TinyOS Tutorial

Greg Hackmann
CSE 467S Spring 2011
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
TinyOS Installation

- TinyOS Documentation Wiki: http://docs.tinyos.net/
  - Various installation options listed under “Getting started” section

- Pre-compiled .rpm and .deb packages for Fedora and Ubuntu Linux users
  - Ubuntu users: be sure to remove brltty package

- All necessary drivers already included with Linux kernel
TinyOS Installation (cont.)

- OS X unofficially supported but works well

- Precompiled packages available at [http://www.cse.wustl.edu/~gwh2/tinyos-2.1.1.dmg](http://www.cse.wustl.edu/~gwh2/tinyos-2.1.1.dmg)

- Can also compile using MacPorts: see TinyOS Wiki
Windows installation uses Cygwin to emulate Linux software layer

- Problematic under XP, refuses to work on some Vista/7 machines (updating Cygwin after the installation may help)
- gcc is also very slow under Cygwin

“Running a XubunTOS Virtual Machine Image in VMware Player” tutorial recommended instead

- Or pick your favorite VM software and install Ubuntu yourself
TinyOS Installation Problems

- If you run `tos-check-env` and it says:
  
  `-> WARNING: CLASSPATH environment variable doesn't exist`

- then add:

  ```bash
  export CLASSPATH=/opt/tinyos-2.1.1/support/sdk/java/tynos.jar:.
  
  to your `.bash_profile` or `.bash_aliases` file
  ```
TinyOS Installation Problems

- If you try to run `make micaz sim` and it gives you a long error message starting with:

  ```
  ...: error: inttypes.h: No such file or directory
  ```

- then you need to install the C library/header files for your own architecture

- On Ubuntu/Debian, run `apt-get install build-essential`
TinyOS Directory Structure

- /opt/tinyos-2.1.1 ($TOSROOT)
  - apps
  - support
    - make
    - sdk
  - tools
  - tos
make System

- $TOSROOT/support/make includes lots of Makefiles to support the build process
- Create a simple stub Makefile in your app directory that points to main component
  
  COMPONENT=[MainComponentC]
  SENSORBOARD=[boardtype] # if needed
  include $(MAKERULES)

- make [platform] in app directory
  - Builds but does not install program
  - platform: one of the platforms defined in $TOSROOT/tos/platforms (mica2, micaz, telosb)
make System

- make [re]install.[node ID] [platform] [programming options]
  - node ID: 0 - 255
  - programming options:
    - mica2/micaz: mib510,/dev/ttyXYZ
    - telosb: bsl,/dev/ttyXYZ

- make clean

- make docs [platform]
  - Generates HTML documentation in $TOSROOT/doc/nesdoc/ [platform]
Build Stages

Preprocess .nc to .c, then compile .c to binary

Set AM address and node ID in binary

Program mote

```
Terminal — bash — 80x35 — x61

compiling BlinkAppC to a telosb binary

ncc -o build/telosb/main.exe -Os -O -std=gnu11 -Wall -Wshadow -Wnesc -all -target=telosb -fnesc-c-file=build/telosb/app.c -board= -DDEFINITION_TOS_AM_GROUP=0x23 -DIDENT_APPNAME="BlinkAppC" -DIDENT_USERNAME="ghw2" -DIDENT_HOSTNAME="Rooster148.cse." -DIDENT_USERHASH=0x9e110b01L -DIDENT_TIMESTAMP=0x48c59807L -DIDENT_TOSHASH=0x463cb61L BlinkAppC.nc -Im

Compiled BlinkAppC to build/telosb/main.exe

2650 bytes in ROM
55 bytes in RAM

msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
writing TOS image

tos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target=ihex build/telosb/main.ihex build/telosb/main.ihex.out-0 TOS_NODE_ID=0 ActiveMessageAddressSaddr=0
Could not find symbol ActiveMessageAddressSaddr in build/telosb/main.exe, ignoring symbol.
Could not find symbol TOS_NODE_ID in build/telosb/main.exe, ignoring symbol.
installing telosb binary using bsl

tos-bl -telosb -c /dev/tty.usbserial-M4A5L524 -r -e -I -p build/telosb/main.hex

MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...  Transmit default password...  Invoking BSL...
Transmit default password...  Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400...
Program...
2682 bytes programmed.
Reset device...
rm -f build/telosb/main.exe.out-0 build/telosb/main.ihex.out-0
```

Washington University in St. Louis
Demo: Installing Blink
“Homework”

- Install TinyOS 2.1 and build Blink

(Not graded, but a good idea to make sure you have everything up and running)
How to Get Help

- TinyOS Documentation Wiki: http://docs.tinyos.net
- TinyOS Programming Manual: 139-page PDF intro to nesC and TinyOS 2.x:
- TinyOS Tutorials: short HTML lessons on using parts of TinyOS (sensors, radio, TOSSIM, etc.):
  http://docs.tinyos.net/index.php/TinyOS_Tutorials
How to Get Help

- nesdoc: annotated API for all interfaces and components in TinyOS:
  [Link](http://docs.tinyos.net/index.php/Source_Code_Documentation)

- TinyOS Enhancement Protocols (TEP): formal documentation for TinyOS features:
  [Link](http://docs.tinyos.net/index.php/TEPs)
Outline

- Installing TinyOS and Building Your First App
- **Hardware Primer**
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
Available Hardware

- Motes, sensor boards, and gateways
- Most up to date list here: http://www.cs.wustl.edu/~gwh2/equipment/
- Hardware is in Mobile Computing Laboratory (Jolley 519)
MICA2 Mote (MPR400CB)

- Atmel ATmega128L µP
- 7.3827MHz (8 MIPS)
- Chipcon CC1000 radio, 38K or 19K baud, Manchester, 315, 433, or 900MHz
- 51 pin I/O Connector
- To Sensors, JTAG, and/or Programming Board
- 128KB Instruction EEPROM
- 4KB Data EEPROM
- 512KB External Flash Memory
- ADC 0-7
- UART 1
- I2C Bus
- SPI bus
- 3 LEDs
- UART 2
- 2 AA
- Quantity: 56
- Quantity: 56

Washington University in St. Louis
MPR2400 MICAz

- Same as Mica2 except with IEEE 802.15.4 radio
  - 2.4GHz
  - 250kbps

- Quantity: 7
Programming Board (MIB510)

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

Quantity: 3
MTS420 Sensor Board

- GPS
- Accelerometer
- Light
- Temperature
- Humidity
- Barometric Pressure
- 2KB EEPROM Conf.

