Introduction to TinyOS and nesC Programming

Topics:

- TinyOS Kernel Design and Implementation
- nesC Software Concepts and Basic Syntax

TinyOS Design Goals

- Support Networked Embedded Systems
- Support Mica Hardware
  - power, sensing, computation, communication
- Support Technological Advances
  - keep scaling down
  - smaller, cheaper, lower power

TinyOS Design Conclusion

- **Objectives**
  - Manage Large number of Concurrent Data Flows
  - Manage Large number of Outstanding Events
  - Managing Application Data Processing
- Conclusion: Need a Multi Threading Engine
  - Extremely Efficient
  - Extremely Simple

TinyOS Kernel Design

- TinyOS Concurrency Model

  2 Level Scheduling Hierarchy Structure:

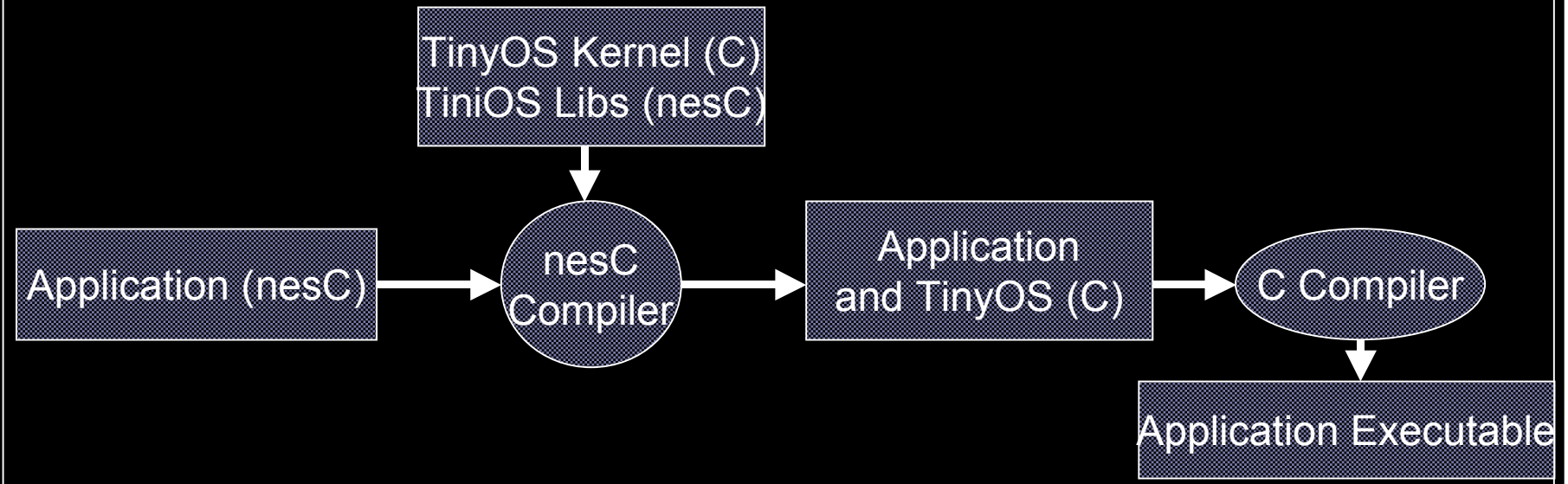
  1. Events preempts Tasks
  2. Tasks do not preempt other Tasks

TinyOS Kernel Design II

- Events
  - Time Critical
  - Small Amount of Processing
  - E.g. Timer, ADC Interrupts
- Tasks
  - Not Time Critical
  - Larger Amount of Processing
  - E.g. Computing an Average on an Array

TinyOS Applications

- Application is created in the nesC Language
- nesC Programming Language Supports the TinyOS Kernel Design (Events and Tasks)



The TinyOS Kernel Under The Hood (nesC Compiler C Code Output)

```
int main(void) {
  RealMain$hardwareInit();
  TOSH_sched_init();

  RealMain$StdControl$init();
  RealMain$StdControl$start();
  RealMain$Interrupt$enable();

  while (1) {
    TOSH_run_task();
  }
}
```

