TinyOS Tutorial

CSE521S, Spring 2015
Dolvara Gunatilaka

Based on tutorial by Mo Sha, Rahav Dor
TinyOS community

- [http://www.tinyos.net/](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](https://github.com) 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](https://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 lowpan/RPL IPv6 stack.
- Support for the ucmi mini platform and ATmega128RFA1 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
WSN Architecture

Sensors → Base station → gateway → Internet
Lecture topics

- 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 Installation

- TinyOS 2.1.2 Installation
  
  http://tinyos.stanford.edu/tinyos-wiki/index.php/Installing_TinyOS#Officially_Supported_Methods

- Linux: .rpm and .deb packages for Fedora and Ubuntu
  *Recommend debian system installation on Ubuntu

- Window: .rpm package, uses Cygwin to emulate Linux software layer

- OS X: Unofficially supported
Installation : Debian Package

- Install TinyOS compiler
  - Edit /etc/apt/sources.list.d/tinyprod-debian.list
    
    ```
    sudo vim /etc/apt/sources.list.d/tinyprod-debian.list
    ```
  - Add two repositories
    
    ```
    deb http://tinyprod.net/repos/debian squeeze main
    deb http://tinyprod.net/repos/debian msp430-46 main
    ```
  - Install the repository security key
    
    ```
    gpg --keyserver keyserver.ubuntu.com --recv-keys 34EC655A
    gpg -a --export 34EC655A | sudo apt-key add -
    ```
  - Install packages
    
    ```
    sudo apt-get update
    sudo apt-get install nesc tinyos-tools msp430-46 avr-tinyos
    ```
Installation : Debian Package

- Get TinyOS
  - Download
    ```
    wget http://github.com/tinyos/tinyos-release/archive/tinyos-2_1_2.tar.gz
    ```
  - Upzip
    ```
    tar xf tinyos-2_1_2.tar.gz
    ```

- Setup TinyOS environment
  - Edit .bashrc
  - Add the following lines:

    ```
    # Setup the environment variables needed by the TinyOS make system
    export TOSROOT="$HOME/tinyos-main"
    echo "setting up TinyOS source path to $TOSROOT"
    export TOSDIR="$TOSROOT/tos"
    export CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java/tinyos.jar:. 
    export MAKERULES="$TOSROOT/support/make/Makerules"
    export PYTHONPATH=$PYTHONPATH:$TOSROOT/support/sdk/python
    ```
Connect Motes

- To check your TinyOS installation
  - `tos-check-env`
  - WARNING about graphviz / JAVA version
- Learn which devices are attached to Linux
  - `ls /dev/tty*`
- When mote is connected, which port a mote attached to
  - `motelist`

```
dolvaragunatilaka@ubuntu:/opt/tinyos-2.1.2/apps$ motelist
Reference  Device                  Description
----------  ------------------------  -----------------------
M4A6J3UG    /dev/ttyUSB0            Moteiv tmote sky
```

- Give access permission
  - `sudo chmod 666 /dev/<devicename>`
TinyOS includes Makefiles to support the build process

Create Makefile in your app directory

COMPONENT=[MainComponentAppC]# the name of your AppC file include $(MAKERULES)
To compile an app without installing it on a mote, run in the app directory:

```bash
make [platform]
```

- platform defined in `$TOSROOT/tos/platforms`
- `mica2`, `micaz`, `telosb`

- `make clean`

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

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

  - node ID: 0–255, for radio transmissions
  - platform: 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
Useful Example 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

- .nc to .c and .c to binary
- set AM address and node ID
- program mote
Homework

- Install TinyOS 2.1.2
- Build Blink
  - $TOSROOT/apps/Blink
  - Simple application that blinks the 3 mote LEDs
  - Tests boot sequence and millisecond timers.

** 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
- 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
- TinyOS Enhancement Protocols (TEP): formal documentation for
  TinyOS features: http://docs.tinyos.net/tinywiki/index.php/TEPs
Lecture topics

- 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
Available Hardware

- Motes, sensor boards, sensors, etc.
  - Motes: Telosb, MICAz

- Most up-to-date list:
Tmote Sky (Telosb)

- IEEE 802.15.4 Radio (2.4 GHz)
  - 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
MICAz

- IEEE 802.15.4 (2.4 to 2.48 GHz ISM band)
- 250 kbps data rate
- MPR2400 based on the Atmel ATmega128L
- 51-pin expansion connector
- MEMSIC Sensor Boards
MICAz Programming board (MIB510)

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

- 4.6KHz Speaker
- 51 pin MICA2 Interface
- Tone Detector
- Light and Temperature
- Microphone
- 2 Axis Accelerometer
- Magnetometer
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

