
<td>TR1000</td>
<td>CC1000</td>
<td>CC2420</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data rate (kbps)</td>
<td>10</td>
<td>40</td>
<td>28.4</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulation type</td>
<td>OOK</td>
<td>ASK</td>
<td>FSK</td>
<td>O-QPSK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive Power (mW)</td>
<td>9</td>
<td>12</td>
<td>29</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmit Power at 0dBm (mW)</td>
<td>36</td>
<td>36</td>
<td>42</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Consumption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum Operation (V)</td>
<td>2.7</td>
<td>2.7</td>
<td>2.7</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Active Power (mW)</td>
<td>24</td>
<td>27</td>
<td>44</td>
<td>89</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming and Sensor Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expansion</td>
<td>none</td>
<td>51-pin</td>
<td>51-pin</td>
<td>none</td>
<td>51-pin</td>
<td>19-pin</td>
<td>51-pin</td>
<td>10-pin</td>
</tr>
<tr>
<td>Communication</td>
<td>IEEE 1284 (programming) and RS232 (requires additional hardware)</td>
<td>USB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrated Sensors</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Telos Platform

- **Radio:**
  - IEEE 802.15.4
    - CC2420 radio
    - 250kbps
    - 2.4GHz ISM band

- **Processor:**
  - TI MSP430 (16bit) @8MHz
    - 1.6 μA sleep
    - 2 mA active
    - 1.8V operation

- **Robustness**
  - Integrated antenna
  - Integrated sensors
  - Soldered connections
Comparison to MicaZ

- **Sleep (majority of the time)**
  - Telos: 5.1 μA, MicaZ: 27 μA

- **Wakeup (as quickly as possible to process and return to sleep)**
  - MCU
    - Telos: 6 μs, MicaZ: 180 μs
  - Radio
    - Telos: 580 μs, MicaZ: 860 μs

- **Active (get your work done and get back to sleep)**
  - Telos (16 bit): 1.8 mA, MicaZ: 8 mA
Networked embedded systems characteristics

- Resource constraints and low power consumption
  - First generation motes had 512 Bytes of RAM (!)
- Concurrency-intensive operation
  - multiple flows, not wait-command-respond
    - In some cases controller needs to service individual bits from the radio
- Limited Physical Parallelism and Controller Hierarchy
  - primitive direct-to-device interface
  - Asynchronous and synchronous devices
- Diversity in Design and Usage
  - application specific, not general purpose
  - huge device variation
- Robust Operation
  - numerous, unattended, critical
Tiny OS Concepts

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

- Component:
  - Written in nesC
    - Commands
    - Event Handlers
    - Frame (storage)
    - Tasks (concurrency)

- Components connect via interfaces
  - Connections called “wiring”

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

- Efficient Layering
  - Events can signal events
Component Types

- Modules are components that have variables and executable code
- Configurations are components that wire other components together
Component Example

- BlinkC wires BlinkP.Timer to TimerC.Timer

```
module BlinkP { ...
} implementation {
  int c;
  void increment() {c++;}
}
```

```
configuration BlinkC { ...
} implementation {
  components new TimerC();
  components BlinkC;
  BlinkC.Timer -> TimerC;
}
```
Interfaces

- Collections of related functions
- Define how components connect
- Interfaces are bi-directional: for A->B
  - Commands are from A to B
  - Events are from B to A
- Can have parameters (types)

```java
interface Timer<tag> {
    command void startOneShot(uint32_t period);
    command void startPeriodic(uint32_t period);
    event void fired();
}
```
Basic TinyOS Applications

- Goal: write an anti-theft device. Let's start simple.
- 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!
The Basics – Let’s Get Started

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

module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}

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

interface Timer<tag> {
    command void startOneShot(uint32_t period);
    command void startPeriodic(uint32_t period);
    event void 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);
}
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

configuration AntiTheftAppC {}
implementation {
components AntiTheftC, MainC, LedsC;
AntiTheftC.Boot -> MainC.Boot;
AntiTheftC.Leds -> LedsC;
components new TimerMilliC() as MyTimer;
AntiTheftC.Check -> MyTimer;
components new PhotoC();
AntiTheftC.Read -> PhotoC;
}

generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
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
Tasks

- TinyOS has a single stack: long-running computation can reduce responsiveness
- Tasks: mechanism to defer computation
  - Tells TinyOS “do this later”
- Tasks run to completion
  - TinyOS scheduler runs them one by one in the order they post
  - Keep them short!
- Interrupts run on stack, can post tasks
More Complex Application

- Let’s improve our anti-theft device. A clever thief could still steal our motes by keeping a light shining on them!
  - But the thief still needs to pick up a mote to steal it.
  - Theft Detection Algorithm 2: Every N ms, sample acceleration at 100Hz and check if variance above some threshold

- What we’ll see
  - (Relatively) high frequency sampling support
  - Use of tasks to defer computation-intensive activities
  - TinyOS execution model
Advanced Sensing, Tasks

uses interface ReadStream;
uint16_t accelSamples[ACCEL_SAMPLES];
event void Timer.fired() {
    call ReadStream.postBuffer(accelSamples, ACCEL_SAMPLES);
    call ReadStream.read(10000);
}

event void ReadStream.readDone(error_t ok, uint32_t actualPeriod) {
    if (ok == SUCCESS)
        post checkAcceleration();
}

task void checkAcceleration() {

ReadStream is an interface for periodic sampling of a sensor into one or more buffers.
• postBuffer adds one or more buffers for sampling
• read starts the sampling operation
• readDone is signalled when the last buffer is full

interface ReadStream<val_t> {
    command error_t postBuffer(val_t* buf, uint16_t count);
    command error_t read(uint32_t period);
    event void readDone(error_t ok, uint32_t actualPeriod);
}
In `readDone`, we need to compute the variance of the sample. We defer this “computationally-intensive” operation to a separate task, using `post`. We then compute the variance and report theft.
TinyOS Execution Model

Stack
- RealMainP
- AntiTheftC
- Timer
- Alarm

Task Queue
- Interrupt table
  - serial receive
  - H/W timer
  - A/D conv.

Diagram:
- RealMainP
- SchedulerP
- Timer
- Alarm
- AccelStreamC
TinyOS Execution Model

Stack

- Alarm
- Timer
- SchedulerP

Task Queue

- Timer task

Diagram:
- RealMainP
- SchedulerP
- Timer
- Alarm
- AccelStreamC

Tasks:
- serial receive
- H/W timer
- A/D conv.
TinyOS Execution Model

Stack
- SchedulerP
- Timer
- AntiTheftC
- AccelStreamC

Task Queue
- timer task

Interrupt table
- serial receive
- H/W timer
- A/D conv.
Acknowledgements

- TinyOS slides originally from TinyOS tutorial given at IPSN 2009