nesC

Prof. Chenyang Lu

How should network msg be handled?

- Socket/TCP/IP?
  - Too much memory for buffering and threads
    - Data buffered in network stack until application threads read it
    - Application threads blocked until data is available
  - Transmit too many bits (sequence #, ack, re-transmission)
  - Tied with multi-threaded architecture
- TinyOS solution: **active messages**

Active Message

- Every message contains the name of an **event handler**
- Sender: split-phase operation
  - Phase I
    - Declaring buffer storage in a frame
    - Naming a handler
    - Requesting Transmission; exit
  - Phase II
    - Done completion signal
- Receiver
  - Event handler is called when message is received

- ✓ No blocked or waiting threads on sender or receiver
- ✓ Behaves like any other events
- ✓ Reduce buffering

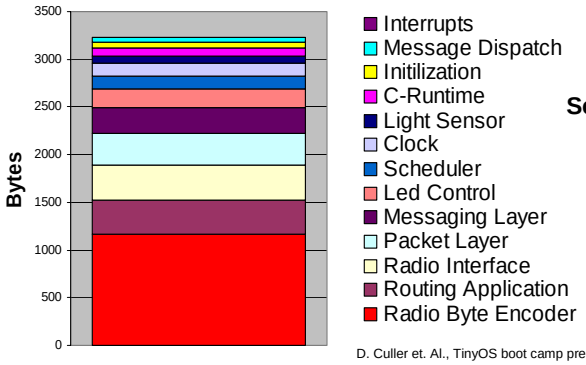
Send Message

```
char TOS_COMMAND(INT_TO_RFM_OUTPUT)(int val){
        int_to_led_msg* message = (int_to_led_msg*)VAR(msg).data;
        if (!VAR(pending)) {
          message->val = val;
          if (TOS_COMMAND(INT_TO_RFM_SUB_SEND_MSG)(TOS_MSG_BCAST,
                          AM_MSG(INT_READING), &VAR(msg))) {
            VAR(pending) = 1;
            return 1;
          }
        }
        return 0;
}
```

access appln msg buffer
cast to defined format
application specific ready check
build msg
request transmission
destination identifier
get handler identifier
msg buffer
mark busy

Space Breakdown…

Code size for ad hoc networking application



Scheduler: 144 Bytes code
Totals: 3430 Bytes code
226 Bytes data

D. Culler et. Al., TinyOS boot camp presentation, Feb 2001

Power Breakdown…

	Active	Idle	Sleep
CPU	5 mA	2 mA	5 µA
Radio	7 mA (TX)	4.5 mA (RX)	5 µA
EE-Prom	3 mA	0	0
LED's	4 mA	0	0
Photo Diode	200 µA	0	0
Temperature	200 µA	0	0

Panasonic CR2354 560 mAh

- Lithium Battery runs for 35 hours at peak load and years at minimum load!
  - That's three orders of magnitude difference!
- A one byte transmission uses the same energy as approx 11000 cycles of computation.

Time Breakdown…

Components	Packet reception work breakdown	CPU Utilization	Energy (nj/Bit)
AM	0.05%	0.20%	0.33
Packet	1.12%	0.51%	7.58
Ratio handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

- 50 cycle thread overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

Advantages

- Small memory footprint
  - Only needed OS components are complied/loaded
  - Non-preemptable FIFO task scheduling
- Power efficient
  - Sleep whenever task queue is empty
- Efficient modularity
  - Function call (event, command) interface between components
- Concurrency-intensive operations
  - Event/command + tasks
  - Efficient interrupt/event handling (function calls, no user/kernel boundary)

Lack of Real-Time Support

- FIFO, non-preemptive task scheduling
  - Urgent task may wait for non-urgent ones

Solution of Impala/ZebraNet

- Timetable for periodic operations
- Prioritize events

Impala: Scheduling

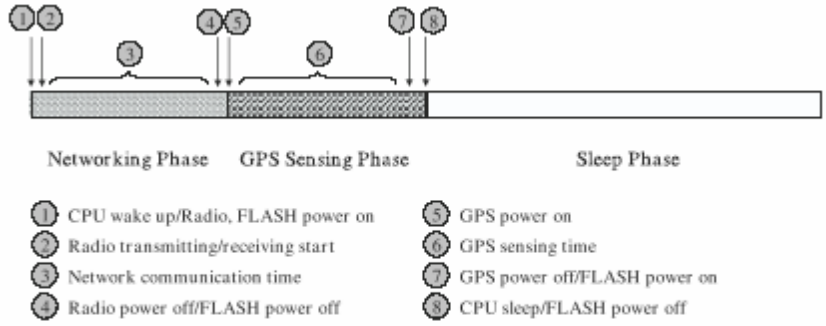


Figure 5: Timeline schedule of Impala regular operations.

