



TinyOS 101

Gaurav Mathur (gmathur@cs.umass.edu)
Sensors Lab, UMass-Amherst





***So why do we need a
new OS... ?***



Traditional OS

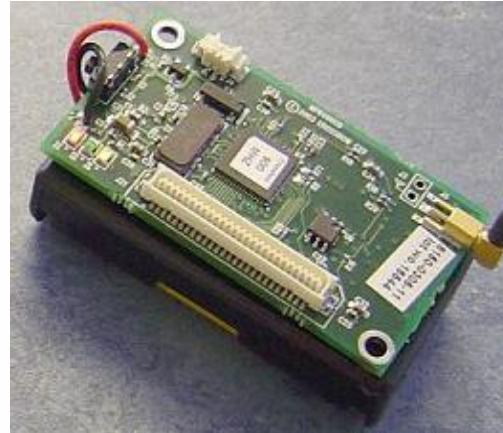
- Big (multiple MBs / GBs) !
- Multi-threaded architecture
 - Processes/threads => large memory footprint
- I/O model not best suited for sensors
 - Blocking I/O (most common)
- Kernel and user space separation
- Typically no energy constraints
- Ample available resources



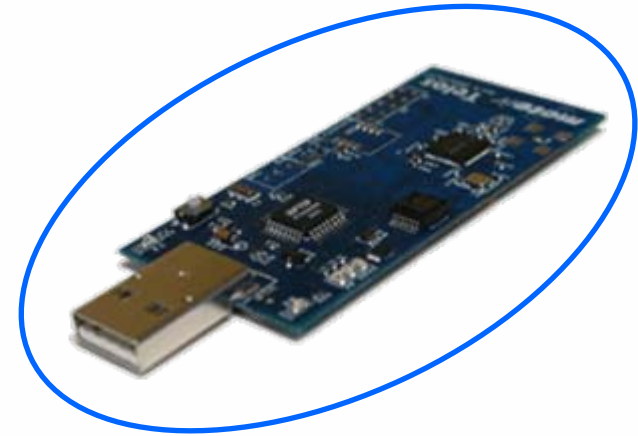
Hardware



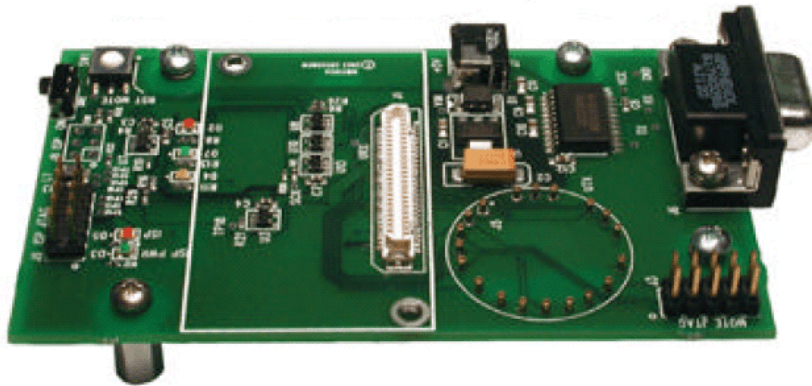
Mica2Dot



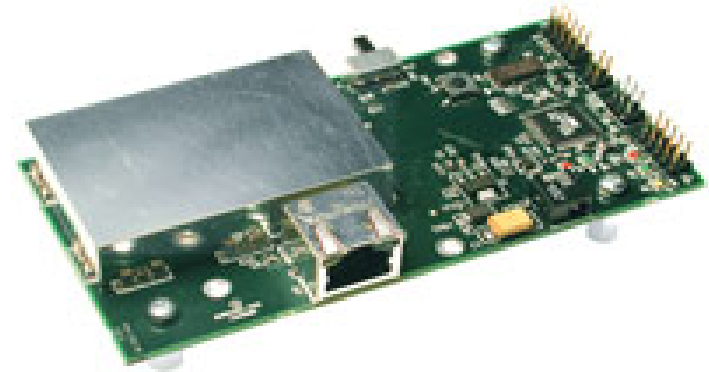
Mica Mote



Telos



MIB510 Serial Programming Board



MIB600 Ethernet Programming Board



Sensor Hardware Constraints

1. Power
2. Limited memory
3. Slow CPU
4. Size
5. Limited hardware parallelisms
6. Communication using radio
 - Low-bandwidth
 - Short range



