Embedded computing projects, Learning TinyOS & nesC

Rahav Dor
Spring 2014

Kevin Klues started this lecture series in 2007. It has been inherited and kept up to date with advancements in TinyOS, nesC, and their development environments by Chien-Liang Fok, Gregory Hackmann, Mo Sha, and now Rahav Dor.
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
Cool projects out there

- Good night lamp.
- Plantduino Greenhouse.
- Twitter mood light.
- LED cube.
- Shoe that lace itself.
- Turn signal biking jacket.
- Secret knock door lock.
- Tree climbing robot.
- Sigh collector.
Cool projects in here

- Compare TinyOS and Contiki.
- Turn your mote into a web server, have it support AllJoin (AllSeen), Bonjour, CoAP, NLNA, MQTT, nanoIP, REST, etc.
- Empirical study of any number of above protocols
- Develop your own transport protocol, discovery protocol, ...
- Turn Raspberry Pi into a 802.15.4 base station (this is one of the few projects that are allowed on RPi).
- Comparative study of security for IoT devices
  - New ideas?
- Make your smartphone into a 802.15.4 remote control
  - Support querying, proximity, and simple commands
- Develop an IoT search engine
Get inspired

- http://makezine.com/
- http://www.instructables.com/

Note that you cannot use the Raspberry Pi as the main platform in this course without my prior approval
Projects – requirements

- Embedded system
- Design, Code, and evaluation – evenly divided among team
- Project scope must be approved by me (Rahav)
- Can repeat past projects as long as they have a twist
  - But you need to design and author your own code
- You will be subject to academic disciplinary actions, including failing this course, if you (see course full policy)
  - Submit program code written outside this course without proper citation
  - Simply copy code not written by you
  - Use code of another student not on your team
  - Collaboration and discussions are great exploration vehicles and are encouraged, but you must attribute the ideas to their originator
Projects – artifacts

- All artifacts will count toward your grade
- Design (diagrams and discourse authored in any tool)
  - Specifications
  - Data flow
  - Logical interactions between the components or functions
  - Design considerations
    - What you have considered (optional)
    - What you have decided to do
- README.txt (or readme.txt)
  - See the readmeFD.txt example, which includes
  - A short description of the app
  - How to install, compile, and run your project (including a command line section, e.g. java MIG)
  - Hardware dependencies
Equipment

- Equipment, motes, ... must be returned in good condition
  - -10% automatic reduction otherwise
  - Physical interfacing must be done via connectors
  - Past evils can be undone
    - Up to a full grade letter (on the project part) of extra credit for good citizenship: Soldering, Double sided tape, Cleaning

- See course site for existing equipment
  - Within reason and time we can buy staff to support your invention
Header comment (1 of 2)

- Author a header comment at the top of all source files. Use the following format:
- The name of the file (e.g., HelloWorldC.nc)
- The name of the deliverable (e.g., Midterm demo. RFID reader.)
- Name of the chief author and email address
- Class: CSE 467S – Embedded Computing Systems
- WUSTL, Spring 2014
- The date on which the first release of the code was ready
- Invariants
A short paragraph describing the functionality. This should include, for example the purpose of the file or its role in the project. Be crisp, precise, and informative. Communicate your approach to solving the problem.

Version log with:
- Date
- Who made the change, email address, and why
- What was the change
HelloWorldC.nc
Purpose: Final project. Agent updater.
Author   : Rahav Dor
            r2d2@wustl.edu
Course  : CSE 467S -- Embedded Computing Systems
            WUSTL, Spring 3017
Date      : Jan. 2, 3017

Invariants:
  10 Hz <= updateFrequency <= 50 Hz
Description:
  Maintain communication between agents and keep a local state when agents are
detached from the hive. If communication channels are open this code guarantees
information exchange according to the invariant.