Quantity: 1
MDA300 Sensor Board

- General measurement platform for motes
  - Temperature/Humidity sensor
  - 7 single-ended or 3 differential ADC channels
  - 6 digital I/O channels
- Quantity: 1
Tmote Sky (aka TelosB)

- IEEE 802.15.4 Radio
  - 250kbps
- TI MSP430 microcontroller
  - 16MHz, 16 MIPS, 10kB RAM
- Integrated antenna & USB interface
- Low power utilization
  - 1.8mA/5.1µA vs. Mica 2’s 8mA/15µA
- Quantity w/o on-board sensors: 6
- Quantity w/temperature, light, and humidity sensors: 20
- Quantity w/SBT80 sensor board: 4
NSLU2 Network Storage Link ("Slug")

- 266MHz Xscale CPU, 32MB SDRAM, 8MB flash, 1x Ethernet port
- Wired power
- No built-in radio, but 2x USB 2.0 ports for add-on 802.11/Bluetooth/mote interface
- Can be easily converted to an embedded Linux box with third-party firmware
  - Our testbed uses the OpenWrt distribution (http://openwrt.org)
- Quantity: 15
MCS410 Cricket Mote

- A Mica2 mote with ultrasound Rx/Tx
- Indoor localization
  - 10.5m range
- Distance accuracy:
  - 1 cm (< 3.5m dist)
  - 2cm (>3.5m dist)
- Mote finds distance, PC finds location
- http://cricket.csail.mit.edu/
- Quantity: 9
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- **Basic nesC Syntax**
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
TinyOS Execution Model

- To save energy, node stays asleep most of the time
- Computation is kicked off by hardware interrupts
- Interrupts may schedule tasks to be executed at some time in the future
- TinyOS scheduler continues running until all tasks are cleared, then sends mote back to sleep
TinyOS Component Model

NetworkHandlerC

NetworkHandlerP

ActiveMessageC

provides SplitControl
provides Receive

uses Receive

start()

startDone()
Components ¥= Objects
Interfaces

- List of exposed events and commands
- Like ordinary C function declarations, except with event or command in front

```c
interface Receive {
    event message_t * Receive(message_t * msg, void * payload, uint8_t len);
    command void * getPayload(message_t * msg, uint8_t * len);
    command uint8_t payloadLength(message_t * msg);
}
```
Modules

- Modules provide the implementation of one or more interfaces.
- They may consume (use) other interfaces to do so.

```plaintext
module ExampleModuleP {
    provides interface SplitControl;
    uses interface Receive;
    uses interface Receive as OtherReceive;
}
```

- “Rename” interfaces with the as keyword -- required if you are using/providing more than one of the same interface!
Modules

- implementation block may contain:
  - Variable declarations
  - Helper functions
  - Tasks
  - Event handlers
  - Command implementations
Modules: Variables and Functions

- Placed inside implementation block exactly like standard C declarations:
  ...
  implementation {
    uint8_t localVariable;
    void increment(uint8_t amount);
  }
  ...
  
  void increment(uint8_t amount) {
    localVariable += amount;
  }
}
Modules: Tasks

- Look a lot like functions, except:
  - Prefixed with `task`
  - Can’t return anything or accept any parameters

Implementation {
  ...  
  task void legalTask() {
    // OK
  }
  task bool illegalTask() {
    // Error: can’t have a return value!
  }
  task void anotherIllegalTask(bool param1) {
    // Error: can’t have parameters!
  }
}
Modules: Task Scheduling

- Tasks are scheduled using the `post` keyword

```c
error_t retval;
retval = post handlePacket();
// retval == SUCCESS if task was scheduled, or FAIL if not
```

- TinyOS guarantees that task will *eventually* run
  - Default scheduling policy: FIFO

![Diagram of tasks and task scheduling](image-url)
Modules: Commands and Events

- Commands and events also look like C functions, except:
  - they start with the keyword command or event
  - the “function” name is in the form InterfaceName.CommandOrEventName

- e.g.

```c
implementation {
  command error_t SplitControl.start() {
    // Implements SplitControl’s start() command
  }

  event message_t * Receive.receive(message_t * msg, void * payload, uint8_t len) {
    // Handles Receive’s receive() event
  }
}
```
Modules: Commands and Events

- Commands are invoked using the `call` keyword:

```plaintext
call Leds.led0Toggle();
// Invoke the led0Toggle command on the Leds interface
```

- Event handlers are invoked using the `signal` keyword:

```plaintext
signal SplitControl.startDone();
// Invoke the startDone event handler on the SplitControl interface
```
A command, event handler, or function can call or signal *any* other command or event from *any* interface wired into the module:

```plaintext
module ExampleModuleP {
    uses interface Receive;
    uses interface Leds;
}
implementation {
    event message_t Receive.receive(message_t * msg, void * payload, uint8_t len) {
        // Just toggle the first LED
        call Leds.led0Toggle();
        return msg;
    }
    ...
}
```
Synchronous vs. Asynchronous