Desired OS Properties

- Small memory footprint
- Efficient in power and computation
- Communication is fundamental
- Real-time
- Support diverse application design



TinyOS Solution

- **Concurrency:** uses event-driven architecture
- **Modularity**
 - Application composed of a graph of *components*
 - OS + Application compiles into single executable
- Code communication
 - Uses event/command model; translated to function calls
 - FIFO and non pre-emptive scheduling
- No kernel/application boundary





A TinyOS Program at a High-level



■ Naming conventions

- nesC files extension: **.nc**
- Clock.nc : either an interface or a configuration
- ClockC.nc : a configuration
- ClockM.nc : a module

```
Clock.nc  
  
interface Clock {  
  ...  
}
```

```
ClockC.nc  
  
configuration ClockC {  
  ...  
}  
  
implementation {  
  ...  
}
```

```
Timer.nc  
  
interface Timer {  
  ...  
}
```

```
TimerC.nc  
  
configuration TimerC {  
  ...  
}  
  
implementation {  
  ...  
}
```

```
TimerM.nc  
  
module TimerM {  
  ...  
}  
  
implementation {  
  ...  
}
```



Interfaces

- Provides the inter-connect fabric between components

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

```
interface X
{
    command result_t doSomething();
    event result_t doSomethingDone();
}
```



Modules

- Implement one or more interfaces
- Can use one or more other interfaces

```
module Provider
{
  provides interface StdControl;
  provides interface X;
  uses interface Z;
}
implementation
{
  // C code
  ....
}
```

MyComp.nc



Modules

```
module Provider
{ provides interface StdControl;
  provides interface X;
  uses interface Z;
}
implementation {
  command result_t StdControl.init()
  { return SUCCESS; }

  command result_t StdControl.start()
  { return SUCCESS; }

  command result_t StdControl.stop()
  { return SUCCESS; }

  task void signaler()
  {
    signal X.doSomethingDone();
  }

  command result_t X.doSomething()
  {
    post signaler();
    return SUCCESS;
  }
}
```

Implementor

User Component

```
module User
{ ...
  uses interface X;
}
implementation {
  ....

  task void A()
  {
    res = call X.doSomething();
  }

  ....

  event result_t X.doSomethingDone()
  {
    // Yay ! doSomething returned ok
    return SUCCESS;
  }
}
```



Modules

- Interfaces can also be parameterized
 - Multiple instances can be instantiated and used

```
module Provider
{ ...
  provides interface X[uint8_t id];
}
implementation {
  uint8_t idd;

  task void signaler()
  {
    signal X.doSomethingDone[idd]();
  }

  command result_t X.doSomething
[uint8_t id] ()
  {
    idd = id;
    post signaler();
    return SUCCESS;
  }
}
```



Configurations

- Two components are linked together in nesC by **wiring** them
- Interfaces on *user* component are wired to **the same interface** on the *provider* component
- Eg. wiring statements in nesC
 - $endpoint_1 = endpoint_2$
 - $endpoint_1 -> endpoint_2$

Application.nc

```
configuration Application {  
}  
implementation {  
  components Main, Provider, User, SomeComp;  
  
  Main.StdControl -> Provider.StdControl;  
  User.X -> Provider.X;  
  Provider.Z -> SomeComp.Z;  
}
```

Component that **uses** an interface is on the **left**,
and the component **provides** the interface is on the **right**



Compile & Run

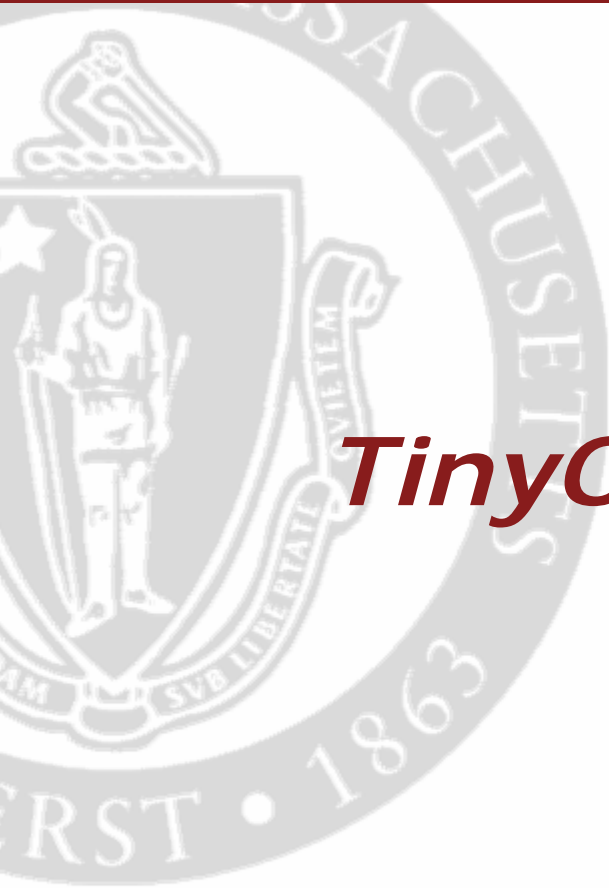
```
make mica2 install mib510,/dev/ttyS0
```

```
make mica2dot install mib510,/dev/ttyS0
```

```
make telosb install bsl,/dev/ttyUSB0
```