- 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 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
Components

- NesC application consists of one or more components
- A component *provides* and *uses* interfaces
- Bidirectional interfaces
- 2 types of components in NesC
  - Modules: implementation of interfaces
  - Configurations: Wiring interfaces of a component to interfaces provided by other components
TinyOS Component model

command Start()

ActiveMessageC

Provides Split Control
Provides Send
Provides Receive

Upper Layer
TinyOS Component model

event StartDone()

Upper Layer

Provides Split Control
Provides Send
Provides Receive

ActiveMessageC
Interfaces

- List of one or more functions
  - Events
  - Commands
- Users call commands and providers signal events.

```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);
}
```
App Bootstrapping

- Each app has a “main” configuration file which wires together the app’s constituent components.
- TinyOS includes a MainC component which provides the Boot interface.
- 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
    }
    ...
}
Modules provide the implementation (logic) of one or more interfaces

They may 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 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`
  - Cannot return anything
  - Cannot accept any parameters

```plaintext
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!
    }
}
```
Task Scheduling

- Tasks are scheduled using the `post` keyword
  ```
  post handlePacket();
  ```
- Can post from within commands, events, and other tasks
- Can be preempted by interrupts, but not by other tasks
  - Design consideration: Break a series of long operations into multiple tasks
- TinyOS guarantees that task will *eventually* run
- 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:

```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:

```c
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;
    }
    ...
}
```
Configurations

- Connect components / wire interfaces

```java
configuration NetworkHandlerC
{
    provides interface SplitControl;
}

implementation
{
    components NetworkHandlerC as NH,
        ActiveMessageP as AM;
    // NH.Receive -> AM.Receive;
    // SplitControl = NH.SplitControl;
    NH.Receive -> AM;
    SplitControl = NH;
}
```

- Wiring is unambiguous, don’t Have to include interface name
- Rename component, more convenient
- -> wire to external interface
- = wire to internal interface
Concurrency Model

- Task
  - deferred execution, run to completion
  - Does not preempt each other

- Event handler
  - Signal asynchronously by HW interrupt
  - Preempt task and other event handler
  - Command/event uses `async` keyword

- Synchronous/Asynchronous code
Race conditions

1. Keep code synchronous (update shared state using task)
   - If timing isn’t crucial, defer code to tasks (synchronous)

```c
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 executed later
Race Condition

2. Atomic Block
   - Interrupts are disabled – use sparingly and make it short

```c
implementation {
    uint8_t sharedCounter;

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

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

- Compiler detects race condition -> false positive
- Absolutely sure that there’s no race condition (or don’t care if there is), use the `norace` keyword:

```cpp
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) or TEP 134
# Example: Blink

## Module

```plaintext
#include "Timer.h"
module BlinkC {
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Leds;
    uses interface Boot;
}
implementation {
    event void Boot.booted() {
        call Timer0.startPeriodic( 250 );
        call Timer1.startPeriodic( 500 );
    }
    event void Timer0.fired() {
        call Leds.led0Toggle();
    }
    event void Timer1.fired() {
        call Leds.led1Toggle();
    }
}
```

## Configuration

```plaintext
configuration BlinkAppC {
}
implementation {
    components MainC, BlinkC, LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;

    BlinkC -> MainC.Boot;
    BlinkC.Timer0 -> Timer0;
    BlinkC.Timer1 -> Timer1;
    BlinkC.Leds -> LedsC;
}
```

## Makefile

```plaintext
COMPONENT=BlindleAppC
include $(MAKERULES)
```
Lecture topics

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

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

```plaintext
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

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

![Diagram]

- 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

provides Receive

RadioP

... but what about one-to-many calls?
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.
Consider the following way to avoid fan-out:

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
- Parameterized interface: array of interfaces with index as a parameter

```cpp
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:

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

```c
g
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
  }
  ...
}
```
Unique parameters

- User does not need 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("RadioP")];
// unique("RadioP") expands to 0  XX
```
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
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
Lecture topics

- 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
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;
```
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

```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 how TinyOS started processing the request

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

```c
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);
}
```

send is a Split-phase operation

Fired on another mote when packet arrives
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();
}

Packet interface
AMPacket interfaces

For querying packets

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

Default behavior: no ACKs
Even with ACKs enabled, no automatic retransmissions