Impala: Events

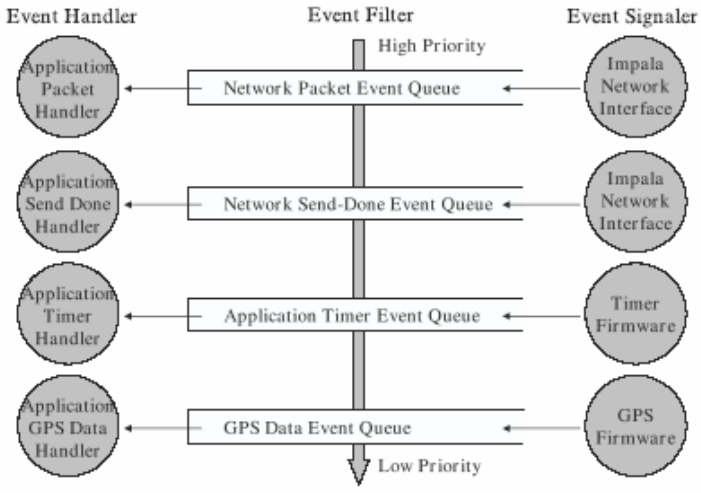


Figure 6: Impala event handling model.

What's missing?

- Support for
  - different workload
  - varying workload
  - predictability
- Leverage on real-time scheduling techniques
  - Static scheduling + schedulability analysis
  - Dynamic scheduling + overload protection
- **Topic for an interesting project!**

nesC

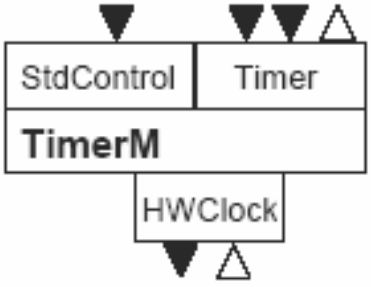
Programming Networked Embedded Systems

Principles

- Support TinyOS components and event/command interfaces
- Static language
  - no malloc, no function pointers
  - Call graph and variable access are known at compile time
- Whole-program analysis at compile time
  - Detect race conditions
  - Cross-component optimization: function inlining, eliminate unreachable code…

nesC Application

- Implementation
  - module: C behavior
  - configuration: select and wire
- Interfaces
  - provides interface
  - requires interface




```
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock as Clk;
} ...
```

Interface

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

interface Send {
    command result_t send(TOS_Msg *msg, uint16_t length);
    event result_t sendDone(TOS_Msg *msg, result_t success);
}

interface ADC {
    command result_t getData();
    event result_t dataReady(uint16_t data);
}
```

Bidirectional interface supports split-phase operation

Module

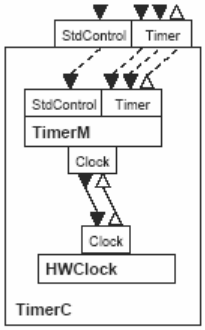
```
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}

implementation {
    uint16_t sensorReading;
    command result_t StdControl.init()
    {
        return call Timer.start(TIMER_REPEAT, 1000);
    }
    event result_t Timer.fired()
    {
        call ADC.getData();
        return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... send message with data in it ...
        return SUCCESS;
    }
    ...
}
```

Configuration

```
configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}

implementation {
    components TimerM, HWClock;
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
    TimerM.Clock -> HWClock.Clock;
}
```



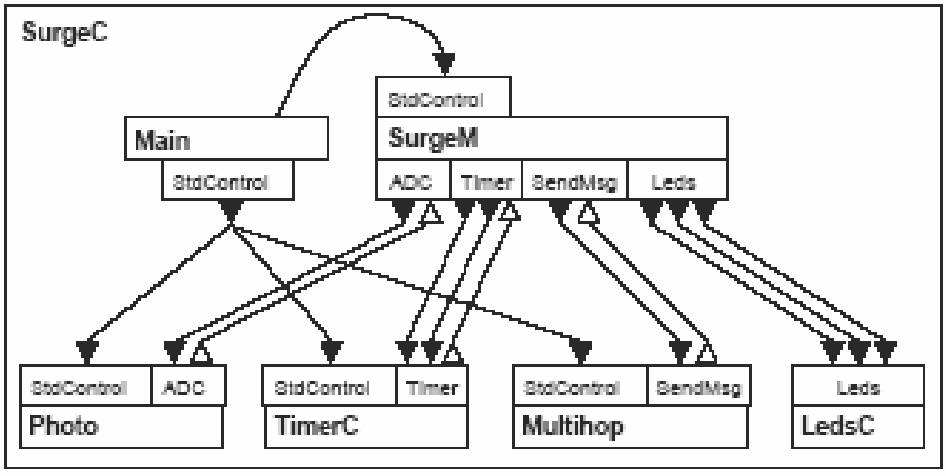

```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

Figure 5: TinyOS's timer service: the `TimerC` configuration.

