TinyOS – an operating system for sensor nets

- Embedded operating systems
  - How do they differ from desktop operating systems?
- Event-based programming model
  - How is concurrency handled?
  - How are resource conflicts managed?
- Programming in TinyOS
  - What new language constructs are useful?

Embedded Operating Systems

- Features of all operating systems
  - Abstraction of system resources
  - Managing of system resources
  - Concurrency model
  - Launch applications
- Desktop operating systems
  - General-purpose – all features may be needed
  - Large-scale resources – memory, disk, file systems
- Embedded operating systems
  - Application-specific – just use features you need, save memory
  - Small-scale resources – sensors, communication ports

System Resources on Motes

- Timers
- Sensors
- Serial port
- Radio communications
- Memory
- Power management

Abstraction of System Resources

- Create virtual components
  - E.g., multiple timers from one timer
- Allow them to be shared by multiple threads of execution
  - E.g., two applications that want to share radio communication
- Device drivers provide interface for resource
  - Encapsulate frequently used functions
  - Save device state (if any)
  - Manage interrupt handling

Very simple device driver

- Turn LED on/off
- Parameters:
  - port pin
- API:
  - on(port_pin) - specifies the port pin (e.g., port D pin 3)
  - off(port_pin)
- Interactions:
  - only if other devices want to use the same port

Simple device driver

- Turning an LED on and off at a fixed rate
- Parameters:
  - port pin
  - rate at which to blink LED
- API:
  - on(port_pin, rate)
    - specifies the port pin (e.g., port D pin 3)
    - specifies the rate to use in setting up the timer (what scale?)
  - off(port_pin)
- Internal state and functions:
  - keep track of state (on or off for a particular pin) of each pin
  - interrupt service routine to handle timer interrupt

Interesting interactions

- What if other devices also need to use timer (e.g., PWM device)?
  - timer interrupts now need to be handled differently depending on which device's alarm is going off
- Benefits of special-purpose output compare peripheral
  - output compare pins used exclusively for one device
  - output compare has a separate interrupt handling routine
- What if we don't have output compare capability or run out of output compare units?

Sharing timers

- Create a new device driver for the timer unit
  - Allow other devices to ask for timer services
  - Manage timer independently so that it can service multiple requests
- Parameters:
  - Time to wait, address to call when timer reaches that value
- API:
  - set_timer(time_to_wait, call_back_address)
    - Set call_back_address to correspond to time+time_to_wait
    - Compute next alarm to sound and set timer
    - Update in interrupt service routine for next alarm
- Internal state and functions:
  - How many alarms can the driver keep track of?
  - How are they organized? FIFO? priority queue?

Concurrency

- Multiple programs interleaved as if parallel
- Each program requests access to devices/services
  - e.g., timers, serial ports, etc.
- Exclusive or concurrent access to devices
  - allow only one program at a time to access a device (e.g., serial port)
  - arbitrate multiple accesses (e.g., timer)
- State and arbitration needed
  - keep track of state of devices and concurrent programs using resource
  - arbitrate their accesses (order, fairness, exclusivity)
  - monitors/locks (supported by primitive operations in ISA - test-and-set)
- Interrupts
  - disabling may effect timing of programs
  - keeping enabled may cause unwanted interactions

Handling concurrency

- Traditional operating system
  - multiple threads or processes
  - file system
  - virtual memory and paging
  - input/output (buffering between CPU, memory, and I/O devices)
  - interrupt handling (mostly with I/O devices)
  - resource allocation and arbitration
  - command interface (execution of programs)
- Embedded operating system
  - lightweight threads
  - input/output
  - interrupt handling
  - real-time guarantees

Embedded operating systems

- Lightweight threads
  - basic locks
  - fast context-switches
- Input/output
  - API for talking to devices
  - buffering
- Interrupt handling (with I/O devices and UI)
  - translate interrupts into events to be handled by user code
  - trigger new tasks to run (reactive)
- Real-time issues
  - guarantee task is called at a certain rate
  - guarantee an interrupt will be handled within a certain time
  - priority or deadline driven scheduling of tasks

