CSEP567– tonight:

TinyOS

TinyOS

- Open-source development environment
- Simple (and tiny) operating system – TinyOS
- Programming language and model – nesC
- Set of services

- Principal elements
  - Scheduler/event model of concurrency
  - Software components for efficient modularity
  - Software encapsulation for resources of sensor networks

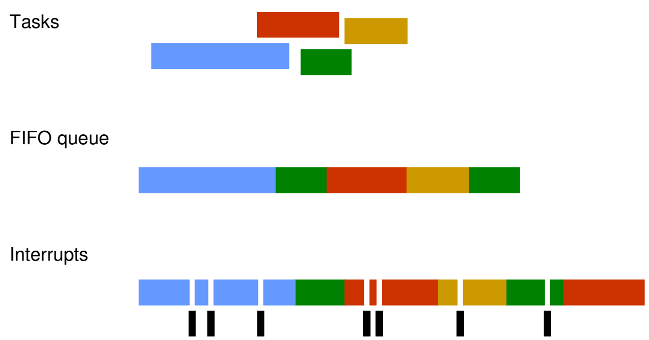
TinyOS Design Goals

- Support networked embedded systems
  - Asleep most of the time, but remain vigilant to stimuli
  - Bursts of events and operations
- Support UCB mote hardware
  - Power, sensing, computation, communication
  - Easy to port to evolving platforms
- Support technological advances
  - Keep scaling down
  - Smaller, cheaper, lower power

TinyOS Kernel Design

- Two-level scheduling structure
  - Events
    - Small amount of processing to be done in a timely manner
    - E.g. timer, ADC interrupts
    - Can interrupt longer running tasks
  - Tasks
    - Not time critical
    - Larger amount of processing
    - E.g. computing the average of a set of readings in an array
    - Run to completion with respect to other tasks
      - Only need a single stack

TinyOS Concurrency Model



Two-level of concurrency: tasks and interrupts

TinyOS Concurrency Model (cont'd)

- Tasks
  - FIFO queue
  - Placed on queue by:
    - Application
    - Other tasks
    - Self-queued
    - Interrupt service routine
  - Run-to-completion
    - No other tasks can run until completed
    - Interruptable, but any new tasks go to end of queue
- Interrupts
  - Stop running task
  - Post new tasks to queue

TinyOS Concurrency Model (cont'd)

- Two-levels of concurrency
  - Possible conflicts between interrupts and tasks
- Atomic statements

```
atomic {
…
}
```

- Asynchronous service routines (as opposed to synchronous tasks)

```
async result_t interface_name.cmd_or_event_name {
…
}
```

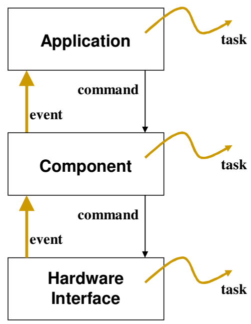
- Race conditions detected by compiler
  - Can generated false positives – `norace` keyword to stop warnings, but be careful

TinyOS Programming Model

- Separation of construction and composition
  - Programs are built out of <u>components</u>
- Specification of component behavior in terms of a set of <u>interfaces</u>
  - Components specify interfaces they <u>use</u> and <u>provide</u>
- Components are statically <u>wired</u> to each other via their interfaces
  - This increases runtime efficiency by enabling compiler optimizations
- Finite-state-machine-like specifications
- <u>Thread of control</u> passes into a component through its interfaces to another component

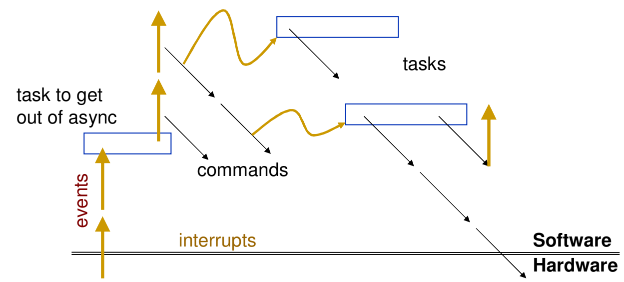
TinyOS Basic Constructs

- Commands
  - Cause action to be initiated
- Events
  - Notify action has occurred
  - Generated by external interrupts
  - Call back to provide results from previous command
- Tasks
  - Background computation
  - Not time critical



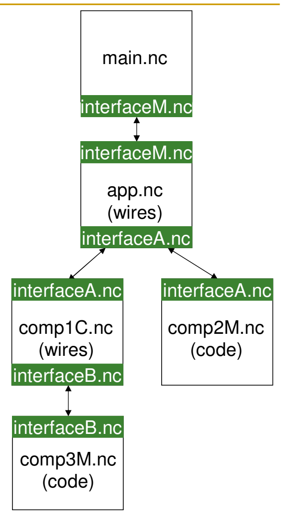
Flow of Events and Commands

- Fountain of events leading to commands and tasks (which in turn issue may issue other commands that may cause other events, …)



