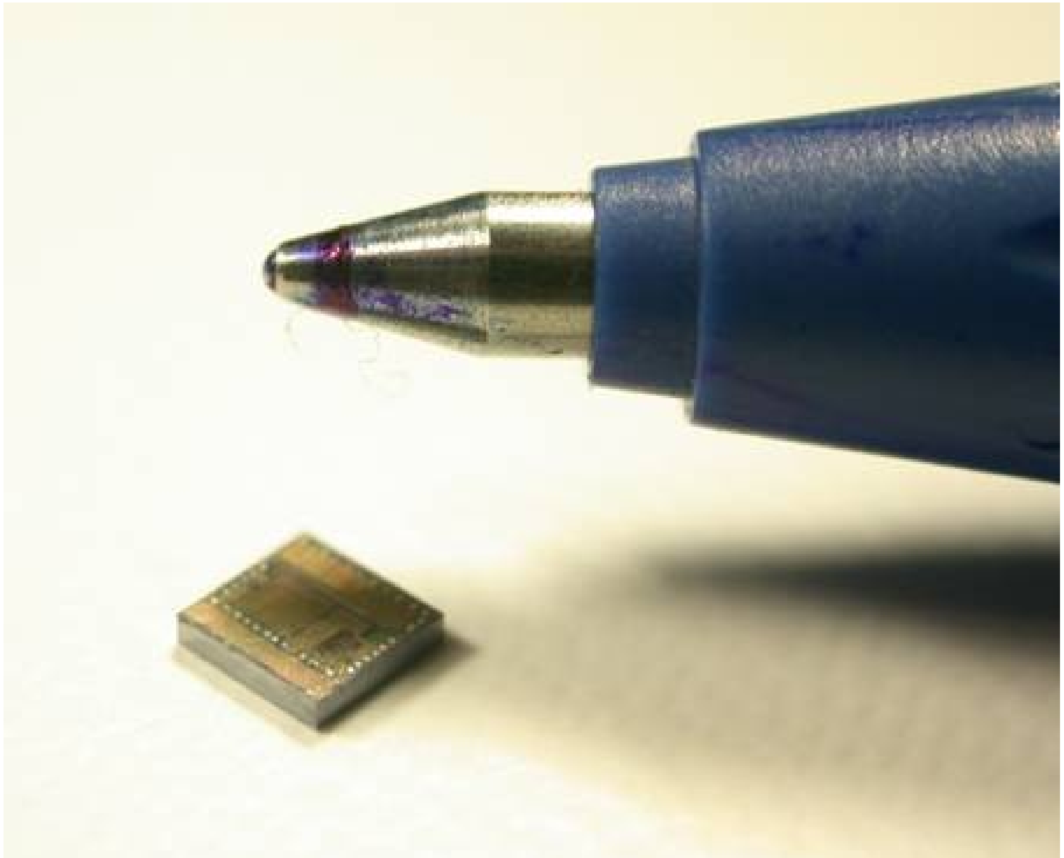
TinyOS and nesC



Outline

- Wireless sensor networks and TinyOS
- Networked embedded system C (nesC)
  - Components
  - Interfaces
  - Concurrency model
  - Tool chain
- Issues / conclusion

Wireless Sensor Networks

- Vision: ubiquitous computing
- Extreme dynamics
- Interact with environment using sensors and radio
- Immense scale
- Limited access
- Small, cheap, low-power systems

Concepts in Sensor Networks

- In-network processing and data aggregation
  - Radio activity 1000 times as expensive as processing
- Duty-cycling: different modes of operation
  - Power down unused hardware
- Systems run a single application
- Applications are deeply tied to hardware
  - Require customized and optimized OS

Challenges

- Limited resources: energy consumption dominates
- Concurrency: driven by interaction with environment
- Soft real-time requirements
- Reliability: reduce run-time errors, e.g. races
- High diversity of platforms
- No well-defined software/hardware boundary

TinyOS

- Component-based architecture
  - Reusable system components: ADC, Timer, Radio
- Tasks and event-based concurrency
  - No user-space or context switching supported by hardware
  - Tasks run to completion only preempted by interrupts
- All long-latency operations are *split-phase*
  - Operation request and completion are separate functions

Introducing nesC

- A “holistic” approach to networked embedded systems
- Supports and reflects TinyOS's design
- Extends a subset of C
- A static language
  - All resources known at compile-time
  - Call-graph fully known at compile-time