```c
interface PacketAcknowledgements {
    async command error_t requestAck(message_t* msg);
    async command error_t noAck(message_t* msg);
    async command bool wasAked(message_t* msg);
}
```
interface PacketLink {
    command void setRetries(message_t *msg, uint16_t maxRetries);
    command void setRetryDelay(message_t *msg, uint16_t retryDelay);
    command uint16_t getRetries(message_t *msg);
    command uint16_t getRetryDelay(message_t *msg);
    command bool wasDelivered(message_t *msg);
}

- Handle retransmissions;
  #define PACKET_LINK
- 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 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
{
    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 *) Packet.getPayload(&output, sizeof(sensor_reading_t));
    reading->temperature = lastTemperatureReading;
    reading->humidity = lastHumidityReading;
    ...
  }
}
```
Finally, use the AMSend interface to send the packet

task void sendData() {
    ...  
    To all nodes
    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.

```
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);

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)
```

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.
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;

Module{
    uses interface SplitControl as RadioPowerControl;
}
Implementation{
    event void Boot.booted()
    {
        call RadioPowerControl.start();
    }
    event void RadioPowerControl.startDone(error_t err) {}
    event void RadioPowerControl.stopDone(error_t err){}
}
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)
- blip: IPv6 stack added to TinyOS 2.1.1 (http://docs.tinyos.net/index.php/BLIP_Tutorial)
Collection Tree Protocol (CTP)
Collection Tree Protocol (CTP)

- Use ETX (expected transmissions) for route selection
- One node as root
- Tree generation
  - Link estimator – determine ETX of links
  - Routing engine – select next-hop by using ETX values
  - Forwarding engine – maintain packet queue / transmit packet
- Address free
Initializing CTP

configuration MyCtpAppC {
} implementation {
components AppLogicP;
components 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();
}
Sending/Receiving Packets

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)

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

```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
Dissemination

- $\$TOSROOT/lib/net$
- Data provider uses DisseminationUpdate interface to disseminate a new value
- Data consumer receives new value through DisseminationValue interface

```java
interface DisseminationUpdate<\texttt{t}>
{
    \texttt{command} \texttt{void} \texttt{change(t* newVal);} \
}

interface DisseminationValue<\texttt{t}>
{
    \texttt{command} \texttt{const t* get();} \
    \texttt{event} \texttt{void changed();} \
}
```
For More Information

- TEP 118: Dissemination (http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html)
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 [platform]
  - Makefile: PFLAGS = -DCC2420_DEF_CHANNEL=xx
Mote 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

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
Base Station

- TinyOS application
  - $TOSROOT/apps/BaseStation
- Radio to serial / serial to radio
- JAVA Listen tool – sniff and print packet
  - $ java net.tinyos.tools.Listen
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
```
JAVA messaging toolchain

- net.tinyos.message
  - MessageListener
    - Provide messageReceived() - signal upon packet reception
  - Message class
    - Get/set data in a packet
  - Moteif class
    - Interface for sending or receiving packet to/from a mote
    - Connect over serial port

- Net.tinyos.packet
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
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
Lecture topics

- 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 provides 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**

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

```plaintext
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
External Sensors

Figure 20: Functionality of the 10-pin expansion connector (U2). Alternative pin uses are shown in gray.
External Sensors

- Wire directly into HplMsp430GeneralIOC component

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

```java
interface HplMsp430GeneralIOC {
    command void makeInput();
    command void makeOutput();
    command bool get();
    command void clr();
    command void set();
    command void toggle();
    ...
}
```

- I²C: read TEP 117 (Low-Level I/O)
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
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
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:

```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

Configuration file:
- Include PrintfC or SerialPrintfC component
- printf.h

- Note: this automatically turns on SerialActiveMessageC subsystem

- Included PrintfClient utility displays printed messages to console
  - i.e. java net.tinyos.tools_PRINTFClient -comm serial@<serial port>:<mote>
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 net.tinyos.tools.Listen [-comm serial@[port]:[speed]]
```

- Extremely helpful for figuring out what data is being sent!
Simulate TinyOS applications

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
    - $TOSROOT/apps/tutorials/RssiDemo
  - Samples included in TinyOS ($TOSDIR/lib/tossim/topologies)
  - Generate one based on various parameters

```
0 1 -90.80
1 0 -95.95
0 2 -97.48
2 0 -102.10
0 3 -111.33
3 0 -115.49
0 4 -104.82
4 0 -110.09
...
```

(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
  http://compilers.cs.ucla.edu/avrora/
- MSPsim: MSP430 (TelosB) emulator
  http://www.sics.se/project/mspsim/

- Profile and benchmark apps, monitor packet transmissions, or interface with gdb
- Slower than TOSSIM, but highly accurate
Demo: Putting it All Together
Demo

- DemoMessage.h
- DemoAppC.nc
- DemoP.nc
- Makefile
- JAVA
  - Main.java
  - Makefile

DemoMessage.h