Version log:
  2/1/2013, Jean-Luc Picard
      Fixed the remote agent bug.
  3/10/2013, Seven of Nine
      An agent can now be detached from the collective, while still maintaining
      its basic operations.
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
Style

- Write self documenting code:
  
  ```
  if (lowTemperature(temp))

  work = force * displacement * cos(moteToFloorAngle)
  ```

- Indent 3 spaces or equal sized tab stop
- Use **DESCRIPTIVE_CONSTANTS**
- And self explanatory `variablesNames`
- FileNames, ClassNames, e.g
  - Modules are named `FallDetectionC`
  - Configuration is named `FallDetectionAppC`
Style

- curlyFunctionBraces {
    ... 
  }

- if (youFollow) {
    your are in a good starting place;
} else {
    you start from less than optimal condition;
}

- Functions must have specs above them that describe the function, invariants if any, parameters, and return value

- Use inline comments as needed
Style – TinyOS / nesC

- Each app has a Configuration (the wiring) and one or more Modules (the code)
- Wiring file are named: HelloWorld\texttt{AppC}.nc
  - This file name is the one specified in the Makefile as the source to be compiled
- Code files are named: HelloWorld\texttt{C}.nc
- Can also end with a \texttt{P} to denote Private
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
TinyOS community

- http://www.tinyos.net/

TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. A worldwide community from academia and industry use, develop, and support the operating system as well as its associated tools, averaging 35,000 downloads a year.

**Latest News**

January, 2013: The transition to hosting at GitHub is now complete. Part of this transition includes slowly retiring TinyOS development mailing lists for bug tracking and issues to using the GitHub trackers. Thanks to all of the developers who are now improving TinyOS and requesting pulls!

August 20, 2012: TinyOS 2.1.2 is now officially released; you can download it from the debian packages on tinyos.stanford.edu. Manual installation with RPMs with the instructions on docs.tinyos.net will be forthcoming. TinyOS 2.1.2 includes:

- Support for updated msp430-gcc (4.6.3) and avr-gcc (4.1.2).
- A complete 6lowpan/RPL IPv6 stack.
- Support for the ucmini platform and ATmega128RF1A chip.
- Numerous bug fixes and improvements.

**FAQ**
Frequently asked questions about TinyOS

**Learn**
Download TinyOS and learn how to use it

**Community**
TinyOS Working Groups, mailing lists, and TEPs
TINYOS INSTALLATION – LINUX

The next number of slides go over the general concepts of installing TinyOS (my recommendations as of Jan. 2014).

To install the system use the handout available on the course web site.
TinyOS recommended installation

- We will be installing TinyOS, on Linux, running on a VM, that runs on a Host OS
- Follow the installation instructions handout
- I recommend you do the large downloads while on campus and while connected to a wired network
- VMware (Fusion version is the one for OS-X and cost 49$), or VirtualBox (free)
  - Known problems: Cut and Paste, Bluetooth
- Ubuntu 12.x (LTE) or 13.x (other debian flavors may work equally well)
  - Consideration for 32 and 64 bit versions: Memory, TinyOS will not compile on 64, and OS response time
TinyOS directory structure

- After TinyOS is installed you will have a new tree branch under your home directory: 
  `/$HOME/tinyos-main` (this must correspond to `$TOSROOT`)
  - apps
  - doc
  - licenses
  - README
  - release-notes.txt
  - support
  - tools
  - tos

- Checkout TinyOS README file for details
Before motes are connected

Learn which devices are attached to Linux before you connect any mote to your machine. It will allow you to identify what is the address of motes when you eventually connect them. Use the `ls /dev/tty*` command to capture data similar to the following:

```
rHAV@Ubuntu:~$ ls /dev/tty*
/dev/tty   /dev/tty23   /dev/tty39   /dev/tty54   /dev/ttyS10   /dev/ttyS26
/dev/tty0   /dev/tty24   /dev/tty4   /dev/tty55   /dev/ttyS11   /dev/ttyS27
/dev/tty1   /dev/tty25   /dev/tty40   /dev/tty56   /dev/ttyS12   /dev/ttyS28
/dev/tty16   /dev/tty26   /dev/tty41   /dev/tty57   /dev/ttyS13   /dev/ttyS29
/dev/tty11   /dev/tty27   /dev/tty42   /dev/tty58   /dev/ttyS14   /dev/ttyS3
/dev/tty12   /dev/tty28   /dev/tty43   /dev/tty59   /dev/ttyS15   /dev/ttyS30
/dev/tty13   /dev/tty29   /dev/tty44   /dev/tty6   /dev/ttyS16   /dev/ttyS31
/dev/tty14   /dev/tty3   /dev/tty45   /dev/tty60   /dev/ttyS17   /dev/ttyS4
/dev/tty15   /dev/tty46   /dev/tty61   /dev/ttyS18   /dev/ttyS5
/dev/tty16   /dev/tty47   /dev/tty62   /dev/ttyS19   /dev/ttyS6
/dev/tty17   /dev/tty48   /dev/tty63   /dev/ttyS2   /dev/ttyS7
/dev/tty18   /dev/tty49   /dev/tty7   /dev/ttyS20   /dev/ttyS8
/dev/tty19   /dev/tty5   /dev/tty8   /dev/ttyS21   /dev/ttyS9
/dev/tty2   /dev/tty50   /dev/tty9   /dev/ttyS22
/dev/tty20   /dev/tty51   /dev/ttyprintk   /dev/ttyS23
/dev/tty21   /dev/tty52   /dev/ttyS0   /dev/ttyS24
/dev/tty22   /dev/tty53   /dev/ttyS1   /dev/ttyS25
```
When motes are connected

