Outline

- Operating Systems for Sensor Nodes
  - Telos
- Language support for Sensor Nodes
  - nesC
Characteristics of Network Sensors

- Small physical size and low power consumption
- Concurrency-intensive operation
  - multiple flows, not wait-command-respond
- 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
  - => efficient modularity
  - => migration across HW/SW boundary
- Robust Operation
  - numerous, unattended, critical
  - => narrow interfaces

Tiny OS Concepts

- Scheduler + Graph of Components
  - constrained two-level scheduling model: tasks + events
- Component:
  - Commands
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)
- Constrained Storage Model
  - frame per component, shared stack, no heap
- Very lean multithreading
- Efficient Layering
  - Events can signal events
Application = Graph of Components

Example: ad hoc, multi-hop routing of photo sensor readings

3450 B code
226 B data

Graph of cooperating state machines on shared stack
Execution driven by interrupts

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

message-event driven
event-driven packet-pump

application comp
data processing
active message
"crc"
event-driven byte-pump
"encode/decode"
event-driven bit-pump
Dynamics of Events and Threads

- **bit event** => end of byte => end of packet => end of msg send
- thread posted to start send next message

### Programming TinyOS - nesC

- TinyOS 1.x is written in an extension of C, called nesC
- Applications are too!
  - just additional components composed with the OS components
- Provides syntax for TinyOS concurrency and storage model
  - commands, events, tasks
  - local frame variables
- Rich Compositional Support
  - separation of definition and linkage
  - robustness through narrow interfaces and reuse
  - interpositioning
- Whole system analysis and optimization
Event-Driven Sensor Access Pattern

- clock event handler initiates data collection
- sensor signals data ready event
- data event handler calls output command
- device sleeps or handles other activity while waiting
- conservative send/ack at component boundary

TinyOS Commands and Events

- `{ ...
  status = call CmdName(args)
  ...
  }`
- `{ ...
  status = signal EvtName(args)
  ...
  }
- `command CmdName(args) {
  ...
  return status;
  }
- `event EvtName)(args) {
  ...
  return status;
  }`
Split-phase abstraction of HW

- Command synchronously initiates action
- Device operates concurrently
- Signals event(s) in response
  - ADC
  - Clock
  - Send (UART, Radio, ...)
  - Recv – depending on model
  - Coprocessor
- Higher level (SW) processes don’t wait or poll
  - Allows automated power management
- Higher level components behave the same way
  - Tasks provide internal concurrency where there is no explicit hardware concurrency
  - Components (even subtrees) replaced by HW and vice versa

TinyOS Execution Contexts

- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks
- Both essential process state transitions
Storage Model

- Local storage associated with each component (or instance of)
  - Internally managed
  - Only objects passed by reference are message buffers

Data sharing

- Passed as arguments to command or event handler
  - Don’t make intra-node communication heavy-weight
- If queuing is appropriate, implement it
  - Send queue
  - Receive queue
  - Intermediate queue
- Bounded depth, overflow is explicit
  - Most components implement 1-deep queues at the interface
- If you want shared state, created an explicit component with interfaces to it.
**Tasks**

- provide concurrency internal to a component
  - longer running operations
- are preempted by events
- are preempted by events
- able to perform operations beyond event context
- may call commands
- may signal events
- not preempted by tasks

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

**Tasks in low-level operation**

- 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
- i2c component
  - i2c bus has long suspensive operations
  - tasks used to create split-phase interface
  - events can proceed during bus transactions
- Timer
  - Post task in-critical section, signal event when current task complete
**Task Scheduling**

- Currently simple fifo scheduler
- Bounded number of pending tasks
- When idle, shuts down node (except clock)

- Uses non-blocking task queue data structure

- Simple event-driven structure + control over complete application/system graph
  - instead of complex task priorities and IPC

**Communication**

- Essentially just like a call
- Receive is inherently asynchronous
- Don’t introduce potentially unbounded storage allocation
- Avoid copies and gather/scatter (mbuf problem)
Tiny Active Messages

- Sending
  - Declare buffer storage in a frame
  - Request Transmission
  - Name a handler
  - Handle Completion signal
- Receiving
  - Declare a handler
  - Firing a handler
    - automatic
    - behaves like any other event
- Buffer management
  - strict ownership exchange
  - tx: done event => reuse
  - rx: must rtn a buffer

Sending a message

```c
bool pending;
struct TOS_Msg buf;
command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)buf.data;
    if (!pending) {
        pending = TRUE;
        message->val = value;
        message->src = TOS_LOCAL_ADDRESS;
        if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &buf))
            return SUCCESS;
        pending = FALSE;
    }
    return FAIL;
}
```