- Commands and event handlers normally run in *synchronous* context
  - i.e., cannot be reached by an interrupt handler
- The `async` keyword notifies nesC that the command/event handler may run in an *asynchronous* context:

```cpp
implementation {
  async event void Alarm.fired() {
    // Handle hardware alarm interrupt
  }
}
```
Reminder: Race Conditions

- Use atomic blocks to avoid race conditions

```c
implementation {
    uint8_t sharedCounter;

    async event void Alarm.fired() {
        atomic {
            sharedCounter++;  // Interrupts are disabled here -- use sparingly and make as short as practical
        }
    }

    event void Receive.receive(...) {
        ...
        sharedCounter++;  // ... and make as short as practical
    }
}
```
Reminder: Race Conditions

- Tasks are always synchronous
- If timing isn’t crucial, defer code to tasks to avoid race conditions

```plaintext
implementation {
    uint8_t sharedCounter;

    task void incrementCounter() { sharedCounter++;
}

    async event void Alarm.fired() {
        post incrementCounter();
    }

    event void Receive.receive(...) {
        ...
        sharedCounter++;
    }
}
```

Task is scheduled immediately, but executes later
nesC and Race Conditions

- nesC can catch some, but not all, potential race conditions
- If you’re absolutely sure that there’s no race condition (or don’t care if there is), use the norace keyword:

```c
implementation {
    uint8_t sharedCounter;

    async event void Alarm1.fired() {
        sharedCounter++; // Race condition is impossible; events are mutually exclusive
        call Alarm2.start(200);
    }

    async event void Alarm2.fired() {
        sharedCounter--; // Race condition is impossible; events are mutually exclusive
        call Alarm1.start(200);
    }
}
```
TOSTThreads

- New in TinyOS 2.1: the TOSTThreads threading library
- Threads add a third execution context to TinyOS’s concurrency layer
  - Lowest priority: only run when TinyOS kernel is idle
  - Threads are preemptable by anything: sync, async, or other threads
- Also adds a library of synchronization primitives (mutex, semaphore, etc.) and blocking wrappers around non-blocking I/O
- Described in TOSTThreads Tutorial (http://docs.tinyos.net/index.php/TOSTThreads_Tutorial) or TEP 134
configuration NetworkHandlerC { 
  provides interface SplitControl;
}
implementation { 
  components NetworkHandlerP as NH, 
              ActiveMessageP as AM;
  //NH.Receive -> AM.Receive;
  //NH.SplitControl = SplitControl;
  NH.Receive -> AM;
  NH = SplitControl;
}
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
High-Level Summary

- nesC includes a lot of complex features that try to alleviate design problems with TinyOS 1.x

- The good news: you will probably never have to write code that incorporates these features

- The bad news: you’re almost certain to use code that incorporates these features

- First, an abstract look at what these features are and what their syntax means

- Second, a concrete example of how to use them to build components
Interfaces with Arguments

Creating new interfaces to support different data types can get redundant fast

```cpp
interface ReadUint16 {
    command error_t read();
    event void readDone(error_t error, uint16_t value);
}

interface ReadBool {
    command error_t read();
    event void readDone(error_t error, bool value);
}
```
Interfaces with Arguments

If you want to make an interface adapt to different underlying types, then put a placeholder in angle brackets:

```cpp
interface Read<type> {
    command error_t read();
    event void readDone(error_t error, type value);
}
```

```cpp
module SixteenBitSensorP {
    provides interface Read<uint16_t>;
}
```

```cpp
module BooleanSensorP {
    provides interface Read<bool>;
}
```
Fan-In: No Big Deal

AppLogicP
- uses Receive

NetworkHandlerP
- uses Receive

AnotherHandlerP
- uses Receive

RadioP
- provides Receive

Many-to-one calls work like you’d expect...
Fan-Out: Bad Things Happen

return &buffer1;

AppLogicP
uses Receive

NetworkHandlerP
uses Receive

AnotherHandlerP
uses Receive

provides Receive

RadioP

return &buffer2;

... but what about one-to-many calls?

return &buffer3;
Fan-Out: What Bad Things Happen?

- If different return values come back, nesC may not be able to make sense of the contradiction and will *arbitrarily* pick one.

- Avoid designs where this is possible.

- If you can’t avoid it, see TinyOS Programming Guide 5.2 for more info on combining return values.
Parameterized Wiring

Consider the following way to avoid fan-out:

```cpp
module RadioP {
  provides interface Receive as Receive0;
  provides interface Receive as Receive1;
  provides interface Receive as Receive2;
  uses interface LowLevelRadio;
  ...
}

implementation {
  event void LowLevelRadio.packetReceived(
    uint8_t * rawPacket) {
    ...
    uint8_t type = decodeType(rawPacket);
    if(type == 0)
      signal Receive0.receive(...);
    else if(type == 1)
      signal Receive1.receive(...);
    ...
  }
  ...
}
```
Parameterized Wiring

- The idea works in concept, but isn’t maintainable in practice
- But nesC can approximate the behavior in a much more maintainable way:

```c
module RadioP {
  provides interface Receive[uint8_t id];
  ...
}
implementation {
  event void LowLevelRadio.packetReceived(uint8_t * rawPacket) {
    ...
    uint8_t type = decodeType(rawPacket);
    signal Receive[type].received(...);
  }
  ...
}
```
Using Parameterized Wiring

- You can wire parameterized interfaces like so:

  ```
  AppLogicP -> RadioP.Receive[0];
  NetworkHandlerP -> RadioP.Receive[1];
  AnotherHandlerP -> RadioP.Receive[2];
  ```

- If each component is wired in with a unique parameter, then fan-out goes away
Unique Parameters

