CS691AA: TinyOS Tutorial

Jeremy Gummeson
(gummeson@cs.umass.edu)
Sensors Lab, UMass-Amherst
So why do we need a new OS...?
Traditional OS

- Big!

- Multi-threaded architecture
  - Large number of processes/threads => large memory footprint

- I/O model
  - Blocking I/O (most common)

- Kernel and user space separation

- Typically no energy constraints

- Ample available resources
Sensor Hardware Constraints

- Power
- Limited memory
- Slow CPU
- Size
- Limited hardware parallelisms
- 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
SOS Solution

- Concurrency: uses event-driven architecture

- Modularity
  - Code: micro-kernel + application modules

- Code communication
  - function calls within a module
  - message passing between modules
  - Scheduling at module level; 2-level priority

- Clear application boundary
The Hardware
Programming Hardware

- Mica2Dot
- Mica Mote
- Telos
- MIB510 Serial Programming Board
- MIB600 Ethernet Programming Board
MIB 510 Serial Board

- Serial interface to laptop
- Mote JTAG
- MICA2Dot interface
- MICA2 interface
- ISP/JTAG
- 5V Power
- Block data to laptop
- Reset

Cost: $95
Part I: 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** (used for split-phase)
  - 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
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 cannot pre-empt another task
- Events can pre-empt tasks (higher priority)

```c
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

Y.multiRead()

Return OK

post A()

(A runs sometime)

If (bytes remain)
  post A()
Else
  signal Y.multiReadDone()

Flash. read()

Flash. readDone()

Y.multiReadDone()
The State Machine

Y.multiRead() -> post A() -> Waiting for A to be executed

If (bytes remain)
    post A()
Else
    signal Y.multiReadDone()

Flash. read() -> post A() -> Waiting for completion

Flash. readDone() -> Y.multiReadDone()
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 2\textsuperscript{nd} byte.
- When `Flash.readDone()` returns, post task A to read 3\textsuperscript{rd} byte.
- ...
- When all bytes are read, signal event `Y.multiReadDone()`.
- If error was encountered, signal event `Y.multiReadDone()` passing an error value.
TinyOS – Practical Information
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 {
    ...
  }
  implementation {
    ...
  }

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

ClockM.nc
  module ClockM {
    ...
  }
  implementation {
    ...
  }

Timer.nc
  interface Timer {
    ...
  }

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

TimerM.nc
  module TimerM {
    ...
  }
  implementation {
    ...
  }
Interfaces

- Provides the inter-connect fabric between components

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

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

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

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

MyComp.nc
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;
    }
}

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

```plaintext
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.
- 3 wiring statements in nesC:
  - `endpoint_1 = endpoint_2`
  - `endpoint_1 -> endpoint_2`
  - `endpoint_1 <- endpoint_2` (equivalent: `endpoint_2 -> endpoint_1`)

```plaintext
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

- 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.)
Tips

- Keep switch in OFF position on serial board (Mica Motes)
- Check radio frequency (Also AM_GROUP)
- A LED is man’s best friend!

- Use TOSSIM to test your program on the PC itself!
  - Uses realistic radio models
  - Repeatable experiments
  - Scales to thousands of motes
  - Compiles directly from TinyOS source!
  - Notes: Helpful to catch common bugs; 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);
TinyOS-2.x Considerations

- Same execution model as TinyOS-1.x
- Many modules / interfaces have changed significantly

For Example:

```plaintext
StdControl.nc
(TinyOS-1.x)

command result_t init();
event result_t initDone();
command result_t start();
event result_t startDone();
command result_t stop();
event result_t stopDone();

SplitControl.nc
(TinyOS-2.x)

command result_t boot();
command result_t bootDone();

Boot.nc

cmd result_t start();
event result_t startDone();
command result_t stop();
event result_t stopDone();
```
More On TinyOS-2.x

- Codebase restructured in more intuitive manner:
  - Each IC used by a sensor platform provides drivers in the “chips” directory
  - Platform is composed of interconnected chips
- Better abstraction for managing different flash chips on different sensor platforms
- Components are now multiply instantiable; previously unique namespace was often needed for each component
- Procedure for programming devices is identical to what is described in TinyOS-1.x; java tools have been re-implemented
- Currently limited support for TOSSIM; only the MicaZ platform and simple datatypes are supported
More on TinyOS-2.x

- TinyOS-1.x only allows one radio message type for a platform “TOS_Msg”; this assumes that a platform will only be using one radio interface.
- TinyOS-2.x uses “msg_t” to send messages over radio; radio stack is responsible for appending a corresponding header to the message.
Resources

- Gaurav’s TinyOS-1.x installation howto:
  [http://www.cs.umass.edu/~gmathur/misc/tinyos_setup.htm](http://www.cs.umass.edu/~gmathur/misc/tinyos_setup.htm)

- Tinyos-2.x installation howto (straightforward):

- The official TinyOS tutorial (pretty good):
  [http://www.tinyos.net/tinyos-1.x/doc/tutorial/](http://www.tinyos.net/tinyos-1.x/doc/tutorial/)

- The official TinyOS-2.x tutorial (very good):


- Other Docs:
Good Luck!

If you need help, email
To: gummeson@cs.umass.edu
CC: cs691aa@cs.umass.edu