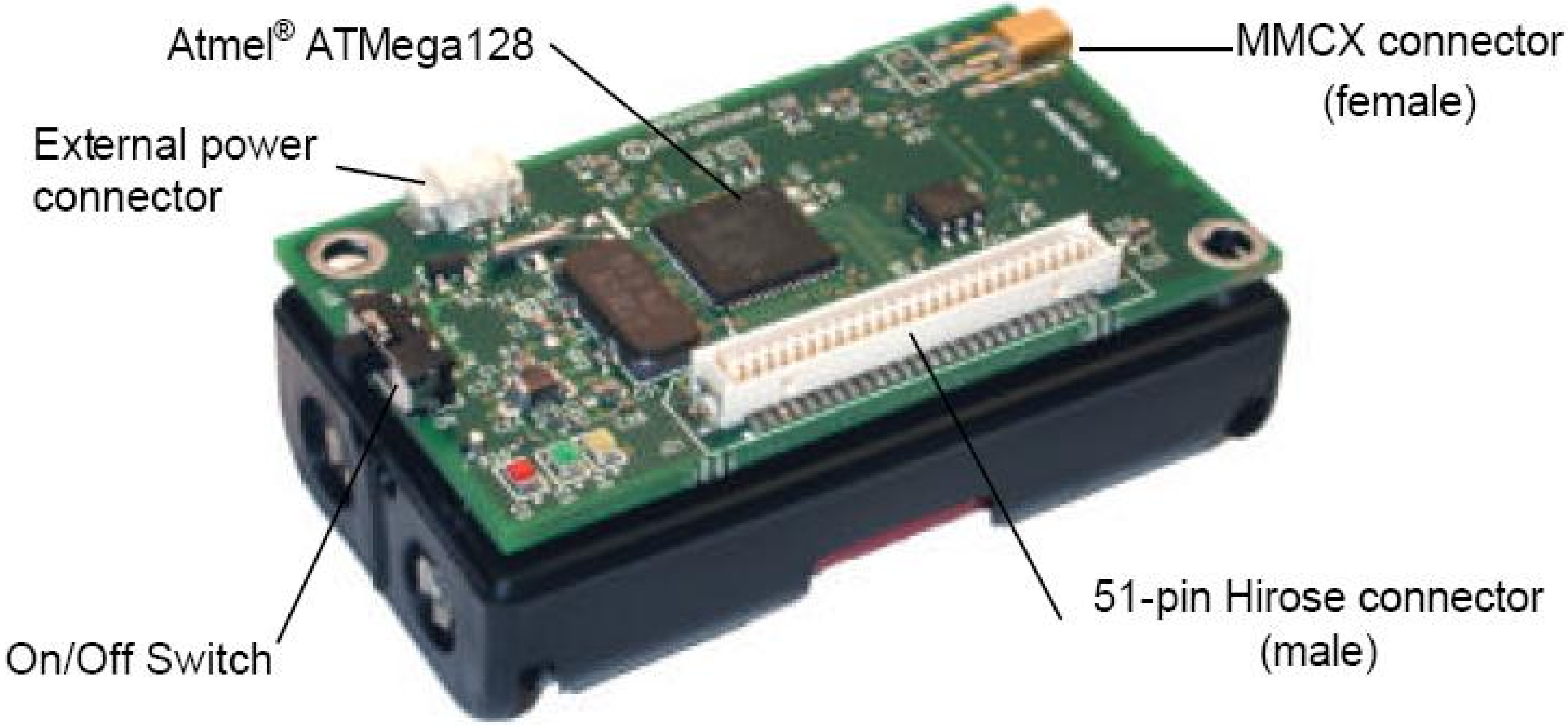
TinyOS/Motes, nesC Tutorial

Presented by Ke Liu
Dept. Of Computer Science, SUNY Binghamton