- Compiler processes nesC files converting them into a gigantic C file
 - Has both your application & the relevant OS components you are using
- Then platform specific compiler compiles this C file
 - Becomes a single executable
- Loader installs the code onto the Mote (Mica2, Telos, etc.)





TinyOS – Some Myths...



Myth 1

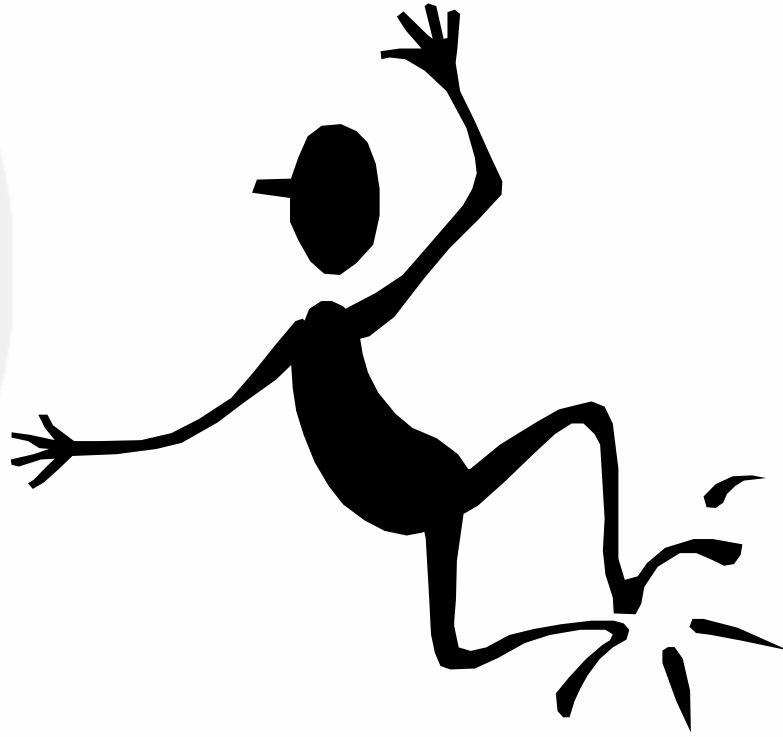


Intel Research Berkeley

TinyOS was created by ~~the Devil himself~~
and is The Ultimate Torture...



Myth 2



Someone became a pro in ~~less than a week~~ !

Way too Long – aim for 2 days...





Setup TinyOS

http://www.cs.umass.edu/~gmathur/misc/tinyos_setup.htm



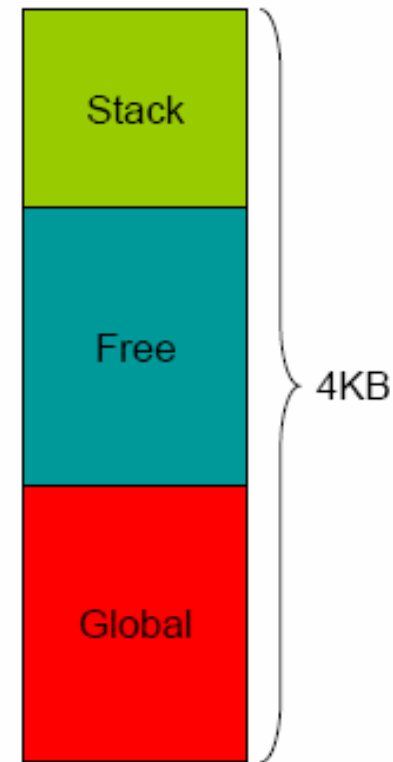


Theoretical side of TinyOS



TinyOS Memory Model

- STATIC memory allocation!
 - No heap (malloc)
 - No function pointers
- Global variables
 - Available on a per-frame basis
- Local variables
 - Saved on the stack
 - Declared within a method



