CSE160: Computer Networks

TinyOS and TOSSIM Tutorial

2020-08-27

Professor
Alberto E. Cerpa

Thanks to David Culler, Stephen Dawson-Haggerty, Omprakash Gnawali, David Gay, Philip Levis, Razvan Musaloiu-E, Keving Klues, John Regehr and more…

Outline

- Overview
- TinyOS and NesC
- TOSSIM
- Debugging

Overview

Sensor code

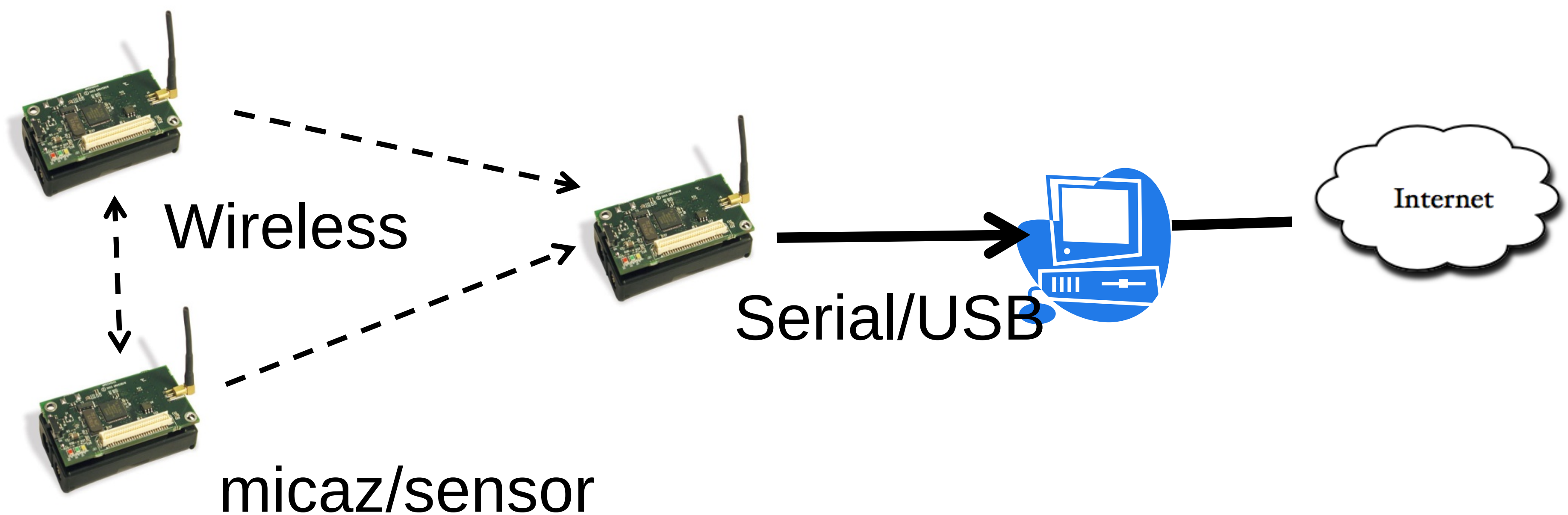
(nesC/TinyOS)

Base station code

(nesC/TinyOS)

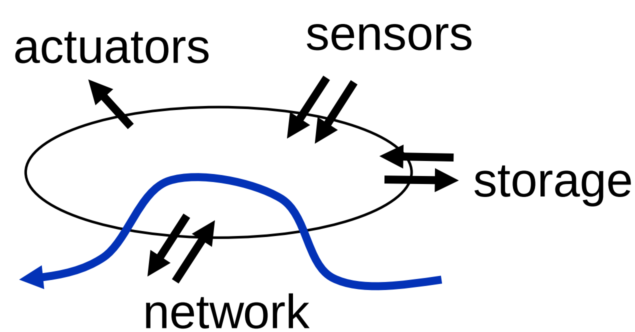
Gateway code

(Java, c, …)



WSNs Features

- Small physical size and low power consumption
- Concurrency-intensive operation
  - multiple flows, not wait-command-respond
- Limited Physical Parallelism and Controller Hierarchy
  - primitive direct-to-device interface
- Diversity in Design and Usage
  - application specific, not general purpose
  - huge device variation
  - efficient modularity
  - migration across HW/SW boundary
- Robust Operation
  - numerous, unattended, critical
  - narrow interfaces



TinyOS 2.0

- Primary Reference:
  - https://github.com/tinyos/tinyos-main/tree/master/doc
  - https://github.com/tinyos/tinyos-main/tree/master/doc/html/tutorial
  - http://nescc.sourceforge.net/papers/nesc-ref.pdf

What is TinyOS?

- An operating system for embedded systems
- Not an operating system for general purpose, it is designed for wireless sensor networks.
  - Official website: http://www.tinyos.net/
- Programming language: NesC (an extension of C)
- It features a component-based architecture.
- Supported platforms include Linux, Windows 10 and MacOSX with VMs.

What is TinyOS? (2)

- An operating system
- An open-source development environment
  - A programming language and model (NesC)
  - A set of services
- Main Ideology
  - HURRY UP AND SLEEP!!
    - Sleep as often as possible to save power
  - Provide framework for concurrency and modularity
    - Commands, events, tasks
  - Interleaving flows, events - never poll, never block