- When you connect a mote, the Virtual Machine will ask you where do you want to connect it – **chose Linux**
- The `ls` command as shown on the previous slide, or the command `motelist` will show you the attached motes
- Change the permissions on any tinyOS related
  - serial (/dev/ttyS<N>)
  - usb (/dev/tts/usb<N>, /dev/ttyUSB<N>)
  - or parallel (/dev/parport)

  devices you are going to use, using the following command:

  ```
  sudo chmod 666 /dev/<devicename>
  ```
  Example:
  ```
  sudo chmod 666 /dev/ttyUSB0
  ```

- See a better alternative in the handout
TinyOS installation verification

- To check your TinyOs installation run `tos-check-env`
- If you did everything properly you will see only one WARNING about graphviz not being version 1.10 (it will appear twice)
- This is actually OK because you will have a newer version (likely 2.x). To check run `dot -V`
- Later in this lecture, and on the installation handout, we will also build Blink to conclude the verification
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 INSTALLATION – LINUX

This concludes the general details about installing TinyOS
make System

- TinyOS includes lots of Makefiles to support the build process
- Create a simple stub Makefile in your app directory that points to main component, and invokes TinyOS make system.

```
COMPONENT=[MainComponentAppC]       # the name of your ...AppC file
SENSORBOARD=[boardtype]            # if needed
include $(MAKERULES)
```

- Two real examples:

```
COMPONENT=FDE_DataCollectionAppC
include $(MAKERULES)
```
make System

COMPONENT=FallDetectionAppC
SENSORBOARD  = mts300
BUILD_EXTRA_DEPS += SenseToRadioMsg.class
CLEAN_EXTRA = *.class SenseToRadioMsg.java
CFLAGS += -DTOSH_DATA_LENGTH=114
# CFLAGS += -DTOSH_DATA_LENGTH=44
SenseToRadioMsg.class: $(wildcard *.java) SenseToRadioMsg.java
   javac SenseToRadioMsg.java
SenseToRadioMsg.java:
   mig java -target=null -java-classname=SenseToRadioMsg FallDetection.h SenseToRadioMsg -o @$include $(MAKERULES)
make System

- To compile an app without installing it on a mote, run in the app directory:

  make [platform]

  Where platform is one of mote names (e.g. micaz, telosb, shimmer2r) defined in $TOSROOT/tos/platforms

- make clean

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

make [re]install.[node ID] [platform] [programmingBoard,address]

- node ID : 0 – 255, for radio transmissions
- platform: as defined in $TOSROOT/tos/platforms
- Programming Board:
  - For mica2/micaz use: mib510
  - For telosb use : bsl
- Address (as reported by ls /dev/tty* or motelist). For example:
  /dev/ttyXYZ
  ttyUSB0
  /dev/tty.usbserial-FTVCZRNVA
Useful commands

- **motelist**
  - See the list of motes physically connected to your pc

- **make telosb**
  - Compile your code for the telosb mote

- **make telosb install.1**
  - Compile your code for telosb, install it on a mote, give it the network id 1

- **make telosb reinstall.7**
  - Use existing runnable, install it on telosb, give it a network id of 7

- **make docs telosb**
  - Generate docs for your application
Build stages

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

Set AM address and node ID in binary

Program mote
Our first TinyOS experience

Provided with TinyOS Blink’s README.txt:
Blink is a simple application that blinks the 3 mote LEDs. It tests that the boot sequence and millisecond timers are working properly. The three LEDs blink at 1Hz, 2Hz, and 4Hz. Because each is driven by an independent timer, visual inspection can determine whether there are bugs in the timer system that are causing drift.