TinyOS & nesC Concepts

- New Language: **nesC**. Basic unit of code = **Component**
- Component
 - Process **Commands**
 - Throws **Events**
 - Has a **Frame** for storing local state
 - Uses **Tasks** for concurrency
- Components *provide* **interfaces**
 - *Used* by other components to communicate with this component
- Components are *wired* to each other in a **configuration** to connect them



TinyOS Application

Component X

provides interface Y

command Y.X₁
throws event Y.X₂

Interface Y

command X₁
event X₂

Component Z

uses Interface Y

call command Y.X₁
handle event Y.X₂

Configuration A

Z.Y -> X.Y



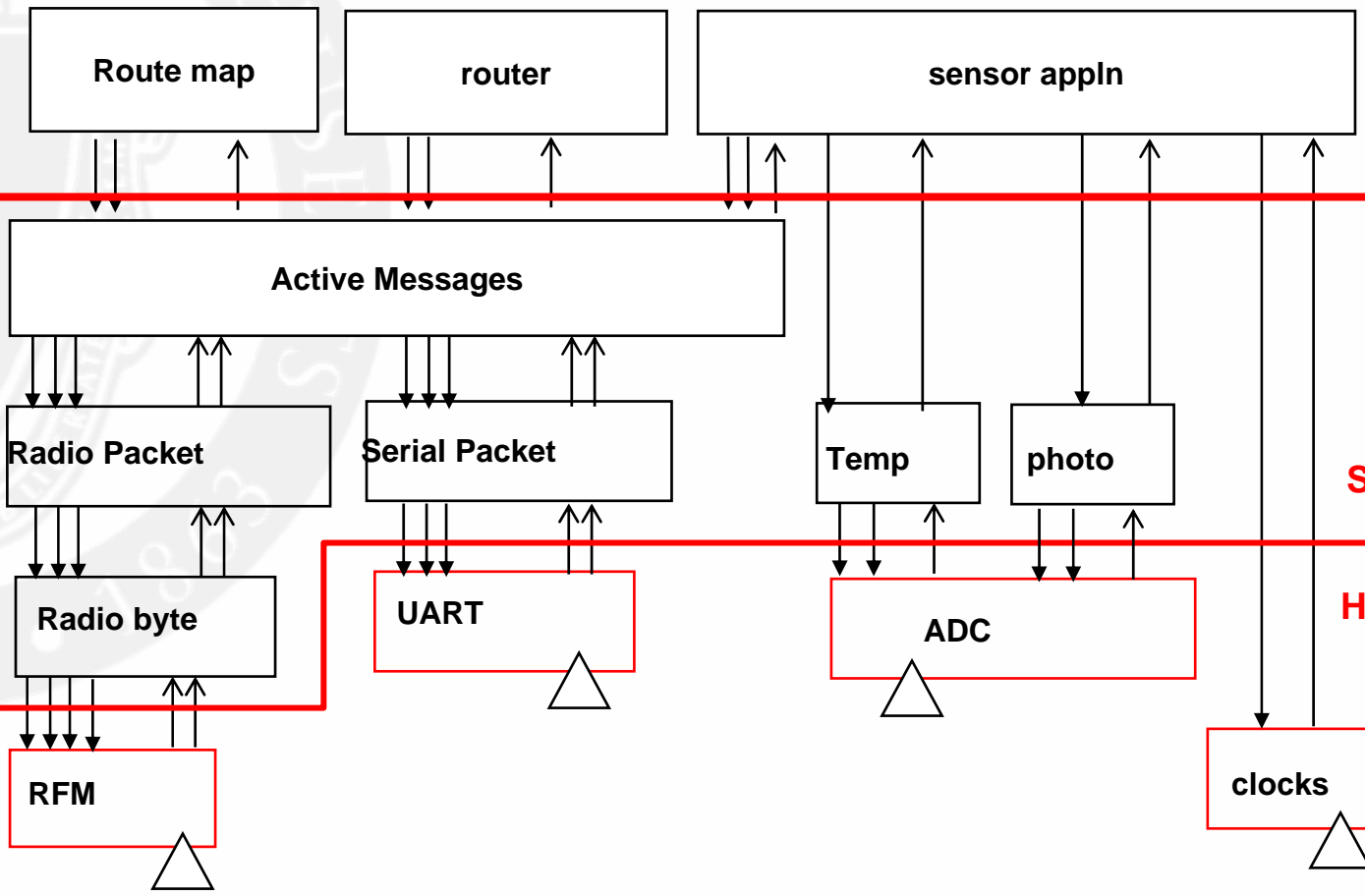
Application = Graph of Components

Application

The OS

SW

HW



Commands/Events/Tasks

■ Commands

- Should be non-blocking
- *i.e.* take parameters start the processing and return to app; postpone time-consuming work by posting a task