Examples

- Palm OS
  - US Robotics Palm Pilot
  - Motorola microcontrollers (68328 – Dragonball, migrating to Xscale)
  - simple OS for PDAs
  - only supports single threads
- Pocket PC
  - PDA operating system
  - spin-off of Windows NT
  - portable to a wide variety of processors (e.g., Xscale)
  - full-featured OS modularized to only include features as needed
- Wind River Systems VxWorks
  - one of the most popular embedded OS kernels
  - highly portable to an even wider variety of processors (tiny to huge)
  - modularized even further than the ones above (basic system under 50K)

embedded operating systems typically reside in ROM (flash)

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 History – www.tinyos.net

- Motivation – create Unix analog (circa 1969)
  - Uniform programming language: C
  - Uniform device abstractions
  - Open source: grow with different developers/needs
  - Support creation of many tools
- Created at UC Berkeley
  - 1st version written by Jason Hill in 2000
  - Large part of development moved to Intel Research Berkeley in 2001 – www.intel-research.net/berkeley
  - Smart Dust, Inc. founded in 2002
- Large deployments
  - Great Duck Island (GDI) – http://www.greatduckisland.net/
  - Center for Embedded Network Sensing (CENS) – http://www.cens.ucla.edu/

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 Design Options

- Can't use existing RTOS's
  - Microkernel architecture
    - VxWorks, PocketPC, PalmOS
  - Execution similar to desktop systems
    - PDA's, cell phones, embedded PC's
  - More than a order of magnitude too heavyweight and slow
  - Energy hogs

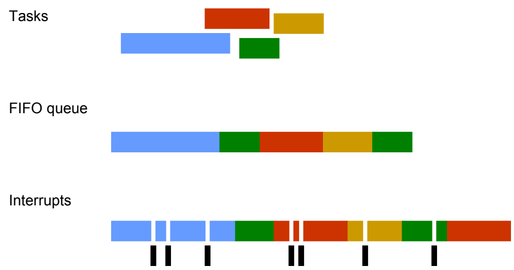
TinyOS Design Conclusion

- Similar to building networking interfaces
  - Data driven execution
  - Manage large # of concurrent data flows
  - Manage large # of outstanding events
- Add: managing application data processing
- Conclusion: need a multi-threading engine
  - Extremely efficient
  - Extremely simple

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 {
…
}
```

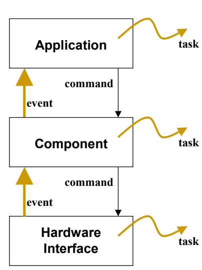
- Race conditions detected by compiler
  - Can generated false positives – `norace` keyword to stop warnings, but be careful

TinyOS Programming Model

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

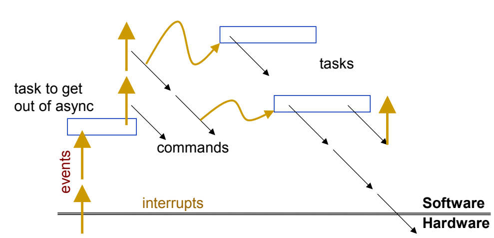
TinyOS Basic Constructs

- 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



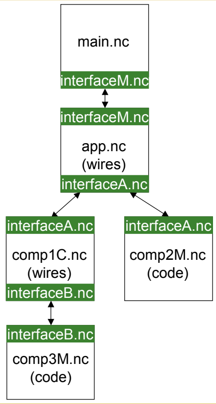
Flow of Events and Commands

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



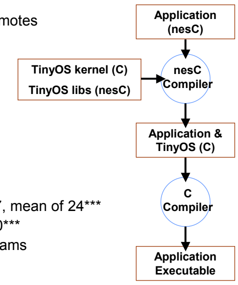
TinyOS File Types

- **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

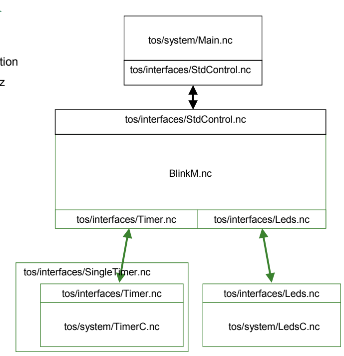
- Directed wire (an arrow: ‘->’) connects components
  - Only 2 components at a time – point-to-point
  - Connection is across compatible interfaces
  - ‘A <- B’ is equivalent to ‘B -> A’
- [component using interface] -> [component providing interface]
  - [interface] -> [implementation]
- ‘=’ can be 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;
  }
```