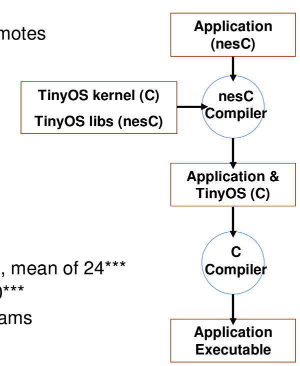
TinyOS File Types

- **Interfaces** (xxx.nc)
  - Specifies functionality to outside world
  - what commands can be called
  - what events need handling
- **Module** (xxxM.nc)
  - Code implementation
  - Code for **Interface** functions
- **Configuration** (xxxC.nc)
  - Wiring of components
  - When top level app, drop C from filename xxx.nc



The nesC Language

- nesC: networks of embedded sensors C
- Compiler for applications that run on UCB motes
  - Built on top of avg-gcc
  - nesC uses the filename extension ".nc“
- Static Language
  - No dynamic memory (no malloc)
  - No function pointers
  - No heap
- Influenced by Java
- Includes task FIFO scheduler
- Designed to foster code reuse
- Modules per application range from 8 to 67, mean of 24***
- Average lines of code in a module only 120***
- Advantages of eliminating monolithic programs
  - Code can be reused more easily
  - Number of errors should decrease



***The NesC Language: A Holistic Approach to Network of Embedded Systems. David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

Commands

- Commands are issued with "call"

```
call Timer.start(TIMER_REPEAT, 1000);
```

- Cause action to be initiated
- Bounded amount of work
  - Does not block
- Act similarly to a function call
  - Execution of a command is immediate

Events

- Events are called with "signal"

```
signal ByteComm.txByteReady(SUCCESS);
```

- Used to notify a component an action has occurred
- Lowest-level events triggered by hardware interrupts
- Bounded amount of work
  - Do not block
- Act similarly to a function call
  - Execution of a event is immediate

Tasks

- Tasks are queued with "post"

```
post radioEncodeThread();
```

- Used for longer running operations
- Pre-empted by events
  - Initiated by interrupts
- Tasks run to completion
- Not pre-empted by other tasks
- Example tasks
  - High level – calculate aggregate of sensor readings
  - Low level – encode radio packet for transmission, calculate CRC

Components

- Two types of components in nesC:
  - **Module**
  - **Configuration**
- A component *provides* and *uses* **Interfaces**

Module

- Provides application code
  - Contains C-like code
- Must implement the ‘provides’ interfaces
  - Implement the “commands” it provides
  - Make sure to actually “signal”
- Must implement the ‘uses’ interfaces
  - Implement the “events” that need to be handled
  - “call” commands as needed

Configuration

- A **configuration** is a **component** that "wires" other **components** together.
- **Configurations** are used to assemble other **components** together
- Connects **interfaces** used by **components** to **interfaces** provided by others.

Interfaces

- Bi-directional multi-function interaction channel between two components
- Allows a single interface to represent a complex event
  - E.g., a registration of some event, followed by a callback
  - Critical for non-blocking operation
- “provides” interfaces
  - Represent the functionality that the component provides to its user
  - Service “commands” – implemented command functions
  - Issue “events” – signal to user for passing data or signalling done
- “uses” interfaces
  - Represent the functionality that the component needs from a provider
  - Service “events” – implement event handling
  - Issue “commands” – ask provider to do something

Application

- Consists of one or more components, wired together to form a runnable program
- Single top-level configuration that specifies the set of components in the application and how they connect to one another
- Connection (wire) to main component to start execution
  - Must implement init, start, and stop commands

Components/Wiring

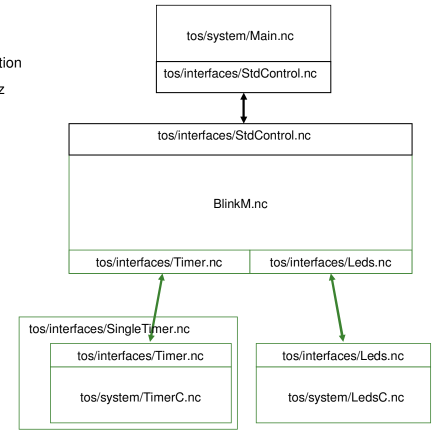
- Directed wire (an arrow: '->') connects components
  - Only 2 components at a time – point-to-point
  - Connection is across compatible interfaces
  - 'A <- B' is equivalent to 'B -> A'
- [component using interface] -> [component providing interface]
  - [interface] -> [implementation]
- '=' can be used to wire a component directly to the top-level object's interfaces
  - Typically used in a configuration file to use a sub-component directly
- Unused system components excluded from compilation