- In most cases, it’s unreasonable to expect the user to count the number of times (s)he is using the interface and wire accordingly.
- nesC can automatically generate a unique parameter for you using the unique() macro:

  ```
  AppLogicP -> RadioP.Receive[unique("RadioP")];
  // unique("RadioP") expands to 0
  
  NetworkHandlerP -> RadioP.Receive[unique("RadioP")];
  // unique("RadioP") expands to 1
  
  AnotherHandlerP -> RadioP.Receive[unique("RaadioP")];
  // unique("RaadioP") expands to 0 (oops)
  ...
  ```
What if your component needs to store different state for each unique parameter?

```c
module RadioP {
    ... 
}
implementation {
    int16_t state[uniqueCount(“RadioP”)];
    ...
}
```

uniqueCount(X) expands to # of times unique(X) appears in the application.
Defaults

- If you provide a parameterized interface and signal an event on it, you must also give a `default` event handler:

```plaintext
module SharedComponentP {
  ...
}
implementation {
  event void LowLevelRadio.packetReceived(uint8_t * rawPacket) {
    ...
    signal Receive[type].received(...);
  }

  default event void Receive.received[uint8_t id](...) {
    // e.g., do nothing
  }
  ...
}
```
Generic Components

- What if you have a component where different users absolutely should not share any state?
- Generic components let you “instantiate” a single component multiple times

```
generic module HashTableP() {
    provides interface HashTable;
}
...

components new HashTableP() as H1, new HashTableP() as H2;
AppLogicP.HashTable -> H1;
NetworkHandlerP.HashTable -> H2;
```
Generic Components

- But wait ... didn’t I say earlier that components aren’t objects?
- nesC internally creates a complete second copy of the component
Generic Components with Parameters

You can give each instantiation of the component slightly different behavior by adding compile-time parameters:

```c
generic module ListP(typedef type, uint8_t size) {
    provides interface List<type>;
}
implementation {
    type data[size];
    command void List.clear() {
        for(uint8_t i = 0; i < size; i++)
            data[i] = 0;
    }
}

components new ListP(bool, 16);
```
Putting It All Together: Building a Timer

- Consider an AlarmC component that exposes a 32 KHz hardware clock using the following interface:

  ```java
  interface Alarm {
      async event void fired();
  }
  ```

- We want to create a high-level timer component that:
  - Runs outside of the asynchronous context
  - Can be hooked into multiple components
  - Each consumer can choose a custom firing interval (every $n$ ticks)
  - Can be transformed into lower frequencies (16 KHz, 1 Hz, etc.)
Step 1: Get Out of Asynchronous Context
Step 2: Virtualize the Timer
Step 3: Reprogram the Timer

```c
interface Timer /* v. 2 */ {  
  event void fired();
}
```
Step 3: Reprogram the Timer

```c
module VirtualizeTimerP /* v. 2 */ {
    ...
}

implementation {

    event void SubTimer.fired() {
        uint8_t i;
        for(i = 0; i < 256; i++) {
            signal Timer.fired[i]();
        }
    }
}

command void Timer.startPeriodic[uint8_t id](uint16_t interval) {
    nextTimeToFire[id] = currentTime + interval;
    intervals[id] = interval;
}

...
Step 3.5: Tidy Up the Wiring

```
Step 3.5:
  Tidy	
  Up	
  the	
  Wiring

implementation {
  components AlarmC;
  components AlarmToTimerP as AtoT;
  components VirtualizeTimerP as Virt;
  AtoT.Alarm -> AlarmC;
  Virt.SubTimer -> AtoT;
  Timer = Virt.Timer[unique("VirtualizedTimerC")];
}
```
Step 3.5: Tidy Up the Wiring
module VirtualizeTimerP /* v. 2.5 */ {
    ...
}
implementation {
    enum {

        }

    uint16_t currentTime = 0;
    uint16_t nextTimeToFire[NUM_SLOTS];
    uint16_t intervals[NUM_SLOTS];

    event void SubTimer.fired() {
        uint8_t i;
        for(i = 0; i < NUM_SLOTS; i++) {
            ...
    
}
Step 4: Transform the Timer’s Frequency

```c
TransformTimerP(uint16_t multiplier)
{
  uses interface Timer as SubTimer;
  provides interface Timer;
}
implementation {
  event void SubTimer.fired()
  {
    signal Timer.fired();
  }
  command void Timer.startPeriodic(uint16_t interval)
  {
    call SubTimer.startPeriodic(interval * multiplier);
  }
}
```

Virtualized TimerC

Transform TimerP(2) SubTimer

Transform TimerP(32) SubTimer

Virtualized TimerC

Virtualized TimerC

Virtualized TimerC
Step 5: Add Type Safety

```c
interface Timer<frequency> /* v. 3 */ {
  event void fired();
  command void startPeriodic(uint16_t interval);
}

typedef struct {
  bool unused;
} T32Khz;

typedef struct {
  bool unused;
} T16Khz;

typedef struct {
  bool unused;
} TMilli;

enum {
  T32Khz,
  T16Khz,
  ...
  TMilli,
};
```
Step 5: Add Type Safety

generic module TransformTimerP(
    uint16_t multiplier) {
    uses interface Timer as SubTimer;
    provides interface Timer;
}
implementation {
    event void SubTimer.fired() {
        signal Timer.fired();
    }

    command void Timer.startPeriodic(uint16_t interval) {
        call SubTimer.startPeriodic(interval * multiplier);
    }
}
The Good News

- This is just an example! It’s already been implemented for you
- TimerMilliC component provides Timer&lt;TMilli&gt; interface
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
Slight Diversion: App Bootstrapping

- Each app has a “main” configuration which wires together the app’s constituent components
- But how do these components start running?
- TinyOS includes a MainC component which provides the Boot interface:

```c
interface Boot {
    event void void booted();
}
```
Slight Diversion: App Bootstrapping

Create one module which initializes your application, then wire MainC’s Boot interface into it:

```plaintext
configuration MyAppC {
} 
implementation {
    components MyAppP;
    components MainC;
    ...
    MyAppP.Boot -> MainC;
} 

module MyAppP {
    uses interface Boot;
} 
implementation {
    event void Boot.booted() {
        // Initialize app here
    }
    ...
} 
```
Slight Diversion #2: error_t Data Type

- TinyOS defines a special error_t data type that describes several different error codes
- Often given as return values to commands or event handlers
- Commonly used values:
  - SUCCESS (everything’s OK)
  - FAIL (general error, deprecated)
  - EBUSY (subsystem is busy with another request, retry later)
  - ERETRY (something weird happened, retry later)
- Others defined in $TOSROOT/types/TinyError.h
Message Addressing

- Each node has a unique 16-bit address (am_addr_t) specified by the make command `make install`.

- Two special address constants:
  - TOS_BCAST_ADDR (0xFFFF) is reserved for broadcast traffic
  - TOS_NODE_ID always refers to the node’s own address

- Each message also has an 8-bit Active Message ID (am_id_t) analogous to TCP ports
  - Determines how host should handle received packets, not which host receives it
TinyOS Active Messages

- message_t structure defined in $TOSROOT/tos/types/message.h
- Each platform defines platform-specific header, footer, and metadata fields for the message_t
- Applications can store up to TOSH_DATA_LENGTH bytes payload in the data field (28 by default)

```c
typedef nx_struct message_t {
    nx_uint8_t header[sizeof(message_header_t)];
    nx_uint8_t data[TOSH_DATA_LENGTH];
    nx_uint8_t footer[sizeof(message_footer_t)];
    nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```
Split-Phase Operation

- Many networking commands take a long time (ms) for underlying hardware operations to complete -- blocking would be bad.

- TinyOS makes these long-lived operations split-phase:
  - Application issues
  - An event is signaled when it's actually done

```c
interface SplitControl {
    command error_t start();
    event void startDone(error_t error);

    command error_t stop();
    event void stopDone(error_t error);
}
```

Error code here indicates whether TinyOS could start processing request.
Error code here indicates whether TinyOS could complete processing request.
Active Messaging Interfaces

interface AMSend {
    command error_t send(am_addr_t addr, message_t * msg,
                          uint8_t len);
    command error_t cancel(message_t * msg);
    event void sendDone(message_t * msg, error_t error);
    command uint8_t maxPayloadLength();
    command void* getPayload(message_t * msg, uint8_t len);
}

interface Receive {
    event message_t* receive(message_t * msg, void *
                              payload, uint8_t len);
}
interface Packet {
  command void clear(message_t * msg);
  command void* getPayload(message_t * msg, uint8_t len);
  command uint8_t payloadLength(message_t * msg);
  command void setPayloadLength(message_t * msg, uint8_t len);
  command uint8_t maxPayloadLength();
}
Other Networking Interfaces

```c
interface AMPacket {
    command am_addr_t address();
    command am_group_t localGroup();

    command am_addr_t destination(message_t* amsg);
    command am_addr_t source(message_t* amsg);
    command am_group_t group(message_t* amsg);
    command bool isForMe(message_t* amsg);

    command am_id_t type(message_t* amsg);
}
```
Other Networking Interfaces

interface PacketAcknowledgements {
    async command error_t requestAck(message_t* msg);
    async command error_t noAck(message_t* msg);
    async command bool wasAced(message_t* msg);
}

- Default behavior: no ACKs
- Even with ACKs enabled, no automatic retransmissions
- Optional packet link layer can handle retransmissions;
  #define PACKET_LINK and see TEP 127
Message Buffer Ownership

- Transmission: Radio driver gains ownership of the buffer until sendDone(...) is signaled
- Reception: Application’s event handler gains ownership of the buffer, but it must return a free buffer for the next message
Network Types

- Radio standards like 802.15.4 mean that you could have communication among different types of motes with different CPUs.

- nesC defines network types (nx_uint16_t, nx_int8_t, etc.) that transparently deal with endian issues for you.

- nesC also defines an nx_struct analogous to C structs:

```c
typedef struct {
    uint16_t field1;
    bool field2;
} bad_message_t;

// Can have endianness problems
// if sent to a host with a different architecture
```

```c
typedef nx_struct {
    nx_uint16_t field1;
    nx_bool field2;
} good_message_t;

// nesC will resolve endian issues for you
```
Sending a Message

- First create a .h file with an nx_struct defining the message data format, and a unique active message ID (127–255)

```c
enum {
    AM_SENSORREADING = 240,
};

typedef nx_struct sensor_reading {
    nx_int16_t temperature;
    nx_uint8_t humidity;
} sensor_reading_t;
```
Sending a Message

- Declare a `message_t` variable in your module to store the packet’s contents
- Get the packet’s payload using the `Packet` interface; cast it to your message type; and store your data

```c
implementation {
...
message_t output;

task void sendData() {
  sensor_reading_t * reading =
      (sensor_reading_t *)call Packet.getPayload(&output,
                                           sizeof(sensor_reading_t));
  reading->temperature = lastTemperatureReading;
  reading->humidity = lastHumidityReading;
  ...
}
```
Finally, use the AMSend interface to send the packet

```c
void sendData() {
  ...

  if(call AMSend.send(AM_BROADCAST_ADDR, &output, sizeof(sensor_reading_t)) != SUCCESS)
    post sendData();
  // Try to send the message, and reschedule the task if it
  // fails (e.g., the radio is busy)
}
```
Sending a Message

- The AM subsystem will signal AMSend.sendDone() when the packet has been completely processed, successfully or not.

```c
event void AMSend.sendDone(message_t * msg, error_t err) {
  if(err == SUCCESS) {
    // Prepare next packet if needed
  }
  else {
    post sendTask();
    // Resend on failure
  }
}
```
Receiving a Message