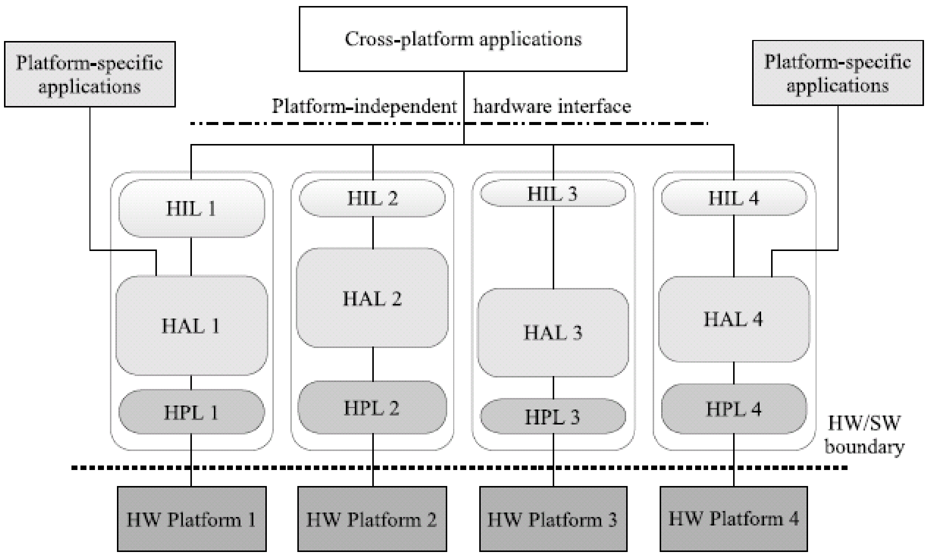
Key TinyOS Concepts

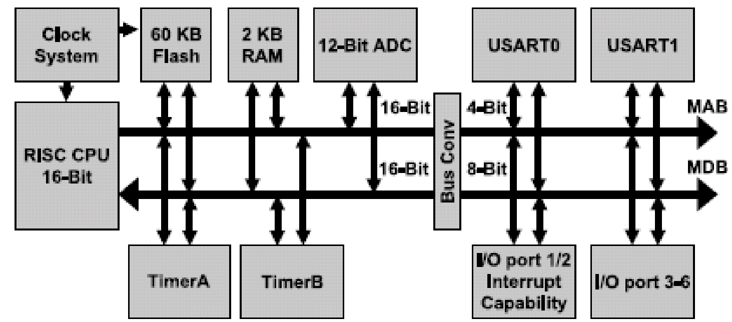
- Application / System = Graph of Components + Scheduler
- *Module*: component that implements functionality directly
- *Configuration*: component that composes components into a larger component by connecting their interfaces
- *Interface*: Logically related collection of commands and events with a strongly typed (polymorphic) signature
  - May be parameterized by type argument
  - *Provided* to components or *Used* by components
- *Command*: Operation performed (called) across components to initiate action.
- *Event*: Operation performed (signaled) across components for notification.
- Task: Independent thread of control instantiated within a component. Non-preemptive relative to other task.
- Synchronous and Asynchronous contexts of execution.

TinyOS Abstraction Architecture

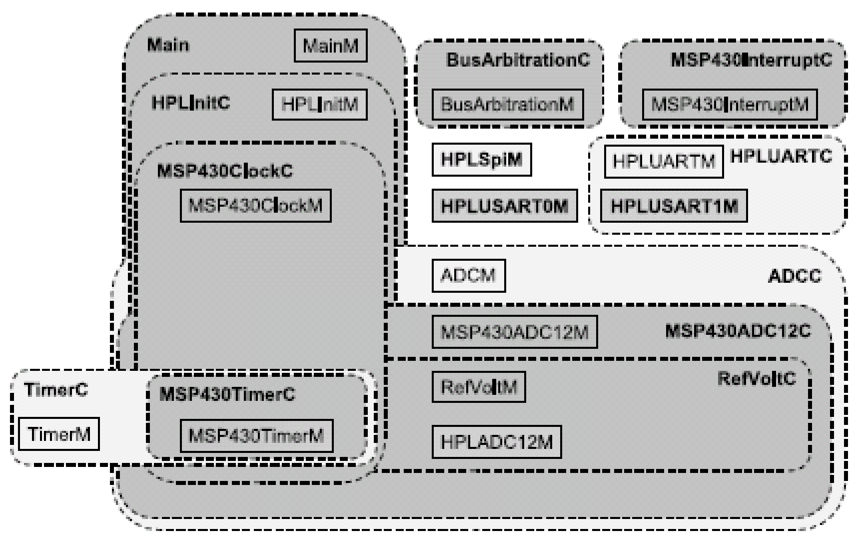
- HPL – Hardware Presentation Layer
  - Components that encapsulate physical hardware units
  - Provide convenient software interface to the hardware.
  - The hardware is the state and computational processes.
  - Commands and events map to toggling pins and wires
- HAL –Hardware Abstraction Layer
  - Components that provide useful services upon the basic HW
  - Permitted to expose any capabilities of the hardware
    - Some platforms have more ADC channels, Timers, DMA channels, capture registers, …
  - Logically consistent, but unconstrained
- HIL – Hardware Independent Layer
  - Components that provide well-defined services in a manner that is the same across hardware platforms.
  - Implement common interfaces over available HAL

Illustration





(a) Functional block diagram of the TI MSP430F149 μC



TinyOS – a tool for defining abstractions

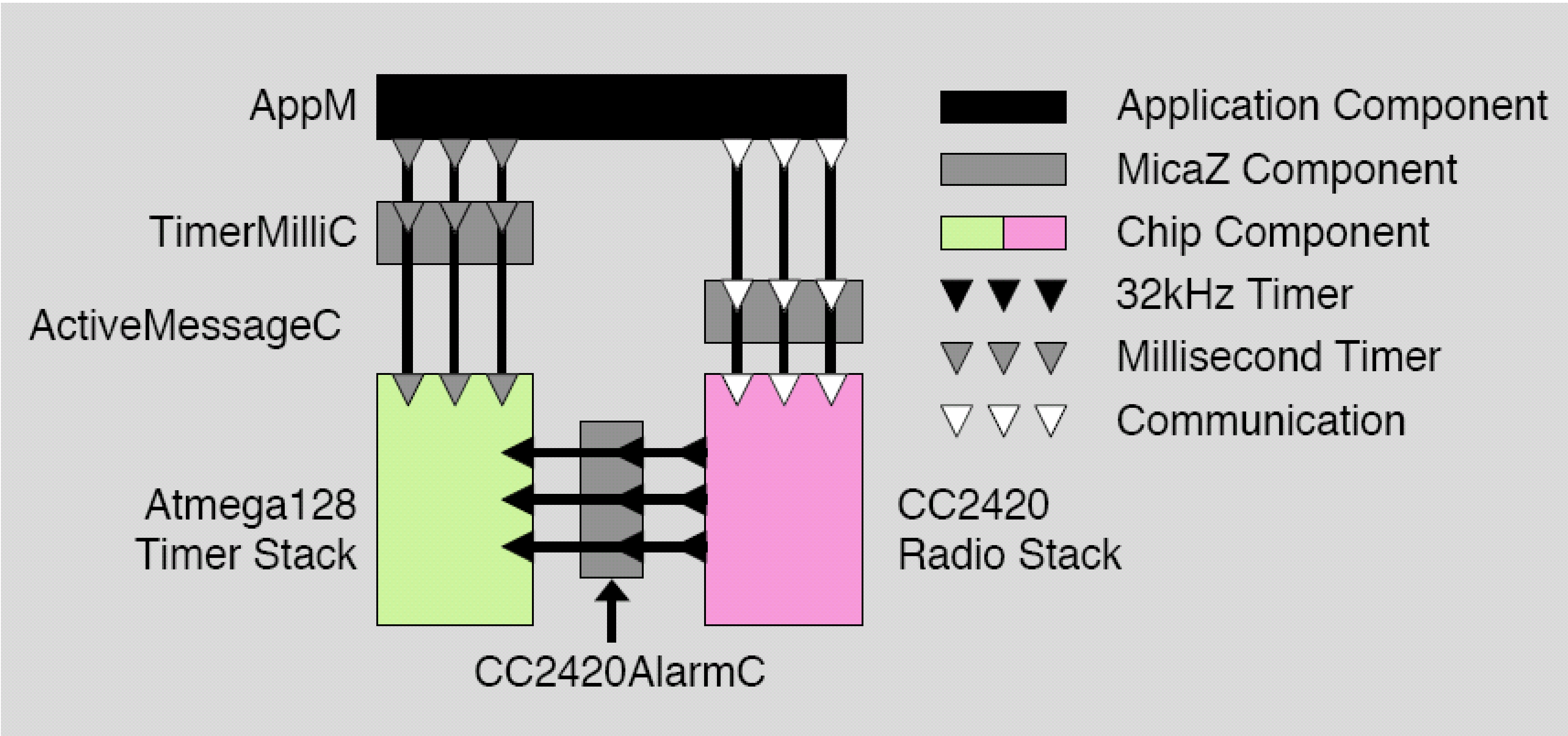
- All of these layers are constructed with the same TinyOS primitives.
- We'll illustrate them from a simple application down.
- Note, components are not objects, but they have strong similarities.
  - Some components encapsulate physical hardware.
  - All components are allocated statically (compile time)
    - Whole system analysis and optimization
  - Logically, all components have internal state, internal concurrency, and external interfaces (Commands and Events)
  - Command & Event handlers are essentially public methods
  - Locally scoped
    - Method invocation and method hander need not have same name (like libraries and objects)
    - Resolved statically by wiring
      - Permits interpositioning

