Sensor Platform: UCB Motes

- Vision
- Hardware overview
- Software overview

The Systems Challenge

- 10 years from now the most of the network will be small, embedded devices
  - today ship 200 M microproc/year, 8.5 B embedded proc/year
- What’s the challenge?
  - (a) They won’t be individually important like your PC, laptop, PDA, cell phone
    - 100s-1,000s per person
    - reside where no system admin can go
  - (b) Highly Application Specific
  - (c) Highly constrained resources (storage, energy)
  - (d) Must be Robust despite changing environment
  - (e) All of the above
Starting Point

- Hands-on Experience with Large Networks of Tiny Network sensors
  - small microcontroller, low-power radio, flash/eeeprom
  - sensor and power boards
  - tiny event driven operating system
  - intense constraints, freedom of abstraction
- Re-explore entire range of networking issues
  - encoding, framing, error handling
  - media access control, transmission rate control
  - discovery, multihop routing
  - broadcast, multicast, aggregation
  - active network capsule (reprogramming)
  - localization, time synchronization
  - security, network-wide protection
  - density independent wake-up and proximity est.
- Fundamentally new aspects in each

Key Characteristics of TNDs

- Small physical size and low power consumption
  => Limited Physical Parallelism and Controller Hierarchy
  => primitive direct-to-device interface
- Concurrency-intensive operation
  - flow-thru, not wait-command-respond
  => must handle multiple inputs and outputs simultaneously
- Diverse in Design and Usage
  - application specific, not general purpose
  - huge device variation
  => efficient modularity
  => migration across HW/SW boundary
- Largely Unattended & Numerous
  => robust operation
  => narrow interfaces
‘Mote’–The Hardware

- 4Mhz, 8bit MCU (ATMEL)
- 512 bytes RAM, 8K ROM
- 900Mhz Radio (RF Monolithics)
  - 10-100 ft. range
- Temperature Sensor
- Light Sensor
- LED outputs
- Serial Port

Second Generation ‘Mote’

- Two Board Sandwich
  - Main CPU board with Radio Communication
  - Secondary Sensor Board
- Allows for expansion and customization

- Current sensors include: Acceleration, Magnetic Field, Temperature, Pressure, Humidity, Light, and RF Signal Strength.
- Can control RF transmission strength & Sense Reception Strength
Networked Sensor/Act Node

- 1" x 1.5" motherboard
  - ATMEL 4Mhz, 8bit MCU, 512 bytes RAM, 8KB pgm flas
  - 900Mhz Radio (RF Monolithics) 1-10+ m range
  - ATMEL network pgming assist
  - Radio Signal strength control and sensing
  - I2C EPROM (logging)
  - Base-station ready
  - stackable expansion connector
    - all ports, i2c, pwr, clock...
- Several sensor boards
  - basic protoboard
  - tiny weather station (temp, light, hum, press)
  - vibrations (2d acc, temp, LIGHT)
  - accelerometers
  - magnetometers
- Integrated "quarter size" node

Basic Power Breakdown...

<table>
<thead>
<tr>
<th></th>
<th>Active</th>
<th>Idle</th>
<th>Sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>5 mA</td>
<td>2 mA</td>
<td>5 µA</td>
</tr>
<tr>
<td>Radio</td>
<td>7 mA (TX)</td>
<td>4.5 mA (RX)</td>
<td>5 µA</td>
</tr>
<tr>
<td>EE-Prom</td>
<td>3 mA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LED's</td>
<td>4 mA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Photo Diode</td>
<td>200 µA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Temperature</td>
<td>200 µA</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- But what does this mean?
  - Lithium Battery runs for 35 hours at peak load and years at minimum load!
    - three orders of magnitude difference!
  - A one byte transmission uses the same energy as approx 11000 cycles of computation.
  - Idleness is not enough, sleep!
Experimenting at Scale

An Operating System for Tiny Devices?

