Mobile and Ubiquitous Computing
Resource Constrained Devices

George Roussos
g.roussos@dcs.bbk.ac.uk

Session Overview
- Resource constrained devices
  - evolution, architecture, components
  - a detailed example
- Energy efficiency
- Programming primitives in Tiny OS
- Concurrency

Moore’s Law:
"the complexity of an integrated circuit, with respect to minimum component cost, will double in about 18 months"

More Drivers

- Cheap and reliable communications:
  - short-range RF, infrared, optical
  - low power
- New interesting sensors
  - light, heat, humidity
  - position, movement, acceleration, vibration
  - chemical presence, biosensor
  - magnetic field, electrical inc. bio-signals (ECG and EEG)
  - RFID
  - acoustic (microphone)

Long-term objective

- Completely integrated
  - one package includes: computation, communication, sensing, actuation, (renewable) power source
  - modular
- Less than a cubic millimeter in volume
- Cheap
- Diverse in design and usage
- Robust
- Main challenge: energy efficiency!

Device evolution

Internet 0 at MIT Centre of Atoms and Bits
http://cba.mit.edu/~neilg

Smart-its  http://www.smart-its.org/

gumstix  http://www.gumstix.org/
What else is out there?

- **pico-TRON**
  - Hardware-software platform from Japan
  - Derived from TRON
  - http://www.t-engine.org/

- **IMEC Sensor Cube**
  - Very low power, modular design for body area applications
  - Tiny OS and embedded C

---

**Tmote Sky**

- Texas Instruments MSP430
  - 16-bit RISC, 8MHz, 10k RAM, 48k Flash, 128b storage
  - Integrated analog-to-digital converter (12 bit ADC)
- Chipcon wireless transceiver
  - IEEE 802.15.4 (Zigbee) compatible
  - 250kbps at 2.4GHz
- Sensirion SHT11/SHT15 sensor module
  - Humidity and temperature
- Hamamatsu light sensors
  - S1087 (photosynthetic)
  - S1087-01 (full visible spectrum)

---

**Module layout (top)**

[Diagram of Tmote Sky module layout]
Module layout (bottom)

Block diagram

Where does the power go?

- Processing
  - excluding low-level processing for radio, sensors, actuators
- Radio
- Sensors
- Actuators
- Power supply

discussion follows Srivastana tutorial
(check module website)
Sky module characteristics

<table>
<thead>
<tr>
<th>Current Consumption</th>
<th>MCU off, Radio RX</th>
<th>21.5</th>
<th>23 mA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Consumption</td>
<td>MCU off, Radio RX</td>
<td>190</td>
<td>15 mA</td>
</tr>
<tr>
<td>Current Consumption</td>
<td>MCU on, Radio off</td>
<td>1800</td>
<td>2400 µA</td>
</tr>
<tr>
<td>Current Consumption</td>
<td>MCU off, Radio off</td>
<td>14.3</td>
<td>1200 µA</td>
</tr>
<tr>
<td>Current Consumption</td>
<td>MCU standby</td>
<td>5.1</td>
<td>210 µA</td>
</tr>
</tbody>
</table>

Need power management to actually exploit energy efficiency:
• idle and sleep modes
• variable voltage
• variable frequency
• in-network storage and processing

Chipcon radio is only a transceiver, and a lot of low-level processing takes place in the main CPU. Contrast this with Wi-Fi radio which will do everything up to MAC and link level encryption in the "radio."

Sensors and power consumption

• Several energy consumption sources
  – transducer
  – front-end processing and signal conditioning
    • analog, digital
  – ADC conversion

• Diversity of sensors: no general conclusions can be drawn
  – Low-power modalities
    • Temperature, light, accelerometer
  – Medium-power modalities
    • Acoustic, magnetic
  – High-power modalities
    • Image, video, chemical

Observations

• Radio benefits less from technology improvements than processors
• The relative impact of the communication subsystem on the system energy consumption will grow
• Using low-power components and trading-off unnecessary performance for power savings can have orders of magnitude impact
• Node power consumption is strongly dependent on the operating mode
• At short ranges, the Rx power consumption > T power consumption
• Idle radio consumes almost as much power as radio in Rx mode
• Processor power fairly significant (30-50%) share of overall power
• In many cases, the sensor overhead is negligible
Programming challenges