Note that this method is different than RadioCountToLeds, which fires a single timer at a steady rate and uses the bottom three bits of a counter to display on the LEDs.

The provided make file:
COMPONENT=BlinkAppC
include $(MAKERULES)

rahavd@ubuntu:~$ cd tinyos-main/apps/Blink/
rahavd@ubuntu:~/tinyos-main/apps/Blink$
Our first TinyOS experience

```
rahav@ubuntu:~/Blink$ make telosb install
mkdir -p build/telosb
  compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -m disable-hwmul -fnesc-separator=-- -Wall -Wc
22 -DIDENT_APPNAME="BlinkAppC" -DIDENT_USERNAME=rahav -DIDENT_HOSTNAME=ubuntu
nkAppC.nc -lm
  compiled BlinkAppC to build/telosb/main.exe
  2648 bytes in ROM
  54 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
  writing TOS image
cp build/telosb/main.ihex build/telosb/main.ihex.out
  found mote on /dev/ttyUSB0 (using bsl, auto)
  installing telosb binary using bsl
tos-bsl --telosb -c /dev/ttyUSB0 -r -e -i -p build/telosb/main.ihex.out
MSP430 Bootstrap Loader Version: 1.39-goodfet-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 ...
2680 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out build/telosb/main.ihex.out
rahav@ubuntu:~/Blink$
```
Our first TinyOS experience

```bash
rahav@ubuntu:-/Blink$ motelist
Reference Device Description
-------- -------- -------------------------
M4AGICPC /dev/ttyUSB0 Moteiv tmote sky

rahav@ubuntu:-/Blink$ make telosb install bsl, /dev/ttyUSB0
mkdir -p build/telosb
  compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -fnesc-separator=__ -Wall -Wl
22 -DIDENT_APPNAME="BlinkAppC" -DIDENT_USERNAME="rahav" -DIDENT_HOSTNAME="ubuntu"
linkAppC.nc -lm
  compiled BlinkAppC to build/telosb/main.exe
    2648 bytes in ROM
    54 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
  writing TOS image
cp build/telosb/main.ihex build/telosb/main.ihex.out
  installing telosb binary using bsl
tos-bsl --telosb -c /dev/ttyUSB0 -r -e -I -p build/telosb/main.ihex.out
MSP430 Bootstrap Loader Version: 1.39-goodfet-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 ...
2680 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out build/telosb/main.ihex.out
rahav@ubuntu:-/Blink$
```
Project 0

- Install your favorite VM or your favorite host OS
- Install ubuntu 12.x or 13.x
- Install TinyOS 2.1.2
- Run `tos-check-env`
- Build Blink
- Answer the questions on the following slide
- Due Feb. 5
- Email me the following screen captures
  - Full window capture of ubuntu running inside a VM and Terminal open, listing the /dev/tty* files
  - What you typed and compilation results of Blink
  - Full browser window (including all panes) showing BlinkAppC documentation as it was generated by the nesC docs system
How to earn an A on project 0?

- Example for first two deliverables are in this lecture
- Here is the third deliverable:
How to get Help

- TinyOS Documentation Wiki: http://docs.tinyos.net
- Text book, TinyOS Programming
  http://www.amazon.com/TinyOS-Programming-Philip-Levis/dp/0521896061
- TinyOS Programming Manual: 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/tinywiki/index.php/TinyOS_Tutorials
How to get Help

nesdoc: annotated API for all interfaces and components in TinyOS:
http://docs.tinyos.net/tinywiki/index.php/Source_Code_Documentation

The nesC code:
call Leds.led1Toggle();    // Yellow

toggle LED 1; if it was off, turn it on, if was on, turn it off. The color of this LED depends on the platform.
How to get Help

- TinyOS Enhancement Protocols (TEP): formal documentation for TinyOS features: http://docs.tinyos.net/tinywiki/index.php/TEPs
- My office hours (Bryan 502D)
  - By email rahav.dor@wustl.edu
  - Subject line MUST start with [CSE467] or [CSE467S]
Reminders

- Team up
- Projects need approvals
  - Iterations may be needed before you can work on the proposals
- Design is part of a successful project (and life after school)
- Start early and spread the work evenly throughout the semester
- I am here to make you successful and enjoy the course
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
Terms we need to know

- **Synchronous**
  - Agreed *timing* for the sending of ones and zeroes (bits)
  - Requires sender-receiver coordination

- **Asynchronous**
  - Ones and zeros can be sent at any point in time

- **Baud**
  - symbols / second

- **BUS**
  - Transfers data between components

- **UART (Universal Asynchronous Receiver/Transmitter)**
  - Translates data between parallel and serial forms
  - Universal = format and transmission speeds are configurable
Terms...