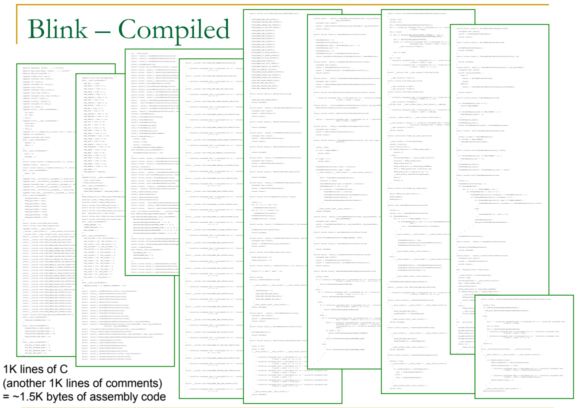
```
async command result_t Leds.redOn() {
   dbg(DBG_LED, "LEDS: Red on.\n");
   atomic {
     TOSH_CLR_RED_LED_PIN();
     ledsOn |= RED_BIT;
   }
   return SUCCESS;
 }

 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

Blink – Compiled – a small piece

```
static inline result_t LedsC$Leds$redToggle(void)
{
  result_t rval;
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
    {
      if (LedsC$ledsOn & LedsC$RED_BIT) { rval = LedsC$Leds$redOff();
      } else { rval = LedsC$Leds$redOn();
      }
    }
    __nesc_atomic_end(__nesc_atomic); }
  return rval;
}
inline static result_t BlinkM$Leds$redToggle(void)
{
  unsigned char result;
  result = LedsC$Leds$redToggle();
  return result;
}
static inline result_t BlinkM$Timer$fired(void)
{
  BlinkM$Leds$redToggle();
  return SUCCESS;
}
```

```
static inline result_t LedsC$Leds$redOn(void)
{
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
    {
      TOSH_CLR_RED_LED_PIN();
      LedsC$ledsOn |= LedsC$RED_BIT;
    }
    __nesc_atomic_end(__nesc_atomic); }
  return SUCCESS;
}

static inline result_t LedsC$Leds$redOff(void)
{
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
    {
      TOSH_SET_RED_LED_PIN();
      LedsC$ledsOn &= ~LedsC$RED_BIT;
    }
    __nesc_atomic_end(__nesc_atomic); }
  return SUCCESS;
}
```


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

```
{
...
 status = call CmdName(args)
...
}
```

```
command CmdName(args) {
...
return status;
}
```

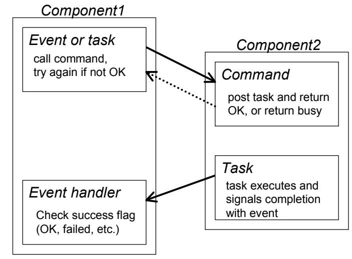
```
event EvtName)(args) {
...
return status;
}
```

```
{
...
 status = signal EvtName(args)
...
}
```

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

```
task void TskName {
...
}
```


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

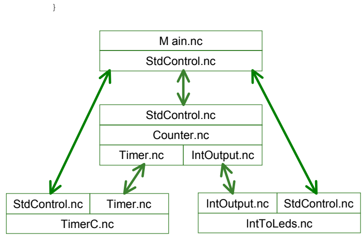
```
result_t ok1, ok2, ok3;
….
ok1 = call UARTControl.init();
ok2 = call RadioControl.init();
ok3 = call Leds.init();
….
return rcombine3(ok1, ok2, ok3);
```

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;
}
```



Exercise

- Which of the following goes inside the module you are implementing if we assume you are the "user" of the interface?
  - NOTE: Not all of these choices are exposed through an interface. Assume those that are not exposed are implemented in your module.
  - post taskA( );
  - call commandB(args);
  - signal eventC(args);
  - taskA implementation
  - commandB implementation
  - eventC implementation

Sense Application

SenseM.nc

```
module SenseM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface ADC;
    interface StdControl as ADCControl;
    interface Leds;
  }
}                                   cont'd
```

Sense.nc

```
configuration Sense {
}
implementation
{
  components Main, SenseM, LedsC, TimerC, DemoSensorC as Sensor;

  Main.StdControl -> Sensor.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> SenseM.StdControl;

  SenseM.ADC -> Sensor.ADC;
  SenseM.ADCControl -> Sensor.StdControl;
  SenseM.Leds -> LedsC.Leds;
  SenseM.Timer -> TimerC.Timer[unique("Timer")];
}
```