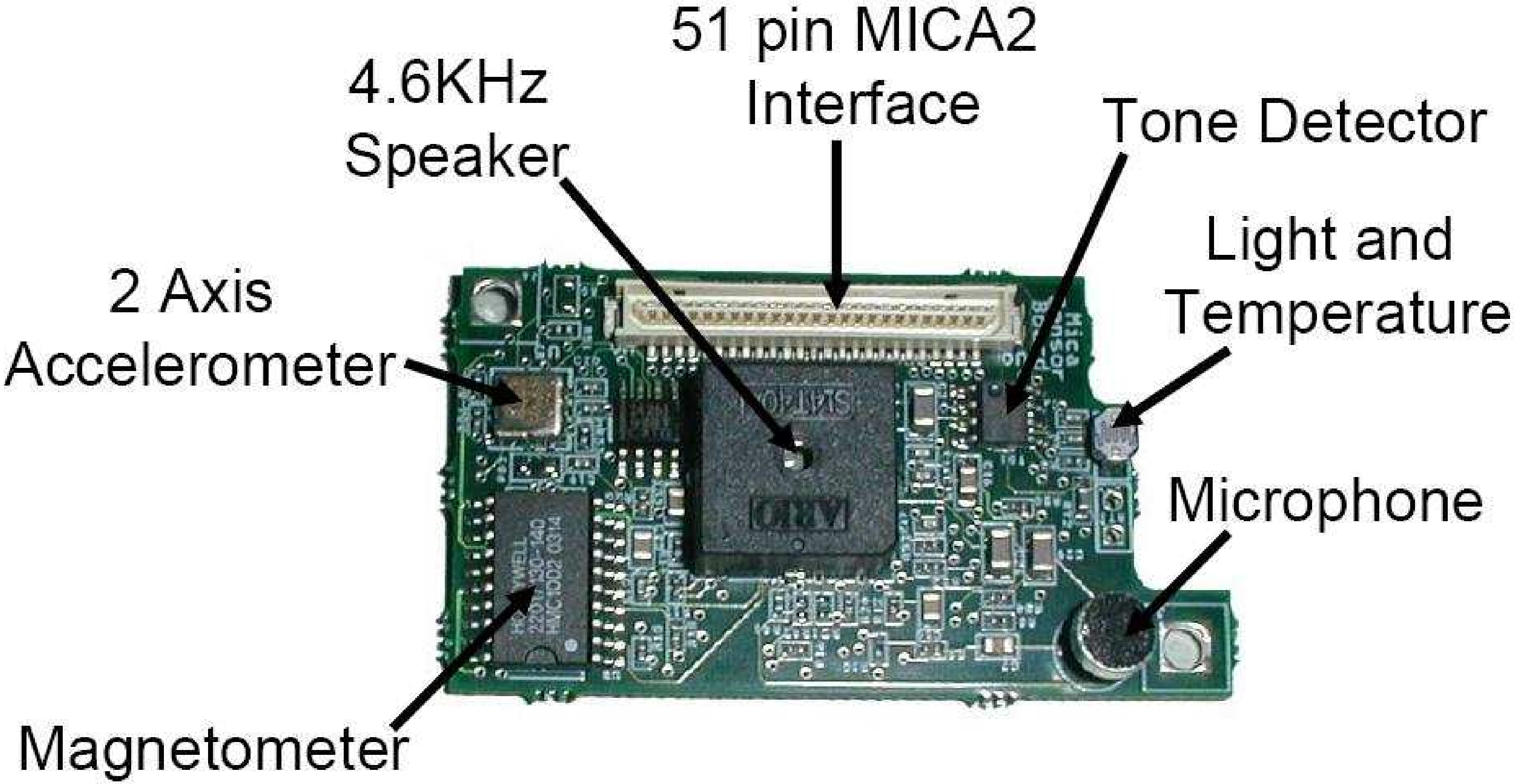
Spring, 2005

TinyOS/Motes Overview

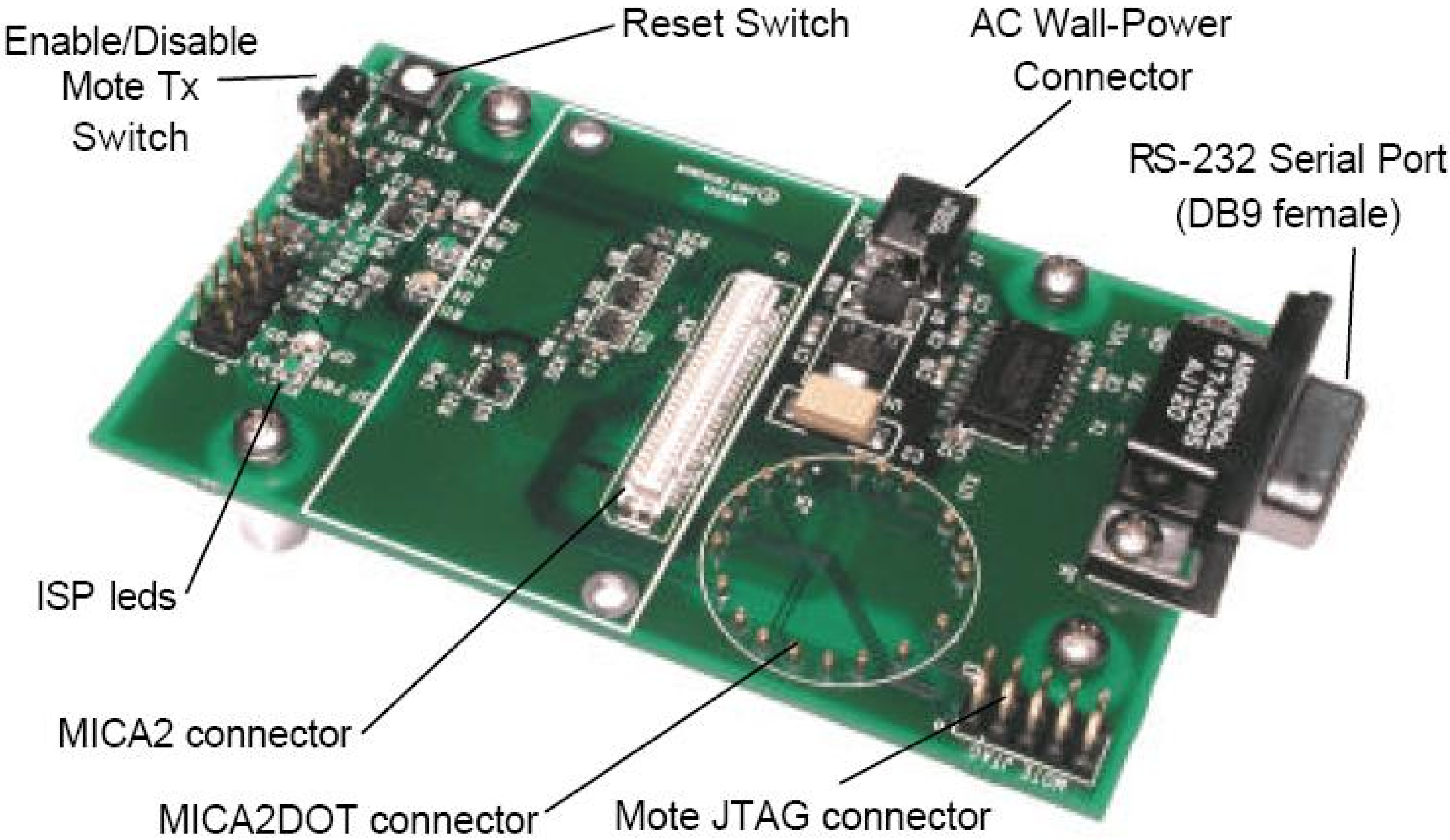
Mica2 Mote


Atmel® ATMega128
MMCX connector
(female)
External power
connector
On/Off Switch
51-pin Hirose connector
(male)