- **Multi-master**
  - Multiple nodes can be the master (can initiate transfer)
  - E.g. allows DMA to work without the CPU

- **SPI (Serial Peripheral Interface)**
  - Synchronous serial bus (pronounced as either ess-pee-eye or spy)

- **I²C**
  - Synchronous two wire multi-master bus
  - For low-speed peripherals

- **ADC**
  - Analog to Digital Converter

- **DAC**
  - Digital to Analog converter

- **JTAG**
  - Today JTAG is used as the primary means of accessing sub-blocks of ICs, or programming devices
Interrupts
Available Hardware

- Motes, sensor boards, sensors, etc.
- With proper justification and time we will be happy to buy additional equipment
- Most up to date list here: http://wsn.cse.wustl.edu/index.php/Equipment_for_course_projects
Tmote Sky (aka TelosB)

- IEEE 802.15.4 Radio
  - 250kbps
- TI MSP430 microcontroller
  - 16MHz, 16 MIPS, 10kB RAM
- Integrated antenna & USB interface
- SBT80: light, IR, acoustic, temperature, magnetometer, and accelerometer sensors, all of dubious accuracy
- Low power utilization
  - 1.8mA/5.1µA
- So how many hours can it work on two AA batteries?

Wifi’s 802.11n max is between 54 Mbits/s to 600 Mbits/s
Tmote front
Tmote back

Texas Instruments
MSP430 F1611 microcontroller

2-pin SVS connector

USB Flash (2kB)

32kHz oscillator

48-bit silicon serial ID

ST Code Flash (1MB)
MICAz Mote (MPR2400)

- 802.15.4 radio, 250 kbps, 2.4 to 2.48 GHz (ISM)
- 128KB Instruction EEPROM
- 4KB config. EEPROM
- Atmel ATmega128L μP, 7.3827MHz (8 MIPS)
- 3 LEDs: Red, Green, Yellow
- 8 ADC
- UART, SPI, I²C Bus
- 51 pin I/O Connector
- 512KB External Flash Memory
- 2 AA batteries
- To Light, Temp, Barometric, Accel, Acoustics, Magnetic, etc. Or Programming Board
MICAz Programming board (MIB510)

- Serial interface to laptop
- Mote JTAG
- MICA2Dot interface
- Mote interface
- ISPJTAG
- Block data to laptop
- 5V Power
- Reset
MTS310CA Sensor board

- 4.6KHz Speaker
- 2 Axis Accelerometer
- Magnetometer
- 51 pin MICA2 Interface
- Tone Detector
- Light and Temperature
- Microphone
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](http://openwrt.org))
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
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
App Bootstrapping

- Each app has a “main” configuration file 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 voidbooted();
}
```
App Bootstrapping

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

```plaintext
Configuration RoutingAppC {
    implementation {
        components RoutingC;
        components MainC;
        ...
        RoutingC.Boot -> MainC;
    }
}

module RoutingC {
    uses interface Boot;
    implementation {
        event void Boot.booted() {
            // Initialize here
        }
        ...
    }
}
```
TinyOS Component model

Upper Layer

Provides Split Control
Provides Send
Provides Receive

ActiveMessageC
TinyOS Component model

command Start()

ActiveMessageC

Provides Split Control
Provides Send
Provides Receive

Upper Layer
TinyOS Component model

event Startdone()
Components != Objects

- AppLogicC
- NetworkHandlerC
- AnotherHandlerC

- ActiveMessageC
- ActiveMessageC
- ActiveMessageC

✗
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);
}
```

Points to any data type
Modules

- Modules provide the implementation (code, logic) of one or more interfaces
- They may consume (use) other interfaces:

```plaintext
module ExampleModuleC {
    provides interface SplitControl;
    uses interface Receive;
    uses interface Receive as OtherReceive;
}
implementation {
    ...
}

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

The 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:

```c
... implementation {
    uint8_t localVariable;
    void increment(uint8_t amount);  // declaration

    ...