DemoSensorC.nc

```
configuration DemoSensorC
{
  provides interface ADC;
  provides interface StdControl;
}
implementation
{
  components Photo as Sensor;

  StdControl = Sensor;
  ADC = Sensor;
}
```


SenseM.nc

```
cont'd

implementation {
/* Module scoped method.  Displays the lowest 3 bits to the LEDs, with RED
   being the most signficant and YELLOW being the least significant */

result_t display(uint16_t value) {
    if (value &1) call Leds.yellowOn(); else call Leds.yellowOff();
    if (value &2) call Leds.greenOn();  else call Leds.greenOff();
    if (value &4) call Leds.redOn();    else call Leds.redOff();
    return SUCCESS;
  }

command result_t StdControl.init()  { return call Leds.init(); }
command result_t StdControl.start() { return call Timer.start(TIMER_REPEAT, 500); }
command result_t StdControl.stop()  { return call Timer.stop(); }

event result_t Timer.fired() { return call ADC.getData(); }

async event result_t ADC.dataReady(uint16_t data) {
    display(7-((data>>7) &0x7));
    return SUCCESS;
  }
}
```


Sense Application Using Task

SenseM.nc

```
module SenseTaskM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface ADC;
    interface Leds;
  }
}                                   cont'd
```

SenseTask.nc

```
configuration SenseTask {
}
implementation
{
  components Main, SenseTaskM, LedsC, TimerC, DemoSensorC as Sensor;

  Main.StdControl -> TimerC;
  Main.StdControl -> Sensor;
  Main.StdControl -> SenseTaskM;

  SenseTaskM.Timer -> TimerC.Timer[unique("Timer")];
  SenseTaskM.ADC   -> Sensor;
  SenseTaskM.Leds  -> LedsC;
}
```

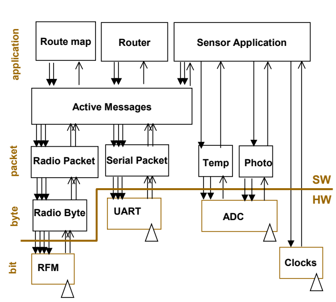

SenseTaskM.nc

```
implementation {
  enum {
    log2size = 3,       // log2 of buffer
    size=1 << log2size, // circular buffe
    sizemask=size - 1,  // bit mask
  };
  int8_t head;          // head index
  int16_t rdata[size];  // circular buffe

  inline void putdata(int16_t val)
  {
    int16_t p;
    atomic {
          p = head;
          head = (p+1) & sizemask;
          rdata[p] = val;
    }
  }
  result_t display(uint16_t value)
  {
      if (value &1) call Leds.yellowOn();
      else call Leds.yellowOff();
      if (value &2) call Leds.greenOn();
      else call Leds.greenOff();
      if (value &4) call Leds.redOn();
      else call Leds.redOff();
      return SUCCESS;
  }
```

```
  task void processData()
  {
     int16_t i, sum=0;
     atomic {
       for (i=0; i<size; i++)
          sum += (rdata[i] >> 7);
     }
     display(sum >> log2size);
  }
  command result_t StdControl.init() {
    atomic head = 0;
    return call Leds.init();
  }
  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 500);
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired() {
    return call ADC.getData();
  }
  async event result_t ADC.dataReady(uint16_t data)
  {
    putdata(data);
    post processData();
    return SUCCESS;
  }
}
```


A More Extensive Application



NOTE: This is NOT the radio stack we will be using

Tips

- Make liberal use of “grep” or “find in files”
- Look at example applications in the /apps directory
- All interfaces are in /interfaces directory
- Utilities are in /system, /lib, /platform, or /sensorboards
- Try to keep commands and events very short
  - Avoid loops, use queues and callbacks

Debugging

- Cover in more detail in later lectures
- Applications can be built to run on the PC (TOSSIM)
  - Good to debug
  - Does not perfectly simulate the hardware
- Toggling LEDs
  - Can only get so much information from 1 LED
  - Useful for indicating specific events (will LED be on long enough?):
    - Radio packet transmit/receive
    - Timer fired
    - Sensor activation