MTS310 Sensor Board



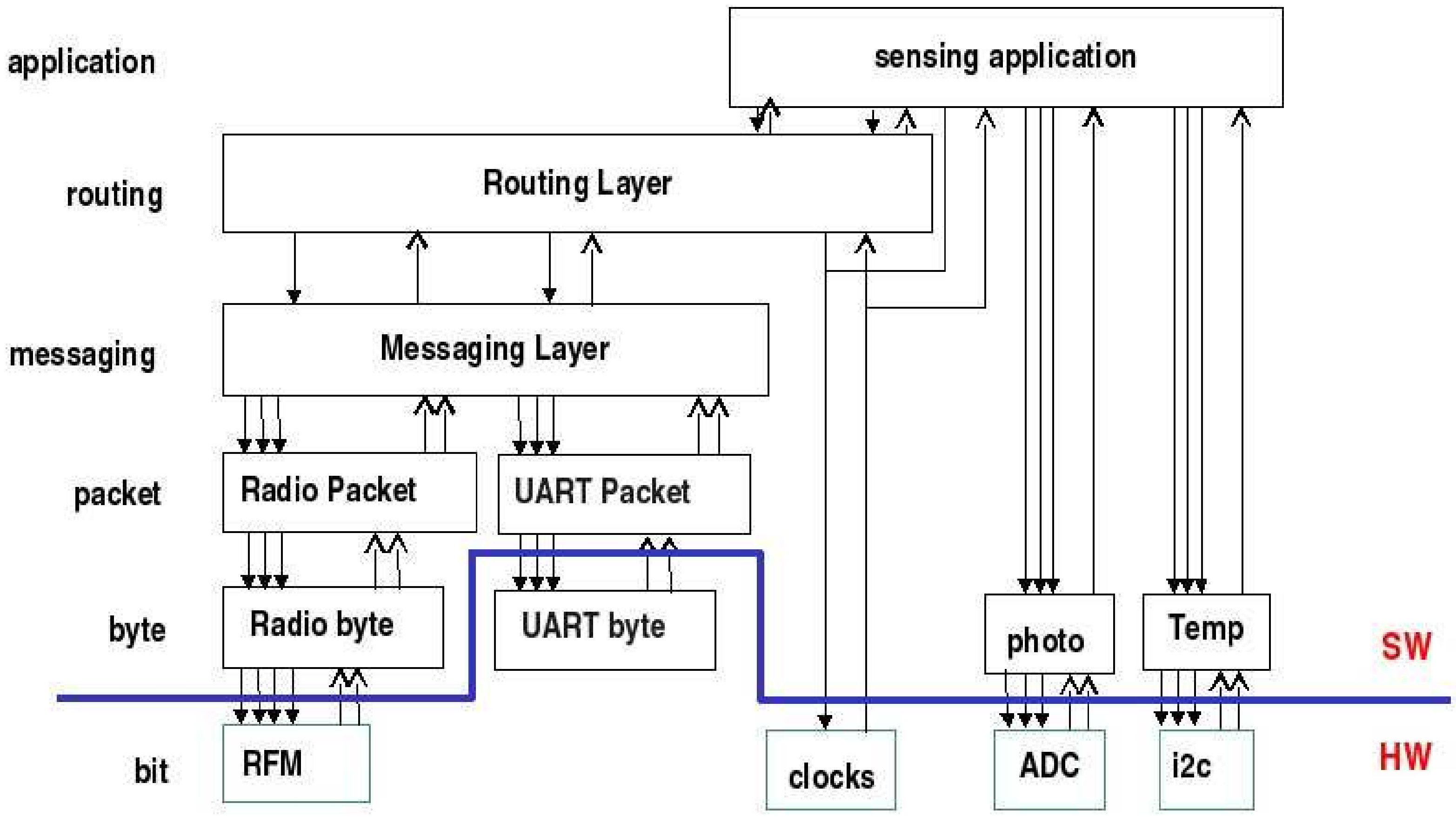
MIB510 Programming Board



What is TinyOS?

- Developed by UC Berkeley, for Berkeley Motes sensor nodes
- Operating system for Motes, Open Source development environment
- The system, library and applications written in *nesC*
- applilcation = scheduler + graph of components
- Event-driven architecture
- Single shared stack
- **NO** kernel, process/memory management

Typical Application architecture



→ stands for interface's user/provider

Components

- Programs are built out of components
- Each component is specified by an interface. A Component has:
  - Frame (internal states)
  - Tasks (data processing)
  - Interface(s) (commands/events)
- Commands and Events are function calls (later)
- Application is a wiring of multiple interfaces(components).
- The components are statically wired together based on their interfaces. (For runtime efficiency)