- Can call commands on other components

■ Events

- Can call commands, signal other events, post tasks but cannot be signal-ed by commands
- Pre-empt tasks, not vice-versa

■ Tasks

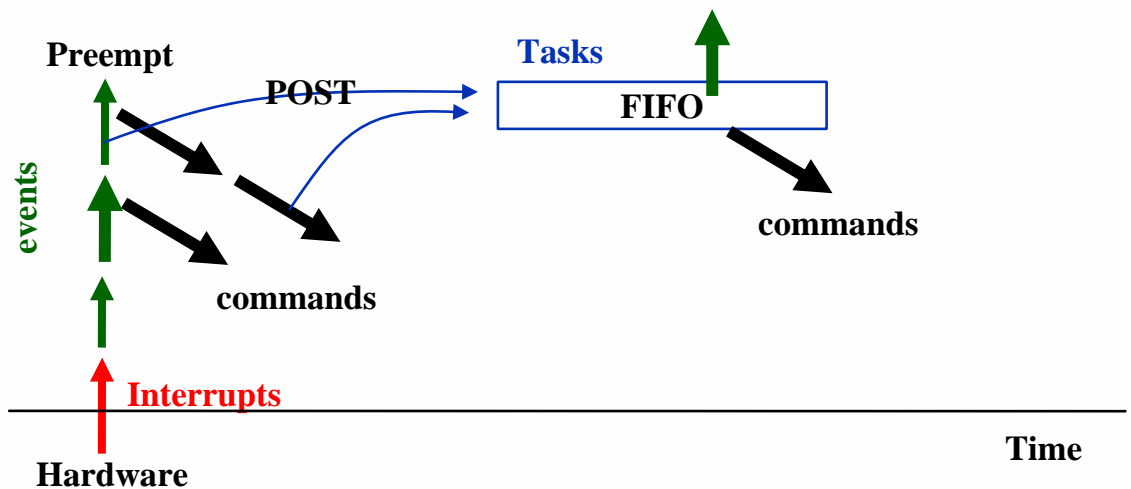
- FIFO scheduling
- Non pre-emptable by other task, pre-emptable by events
- Used to perform computationally intensive work
- Can be posted by commands and/or events



Scheduler

- Two level scheduling: events and tasks
- Scheduler is simple FIFO
- a task can not pre-empt another task
- events can pre-empt tasks (higher priority)

```
main {  
  ...  
  while(1) {  
    while(more_tasks)  
      schedule_task;  
    sleep;  
  }  
}
```



So what should I *really* care about ?

- No dynamic memory
- Single process execution; event-driven
- *commands* and *events* should do little work
- Post a *task* to do long processing

Your entire code should be a state-machine (arrgh !)