Platform is a collection of Chips

Platform	MCU	Radio	Storage
Mica2	ATMega128	CC1000	at45db
mica	ATMega128	CC2420	at45db
iMote2	pxa27x	CC2420	
Telos	MSP430	CC2420	at45db stm25p
eyes	MSP430	TDA5250	at45db
tinynode	MSP430	XE1205	stm25p

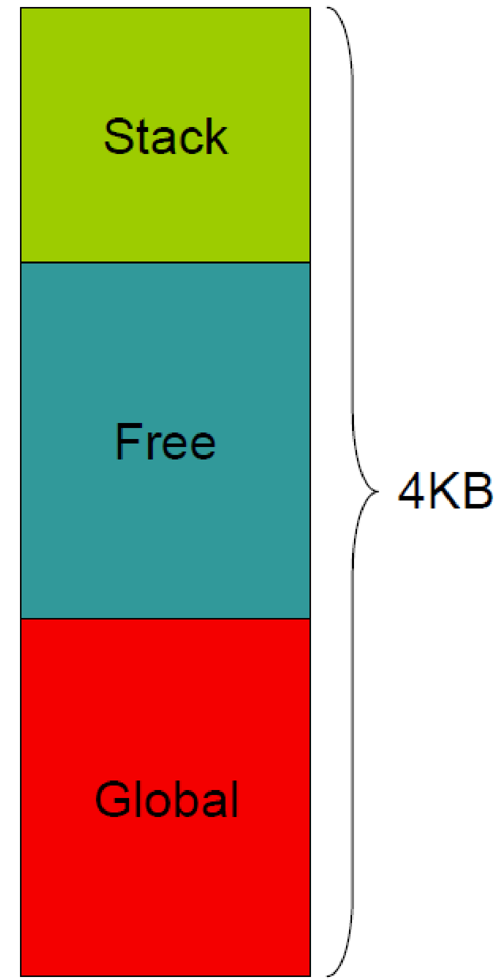
- TinyOS 2.x components provide the capabilities of the chips.
- TinyOS 2.x components glue to together those capabilities into a consistent system.
- TinyOS 2.x components provide higher level capabilities and services

Overall System Configuration (std)



Data Memory Model

- STATIC memory allocation!
  - No heap (malloc)
  - No function pointers
- Global variables
  - Available on a per-frame basis
- Local variables
  - Saved on the stack
  - Declared within a method



Programming Model

- Separation of construction and composition
- Programs are built out of components
  - Libraries and components are written in nesC.
  - Applications are too -- just additional components composed with the OS components
- Each component is specified by an interface
  - Provides "hooks" for wiring components together
- Components are statically wired together based on their interfaces
  - Increases runtime efficiency

Components

- Components **use** and **provide** interfaces, commands, and events
  - Specified by a component's interface
  - The word "interface" has two meanings in TinyOS
- Components implement the events they use and the commands they provide:

Component	Commands	Events
Use	Can call	Must Implement
Provide	Must Implement	Can signal

Types of Components

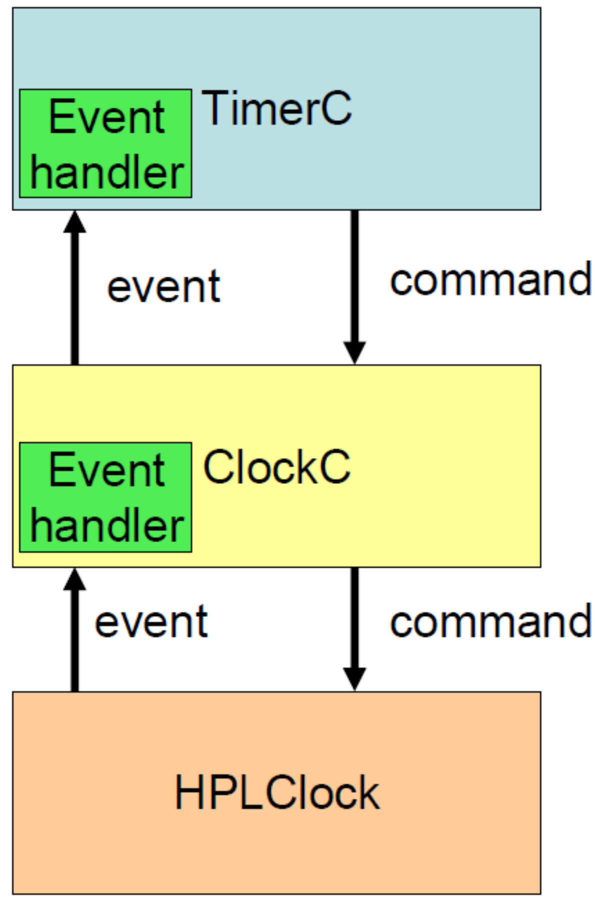
- There are two types of components:
  - **Modules**: Implement the application behavior
  - **Configurations**: Wires components together
- A component does not care if another component is a module or configuration
- A component may be composed of other components