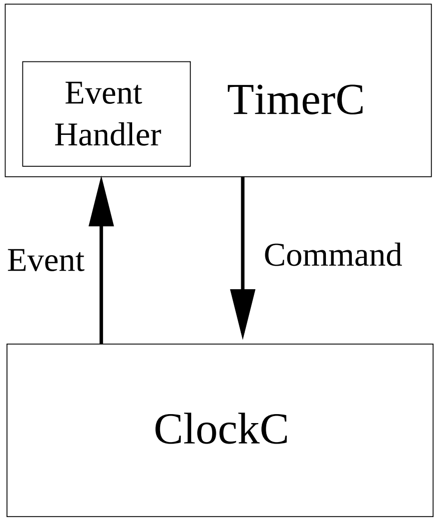
Interface: Commands/Events

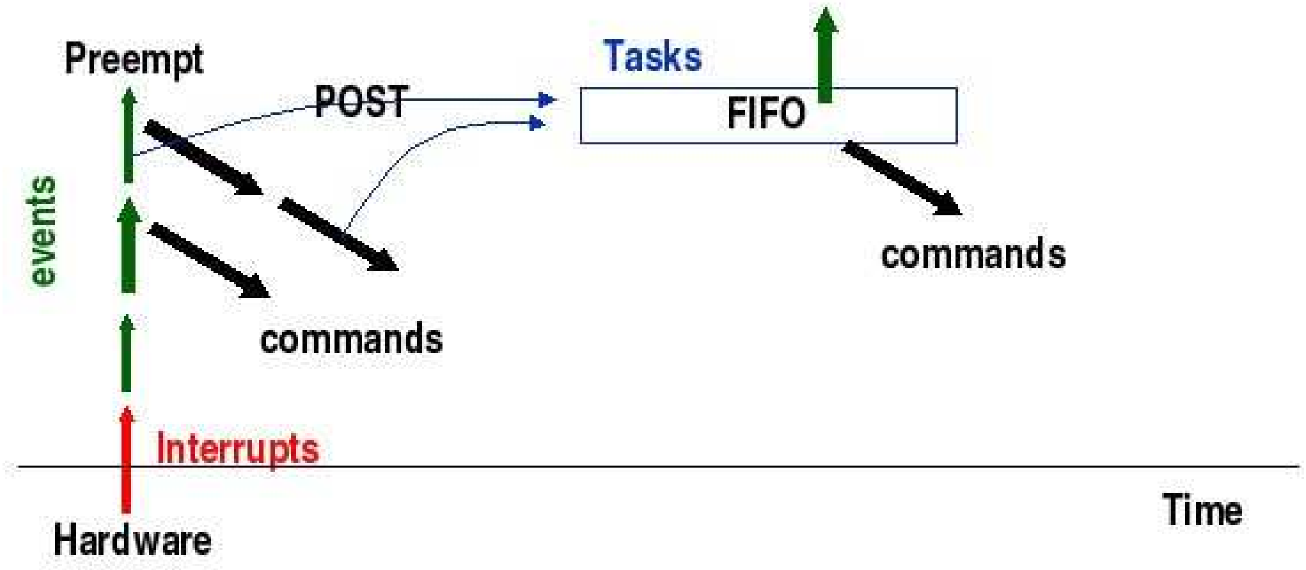
Components implement the events they use, and the commands they provide

- Commands
  - deposit request parameters into the frame
  - are non-blocking
  - need to return status $\Rightarrow$ postpone time consuming work by posting a task
  - can call lower level commands
- Events
  - can call commands, signal events, post tasks
  - can **Not** be signaled by commands
  - preempt tasks, not vice-versa
  - interrupt trigger the lowest level events
  - deposit the information into the frame

Scheduler

- two level scheduling: events and tasks
- scheduler is simple FIFO
- a task can not preempt another task
- events preempt tasks (higher priority)
- event may preempt another event ⇒ post task to make event smaller