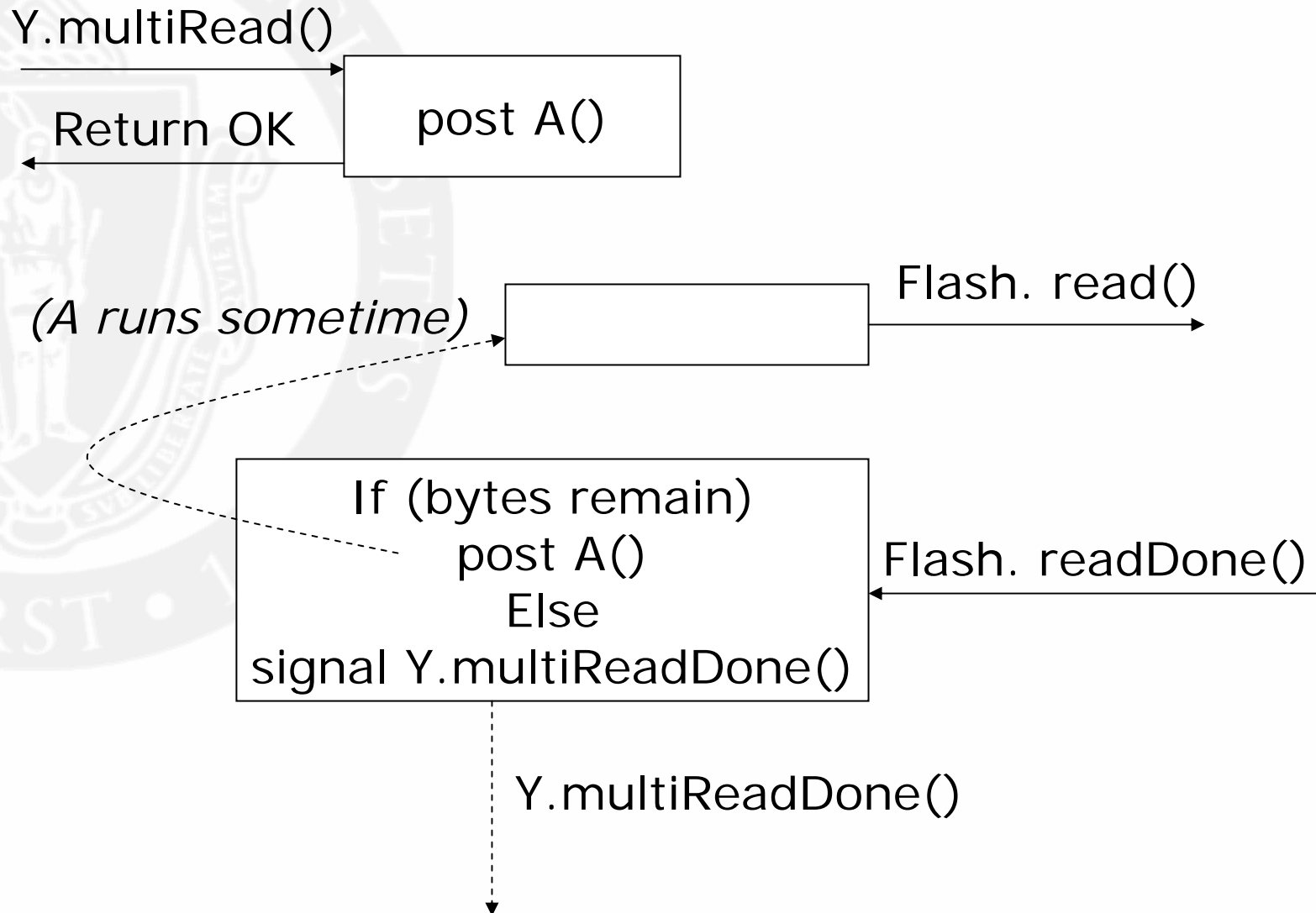
i.e. Code should be ***split-phase***, for example...