Blink Application

What the executable does:

1. Main initializes and starts the application
2. BlinkM initializes ClockC's rate at 1Hz
3. ClockC continuously signals BlinkM at a rate of 1 Hz
4. BlinkM commands LedsC red led to toggle each time it receives a signal from ClockC

Note: The StdControl interface is similar to state machines (init, start, stop); used extensively throughout TinyOS apps & libs



Blink.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC.Leds;
}
```


StdControl.nc

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

BlinkM.nc

```
BlinkM.nc module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
```

```
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```


SingleTimer.nc (should have been SingleTimerC.nc)

- Parameterized interfaces
  - allows a component to provide multiple instances of an interface that are parameterized by a value
- Timer implements one level of indirection to actual timer functions
  - Timer module supports many interfaces
  - This module simply creates one unique timer interface and wires it up
  - By wiring Timer to a separate instance of the Timer interface provided by TimerC, each component can effectively get its own "private" timer
  - Uses a compile-time constant function `unique()` to ensure index is unique

```
configuration SingleTimer {
  provides interface Timer;
  provides interface StdControl;
}
implementation {
  components TimerC;

  Timer = TimerC.Timer[unique("Timer")];
  StdControl = TimerC.StdControl;
}
```


Blink.nc without SingleTimer

```
configuration Blink {
}
implementation {
  components Main, BlinkM, TimerC, LedsC;
  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> TimerC.Timer[unique("Timer")];
  BlinkM.Leds -> LedsC.Leds;
}
```


Timer.nc

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

TimerC.nc

- Implementation of multiple timer interfaces to a single shared timer
- Each interface is named
- Each interface connects to one other module

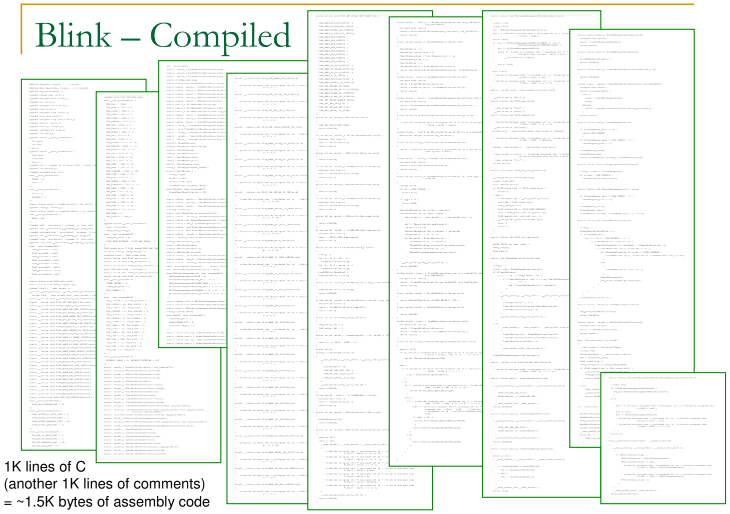
Leds.nc (partial)

```
interface Leds {

/**
 * Initialize the LEDs; among other things, initialization turns them all off.
 */
  async command result_t init();

/**
 * Turn the red LED on.
 */
  async command result_t redOn();

/**
 * Turn the red LED off.
 */
  async command result_t redOff();

/**
 * Toggle the red LED. If it was on, turn it off. If it was off,
 * turn it on.
 */
  async command result_t redToggle();

...
```


LedsC.nc (partial)

```
module LedsC {
  provides interface Leds;
}
implementation
{
  uint8_t ledsOn;

  enum {
    RED_BIT = 1,
    GREEN_BIT = 2,
    YELLOW_BIT = 4
  };

  async command result_t Leds.init() {
    atomic {
      ledsOn = 0;
      dbg(DBG_BOOT, "LEDS: initialized.\n");
      TOSH_MAKE_RED_LED_OUTPUT();
      TOSH_MAKE_YELLOW_LED_OUTPUT();
      TOSH_MAKE_GREEN_LED_OUTPUT();
      TOSH_SET_RED_LED_PIN();
      TOSH_SET_YELLOW_LED_PIN();
      TOSH_SET_GREEN_LED_PIN();
    }
    return SUCCESS;
  }

async command result_t Leds.redOn() {
    dbg(DBG_LED, "LEDS: Red on.\n");
    atomic {
      TOSH_CLR_RED_LED_PIN();
      ledsOn |= RED_BIT;
    }
    return SUCCESS;
}

async command result_t Leds.redOff() {
    dbg(DBG_LED, "LEDS: Red off.\n");
     atomic {
       TOSH_SET_RED_LED_PIN();
       ledsOn &= ~RED_BIT;
     }
     return SUCCESS;
}

async command result_t Leds.redToggle() {
    result_t rval;
    atomic {
      if (ledsOn & RED_BIT)
          rval = call Leds.redOff();
      else
          rval = call Leds.redOn();
    }
    return rval;
}

...
```