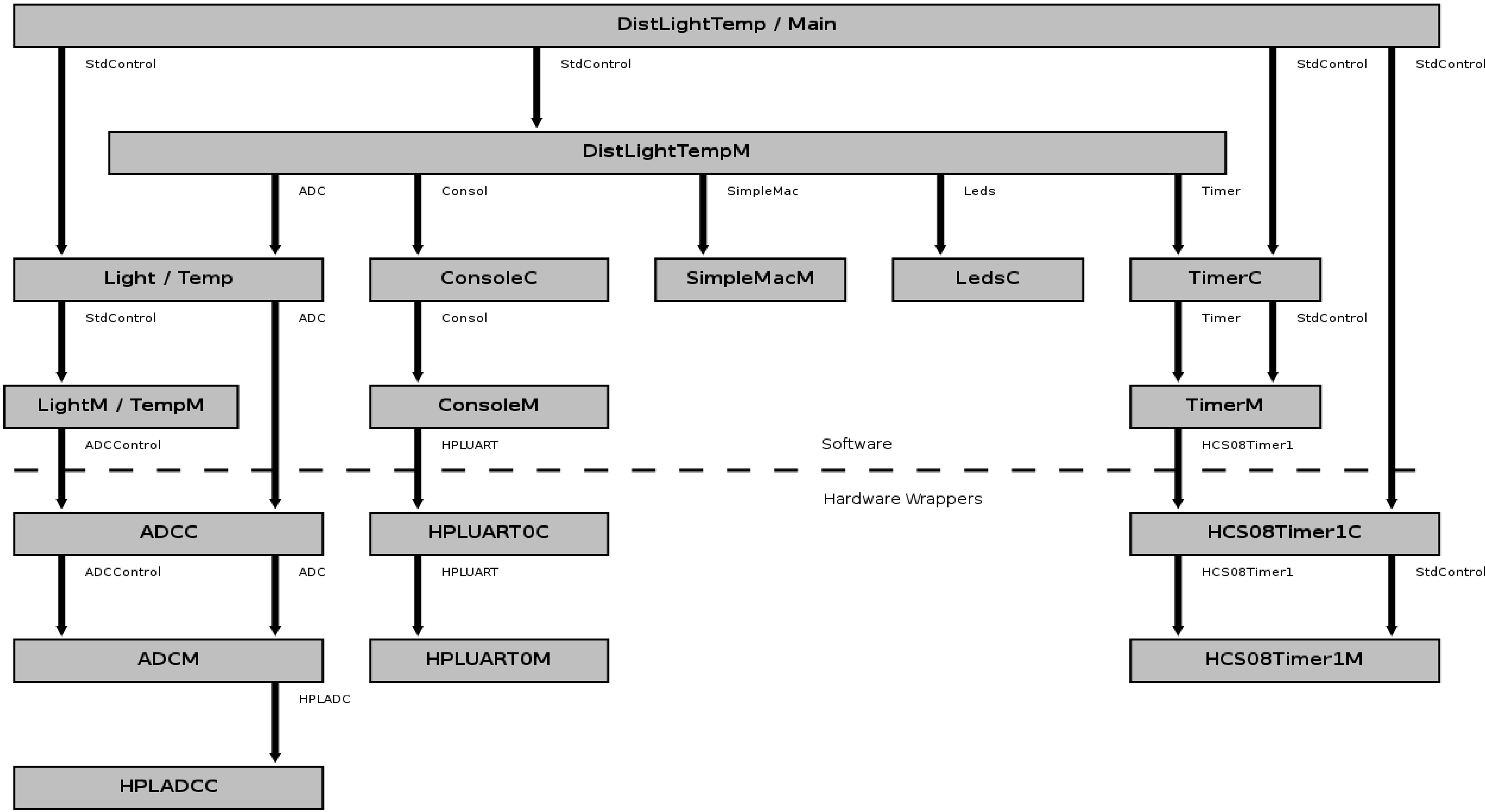
Design Decisions for nesC

- Components
- Bidirectional interfaces
- Simple expressive concurrency model
- Whole-program analysis

Components

- Challenge: platform diversity, flexible SW/HW boundary, applications deeply tied to hardware
- Encourages modular design
- Restrict access to private data
- Allow underlying implementations to be replaced easily
- Can abstract HW using thin wrappers
- Allow specialization of applications to hardware

Example Component Graph



Module Components

- Modules implement application code
- Modules have private state
  - Sharing of data among components is discouraged
- Convention:
  - Module names end with 'M', e.g. BlinkM
  - Module variables start with 'm_', e.g. m_timers