Fn exists to read a *single byte* at a time from flash

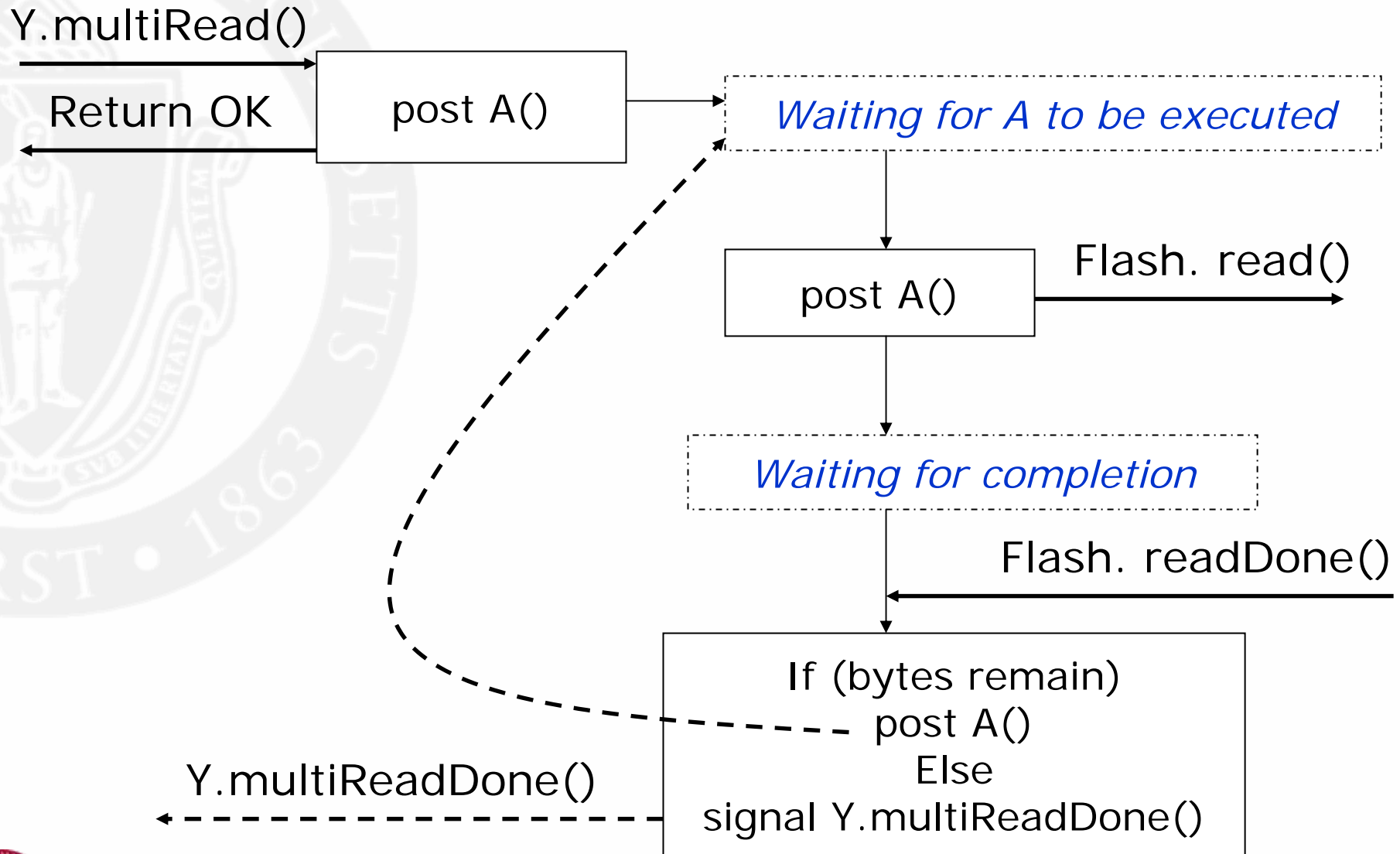
Write a wrapper to read *multiple bytes* together from flash



Code Structure



The State Machine



Example

- X calls your component Y to read some bytes from flash using a command *Y.multiRead()*
- You post a task to read the first byte calling *Flash.read()*
- Return to caller with status OK
- When *Flash.readDone()* returns, post task A to read 2nd byte
- When *Flash.readDone()* returns, post task A to read 3rd byte
- ...
- When all bytes are read, signal event *Y.multiReadDone()*
- If error was encountered, signal event *Y.multiReadDone()* passing an error value



Tips

- Keep switch on **OFF** on serial board
- Check radio frequency
- A LED is man's best friend!
- Use TOSSIM to test your program on the PC itself !
 - Uses realistic radio models
 - Scales to thousands of motes
 - Compiles directly from TinyOS source !
 - Notes: Helpful to catch common bugs as gdb can be used;
But, bad idea to reproduce timing issues
- DBG_USR flag (debug macro statements – works **only** on PC)
- export DBG=usr1
- `dbg(DBG_USR1, "Counter: Value is %i\n", state);`



Resources

- My TinyOS installation howto:
http://www.cs.umass.edu/~gmathur/misc/tinyos_setup.htm
- The official TinyOS tutorial (pretty good):
<http://www.tinyos.net/tinyos-1.x/doc/tutorial/>
- nesC: <http://nesc.sourceforge.net/>
- Other Docs:
 - nesC paper: <http://none.cs.umass.edu/~gmathur/tinyos/nesc-pldi-2003.pdf>
 - nesC manual: <http://none.cs.umass.edu/~gmathur/tinyos/nesc-ref.pdf>
 - TinyOS abstractions:
<http://none.cs.umass.edu/~gmathur/tinyos/tinyos-nsdi04.pdf>





Good Luck !

If you need help, email

To: gmathur@cs.umass.edu

CC: cs691aa@cs.umass.edu