- Refuses to accept command if buffer is still full or network refuses to accept send command
- User component provide structured msg storage
Send done event

```c
event result_t IntOutput.sendDone(TOS_MsgPtr msg,
                                  result_t success)
{
    if (pending && msg == &buf) {
        pending = FALSE;
        signal IntOutput.outputComplete(success);
    }
    return SUCCESS;
}
```

- Send done event fans out to all potential senders
- Originator determined by match
  - free buffer on success, retry or fail on failure
- Others use the event to schedule pending communication

Receive Event

```c
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
    IntMsg *message = (IntMsg *)m->data;
    call IntOutput.output(message->val);
    return m;
}
```

- Active message automatically dispatched to associated handler
  - knows the format, no run-time parsing
  - performs action on message event
- Must return free buffer to the system
  - typically the incoming buffer if processing complete
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

A Complete Application
Composition

- 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
- Interface methods are the external namespace of the component

Timer Component

- StdControl
- Timer
- Clock

Components

- Modules
  - provide code that implements one or more interfaces and internal behavior
- Configurations
  - link together components to yield new component
- Interface
  - logically related set of commands and events

StdControl.nc

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

Clock.nc

```plaintext
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```
Example top level configuration

classification SenseToRfm {
// this module does not provide any interface
}
implementation{
components Main, SenseToInt, IntToRfm, ClockC, Photo as Sensor;
Main.StdControl -> SenseToInt;
Main.StdControl -> IntToRfm;
SenseToInt.Clock -> ClockC;
SenseToInt.ADC -> Sensor;
SenseToInt.ADCClock -> Sensor;
SenseToInt.IntOutput -> IntToRfm;
}

IntToRfm Module

includes IntMsg;

module IntToRfmM {
uses {
    interface StdControl as SubControl;
    interface SendMsg as Send;
}
provides {
    interface IntOutput;
    interface StdControl;
}
implementation{
    bool pending;
    struct TOS_Msg data;
    command result_t StdControl.init() {
        pending = FALSE;
        return call SubControl.init();
    }
Given the framework, what’s the system?

- Core Subsystems
  - Simple Display (LEDS)
  - Identity
  - Timer
  - Bus interfaces (i2c, SPI, UART, 1-wire)
  - Data Acquisition
  - Link-level Communication
  - Power management
  - Non-volatile storage
- Higher Level Subsystems
  - Network-level communication
    - Broadcast, Multihop Routing
  - Time Synchronization, Ranging, Localization
  - Network Programming
  - Neighborhood Management
  - Catalog, Config, Query Processing, Virtual Machine

Timer

- Clock abstraction over HW mechanism
  - 3-4 registers of various sizes
  - Operations to set scale, phase, limit
  - One-shot, periodic
  - Signals critical event
- One physical clock must operate in minimum energy state with rest of the device power off
- Timer provides collection of logical clocks
  - Clean units
  - One-shot or periodic
  - Manage Underlying Clock Resources
    - Mapping to physical clock
    - Minimum spacing and accuracy
  - Signals non-critical event
Communication

- Parameterized Active Message Abstraction
- Multiple Media
  - Radio
    - RFM, Mica-RFM, ChipCon
  - UART
  - i2C
- Media routing based on destination address
- Side Attributes
  - Link-level ack
  - Timestamp
  - Signal-strength
- Security can be interposed (TinySec)
- Inherently Bursty and Unpredictable
  - Unless higher level protocols make it predictable

Composable Power Management

- Scheduler
  - On idle drops into preset sleep state till interrupt
    - Proc: Active: 5 mA, Idle: 2 mA, Sleep: 5 uA
    - Radio, Sensor, Co-Processors
- All Components implement Std_Control Interface
  - Power mgmt of non-processor resources
  - Start: bring subsystem to operational level, inform pwr_mgmt
  - Stop: bring subsystem to sleep, inform pwr_mgmt
- Power-management Component
  - Establishes sleep state should scheduler idle
    - Knows what is shut down, potential sleep duration
- Timer
  - Pulls system out of deep sleep
  - What happens after depends on what event handler does
- Applications compose policy
Typical Operational Mode

- Major External Events
  - Trigger collection of small processing steps (tasks and events)
  - May have interval of hard real time sampling
    - Radio
    - Sensor
  - Interleaved with moderate amount of processing in small chunks at various levels
- Periods of sleep
  - Interspersed with timer mgmt

What is Em*?

- Software environment for sensor networks built from Linux-class devices (microservers)
Em* Modularity

- Dependency DAG
- Each module (service)
  - Manages a resource and resolves contention
  - Well defined interface
  - Well scoped task
  - Encapsulate mechanism
  - Expose control of policy
  - Minimize work done by client library
- Application has same structure as "services"

Em* Robustness

- Fault isolation via multiple processes
- Active process management (EmRun)
- Auto-reconnect built into libraries
  - "Crashproofing" prevents cascading failure
- Soft state design style
  - Services periodically refresh clients
  - Avoid "diff protocols"
Comments

- Positive

- Negative