- Would love to have theoretically-sound tools to go from req. to implementation, but...
- Traditional approaches
  - command processing loop (wait request, act, respond)
  - monolithic event processing
  - bring full thread/socket posix regime to platform
- Alternative
  - provide framework for concurrency and modularity
  - never poll, never block
  - interleaving flows, events, energy management
  - => allow appropriate abstractions to emerge
Tiny OS Concepts

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

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

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

- Very lean multithreading
- Efficient Layering

TinyOS Program Structure

- Application = graph of components + schedule
TOS Execution Model

- commands request action
  - ack/nack at every boundary
  - call cmd or post task
- events notify occurrence
  - HW intrpt at lowest level
  - may signal events
  - call cmds
  - post tasks
- Tasks provide logical concurrency
  - preempted by events
- Migration of HW/SW boundary

TinyOS Execution Contexts
Dynamics of Events and Threads

bit event =>
end of byte =>
end of packet =>
end of msg send

thread posted to start
send next message

radio takes clock events to detect recv

Quantitative Analysis...

Power down when task queue empty

<table>
<thead>
<tr>
<th>Components</th>
<th>Packet reception unit breakdown</th>
<th>Percent CPU utilization</th>
<th>Energy (nj/Bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>0.05%</td>
<td>0.20%</td>
<td>0.33</td>
</tr>
<tr>
<td>Packet</td>
<td>1.12%</td>
<td>0.51%</td>
<td>7.58</td>
</tr>
<tr>
<td>Radio handler</td>
<td>26.87%</td>
<td>12.16%</td>
<td>182.38</td>
</tr>
<tr>
<td>Radio decode thread</td>
<td>5.48%</td>
<td>2.48%</td>
<td>37.2</td>
</tr>
<tr>
<td>RFM</td>
<td>66.48%</td>
<td>30.08%</td>
<td>451.17</td>
</tr>
<tr>
<td>Radio Reception</td>
<td>-</td>
<td>-</td>
<td>1350</td>
</tr>
<tr>
<td>idle</td>
<td>-</td>
<td>54.75%</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>100.00%</td>
<td>100.00%</td>
<td>2028.66</td>
</tr>
</tbody>
</table>
Maintaining Scheduling Agility

- Need logical concurrency at many levels of the graph
- While meeting hard timing constraints
  - sample the radio in every bit window

- Retain event-driven structure throughout application
- Tasks extend processing outside event window
- All operations are non-blocking
  - lock-free scheduling queue

Typical split-phase pattern

```c
char TOS_EVENT(SENS_OUTPUT_CLOCK_EVENT)(){
    return TOS_CALL_COMMAND(SENS_GET_DATA());
}
```

```c
char TOS_EVENT(SENS_DATA_READY)(int data){
    VAR(buffer)[VAR(index)++] = data;
    if (full()) TOS_POST_TASK(FILTER_DATA);
    return 1;
}
```

- clock event handler initiates data collection
- sensor signals data ready event
- data event handler posts task to process data
Conservative Component Coupling

- Each component has bounded state and concurrency
- Each command interface has explicit handshake
- Each component must deal with rejections

- Message layer may reject send request if full or busy
- Requestor cannot busy wait
  - send\_done event broadcast to all potential senders
    - send\_buffer pointer used to disambiguate
    - can elect to retry or drop

Communication Storage Management

- Strict ownership protocol at appln components
- Each component ‘owns’ set of buffers
- send => comp. ‘gives’ out-buffer till send\_done
  - component tracks state
- receive => system ‘gives’ in-buffer to handler
  - handler must return a free buffer to the system
  - if completely consumed, returns same
  - otherwise, returns another one it ‘owns’
    - if none available, must give back incoming
Crossing Layers without buffering

- stack consists of series of data pumps
- each peels off portion of packet and feeds to lower layer
- task starts event-driven data pump

Low-Power Listening

- Costs about as much to listen as to xmit, even when nothing is received
- Only way to save power is to turn radio off when there is nothing to hear.
- Can turn radio on/of in <1 bit
  - 30 ms on every 300 ms
  - Can detect transmission at cost of ~2 bit times
  ⇒ Small sub-msg recv sampling