When messages with the correct AM ID are received, the Receive interface fires the receive() event

```c
implementation {
...
    event message_t * Receive.receive(message_t * msg,
                             void * payload, uint8_t len) {
        am_addr_t from = call AMPacket.source(msg);
        sensor_reading_t * data = (sensor_reading_t *)payload;
        ...
        return msg;
    }
}
```
Networking Components

- Note that we didn’t mention the packet’s AM ID anywhere in the code.
- That’s because TinyOS includes generic components to manage the AM ID for you when you send/receive:

```java
components new AMSenderC(AM_SENSORREADING);
components new AMReceiverC(AM_SENSORREADING);
```

```java
MyAppP.AMSend -> AMSenderC;
// AMSenderC provides AMSend interface
MyAppP.Receive -> AMReceiverC;
// AMReceiverC provides Receive interface
MyAppP.Packet -> AMSenderC;
MyAppP.AMPacket -> AMSenderC;
// AMSenderC and AMReceiverC provide Packet and AMPacket
// interfaces (pick one or the other)
```
Networking Components

- Before you can send/receive, you need to turn the radio on
- `ActiveMessageC` component provides a `SplitControl` interface to control the radio’s power state

```
components ActiveMessageC;
MyAppP.RadioPowerControl -> ActiveMessageC;
```
What About Multi-Hop?

- Until recently, TinyOS did not include a **general-purpose, point-to-point** multi-hop routing library.
- Two special-purpose algorithms instead:
  - Collection Tree Protocol (CTP)
  - Dissemination
- Experimental TYMO point-to-point routing library added to TinyOS 2.1 ([http://docs.tinyos.net/index.php/Tymo](http://docs.tinyos.net/index.php/Tymo))
- blip: IPv6 stack added to TinyOS 2.1.1 ([http://docs.tinyos.net/index.php/BLIP_Tutorial](http://docs.tinyos.net/index.php/BLIP_Tutorial))
Collection Tree Protocol (CTP)

Basic Operation
configuration MyCtpAppC {
}
implementation {
  components AppLogicP;
  components CollectionC;
  ...
  MyAppP.RoutingControl -> CollectionC;
  MyAppP.RootControl -> CollectionC;
  ...
}

module AppLogicP {
  uses interface StdControl as RoutingControl;
  uses interface RootControl;
  ...
}
implementation {
  ...
  event void RadioControl.startDone(
    error_t err) {
    ...
    if(TOS_NODE_ID == 100)
      call RootControl.setRoot();
      call RoutingControl.start();
    }
  ...
}
configuration MyCtpAppC {
}
implementation {
    components AppLogicP;
    components CollectionC;
    ...
    MyAppP.Send -> CollectionC.
    Send[MY_MSG_ID];
    MyAppP.Receive -> CollectionC.
    Receive[MY_MSG_ID];
    MyAppP.Packet -> CollectionC;
    ...
}

module AppLogicP {
...
uses interface Send;
uses interface Receive;
uses interface Packet;
...
}
implementation {
...
    task void sendPacket() {
        result_t err = call Send.send(
            &msg, sizeof(MyMsg));
    }
    ...
}

event message_t * Receive.receive(
    message_t * msg, void * payload,
    uint8_t len) {
    // Only signaled on root node
    ...
}

Collection Tree Protocol (CTP)  Sending/Receiving Packets

Washington University in St. Louis
Collection Tree Protocol (CTP)

- To link into your app, include these lines in your Makefile:

  ```
  CFLAGS += -I$(TOSDIR)/lib/net
  CFLAGS += -I$(TOSDIR)/lib/net/4bitle
  CFLAGS += -I$(TOSDIR)/lib/net/ctp
  ```

- CTP automatically turns on packet ACKs, retransmits up to 30 times at each hop
  - But no end-to-end acknowledgments;
  - `PacketAcknowledgments.wasAked()` only tells you if the packet made it to the first hop
Dissemination

Basic Operation
For More Information

- TinyOS Tutorial 12: Network Protocols
  (http://docs.tinyos.net/index.php/Network_Protocols)
- TEP 123: Collection Tree Protocol
  (http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html)
- TEP 118: Dissemination
  (http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html)
Sending Data to a PC

- TinyOS apps can also send or receive data over the serial/USB connection to an attached PC
- The SerialActiveMessageC component provides an Active Messaging interface to the serial port:

```c
components SerialActiveMessageC;
MyAppP.SerialAMSend ->
    SerialActiveMessageC.Send[AM_SENSORREADING];
MyAppP.SerialReceive ->
    SerialActiveMessageC.Receive[AM_SENSORREADING];
// SerialActiveMessageC provides parameterized AMSend and Receive interfaces
MyAppP.SerialPowerControl -> SerialActiveMessageC;
```
Interfacing With Motes

- Mote serial stack adds packet delimiting, byte escaping, CRC, retransmissions, etc. on top of raw serial packets
- If you want to talk directly to motes (not recommended), TEP 113 documents serial protocol
Interfacing With Motes

- TinyOS includes a Java-based SerialForwarder utility that implements PC side of TEP 113
  - `java net.tinyos.sf.SerialForwarder -comm serial@[port]:[speed]`
  - `[speed]` may be a specific baud rate or a platform name (e.g., telosb)

- Listens on TCP port and sends/receives TinyOS messages from local or remote applications
Interfacing With Motes