    void increment(uint8_t amount) {  // implementation
        localVariable += amount;
    }
}
```

Modules: Tasks

- Look a lot like functions, except:
  - Prefixed with task
  - Can’t return anything
  - Can’t 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
  ```
  post handlePacket();
  ```
- Can post from within commands, events, and other tasks
- For longer computation, background data processing, etc.
- Can be preempted by interrupts, but not by other tasks
  - So, design consideration: Break a series of long operations
- TinyOS guarantees that task will *eventually* run
  - Default scheduling policy: FIFO
- Cannot post the same task while its on the queue
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`

```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:

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

- Event handlers are invoked using the signal keyword:

```java
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 ExampleModuleC {
    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;
    }
    ...
}
```
configuration NetworkHandlerC {  
  provides SplitControl;  
}  

implementation {  
  components NetworkHandlerC as NH,  
  ActiveMessageP as AM;  

  // NH.Receive -> AM;  
  // SplitControl = NH.SplitControl;  
  NH.Receive -> AM;  
  SplitControl = NH;  
}
Quote from the people who made it

- Modules implement program logic*1
- Configurations compose modules into larger abstractions*1

*1 Philip Levis
Race conditions

- Use atomic blocks to avoid race conditions, they do not get preempted.

```plaintext
implementation {
    uint8_t sharedCounter;

    async event void Alarm.fired() {
        ...
        sharedCounter++;
    }

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

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

```
implementation {
  uint8_t sharedCounter;

  task void incrementCounter() {
    sharedCounter++;
  }

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

  event void Receive.receive(...) {
    ...
    sharedCounter++;
  }
}
```
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 {
  norace uint8_t sharedCounter;

  async event void Alarm1.fired() {
    sharedCounter++;
    call Alarm2.start(200);
  }

  async event void Alarm2.fired() {
    sharedCounter--;
    call Alarm1.start(200);
  }
}
```

Race condition is impossible; these Alarms are mutually exclusive
TOSThreads

- New in TinyOS 2.1: the TOSThreads 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 everything: 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 TOSThreads Tutorial [http://docs.tinyos.net/index.php/TOSThreads_Tutorial](http://docs.tinyos.net/index.php/TOSThreads_Tutorial) or TEP 134
Example-Blink (...\tinyos-main\apps\Blink)

Three files:

1. Makefile
2. BlinkC.nc (module = logic)
3. BlinkAppC.nc (configuration = wiring)
COMPONENT=BlinkAppC
include $(MAKERULES)
BlinkC.nc (the logic or code)

#include "Timer.h"

module BlinkC {
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Timer<TMilli> as Timer2;
    uses interface Leds;
    uses interface Boot;
}

Implementation {
    event void Boot.booted() {
        call Timer0.startPeriodic(250);
        call Timer1.startPeriodic(500);
        call Timer2.startPeriodic(1000);
    }
    event void Timer0.fired() {
        call Leds.led0Toggle();
    }
    event void Timer1.fired()  {
        ...
    }
    event void Timer2.fired()  {
        ...
    }
}
configuration BlinkAppC {
}

implementation {
    components MainC, BlinkC, LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;
    components new TimerMilliC() as Timer2;

    BlinkC -> MainC.Boot;
    BlinkC.Timer0 -> Timer0;
    BlinkC.Timer1 -> Timer1;
    BlinkC.Timer2 -> Timer2;
    BlinkC.Leds -> LedsC;
}
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- 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 other news: you’re almost certain to use code that incorporates these features
Creating new interfaces to support different data types can get redundant fast.

```java
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:

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

module SixteenBitSensorP {
    provides interface Read<uint16_t>;
}

module BooleanSensorP {
    provides interface Read<bool>;
}
```
Fan-In: No big deal

- Behavior of shared component depends on its design. Can return FAILURE (requests rejected), buffer them, or screw the data up.