TinyOS Thread Model

- Tasks:
  - Time flexible
  - Longer background processing jobs
  - Atomic with respect to other tasks (single threaded)
  - Preempted by events
- Events:
  - Time critical
  - Shorter duration (hand off to task if need be)
  - Interrupts task
  - Last-in first-out semantics (no priority among events)
- **Do not confuse an event from the NesC event keyword!!**

Component Hierachy

- Components are wired together by connecting users with providers
  - Forms a hierarchy
- Commands:
  - Flow downwards
  - Control returns to caller
- Events:
  - Flow upwards
  - Control returns to signaler
- Events can call Commands but not vice versa



TinyOS Directory Structure

- *tos/system/* - Core TinyOS components. This directory's
  - components are the ones necessary for TinyOS to actually run.
- *tos/interfaces/* - Core TinyOS interfaces, including
  - hardware-independent abstractions. Expected to be heavily used not just by *tos/system* but throughout all other code. *tos/interfaces* should only contain interfaces named in TEPs.
- *tos/platforms/* - code specific to mote platforms, but chip-independent.
- *tos/chips/***/* - code specific to particular chips and to chips on particular platforms.
- *tos/lib/***/* - interfaces and components which extend the usefulness of TinyOS but which are not viewed as essential to its operation.
- *apps/*, *apps/demos*, *apps/tests*, *apps/tutorials*.

Build Tool Chain

Convert NesC into C and compile to exec

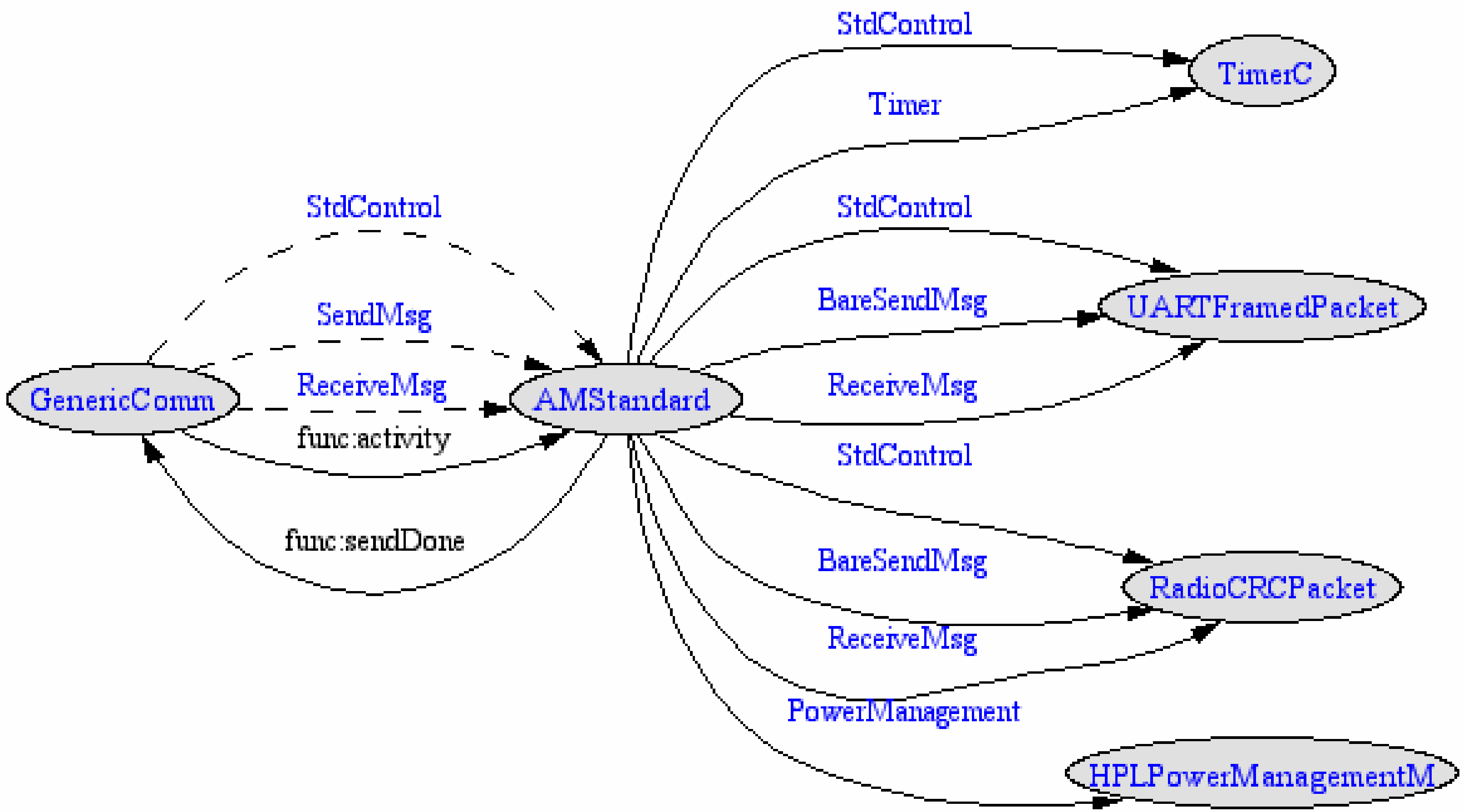
Modify exec with platform-specific options