- Driven by interaction with environment
  - Data collection and control, not general purpose computation
  - Reactive, event-driven programming model
- Extremely limited resources
  - Very low cost, size, and power consumption
  - Typical embedded OSs consume hundreds of KB of memory
- Reliability for long-lived applications
  - Apps run for months/years without human intervention
  - Reduce run time errors and complexity
- Soft real-time requirements
  - Few time-critical tasks (sensor acquisition and radio timing)
  - Timing constraints through complete control over app and OS

Current popular platform

- **NesC**: a C dialect for embedded programming
  - Components, “wired together”
  - Quick commands and async events
- **TinyOS**: a set of NesC components
  - Hardware components
  - ad-hoc network formation & maintenance
  - time synchronization

Tiny OS facts

- Very small “operating system” for sensor networks
  - Core OS requires 396 bytes of memory
- Component-oriented architecture
  - Set of reusable system components: sensing, communication, timers, etc.
  - No binary kernel - build app specific OS from components
- Concurrency based on **tasks and events**
  - Task: deferred computation, runs to completion, no preemption
  - Event: invoked by module (upcall) or interrupt, may preempt tasks or other events
  - Very low overhead, no threads
- Split-phase operations
  - No blocking operations
  - Long-latency ops (sensing, comm, etc.) are split phase
  - Request to execute an operation returns immediately
  - Event signals completion of operation

*discussion follows Welsh check module website*
**nesC facts**

- Dialect of C with support for **components**
  - Components provide and require interfaces
  - Create application by wiring together components using configurations
- Whole-program compilation and analysis
  - nesC compiles entire application into a single C file
  - Compiled to mote binary by back-end C compiler (e.g., gcc)
  - Allows aggressive cross-component inlining
  - Static data-race detection
- Important restrictions
  - No function pointers (makes whole-program analysis difficult)
  - No dynamic memory allocation
  - No dynamic component instantiation/destruction
  - These static requirements enable analysis and optimization

**nesC interfaces**

nesC interfaces are bidirectional
- **Command**: Function call from one component requesting service from another
- **Event**: Function call indicating completion of service by a component
- Grouping commands/events together makes inter-component protocols clear

```c
interface Timer {
    command result t start(timer_type, start2 interval);  
    command result t step();
    event result t finished();
}

interface SendMsg {
    command result t send(Msg msg, uint8_t length);
    event result t sendDone(Msg msg, result t success);
}
```

**nesC components**

- Two types of components
  - **Modules** contain implementation code
  - **Configurations** wire other components together
  - An application is defined with a single top-level configuration

```c
module TimerM {
    provides {
        interface社保Interval;
        interface Timer;
    }
    use interface Clock;
}
```

```c
implementation {
    command result t timer.start(timer_type, start2 interval) {
    }
    command result t timer.step() {
    }
    event void clock tick() {
    }
```
nesC configurations

```nesC
configuration TimerC {
    provides {
        interface StartControl;
        interface Timer;
    }
}
}

implementation {
    component TimerM, HWLock;

    // Start-through: Connect our "provides" to TimerM "provides"
    StartControl = TimerM.StartControl;
    Timer = TimerM.Timer;

    // Normal wiring: Connect "requires" to "provides"
    TimerM.Clock = HWLock.Clock;
}
```

Concurrency in nesC

- **Tasks** used as deferred computation mechanism
  - Commands and events cannot block
  - Tasks run to completion, scheduled non-preemptively
  - Scheduler may be FIFO, EDF, etc.

```nesC
// Synchronized by interrupt handler
void TaskHandler()
{
    if (eventOccurred) {
        return; // Drop!
    }
    error_task.key = TASK;  
    error_task.mag = msg;  
    post(error_task); 
}
```

- **More on concurrency**
  - All code is classified as one of two types:
    - **Asynchronous code (AC):** Code reachable from at least one interrupt handler
    - **Synchronous code (SC):** Code reachable only from tasks
  - Any update to shared state from AC is a potential data race
    - SC is atomic with respect to other SC (no preemption)
    - Race conditions are shared variables between SC and AC, and AC and AC
    - Compiler detects data races by walking call graph from interrupt handlers
Avoiding a data race

• Two ways to fix a data race
  – Move shared variable access into tasks
  – Use an atomic section
    or
  – Short, run-to-completion atomic blocks
    – Currently implemented by disabling interrupts

```
atomic {
  sharedvar = sharedvar+1;
}
```

Summary

• Resource constrained devices
  – evolution, architecture, components
  – a detailed example
• Energy efficiency
• Programming primitives in Tiny OS
• Concurrency