Diagram:
- AppLogicP
  - uses Receive
- NetworkHandlerP
  - uses Receive
- AnotherHandlerP
  - uses Receive
- RadioP
  - provides Receive

Many-to-one calls may work like you’d expect.
Fan-Out: Bad things happen

return &buffer1;

AppLogicP
uses Receive

return &buffer2;

NetworkHandlerP
uses Receive

return &buffer3;

AnotherHandlerP
uses Receive

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

provides Receive

RadioP
Fan-Out: When 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:

```c
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(...);
        ...
    }
    ...
}
```

![Diagram showing network handler and radio components]
Parameterized Wiring

- The idea works in concept, but isn’t maintainable in practice
- nesC can yield the desired behavior in a 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:

```c
AppLogicP -> RadioP.Receive[unique("RadioP")];
// unique("RadioP") expands to 0

NetworkHandlerP -> RadioP.Receive[unique("RadioP")];
// unique("RadioP") expands to 1

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

- We can use an array. But of what size?

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

  ...
}
uniqueCount("strName") expands to # of times unique(X) appears in the application
```
If you provide a parameterized interface and signal an event on it, you must also give a default event handler:

```
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
  }
  ...
}
```
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- Basic nesC syntax
- Advanced nesC syntax
- Network communication
- Sensor data acquisition
- Debugging tricks and techniques
**error_t data type**

- TinyOS defines a special `error_t` data type that describes several error codes
- Often given as return values to commands or event handlers
- Commonly used values:
  - SUCCESS (everything is OK)
  - FAIL (general error)
  - 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 can have a unique 16-bit address (am_addr_t) specified on the make command `make install.[address] platform`
- Two special constants available for code authoring:
  - 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
  - 0 - 126 are reserved for TinyOS internal use
TinyOS Active Messages (AM)

- `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, 114 max)

```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;
```

<table>
<thead>
<tr>
<th>Header</th>
<th>Payload (TOSH_DATA_LENGTH)</th>
<th>Footer</th>
<th>Metadata</th>
</tr>
</thead>
</table>

Washington University in St. Louis
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 start...() command that returns immediately
  - An event is signaled when it’s actually done

```cpp
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 how TinyOS started processing the request

Error code here indicates how TinyOS completed processing the request
Active Message interface

send is a Split-phase operation

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

Fired on another mote when packet arrives
Packet interface

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();
}
AMPacket interfaces

For querying packets

```cpp
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);
}
```
Other networking interfaces

```c
interface PacketAcknowledgements {
    async command error_t requestAck(message_t* msg);
    async command error_t noAck(message_t* msg);
    async command bool wasAicked(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
nx_ (network external) 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 \textbf{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

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 to store the packet’s contents
- Get the packet’s payload using the `Packet` interface
  - Note the `sizeof()` function
  - Cast it to your message type
- Set 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
task 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:

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

```c
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.

```cpp
components ActiveMessageC;
MyAppP.RadioPowerControl -> ActiveMessageC;
```

- Think about design when you have both radio and periodical sensing (how does your boot process should be processed?)
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))

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

Collection Tree Protocol (CTP) Initializing CTP
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
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.wasAcked() only tells you if the packet made it to the first hop
Dissemination: Basic Operation
For More Information

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.SerialPowerControl1 -> SerialActiveMessageC;
```
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
  ```
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);
}
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- 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
configuration MyAppC {
}
implementation {
    components MyAppP;
    components new AccelXC();
    // X axis accelerometer component
    // defined by mts300 sensorboard
    MyAppP.AccelX -> AccelXC;
    ...}

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
    }
    ...
}
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

```java
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)
Lecture topics

- Embedded Computing projects
- Coding style, coherent and useful programming
- Installing TinyOS and compiling your first app
- Introduction to motes
- 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: `CC2420CHANNEL=xx` make `[platform]`
- Makefile: `PFLAGS = -DC2420_DEF_CHANNEL=xx`
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
LEDs

- The easiest way to display runtime information is to use the mote’s LEDs:

```c
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
printf()

- 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]]
BaseStation

- 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
  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/usctopologies.html)

<table>
<thead>
<tr>
<th>Link</th>
<th>Gain (dBm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>-90.80</td>
</tr>
<tr>
<td>1 0</td>
<td>-95.95</td>
</tr>
<tr>
<td>0 2</td>
<td>-97.48</td>
</tr>
<tr>
<td>2 0</td>
<td>-102.10</td>
</tr>
<tr>
<td>0 3</td>
<td>-111.33</td>
</tr>
<tr>
<td>3 0</td>
<td>-115.49</td>
</tr>
<tr>
<td>0 4</td>
<td>-104.82</td>
</tr>
<tr>
<td>4 0</td>
<td>-110.09</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)
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

- **MSPsim**: MSP430 (TelosB) emulator
  [http://www.sics.se/project/mspsim/](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

- http://www.cs.utah.edu/~coop/safetinyos/

Demo: Putting it All Together
Demo Example

- See the zip file on the course site