Set the mote ID

Reprogram the mote

```
wenyuan@tinyos-laptop: /opt/tinyos-1.x/apps/Blink
File  Edit  View  Terminal  Tabs  Help
wenyuan@tinyos-laptop:/opt/tinyos-1.x/apps/Blink$ ls
BlinkM.nc  Blink.nc  build  Makefile  README  SingleTimer.nc
wenyuan@tinyos-laptop:/opt/tinyos-1.x/apps/Blink$ make mica2 install.0
    compiling Blink to a mica2 binary
ncc -o build/mica2/main.exe -Os -board=micasb -target=mica2 -DCC1K_DEF_FREQ=4330
02000 -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x33 -Wnesc-all -finline-limit=100000 -f
nesc-cfile=build/mica2/app.c  Blink.nc -lm
    compiled Blink to build/mica2/main.exe
            1626 bytes in ROM
              49 bytes in RAM
avr-objcopy --output-target=srec build/mica2/main.exe build/mica2/main.srec
make mica2 reinstall.0 PROGRAMMER="STK" PROGRAMMER_FLAGS="-dprog=mib510 -dserial
=/dev/ttyUSB0 -dpart=ATmega128 --wr_fuse_e=ff "
make[1]: Entering directory `/opt/tinyos-1.x/apps/Blink'
    installing mica2 binary
set-mote-id build/mica2/main.srec build/mica2/main.srec.0.out `echo reinstall.0
|perl -pe 's/^reinstall.//; $_=hex if /^0x/i;'`
Could not find symbol TOS_LOCAL_ADDRESS in build/mica2/main.exe, ignoring symbol
.
uisp -dprog=mib510 -dserial=/dev/ttyUSB0 -dpart=ATmega128 --wr_fuse_e=ff  --eras
e --upload  if=build/mica2/main.srec.0.out
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash

Fuse Extended Byte set to 0xff
make[1]: Leaving directory `/opt/tinyos-1.x/apps/Blink'
wenyuan@tinyos-laptop:/opt/tinyos-1.x/apps/Blink$
```

Example Components



Interface Syntax

- Look in **<tos>/tos/interfaces/SendMsg.nc**

```
includes AM;    // includes AM.h located in <tos>\tos\types\

interface SendMsg {
  // send a message
  command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg);

  // an event indicating the previous message was sent
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

- Multiple components may **provide** and **use** this interface

Interface StdControl

- Look in **<tos>/tos/interfaces/StdControl.nc**

```
interface StdControl {

  // Initialize the component and its subcomponents.
  command result_t init();

  // Start the component and its subcomponents.
  command result_t start();

  // Stop the component and pertinent subcomponents
  command result_t stop();
}
```

- Every component *should* **provide** this interface
  - This is good programming technique, it is not a language specification

Module Syntax: Interface

- Look in **<tos>/tos/system/AMStandard.nc**

Component Interface

```
module AMStandard {
  provides {
    interface StdControl as Control;
    interface SendMsg[uint8_t id];        // parameterized by AM ID
    command uint16_t activity();  // # of packets sent in past second
    ...
  }
  uses {
    event result_t sendDone();
    interface StdControl as UARTControl;
    ...
  }
}
implementation {
  ...// code implementing all provided commands and used events
}
```

Module Syntax: Implementation

```
module AMStandard {
  provides { interface SendMsg[uint8_t id]; … }
  uses {event result_t sendDone(); … }
}
implementation {
  task void sendTask() {
    …
    signal sendDone(); signal SendMsg.SendDone(….);
  }
  command result_t SendMsg.send[uint8_t id](uint16_t addr,
    uint8_t length, TOS_MsgPtr data) {
    …
    post sendTask();
    …
    return SUCCESS;
  }
  default event result_t sendDone() { return SUCCESS; }
}
```

Async and Atomic

- Anything executed as a direct result of a hardware interrupt must be declared **async**
  - E.g., **async command** result_t cmdName(…)
  - See **<tos>/tos/system/TimerM.nc** for cross-boundary example
- Variables shared across sync and async boundaries should be protected by **atomic**{…}
  - Can skip if you put **norace** in front of variable declaration (Use at your own risk!!)
  - There are lots of examples in HPL*.nc components found under <tos>/tos/platform (e.g., HPLClock.nc)

Configuration Syntax: Interface

- Look in **<tos>/tos/system/GenericComm.nc**

Component Interface

Component Selection

```
configuration GenericComm {
  provides {
    interface StdControl as Control;
    interface SendMsg[uint8_t id]; //parameterized by active message id
    interface ReceiveMsg[uint8_t id];
    command uint16_t activity();
  }
  uses { event result_t sendDone();}
}
implementation {
  components AMStandard, RadioCRCPacket as RadioPacket, TimerC,
    NoLeds as Leds, UARTFramedPacket as UARTPacket,
    HPLPowerManagementM;
  … // code wiring the components together
}
```

Configuration Syntax: Wiring

- Still in **<tos>/tos/system/GenericComm.nc**

```
configuration GenericComm {
  provides {
    interface StdControl as Control;
    interface SendMsg[uint8_t id]; //parameterized by active message id
    command uint16_t activity(); …
  }
  uses {event result_t sendDone(); …}
}
implementation {
  components AMStandard, TimerC, …;
  Control = AMStandard.Control;
  SendMsg = AMStandard.SendMsg;
  activity = AMStandard.activity;
  AMStandard.TimerControl -> TimerC.StdControl;
  AMStandard.ActivityTimer -> TimerC.Timer[unique("Timer")]; …
}
```

Configuration Wires

- A configuration can bind an interface user to a provider using **->** or **<-**
  - User.interface **->** Provider.interface
  - Provider.interface **<-** User.interface
- Bounce responsibilities using **=**
  - User1.interface **=** User2.interface
  - Provider1.interface **=** Provider2.interface
- The interface may be implicit if there is no ambiguity
  - e.g., User.interface -> Provider == User.interface -> Provider.interface

Fan-Out and Fan-In

- A user can be mapped to multiple providers (fan-out)
  - Open **<tos>\apps\CntToLedsAndRfm\CntToLedsAndRfm.nc**

```
configuration CntToLedsAndRfm { }
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;
}
```

- A provider can be mapped to multiple users (fan-in)

Potential Fan-Out Bug

- Whenever you fan-out/in an interface, ensure the return value has a combination function
  - Can do:

```
App.Leds -> LedsC;
App.Leds -> NoLeds;
```

  - **CANNOT** do:

```
AppOne.ReceiveMsg -> GenericComm.ReceiveMsg[12];
AppTwo.ReceiveMsg -> GenericComm.ReceiveMsg[12];
```

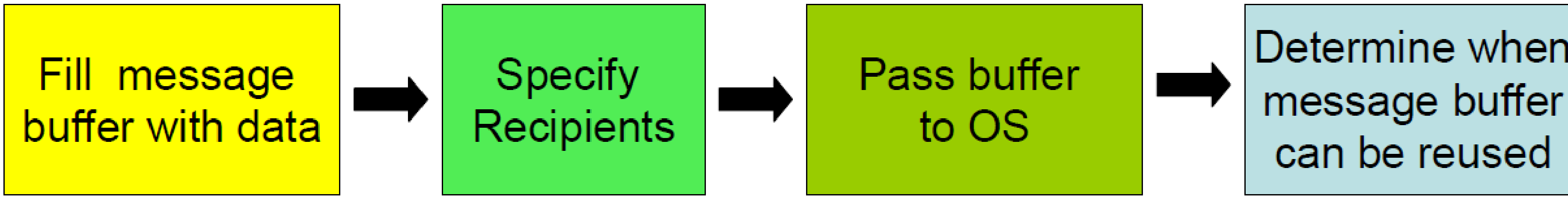
Top-Level Configuration

- All applications must contain a top-level configuration that uses **Main.StdControl**
  - Open **<tos>/apps/BlinkTask/BlinkTask.nc**