Hardware and Kernel initialization

Application initialization

Infinite loop

```
static inline void TOSH_run_task(void) {
  while (TOSH_run_next_task())
    ;
  TOSH_sleep();
  TOSH_wait();
}
```

1. First Run All Tasks in the Task Queue (Strictly a FIFO)
2. Then Sleep (In Low Power Mode)
3. And Wait for an Interrupt

```
bool TOSH_run_next_task(void) {
  uint8_t old_full;
  void (*func)(void );
  if (TOSH_sched_full == TOSH_sched_free) {
    return 0;
  }
  else {
    old_full = TOSH_sched_full;
    TOSH_sched_full++;
    TOSH_sched_full &= TOSH_TASK_BITMASK;
    func = TOSH_queue[(int )old_full].tp;
    TOSH_queue[(int )old_full].tp = 0;
    func();
    return 1;
  }
}
```

Task Runs To Completion (But May Be Interrupted By An Event)

Overhead of TinyOS Primitive Operations

Operation	Cost(cycles)	Time(uSecs)	Normalized to Byte Copy
Byte Copy	**8**	**2**	**1**
Signal an Event	10	2.5	1.25
Call a Command	10	2.5	1.25
Schedule a Task	46	11.5	6
Context Switch	51	12.75	6
Hardware Interrupt (hw)	9	2.25	1
Hardware Interrupt (sw)	71	17.75	9

Code and Data Size of the TinyOS kernel

	Code Size(bytes)	Data Size(bytes)
Processor Init	172	30
Scheduler	178	16
C runtime	82	0
	432	**46**

TinyOS/nesC Application Notes

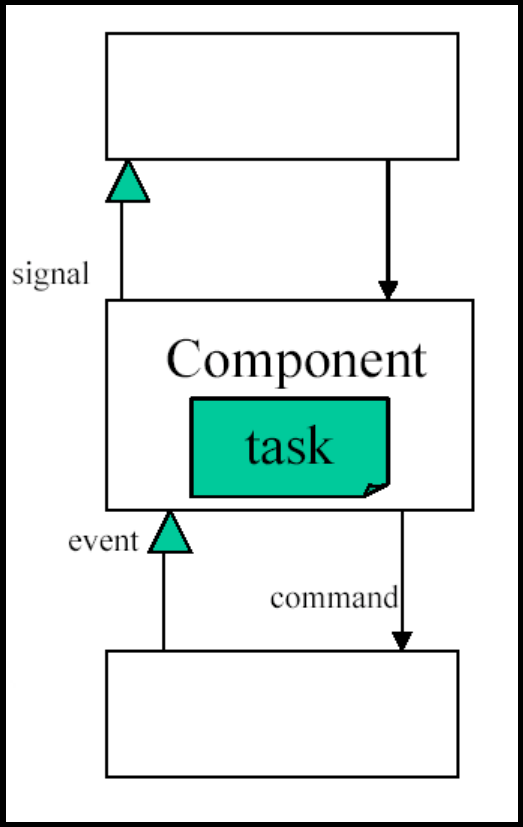
- Everything is Static
  - No Dynamic Memory (no malloc)
  - No Function Pointers
  - No Heap

Application Memory Map

- Text/code - Executable Code
  - In the 128K Program Flash
- data – Program Constants
  - In the 128K Program Flash
- bss - Variables
  - In the 4K SRAM
- Free Space - Fixed (No Dynamic Memory)
- stack - Grows Down in the Free Space

TinyOS Concepts Embodied by nesC – Tasks, Events, Commands

- **Tasks**
  - Background computation, non-time critical
- **Events**
  - Time critical
  - Originator gives a '**Signal**'
  - Receiver gets/accepts an '**Event**'
- **Command**
  - Function call to another Component
  - Cannot **Signal**