Configuration Components

- Configurations wire other components together
- All applications have a top-level configuration
- A component interface may be wired zero or more times
  - Used for StdControl to implement power management
- Convention:
  - Configuration names end with 'C', e.g. TimerC
    (unless it is the top-level configuration ;-)

Modules and Configurations

```
/* BlinkM.nc */
module BlinkM {
    provides interface StdControl as Control;
    uses interface Timer;
    uses interface Leds;
} implementation {
    command result_t Control.init() {
        call Leds.init();
        return SUCCESS;
    }
    command result_t Control.start() { /* ... */ }
    command result_t Control.stop() { /* ... */ }

    event result_t Timer.fired() {
        call Leds.redToggle();
        return SUCCESS;
    }
}
```

```
/* Blink.nc */
configuration Blink {
} implementation {
    /* Declare used components. */
    components Main, BlinkM, SingleTimer, LedsC;

    /* Wire components together. */
    Main.StdControl -> SingleTimer.StdControl;
    Main.StdControl -> BlinkM.StdControl;
    BlinkM.Timer        -> SingleTimer.Timer;
    BlinkM.Leds         -> LedsC;
}
```

Bidirectional Interfaces

- Challenge: flexible SW/HW boundary and concurrency
- Support split-phase operations
- Commands: call *down* the component graph
  - Implemented by provider
- Events: call *up* the component graph
  - Implemented by user

Interfaces

```
/* Timer.nc */
includes Timer; /* Include C types from Timer.h */

interface Timer {
	command result_t start(char type, uint32_t interval);
	command result_t stop();
	event result_t fired();
}

/* SyncAlarm.nc */
interface SyncAlarm<Precision_t> {
	command result_t armCountdown(Precision_t timeout);
	command result_t armAlarmClock(Precision_t time);
	command result_t stop();
	event result_t alarm();
}
```

Parameterized Interfaces

```
module TimerM {
    provides interface Timer[uint8_t id];
} implementation {
    /* ... */
    Timer_t m_timers[NUM_TIMERS];
    command result_t Timer.isSet[uint8_t timer]() {
        return m_timers[timer].isset;
    }
    task void timerCheck() {
        uint8_t timer;
        for (timer = 0; timer < NUM_TIMERS; timer++)
            if (m_timers[timer].fired)
                signal Timer.fired[timer]();
    }
    /* ... */
}

configuration  MyApp { /* ... */ }
implementation {
    components MyAppM, TimerC, /* ... */;
    MyAppM.SampleTimer -> TimerC.Timer[unique("Timer")];
}
```

Concurrency Model

- Challenge: extreme dynamics and soft real-time requirements
- Cooperative scheduling
- Light-weight tasks
- Split-phase operations: non-blocking requests
- Built-in atomic sections
  - Limited crossing of module boundaries

Sources of Concurrency

- Tasks
  - Deferred computation
  - Run *sequential* and *to completion*
  - Do not preempt
- Events
  - Run to completion, and may preempt tasks and events
  - Origin: hardware interrupts or split-phase completion

Tasks and Events

```
module LightM {
    /* ... */
} implementation {
    uint16_t light_data
    task void processLightdata() {
        uint16_t local_light_data;
        atomic local_light_data = light_data;
        /* Process light data. */
        if (!done)
            post anotherTask()
    }
    async event result_t Light.dataReady(uint16_t data) {
        atomic lightData = data;
        post processLightData();
        return SUCCESS;
    }
    event result_t SensorTimer.fired() {
        return call Light.getData();
    }
}
```

Whole-Program Analysis

- Compilation can examine complete call-graph
  - Remove dead-code
  - Eliminate costly module boundary crossings
  - Inline small functions
- Back-end C compiler can optimize whole program
  - Perform cross component optimizations
  - Constant propagation, common subexpression elimination
- Allows detection of race conditions

Synchronous and Asynchronous

- **Asynchronous code (AC)**:
  - Code reachable from at least one interrupt handler
  - Events signaled directly or indirectly by hardware interrupts
- **Synchronous code (SC):**
  - "Everything else ..."
  - Primarily tasks