```
configuration BlinkTask { }
implementation {
  components Main, BlinkTaskM, SingleTimer, LedsC;

  Main.StdControl -> BlinkTaskM.StdControl;
  Main.StdControl -> SingleTimer;

  BlinkTaskM.Timer -> SingleTimer;
  BlinkTaskM.Leds -> LedsC;
}
```

Inter-Node Communication

- General idea:
  - Sender:



  - Receiver:

Group IDs and Addresses

- Group IDs create a virtual network
  - Group ID is an 8 bit value specified in <tos>/apps/Makelocal
- The address is a 16-bit value specified by the make command
  - make install.<id> mica2
  - Reserved addresses:
    - 0x007E - UART (TOS_UART_ADDR)
    - 0xFFFF - broadcast (TOS_BCAST_ADDR)
  - Local address: TOS_LOCAL_ADDRESS

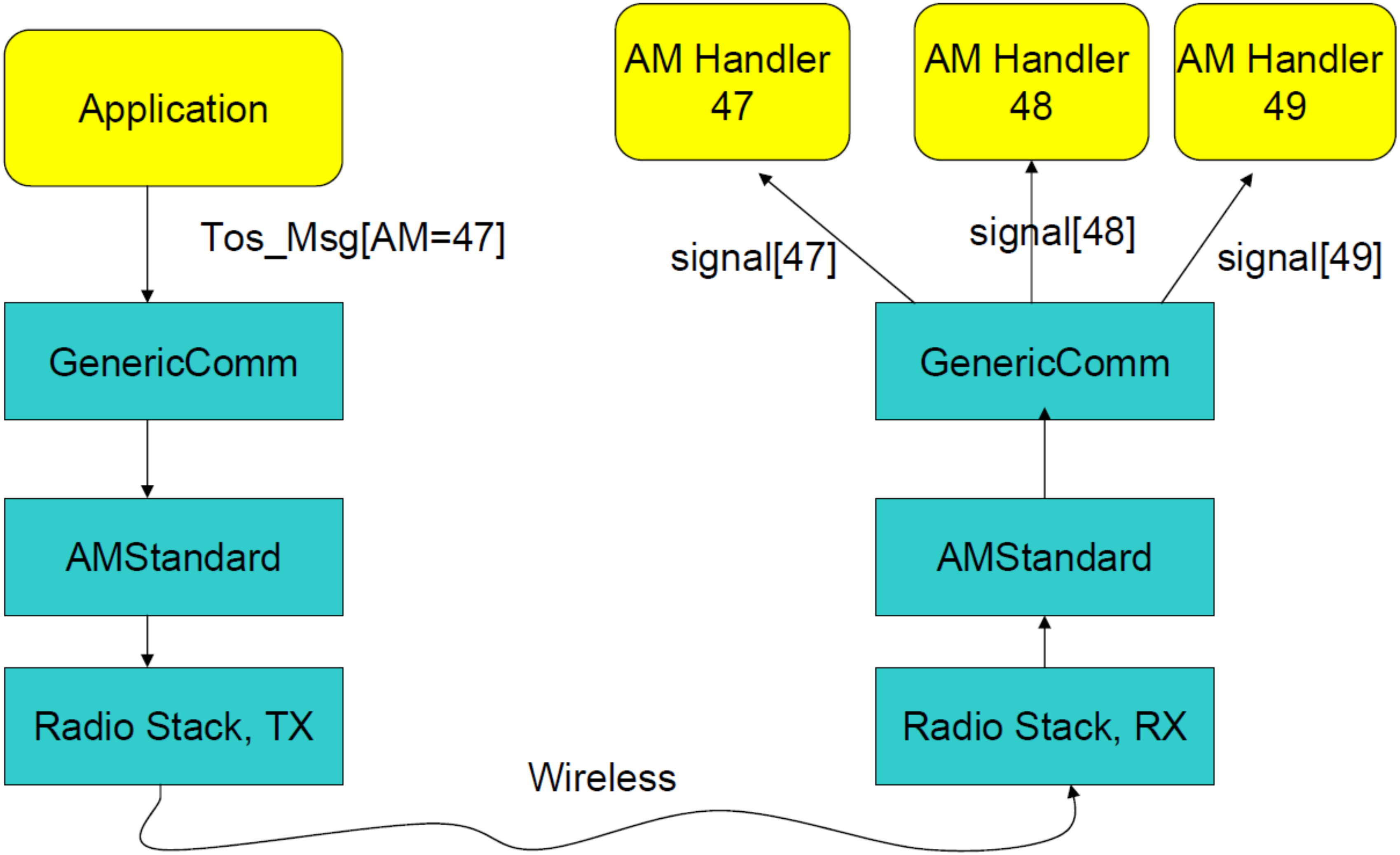
TOS Active Messages

- TOS uses active messages as defined in <tos>/system/types/AM.h
- Message is “active” because it contains the destination address, group ID, and type
- TOSH_DATA_LENGTH = 29 bytes
  - Can change via MSG_SIZE=x in Makefile

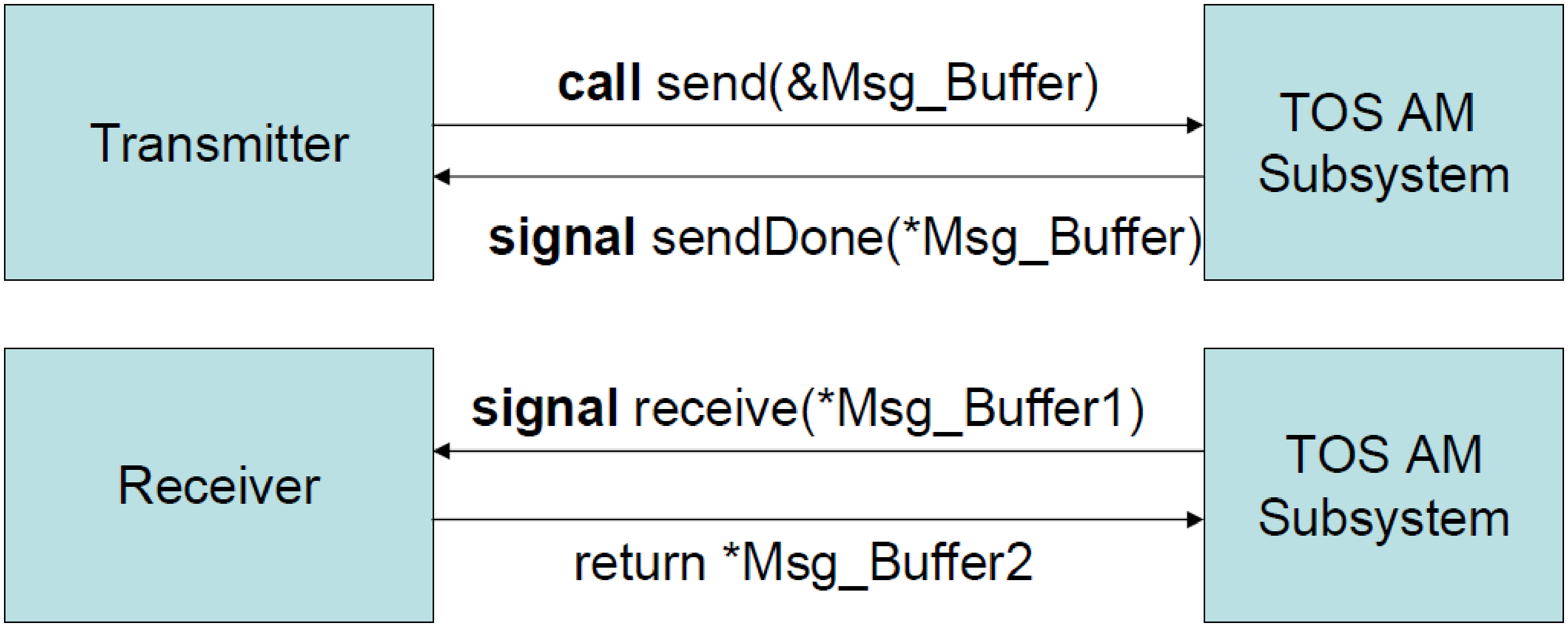
```
typedef struct TOS_Msg {
  // the following are transmitted
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[TOSH_DATA_LENGTH];
  uint16_t crc;
// the following are not transmitted
  uint16_t strength;
  uint8_t ack;
  uint16_t time;
  uint8_t sendSecurityMode;
  uint8_t receiveSecurityMode;
} TOS_Msg;
```

Header (5)	Payload (29)	CRC

Active Messaging (2)



Message Buffer Ownership



- Transmission: AM 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

Sending a Message

- First create a .h file with a struct defining the message data format, and a unique active message number
  - Open <tos>/apps/Oscilloscope/OscopeMsg.h