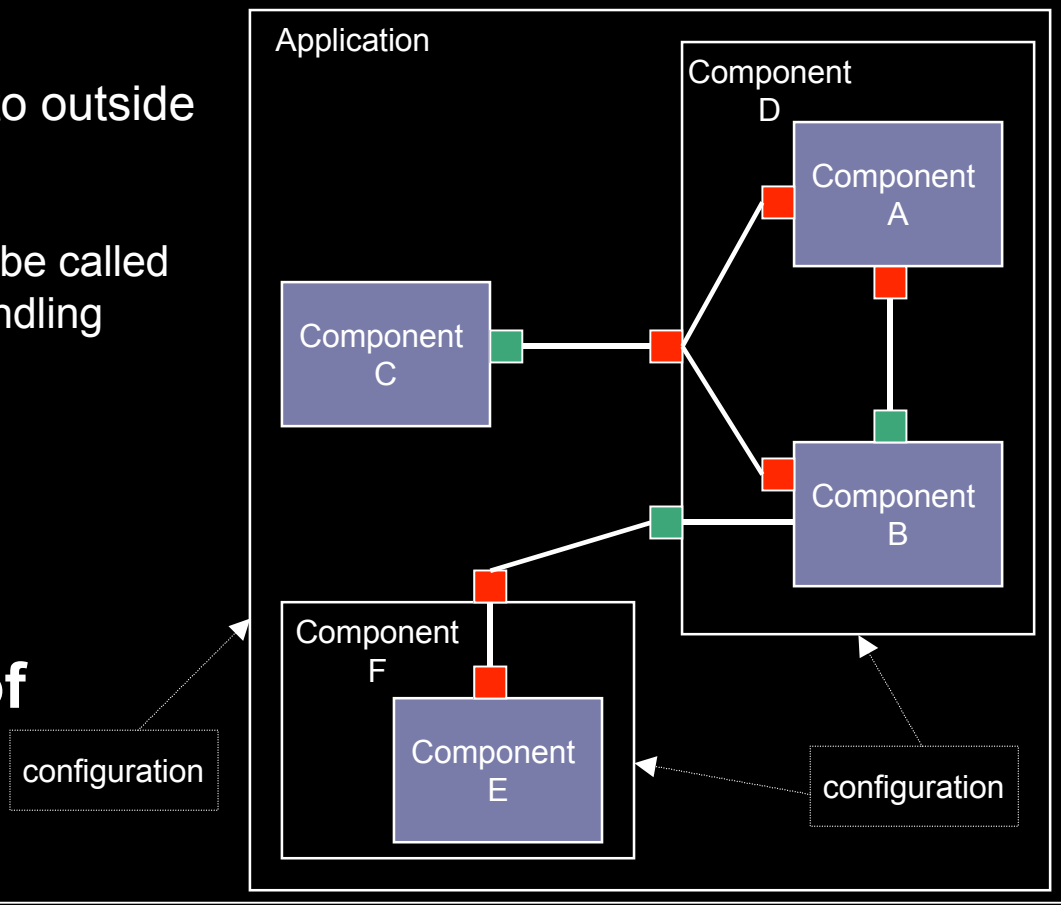
Concepts of SW Components

- **Interfaces**
  - Specifies functionality to outside world
  - Tell outside world
    - what commands can be called
    - what events need handling
    - USES - PROVIDES

Software Components:

- **Module**
- **Configuration**

application:= graph of components

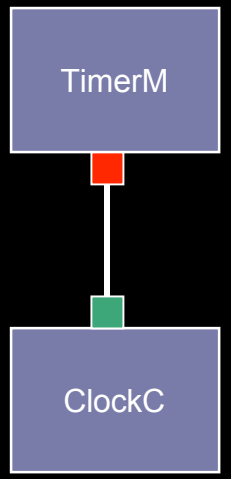


Interfaces

- describe bidirectional interaction:

```
interface Clock {
   command result_t setRate (char interval, char scale);
   event result_t fired ();
}
```

Clock.nc



- interface provider must implement commands
- interface user must implement events

Interfaces

- examples of interfaces:

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

StdControl.nc

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

Timer.nc

```
interface SendMsg {
  command result_t send (uint16_t addr,
                         uint8_t len,
                         TOS_MsgPtr p);
  event result_t sendDone ();
}
```

SendMsg.nc

```
interface ReceiveMsg {
  event TOS_MsgPtr receive (TOS_MsgPtr m);
}
```

ReceiveMsg.nc

Modules

- implements a component's specification with C code:

```
module MyComp {
   provides interface X;
   provides interface Y;
   uses interface Z;
}
implementation {
…// C code
}
```