⇒ Application-level synchronization rendezvous to determine when to sample

Optimal Preamble = \((2/3 \text{ Sxb})^{1/2}\)
Network Discovery

Rich set of additional challenges

- Efficient and robust security primitives
- Application specific virtual machines
- Time & space information in every packet
- Density independent wake-up, aggregation
  - sensor => can use radio in ‘analog’ mode
- Resilient aggregators
- Programming support for systems of generalized state machines
- Programming the unstructured aggregate
  - SPMD, Data Parallel, Query Processing, Tuples
- Understanding how an extreme system is behaving and what is its envelope
  - adversarial simulation
- Self-configuring, self-correcting systems
Summary

- Distribute the embedded system over many small devices
- Integrated them with communication
- New set of embedded software challenges
  - local scheduling, synthesis, etc. must address resource constraints
  - plus the distributed aspects
- Operating against energy constraints
  - rather than overload
- Inherent asynchrony
- NEST platform due in Jan
  - working 10/5

Tiny OS – The Software

- Provides a component based model abstracting hardware specifics from application programmer.
- Capable of maintaining high levels of concurrency.
- Allows multiple applications to be “running.”
- Services Provided Include:
  - RF messaging protocols.
  - Periodic Timer Events.
  - Asynchronous access to UART data transfers.
  - Mechanism for Static, Persistent Storage.
- Can “Swap Out” system components to get necessary functionality.
- Complete applications fit in 4KB of ROM and 256B RAM.

webs.cs.berkeley.edu or www.tinyos.org
Tiny OS Internals

- Scheduler and Graph of Components
  - constrained two-level scheduling model: tasks + events
- Component:
  - Frame (storage)
  - Tasks (concurrency)
  - Commands, and Handlers (events)
- Constrained Storage Model
  - frame per component, shared stack, no heap
- Very lean multithreading
- Layering
  - components issue commands to lower-level components
  - event signal high-level events, or call lower-level commands
    - Guarantees no cycles in call chain

Networked Sensor System Challenge

- Managing multiple concurrent flows
  - some with real-time requirements
- Very limited I/O controller hierarchy
  - process every bit, or perhaps byte in CPU
- Asynchronous and synchronous devices
- Limited storage and processing
- At very low power
TinyOS Execution Contexts

![Diagram showing TinyOS Execution Contexts]

TinyOS Storage Model

- Single shared stack
- Each component has a static frame
  - only accessible locally (except msg buffers)
- Msg buffers allocated statically by components, but shared dynamically by ownership discipline
TinyOS Commands and Events

Commands

- Function call across component boundary
  - cause action to be initiated
  - bounded amount of work
    - cannot block
  - always return status (0 => error)
    - component can refuse to perform command
- share call stack and execution context
- command body has access to local frame
- commands may post tasks or call commands
- commands may not signal events
Events

- Upcall to notify action has occurred
  - must do bounded (and small) amount of work
  - cannot block
  - access local frame, shares stack
- Lowest-level events triggered by hardware interrupts
  - hardware abstraction components perform critical section and enable interrupts
- Events may signal events
- Events may call commands

- Entire event “fountain” must stay within overall application jitter tolerance

FSM Programming Style

- Most components are essentially FSMs
- Composed via events and commands
- non-blocking!
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

Dynamics of Events and Tasks

- consecutive message sends
- event propagation
- generation of tasks, preempted by events
Tasks in low-level operation

- transmit packet
  - send command schedules task to calculate CRC
  - task initiated byte-level datapump
  - 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
- i2c component
  - i2c bus has long suspensive operations
  - tasks used to create split-phase interface
  - events can procede during bus transactions

High-level use of tasks

- virtual machine interpreter schedules an “interpretation task” on clock event
  - similar for data event
- may be a long running activity
- utilizes many low-level components
- reschedules itself on each virtual machine instruction
Scheduling

- current sched.c is simple fifo scheduler
- Bounded number of pending tasks
- When idle, shuts down node except clock

- Uses non-blocking task queue data structure