```
struct OscopeMsg
{
    uint16_t sourceMoteID;
    uint16_t lastSampleNumber;
    uint16_t channel;
    uint16_t data[BUFFER_SIZE];
};
```

```
struct OscopeResetMsg
{
    /* Empty payload! */
};

enum {
  AM_OSCOPEMSG = 10,
  AM_OSCOPERESETMSG = 32
};
```

Sending a Message (2)

```
module OscilloscopeM { …
  uses interface SendMsg as DataMsg; …
}
implementation{
  TOS_Msg msg; …

  task void dataTask() {
    struct OscopeMsg *pack = (struct OscopeMsg *)msg.data;
    … // fill up the message
    call DataMsg.send(TOS_BCAST_ADDR, sizeof(struct OscopeMsg),
                                  &msg[currentMsg]);
  }

  event result_t DataMsg.sendDone(TOS_MsgPtr sent, result_t success) {
    return SUCCESS;
  }
}
```

Question: How does TOS know the AM number?

Sending a Message (3)

- The AM number is determined by the configuration file
  - Open <tos>/apps/OscilloscopeRF/Oscilloscope.nc

```
configuration Oscilloscope { }
implementation {
  components Main, OscilloscopeM, GenericComm as Comm, …;
  …
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];
}
```

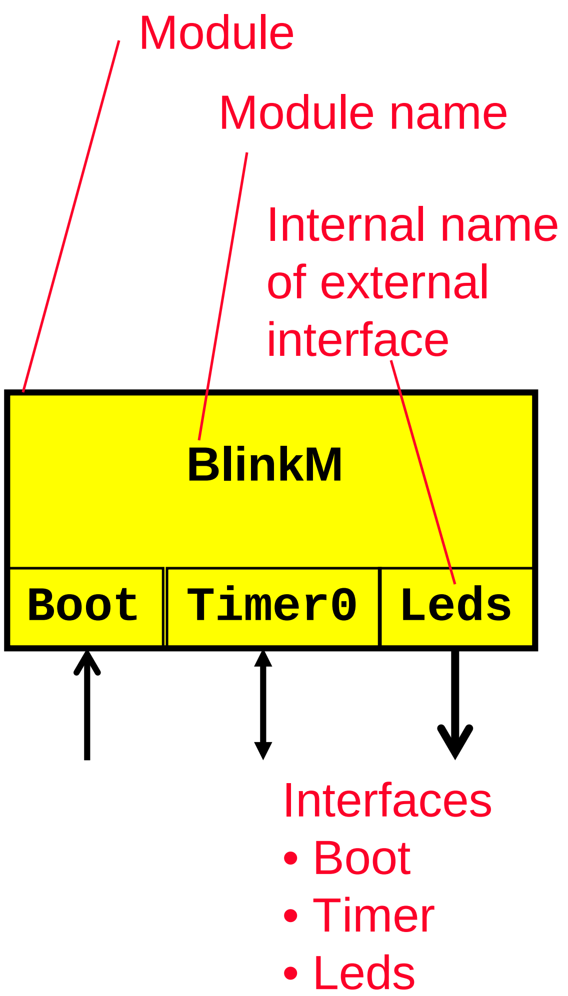
Receiving a Message

```
configuration Oscilloscope { }
implementation {
  components Main, OscilloscopeM,  UARTComm as Comm, ….;
  …
  OscilloscopeM.ResetCounterMsg ->
    Comm.ReceiveMsg[AM_OSCOPERESETMSG];
}
```