- Java SDK connects to SerialForwarder and converts TinyOS messages to/from native Java objects
- `mig` application auto-generates these classes from your app’s header files
  - `mig java -java-classname=[classname] [header.h] [message-name] -o [classname].java`
import net.tinyos.message.*;

public class MyTinyOSApp implements MessageListener {
    private final MoteIF moteIF = new MoteIF();

    public MyTinyOSApp() {
        moteIF.registerListener(new MyMsg(), this);
    }

    public void messageReceived(int to, Message m) {
        if (m.amType() == MyMsg.AM_TYPE) {
            MyMsg msg = (MyMsg)m;
            System.out.println("someField = " + msg.get_someField());
            MyResponse response = new MyResponse();
            response.set_someOtherField(msg.get_someField() + 1);
            moteIF.send(0, response);
        }
    }

    public static void main(String [] args) {
        new MyTinyOSApp();
    }
}
SDK Support for Other Languages

- **C/C++**
  - C reimplementation of SerialForwarder (sf) and a few test apps found in $TOSROOT/support/sdk/c/sf
  - Building sf also builds libmote.a for accessing the motes in your own code
  - See sfsource.h and serialsource.h to get started
SDK Support for Other Languages

- **Python**
  - Python classes in $TOSROOT/support/sdk/python closely mirror Java SDK
  - Not completely stand-alone; Python MoteIF implementation talks to Java or C SerialForwarder
  - See tinyos/message/MoteIF.py to get started

- **C#**
  - mig can generate C# classes to parse/generate raw TinyOS packets
  - But it’s up to the user to actually get those packets from the serial port or SerialForwarder
interface CC2420Config {
  command uint8_t getChannel();
  command void setChannel(uint8_t channel);

  async command uint16_t getShortAddr();
  command void setShortAddr(uint16_t address);

  async command uint16_t getPanAddr();
  command void setPanAddr(uint16_t address);

  command error_t sync();
  event void syncDone(error_t error);
}

(Provided by CC2420ControlC component)
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- **Sensor Data Acquisition**
- Debugging Tricks and Techniques
Obtaining Sensor Data

- Each sensor has components that provide one or more split-phase Read interfaces

```cpp
interface Read<val_t> {
    command error_t read();
    event void readDone(error_t result, val_t val);
}
```

- Some sensor drivers provide additional interfaces for bulk (ReadStream) or low-latency (ReadNow) readings
  - See TEPs 101 and 114 for details
Sensor Reading Example

module MyAppP {
  uses interface Read<uint16_t> as AccelX;
  ...
}
implementation {
  ...
  task void readAccelX() {
    if(call AccelX.read() != SUCCESS)
      post readAccelX();
  }
  event void AccelX.readDone(error_t err, uint16_t reading) {
    if(err != SUCCESS) {
      post readAccelX();
      return;
    }
    // Handle reading here
  }
  ...
}

configuration MyAppC {
}
implementation {
  components MyAppP;
  components new AccelXC();
  // X axis accelerometer component
  // defined by mts300 sensorboard
  MyAppP.AccelX -> AccelXC;
  ...
}
Sensor Components

Sensor components are stored in:

- $TOSROOT/tos/platform/[[platform]] (for standard sensors)
  - Note that telosb “extends” telosa, so look in both directories if you’re using a TelosB or Tmote Sky mote!
- $TOSROOT/tos/sensorboard/[[sensorboard]] (for add-on sensor boards)

Additional sensor board components may be available from TinyOS CVS in tinyos-2.x-contrib

- Unfortunately, some third-party sensor board drivers have yet to be ported from TinyOS 1.x to 2.x
interface HplMsp430GeneralIO {
  command void makeInput();
  command void makeOutput();
  command bool get();
  command void clr();
  command void set();
  command void toggle();
}
External Sensors

- Digital I/O: wire directly into HplMsp430GeneralIOC component

  component HplMsp430GeneralIOC {
    provides interface HplMsp430GeneralIO as ADC0;
    provides interface HplMsp430GeneralIO as ADC1;
    provides interface HplMsp430GeneralIO as ADC2;
    provides interface HplMsp430GeneralIO as ADC3;
    provides interface HplMsp430GeneralIO as ADC4;
    provides interface HplMsp430GeneralIO as ADC5;
    provides interface HplMsp430GeneralIO as ADC6;
    provides interface HplMsp430GeneralIO as ADC7;
    provides interface HplMsp430GeneralIO as DAC0;
    provides interface HplMsp430GeneralIO as DAC1;
  }

- I²C: read TEP 117 (Low-Level I/O)
- Analog I/O: read TEP 101 (Analog-to-Digital Converters)
Outline

- Installing TinyOS and Building Your First App
- Hardware Primer
- Basic nesC Syntax
- Advanced nesC Syntax
- Network Communication
- Sensor Data Acquisition
- Debugging Tricks and Techniques
Hard-Learned Lessons

- Be sure to check return values -- don’t assume SUCCESS!
  - At the very least, set an LED when something goes wrong

- The TinyOS toolchain doesn’t always warn about overflowing integers

```c
uint8_t i;
for(i = 0; i < 1000; i++) {
    ...
} // This loop will never terminate
```

- Not all the Tmote Sky motes have sensors
msp430-gcc Alignment Bugs

If you’re unlucky, msp430-gcc will crash with internal errors like these:

```
/opt/tinyos-2.x/tos/interfaces/TaskBasic.nc: In function `SchedulerBasicP$TaskBasic$runTask':
/opt/tinyos-2.x/tos/interfaces/TaskBasic.nc:64: unable to generate reloads for:
(call_insn 732 3343 733 (set (reg:SI 15 r15)
  (call (mem:HI (symbol_ref:HI ("AsyncQueueC$1$Queue$dequeue")) [0 S2 A8])
   (const_int 0 [0x0])))) 14 {*call_value_insn} (nil)
  (nil)
  (nil))
```

It’s almost always because of alignment bugs (msp430-gcc doesn’t always like it when fields straddle 16-bit boundaries)

```c
typedef nx_struct my_msg {
  nx_uint8_t field1;
  nx_uint8_t pad;
  nx_uint16_t field2;
} my_msg_t;
```
802.15.4 Radio Channels