Tasks

- FIFO scheduling
- non-preemptable by other task, preemtable by events
- perform computationally intensive work
- handling of multiple data flows:
  - a sequence of non-blocking command/event through the component graph
  - post task for computational intensive work
  - preempt the running task, to handle new data

nesC

Note: this part is mostly from the TinyOS website tutorial

The programming evironment

- Download from http://www.tinyos.net/download.html
- Installation: http://www.tinyos.net/tinyos-1.x/doc/install.html
- nesC reference manual: http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf
- Components
  - AVR package: for processor
  - nesC compbiler
  - JDK (Important):
    - Linux (Red-Hat): IBM JDK/JavaComm (not properly for Debian);
    - Windows(Cygwin): Sun JSDK
  - TinyOS distribution

Directory Structure

```
/apps
        /CntToLedsAndRfm
        /Sense
        ...
/doc
/tools
        /java
        /matlab
        ...
/tos
        /interface
        /lib
        /platform
                /mica
                /mica2
                /mica2dot
                /pc
        /sensorboard
                /micasb
        /system
        /types
```

nesC files

interface	StdControl.nc	Declares the services provided and the services used. In*tos/interface/*
module	BlinkM.nc	provides application code implementing one or more interfaces
configuration	Blink.nc	wires components, makes control-flow

Naming Conventions

Identifier	Rules for Naming	Examples
Interfaces	Verbs or Nouns the mixed case: TheMixedCase	ADC SendMsg
Components	Nouns: terminating with C: components, providing interfaces. M: Modules, implementation.	TimerC TimerM
Files	with suffix ".nc"	TimerC.nc TimerM.nc
Commands, Events and Tasks	Verbs, the mixed case: theMixedCase. Command/Event pair: suffixing the command with 'Done' or 'Complete'	sendMsg put putDone
Variables	Nouns: the mixed case: theMixedCase	bool state
Constants	all in caps, with underscores delimiting internal words	TOS_BCAST_ADDR

Simple Application: Blink

This application simply causes the red LED on the mote to turn on and off at 1Hz

- Files:
  - Blink.nc : the definition of component (top-level configuration file)
  - BlinkM.nc : the definition of Blink Module and implementation of interface
  - SingleTimer.nc : single timer component used by Blink Module.

- *Blink.nc* is used to wire the *BlinkM.nc* module to other components that the Blink application requires.

- *SingleTimer* is just an extension of the TimerC component.

Blink.nc

Components it uses: **Main, BlinkM, SingleTimer, LedsC**

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Main is a component that is executed first in a TinyOS application.

-> stands for ”binds to” or ”wired to”.

StdControl interface

StdControl is a common interface used to initialize and start TinyOS components. Every component *should* provide this interface. It is defined at *tos/interfaces/StdControl.nc*:

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

- `init()` can be called multiple times, but will never be called after either `start()` or `stop()` are called. Specifically, the valid call patterns of StdControl are init*(start|stop)*
- All three of these commands have "deep" semantics: calling `init()` on a component must make it call `init()` on all of its subcomponents

Blink.nc (Cont')

```
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
```

- These 2 lines wire the *StdControl* interface in **Main** to the *StdControl* interface in both **BlinkM** and **SingleTimer**.
- *SingleTimer.StdControl.init()* and *BlinkM.StdControl.init()* will be called by *Main.StdControl.init()*.
- The same rule applies to the start() and stop() commands.

BlinkM.nc Module file

- This is a Module called **BlinkM**
- It provides interface(s): *StdControl*
- It uses interface(s): *Timer, Leds*

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

Timer.nc Interface file

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

- *start()*: to specify the type of the timer and the interval at which the timer will expire;
  - Unit of the interval; millisecond
  - the valid types are TIMER_REPEAT and TIMER_ONE_SHOT
- the *fired()* event is signaled when the specified interval has passed
- a **bi-directional** interface:
  - interface provider must implement commands
  - commands are called by user
  - interface user must implement events
  - events are called by provider, handled by user

BlinkM.nc (Cont')

```
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;
  }
}
```

- component's specification is implemented in C code
- Each time *Timer.fired()* event is triggered, the *Leds.redToggle()* toggles the red LED.