MyComp.nc

Modules

- parameterised interfaces:

```
module GenericComm {
  provides interface SendMsg [uint8_t id];
  provides interface ReceiveMsg [uint8_t id];
  …
}
implementation {…
}
```

GenericComm.nc

  - i.e., it provides 256 instances of SendMsg and RecvMsg interfaces

Modules

- implementing the specification:

```
module DoesNothing {
  provides interface StdControl as Std;
}
implementation {
  command result_t Std.init() {
    return SUCCESS;
  }
  command result_t Std.start() {
    return SUCCESS;
  }
  command result_t Std.stop() {
    return SUCCESS;
}
```

DoesNothing.nc

Modules

- calling commands:

```
module TimerM {
  provides interface StdControl;
  provides interface Timer[uint8_t id];
  uses interface Clock;…
}

implementation {
  command result_t StdControl.stop() {
    call Clock.setRate(TOS_I1PS, TOS_S1PS);
  }
  …
}
```

TimerM.nc

Modules

- posting tasks:

```
module BlinkM {…
}
implementation {…
  task void processing () {
    if(state) call Leds.redOn();
    else call Leds.redOff();
  }

  event result_t Timer.fired () {
    state = !state;
    post processing();
    return SUCCESS;
  }…
}
```

BlinkM.nc

Configurations

- implements a component by wiring together multiple components:

```
configuration MyComp {
   provides interface X;
   provides interface Y;
   uses interface Z;
}
implementation {
…// wiring code
}
```

MyComp.nc

- wiring := connects interfaces, commands, events together

Configurations

- connected elements must be compatible (interface-interface, command-command, event-event)
- 3 wiring statements in nesC:
  - endpoint_1 = endpoint_2
  - endpoint_1 -> endpoint_2
  - endpoint_1 <- endpoint_2 (equivalent: endpoint_2 -> endpoint_1)

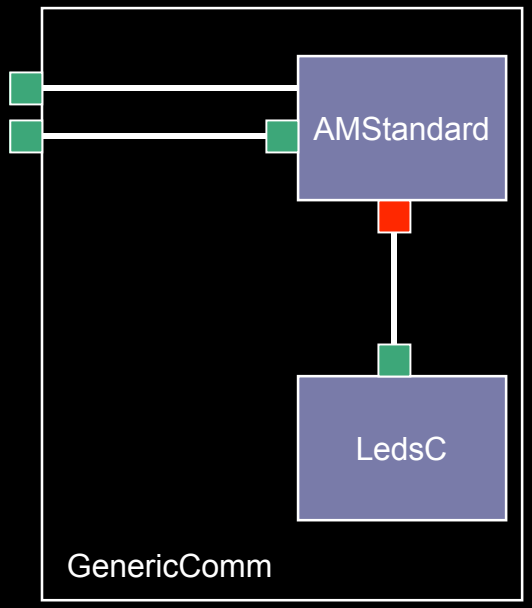
Configurations

- wiring example:

```
configuration GenericComm {
      provides interface StdControl as Control;
      command result_t activity();…
}

implementation {
   components AMStandard, LedsC;

   Control = AMStandard.Control;
   AMStandard.Leds -> LedsC.Leds;
   activity = AMStandard.activity;
}                                          GenericComm.nc
```



Configurations

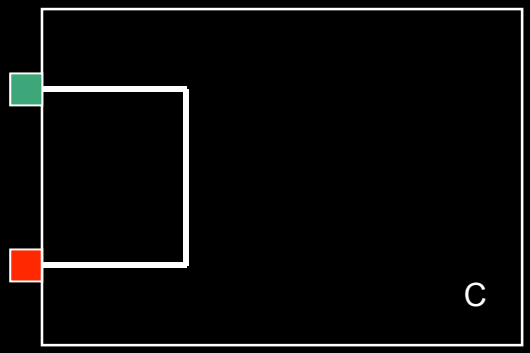
- other examples:

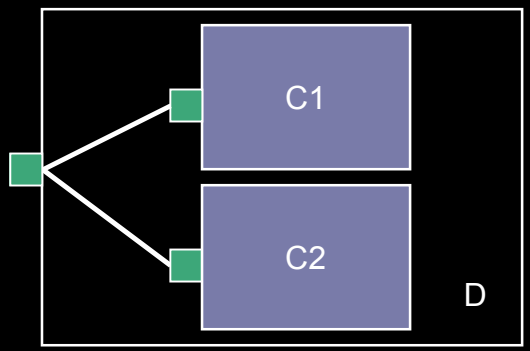
```
configuration C {
      provides interface X as Xprovider;
      uses interface X as Xuser;
}
implementation {
   Xuser = Xprovider;
}
```

C.nc

```
configuration D {
      provides interface X;
}
implementation {
   components C1, C2;
   X = C1.X;
   X = C2.X;
}
```

D.nc





Example

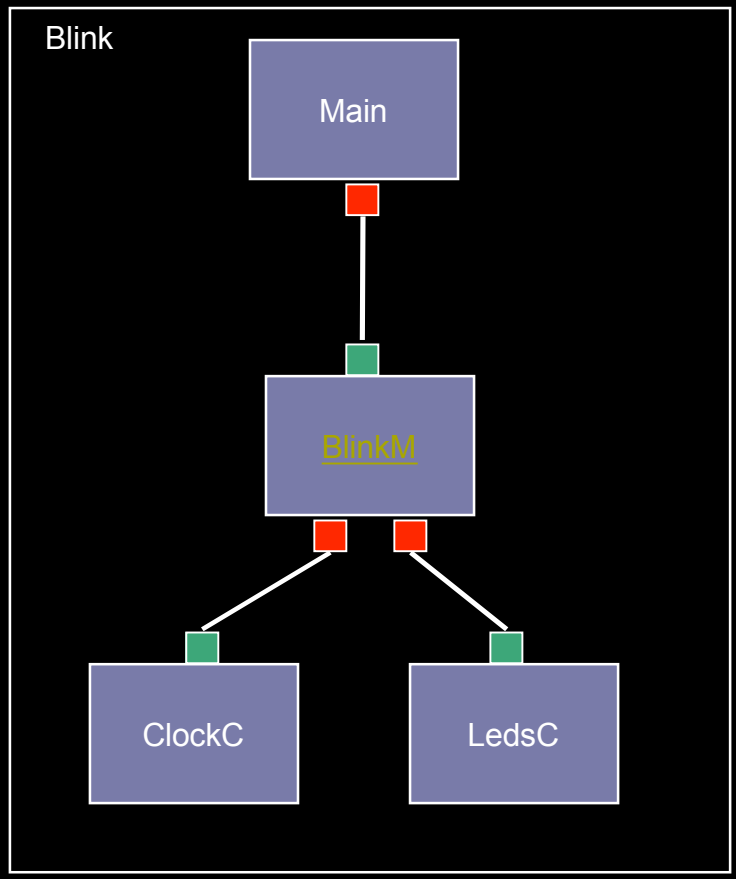
- Blink application

```
configuration Blink {
}

implementation {
  components Main, BlinkM, ClockC, LedsC;

  Main.StdControl->BlinkM.StdControl;
  BlinkM.Clock->ClockC;
  BlinkM.Leds->LedsC;
}
```

Blink.nc



Example

- BlinkM module:

```
module BlinkM {
  provides interface StdControl;
  uses interface Clock;
  uses interface Leds;
}

implementation {
  bool state;

  command result_t StdControl.init() {
    state = FALSE;
    call Leds.init();
    return SUCCESS;
  }
```

Blink.nc

```
  command result_t StdControl.start() {
    return call Clock.setRate(128, 6);
  }

  command result_t StdControl.stop() {
    return call Clock.setRate(0, 0);
  }

  event result_t Clock.fire() {
    state = !state;
    if (state) call Leds.redOn();
    else call Leds.redOff();
  }
}
```

Blink.nc

References

[1] System Architecture Directions for Networked Sensors, J.Hill, R.Szewczyk, A.Woo, S.Hollar, D.Culler, K.Pister, University of Virginia

[2] nesC/TinyOS Programming Basics

[3] D.Gay, D.Culler, P.Levis “nesC Language Reference Manual”, 9/2002

[4] **TinyOS Tutorial**