- The CC2420 chip on the Tmote and MicaZ supports 802.15.4 channels 11 - 26
- 802.15.4 uses 2.4 GHz spectrum
- This can lead to interference between motes and with 802.11, Bluetooth, and all sorts of other things
802.15.4 Radio Channels

> If you’re seeing weird network behavior, set your CC2420 channel to something else:

- Defaults to 26
- Command-line: CC2420_CHANNEL=xx make ...
- Makefile: PFLAGS = -DCC2420_DEF_CHANNEL=xx

(Small request: please do not use channels 12, 14, 22, or 25!)
Active Message Groups

- To avoid address collision with other applications or networks, you can also change the AM group:
  - Defaults to 0x22
  - Makefile: DEFAULT_LOCAL_GROUP=xx (any 16-bit value)

- On 802.15.4 compliant chips, maps to PAN ID

- Does not prevent *physical* interference of packets: only instructs radio chip/driver to filter out packets addressed to other groups
The easiest way to display runtime information is to use the mote’s LEDs:

```cpp
interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();
  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();
  async command uint8_t get();
  async command void set(uint8_t val);
}
```

Provided by the components LedsC and NoLedsC
You can use `printf()` to print debugging messages to the serial port.

Messages are buffered and sent to serial port in bulk; `printfflush()` asks TinyOS to flush buffer.

**DON’T USE `printf()` FOR CRITICAL MESSAGES**

When its buffer fills up, `printf()` starts throwing away data.
printf()

- To enable the printf library, add the following line to your Makefile:

  CFLAGS += -I$(TOSDIR)/lib/printf

- Note: this automatically turns on SerialActiveMessageC subsystem

- Included PrintfClient utility displays printed messages to console
  
  java net.tinyos.tools.PrintfClient
  [-comm serial@[port]:[speed]]
The BaseStation app in $TOSROOT/apps/
BaseStation will sniff all wireless traffic and forward it to the serial port

Listen tool prints hex-dump of packets to console:
java net.tinyos.tools.Listen
   [-comm serial@[port]:[speed]]

Extremely helpful for figuring out what data is being sent!
The CPU on the Tmote Sky motes supports interactive debugging using gdb

- Set breakpoints, inspect the contents of variables, etc.

- The catch: it needs a special cable and modified motes -- and they don’t make the motes anymore
  - We have 5 motes and one cable
TOSSIM

- make micaz sim compiles application to native C code for your own machine, which can be loaded into Python or C++ simulator ("TOSSIM")
- Good way to rapidly test application logic, at the cost of some realism
  - e.g., does not emulate sensing and does not reproduce timing of real microcontrollers

- Besides app code, need two configuration details:
  - Topology of simulated network
  - Noise trace from simulated environment
TOSSIM Configuration: Topology

- List of links in the network and associated gain (signal strength in dBm)

- Several sources:
  - Real measurements
  - Samples included in TinyOS ($TOSDIR/lib/tossim/topologies)
  - Generate one based on various parameters ([http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/usc-topologies.html](http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/usc-topologies.html))

<table>
<thead>
<tr>
<th>Source</th>
<th>Link 1</th>
<th>Gain</th>
<th>Link 2</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>0</td>
<td>-90.80</td>
<td>1</td>
<td>-95.95</td>
</tr>
<tr>
<td>Samples</td>
<td>0</td>
<td>-97.48</td>
<td>2</td>
<td>-102.10</td>
</tr>
<tr>
<td>Parameters</td>
<td>0</td>
<td>-111.33</td>
<td>3</td>
<td>-115.49</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-104.82</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-110.09</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(from 15-15-sparse-mica2-grid.txt)
TOSSIM Configuration: Noise Trace

- Trace of ambient noise readings in dBm
- Must contain at least 100 entries; more is better, but RAM consumption increases with larger traces
- Two sources:
  - Real measurements
  - Samples included in TinyOS ($TOSDIR/lib/tossim/noise)

(from meyer-heavy.txt)
import TOSSIM
sim = TOSSIM.Tossim([])
radio = sim.radio()

topology = ... # load topology from disk
all_motes = set()
for src, dest, gain in topology:
    radio.add(src, dest, gain)
    all_motes.add(src)

noise_trace = ... # load noise trace from disk
for i in all_motes:
    node = sim.getNode(i)
    for entry in noise_trace:
        node.addNoiseTraceReading(entry)
    node.createNoiseModel()
    node.bootAtTime(5 * sim.ticksPerSecond())

simulation_length = 60 * sim.ticksPerSecond()
while sim.time() < simulation_length:
    sim.runNextEvent()
Other TOSSIM Features

- Log debug messages to console or to a file
- Inject packets into network
- Debugging support
  - Python TOSSIM: read variables’ contents
  - C++ TOSSIM: use gdb
- TOSSIM Live fork: TOSSIM acts as SerialForwarder, send/receive serial packets to simulated motes
  - [http://docs.tinyos.net/index.php/TOSSIM_Live](http://docs.tinyos.net/index.php/TOSSIM_Live)

- See TinyOS Tutorial 11 for more details
Avrora + MSPsim

- Avrora: cycle-accurate Mica2 and MicaZ emulator
  [Link](http://compilers.cs.ucla.edu/avrora/)
- MSPsim: MSP430 (TelosB) emulator
  [Link](http://www.sics.se/project/mspsim/)
- Profile and benchmark apps, monitor packet transmissions, or interface with gdb
- Slower than TOSSIM, but highly accurate
Safe TinyOS

- New in TinyOS 2.1: make [platform] safe
- Augments code to enforce pointer and type safety at runtime (bad casts, out-of-bounds array accesses, NULL pointer dereferences, etc.)
- When safety violations detected, LEDs blink error code


Demo: Putting it All Together