```c
#ifndef __DEMOAPP_H
#define __DEMOAPP_H

enum
{
    AM_DEMO_MESSAGE = 150,
};
typedef nx_struct demo_message
{
    nx_uint16_t photoReading;
} demo_message_t;
#endif // __DEMOAPP_H
```

Makefile

```makefile
COMPONENT=DemoAppC
include $(MAKERULES)
```

JAVA

- Main.java
- Makefile
#include "DemoApp.h"

configuration DemoAppC
{
}
implementation
{
    components DemoP, MainC, new HamamatsuS10871TsrC() as PhotoC;
    components ActiveMessageC;
    components new AMSenderC(AM_DEMO_MESSAGE), new AMReceiverC(AM_DEMO_MESSAGE);
    components LedsC;
    components new TimerMilliC();
    components SerialActiveMessageC as SerialC;

    DemoP.Boot -> MainC;
    DemoP.Photo -> PhotoC;
    DemoP.RadioControl -> ActiveMessageC;
    DemoP.AMSend -> AMSenderC;
    DemoP.Receive -> AMReceiverC;
    DemoP.Packet -> ActiveMessageC;
    DemoP.SerialControl -> SerialC;
    DemoP.SerialAMSsend -> SerialC.AMSend[AM_DEMO_MESSAGE];
    DemoP.SerialPacket -> SerialC;
    DemoP.Leds -> LedsC;
    DemoP.Timer -> TimerMilliC;
}
module DemoP
{
    uses interface Boot;

    uses interface Read<uint16_t> as Photo;

    uses interface SplitControl as RadioControl;
    uses interface AMSend;
    uses interface Receive;
    uses interface Packet;

    uses interface SplitControl as SerialControl;
    uses interface Packet as SerialPacket;
    uses interface AMSend as SerialAMSend;

    uses interface Leds;
    uses interface Timer<TMilli>;
}

implementation
{
    message_t buf;
    message_t *receivedBuf;

    task void readSensor();
    task void sendPacket();
    task void sendSerialPacket();
event void Boot.booted()
{
    call RadioControl.start();
    call SerialControl.start();
}

event void RadioControl.startDone(error_t err)
{
    if(TOS_NODE_ID == 0)
        call Timer.startPeriodic(256);
}

event void Timer.fired()
{
    post readSensor();
}

event void RadioControl.stopDone(error_t err){}

event void SerialControl.startDone(error_t err){}

event void SerialControl.stopDone(error_t err){}
task void readSensor()
{
    if(call Photo.read() != SUCCESS)
        post readSensor();
}

event void Photo.readDone(error_t err, uint16_t value)
{
    if(err != SUCCESS)
        post readSensor();
    else
    {
        demo_message_t * payload = (demo_message_t *)call
        Packet.getPayload(&buf,sizeof(demo_message_t));
        payload->photoReading = value;
        post sendPacket();
    }
}

task void sendPacket()
{
    if(call AMSend.send(AM_BROADCAST_ADDR, &buf, sizeof(demo_message_t)) != SUCCESS)
        post sendPacket();
}

event void AMSend.sendDone(message_t * msg, error_t err)
{
    if(err != SUCCESS)
        post sendPacket();
}
event message_t * Receive.receive(message_t * msg, void * payload, uint8_t len) {
    demo_message_t * demoPayload = (demo_message_t *)payload;
    call Leds.set(demoPayload->photoReading / 200);
    receivedBuf = msg;
    post sendSerialPacket();
    return msg;
}

task void sendSerialPacket() {
    if(call SerialAMSend.send(AM_BROADCAST_ADDR, receivedBuf, sizeof(demo_message_t))! = SUCCESS)
        post sendSerialPacket();
}

event void SerialAMSend.sendDone(message_t* ptr, error_t success) {
    if(success!=SUCCESS)
        post sendSerialPacket();
}
BUILD_EXTRA_DEPS = Main.class

Main.class: DemoAppMsg.java
   javac *.java

DemoAppMsg.java: ../DemoApp.h
   mig java -target=null -java-classname=DemoAppMsg ../DemoApp.h
demo_message -o @$

clean:
   rm -f DemoAppMsg.java  *.class
```java
import java.io.*;
import net.tinyos.message.*;

public class Main implements MessageListener {
    MoteIF mote;

    public Main() {
        try {
            mote = new MoteIF();
            mote.registerListener(new DemoAppMsg(), this);
        } catch (Exception e) {} 
    }

    public void messageReceived(int dest, Message m) {
        DemoAppMsg msg = (DemoAppMsg)m;
        String output = msg.get_photoReading();
        System.out.println("reading: " + output);
    }

    public static void main(String args[]) {
        new Main();
    }
}
```