Surge



Race Conditions

Example: Race Conditions

```
module SurgeM { ... }
implementation {
bool busy;
uint16_t sensorReading;

event result_t Timer.fired() {
   if (!busy)
        busy = TRUE;
        call ADC.getData();
   }
   return SUCCESS;
}
```

Asynchronous Code

Example: Race Conditions

In a command handler:

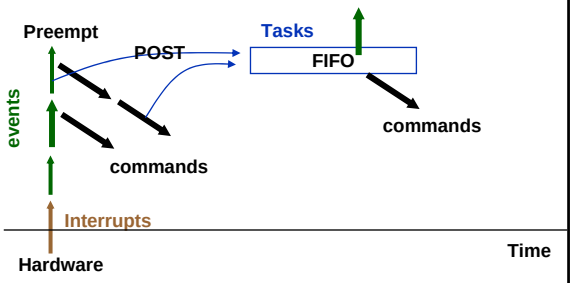
```
/* Contains a race: */
if (state == IDLE) {
   state = SENDING;
   count++;
   // send a packet
}
```


Concurrency in TinyOS

- Asynchronous code (AC): code that is reachable from at least one interrupt handler
- Synchronous code (SC): code that is only reachable from tasks
- Key properties
  - Any update to shared state (variable) from AC is a potential race condition
  - Any update to shared state from SC that is also updated from AC is a potential race condition

TinyOS Two-level Scheduling

- Two priorities
  - Event/command
  - Tasks
- Event/command can preempt task
- Tasks cannot preempt another task or event/command



Concurrency in TinyOS

- Asynchronous code (AC): code that is reachable from at least one interrupt handler
  - Event/command
- Synchronous code (SC): code that is only reachable from tasks
- Key properties
  - Any update to shared state (variable) from AC is a potential race condition
  - Any update to shared state from SC that is also updated from AC is a potential race condition

Solution

- **Race-Free Invariant**: Any update to shared state is either not a potential race condition (SC only), or occurs within an atomic section.
- Compiler check: identifies all shared states and return errors if the above invariant is violated
- Fix code
  - Make the access to all shared states with potential race conditions atomic
  - Move access to SC

Atomic Sections

```
atomic {
    <Statement list>
}
```

- Disable interrupt when atomic code is being executed
  - Alternative: semaphores based on test-set operations
- But cannot disable interrupt for long! → restrictions
  - No loops
  - No commands/events
  - Calls OK, but callee must meet restrictions too

```
module SurgeM { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;

    event result_t Timer.fired() {
        bool localBusy;
        atomic {
          localBusy = busy;
          busy = TRUE;
        }
        if (!localBusy)
          call ADC.getData();
        return SUCCESS;
    }

    task void sendData() { // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof adcPacket.data);
        return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
```

disable interrupt

enable interrupt

test-and-set

Example 2

```
/* Contains a race: */
if (state == IDLE) {
    state = SENDING;
    count++;
    // send a packet
}
```

```
/* Fixed version: */
uint8_t oldState;
atomic {
    oldState = state;
    if (state == IDLE) {
        state = SENDING;
    }
}
if (oldState == IDLE) {
    count++;
    // send a packet
}
```

Results

Results

- Tested on full TinyOS tree, plus applications
  - 186 modules (121 modules, 65 configurations)
  - 20-69 modules/app, 35 average
  - 17 tasks, 75 events on average (per app)
    - Lots of concurrency!
- Found 156 races: 103 real (!)
  - About 6 per 1000 lines of code
  - 53 false positives
- Fixing races:
  - Add atomic sections
  - Post tasks (move code to task context)

Optimization: inlining

App	Code size		Code reduction	Data size	CPU reduction
	inlined	*noninlined*			
Surge	14794	16984	12%	1188	15%
Maté	25040	27458	9%	1710	34%
TinyDB	64910	71724	10%	2894	30%

- Inlining reduces code size AND improves performance!

Issues: Programming Model

- No dynamic memory allocation
  - How to size buffer when amount of data varies?
  - **Bound memory footprint**
    - **Prevent run-time corruption**
    - **Allow offline footprint analysis** (not done)
- Restriction: no "long-running" code in
  - command/event handlers
  - atomic sections
- Push errors to applications
  - Burden application programmers
  - **Allow application-specific optimization**
    - **Ex., which message needs retransmission?**
- No kernel/user protection
  - Application can corrupt an entire system

Reminder

- Project
  - You should (at least) have a team now!
    - If not, email **cs537s@cse** ASAP
  - Discuss your ideas with me
- Critiques
  - **No** critique is due on Tuesday
  - This **is** a critique (on MAC) due on Sunday
- Presentations
  - Go over your slides with me **one week before** your presentation

Proposal Outline

1. Team
2. Motivate the problem
3. Define the problem
   - Assumptions, requirements, goals and non-goals
4. Related work
   - What have been done before?
   - How does your solution compare to them?
5. Approach (optional)
   - Intuition and rationale: Why is it a good idea?
   - Sketch of your approach and design.
6. Experimental plan
   - What hardware (e.g., sensor boards) or simulation tools are you going to use?
   - What experiments are you going to run?
   - What criteria and metrics are you going to use to evaluate your solutions?
7. Milestones (with dates)
8. References

Proposal Requirements

- 4-5 pages, 10 point, double column, single space.
- **Email me** your proposal by **midnight, 9/27 (Monday)**.

- See project Web page (updated today) for outline and requirements.