Detecting Race Conditions

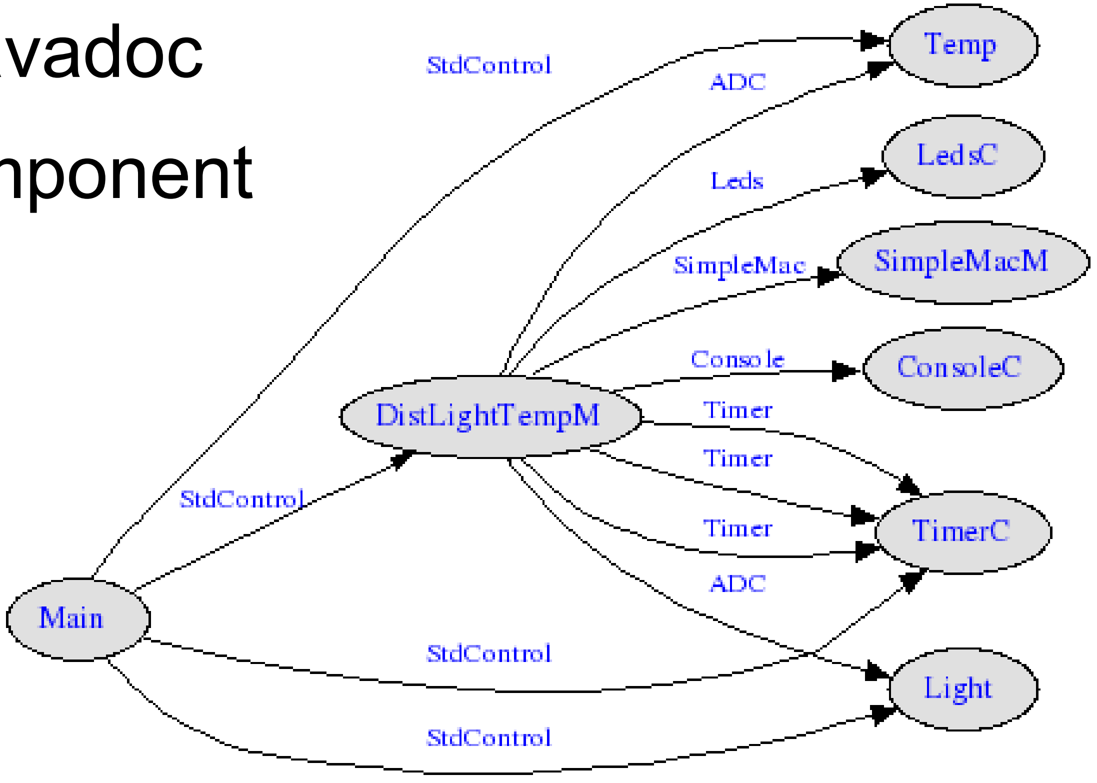
- Invariant: *SC is atomic with respect to other SC*
- Two claims about updates for AC/AC and SC/AC:
  - *Any update to shared state from AC is a potential race condition*
  - *Any update to shared state from SC that is also updated from AC is a potential race condition*
- Race-free invariant enforced at compile time:
  - *Updates to shared state is either SC only or in atomic section*

Dealing with Race Conditions

- Use atomic sections to update shared state
  - **atomic** { shared_var = 1; }
- Convert code with updates to shared state to tasks
- Mark false positive with norace qualifier
  - **norace** uint8_t variable;

The nesC Toolchain: nesdoc

- Generate code documentation using simple tags
- Same concept as javadoc
- Can generate a component graph using dot



The nesC Toolchain: nescc

- The nesC compiler for TinyOS
- Implemented as an extension to GCC
- Called via TinyOS wrapper ncc
- Input: path to TinyOS code + nesC files
  - Platform code implements API of macros and functions in C
- Output: C code or object code (if supported by GCC)

The nesC Toolchain: ncg and mig

- Allows integration with Java code
- Typical use: interact with network through base station
- ncg - extract constants from nesC files
  - Generates class that contains constants
- mig - message interface generator for nesC
  - Generates class that encodes and decodes messages

Issues for nesC

- Problem for data shared across components
  - False positives for buffer swapping
- Problem for data shared between split-phase operations
  - Event can potentially fire if other components access HW
- Some TinyOS idioms are not well expressed
  - Parameterized interfaces each with private state

Issues for Applications

- Focus early on modeling it as a state-machine
- Design duty-cycling from the start
  - Affect the state-machine so hard to add later
- Abstracting functionality into components
  - Makes it harder to access shared state: encapsulate shared state in a separate module
- Configuring TinyOS for the application needs
  - By default there can only be 8 posted tasks

Conclusions for nesC

- Bidirectional interfaces fit the TinyOS model
- Components are a good abstraction
- Concurrency model meets requirements in applications
- The restrictions in nesC introduce practical problems
- Not limited to the domain of embedded systems