SingleTimer.nc configuration file

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

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

- Module **SingleTimer** provides *Timer* interface but it does not implement it;
- The implementation of *Timer* interface in module **SingleTimer** is provided by `TimerC.Timer[unique("Timer")]`, which is an external specification element;
- Implicit wiring: `StdControl=TimerC` ⇔ `StdControl=TimerC.StdControl`

Compile the application

- Our hardware platform is mica2
- Go to the *tos/apps/Blink*
- `make mica2` is used to make the target executable for platform **mica2**
- `make pc` is used to make the target for **TOSSIM**, a simulator for TinyOS

Programming the Motes and Runnig Blink

- Go to the *tos/apps*, add the following lines in the *Makelocal* file:

  ```
  PFLAGS += -DCC1K_DEF_FREQ=916700000
  DEFAULT_LOCAL_GROUP=0x01
  MIB510=/dev/ttyS0
  ```

  Create it if it is not there.

- Connect the programming board to the PC (serial port);

- Connect the Mote node to the programming board;

- Turn on the switch on the Mote if you are using the battery;

- `make mica2 install.<addr>` to upload the program

Wiring

- Not only the interfaces can be wired together, commands/events also can be;
- any wired elements must be compatible;
- Wiring statements:
  - $S_1 = S_2$
    - S1 and S2 are both external, one is provided and the other is used
    - one is internal, the other is external; and both are provided or used.
  - $S_1 -> S_2$ or $S_2 <- S_1$:
    - Both are internal. One is provided and other used.
- internal specification element: from a configuration's specification
- external specification element: from a configuration's component's specification

Concurrency in TinyOS/nesC

- The execution model consists of
  - *run-to-completion tasks* that typically represent the ongoing computation
  - *interrupt handlers* that are signaled asynchronously by hardware.

- 2 types of code in *nesC*:
  - Synchronous Code (**SC**): code (functions, commands, events, tasks) that is only reachable from tasks.
  - Asynchronous Code (**AC**): code that is reachable from at least one interrupt handler.

- **Race-Free Invariant**: Any update to shared state is either SC-only or occurs in an atomic statement.
  - This would be enforced at compiling time

Concurrency (cont')

- To handle events and concurrency, nesC provides 2 tools:
  - *atomic* sections
  - *task*(s)
- Atomicity is implemented by simply **disabling/enabling** interrupts (this only takes a few cycles). Disabling interrupts for a long time can delay interrupt handling and make systems less responsive.
- If potential race condition is present and programmer knows it's not an actual race condition, can specify something as **norace**

SurgeM

```
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(){
    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;
  }
  ...
} //implementation
```

Data Race Conditions

- Tasks may be preempted by other asynchronous code
- Races are avoided by:
  - Accessing shared data exclusively within **task**(s)
  - Having all accesses within **atomic** statements
- The *nesC* compiler(ncc) reports potential data races to the programmer at compiling time
- Variables can be declared with the **norace** keyword (should be used with extreme caution)

Application Surge

- It is a simple example of a mutlihop application
- it takes light sensor readings and sends them to the base node (Node 0)
- The Multihop routing in TinyOS is shortest-path routing
- `make mica2 install.<addr>`
- Using Java tools to collect the data from node 0

TinyOS/nesC messgae

- A standard message format is used for passing information between nodes
- Messages include: Destination Address, Group ID, Message Type, Message Size and Data
-
```
#define TOSH_DATA_LENGTH 29
typedef struct TOS_Msg{
  /* The following fields are transmitted/received on the radio. */
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[TOSH_DATA_LENGTH];
  uint16_t crc;
  /* The following fields are used for internal accounting only. */
  uint16_t strength;
  uint8_t ack;
  uint16_t time;
  uint8_t sendSecurityMode;
  uint8_t receiveSecurityMode;
} TOS_Msg;
```

Active Messaging

- All the messages sending/receiving in TinyOS are implemented as active messages
- The definitions are found in *tos/types/AM.h*
- Each message on the network specifies a HANDLER ID in the header.
- HANDLER ID invokes specific handler on recipient nodes
- When a message is received, the EVENT wired that HANDLER ID is signaled
- Different nodes can associate different receive event handlers with the same HANDLER ID

Topics are not coverred

- Obtaining the sensing data
- Implementing of Sending/Receiving data
- Using **TOSSIM** to simplify your work
- Display your data on your PC
- Implementing a subsystem based on TinyOS

Thank You !

More Documentation can be found on

http://www.tinyos.net/

Any Question?