Blink – Compiled

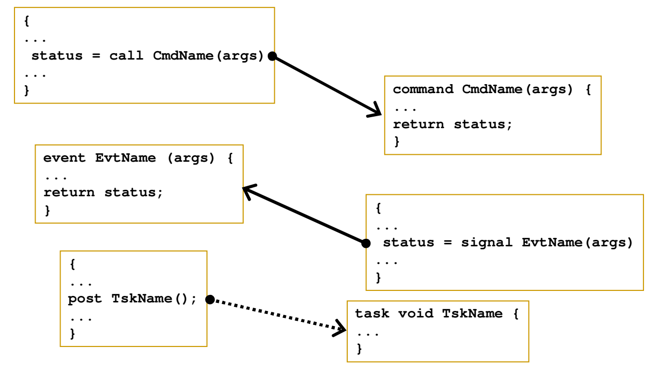


1K lines of C
(another 1K lines of comments)
= ~1.5K bytes of assembly code

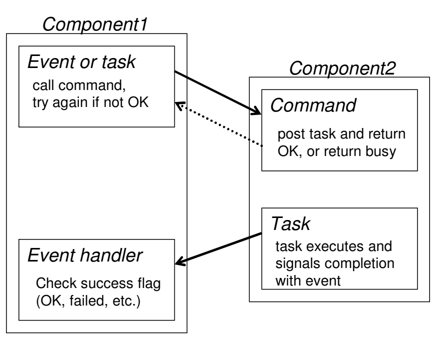
Concurrency Model

- Asynchronous Code (AC)
  - Any code that is reachable from an interrupt handler
- Synchronous Code (SC)
  - Any code that is ONLY reachable from a task
  - Boot sequence
- Potential race conditions
  - Asynchronous Code and Synchronous Code
  - Asynchronous Code and Asynchronous Code
  - Non-preemption eliminates data races among tasks
- nesC reports potential data races to the programmer at compile time (new with version 1.1)
- Use `atomic` statement when needed
- `async` keyword is used to declare asynchronous code to compiler

Commands, Events, and Tasks



Split Phase Operations



Phase I
- call command with parameters
- command either posts task to do real work or signals busy and to try again later

Phase II
- task completes and uses event (with return parameters) to signal completion
- event handler checks for success (may cause re-issue of command if failed)

Naming Convention

- Use mixed case with the first letter of word capitalized
  - Interfaces (Xxx.nc)
  - Components
    - Configuration (XxxC.nc)
    - Module (XxxM.nc)
  - Application – top level component (Xxx.nc)
- Commands, Events, & Tasks
  - First letter lowercase
  - Task names should start with the word “task”, commands with “cmd”, events with “evt” or “event”
  - If a command/event pair form a split-phase operation, event name should be same as command name with the suffix “Done” or “Complete”
  - Commands with “TOSH_” prefix indicate that they touch hardware directly
- Variables – first letter lowercase, caps on first letter of all sub-words
- Constants – all caps

Interfaces can fan-out and fan-in

- nesC allows interfaces to *fan-out* to and *fan-in* from multiple components
- One “provides” can be connected to many “uses” and vice versa
- Wiring fans-out, fan-in is done by a *combine* function that merges results

```
implementation {
  components Main, Counter, IntToLeds, TimerC;

  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> TimerC.StdControl;
```

Fan-out by wiring

```
result_t ok1, ok2, ok3;
….
ok1 = call UARTControl.init();
ok2 = call RadioControl.init();
ok3 = call Leds.init();
….
return rcombine3(ok1, ok2, ok3);
```

Fan-in using rcombine
- rcombine is just a simple logical AND for most cases

Example

```
configuration CntToLeds {
}
implementation {
  components Main, Counter, IntToLeds, TimerC;

  Main.StdControl    -> IntToLeds.StdControl;
  Main.StdControl    -> Counter.StdControl;
  Main.StdControl    -> TimerC.StdControl;
  Counter.Timer      -> TimerC.Timer[unique("Timer")];
  Counter.IntOutput -> IntToLeds.IntOutput;
}
```