```
module OscilloscopeM {
  uses interface ReceiveMsg as ResetCounterMsg; …
}
implementation {
  uint16_t readingNumber;
  event TOS_MsgPtr ResetCounterMsg.receive(TOS_MsgPtr m) {
    atomic { readingNumber = 0; }
    return m;
  }
}
```

A simple event-driven module – BlinkM.nc

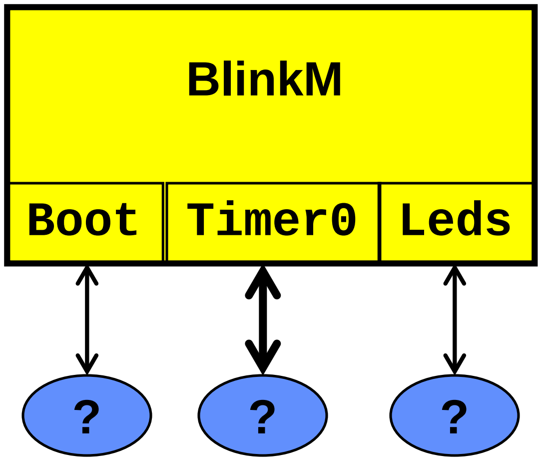
```
#include "Timer.h"
module BlinkM
{
  uses interface Boot;
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
```



- Coding conventions: TEP3

A simple event-driven module (cont.)

```
#include "Timer.h"
module BlinkM
{
  uses interface Boot;
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
```



Two Event Handlers

Each services
external event
by calling command
on some subsystem

Simple example: Boot interface

```
interface Boot {
  /**
   * Signaled when the system has booted successfully. Components can
   * assume the system has been initialized properly. Services may
   * need to be started to work, however.
   *
   * @see StdControl
   * @see SplitConrol
   * @see TEP 107: Boot Sequence
   */
  event void booted();
}
```

- $tinyOS-2.x/tos/interfaces/
- Defined in TEP 107 – Boot Sequence
- Consists of a single event.
- Hardware and operating system actions prior to this simple event may vary widely from platform to platform.
- Allows module to initialize itself, which may require actions in various other parts of the system.

Simple example: LEDs interface

```
#include "Leds.h"

interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();  ...
 /*
  * @param  val   a bitmask describing the on/off settings of the LEDs
  */
  async command uint8_t get();
  async command void set(uint8_t val);
}
```

- $tinyOS-2.x/tos/interfaces/
- set of Commands
  - Cause action
  - get/set a physical attribute (3 bits)
- async => OK to use even within interrupt handlers
- Physical wiring of LEDs to microcontroller IO pins may vary

Timer

```
interface Timer<precision_tag>
{
  command void startPeriodic(uint32_t dt);
  event void fired();
  command void startOneShot(uint32_t dt);
  command void stop();

  command bool isRunning();
  command bool isOneShot();
  command void startPeriodicAt(uint32_t t0, uint32_t dt);
  command void startOneShotAt(uint32_t t0, uint32_t dt);
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

- $tinyOS-2.x/tos/lib/timer/Timer.nc
- Rich application timer service built upon lower level capabilities that may be very different on different platform
  - Microcontrollers have very idiosyncratic timers
- Parameterized by precision

Timers

```
#include "Timer.h“

…
typedef struct { } TMilli; // 1024 ticks per second
typedef struct { } T32khz; // 32768 ticks per second
typedef struct { } TMicro; // 1048576 ticks per second
```

- Timers are a fundamental element of Embedded Systems
  - Microcontrollers offer a wide range of different hardware features
  - Idiosyncratic
- Logically Timers have
  - Precision - unit of time the present
  - Width - # bits in the value
  - Accuracy - how close to the precision they obtain
- TEP102 defines complete TinyOS timer architecture
- Direct access to low-level hardware
- Clean virtualized access to application level timers

Example – multiple virtual timers

```
#include "Timer.h"

module Blink3M
{
  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()
  {
     call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
     call Leds.led2Toggle();
  }
}
```

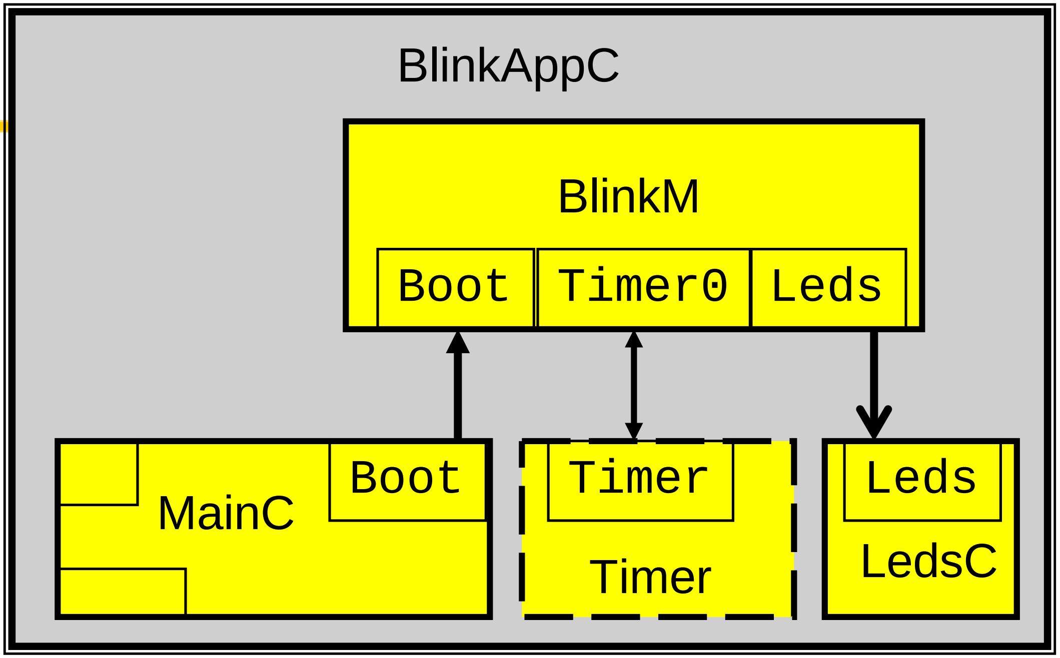
Composition

- Our event-driven component, Blink, may be built directly on the hardware
  - For a particular microcontroller on a particular platform
- or on a simple layer for a variety of platforms
- or on a full-function kernel

- Or it may run in a simulator on a PC,
- Or…

- As long as it is wired to components that provide the interfaces that this component uses.
- And it can be used in a large system or application

Configuration

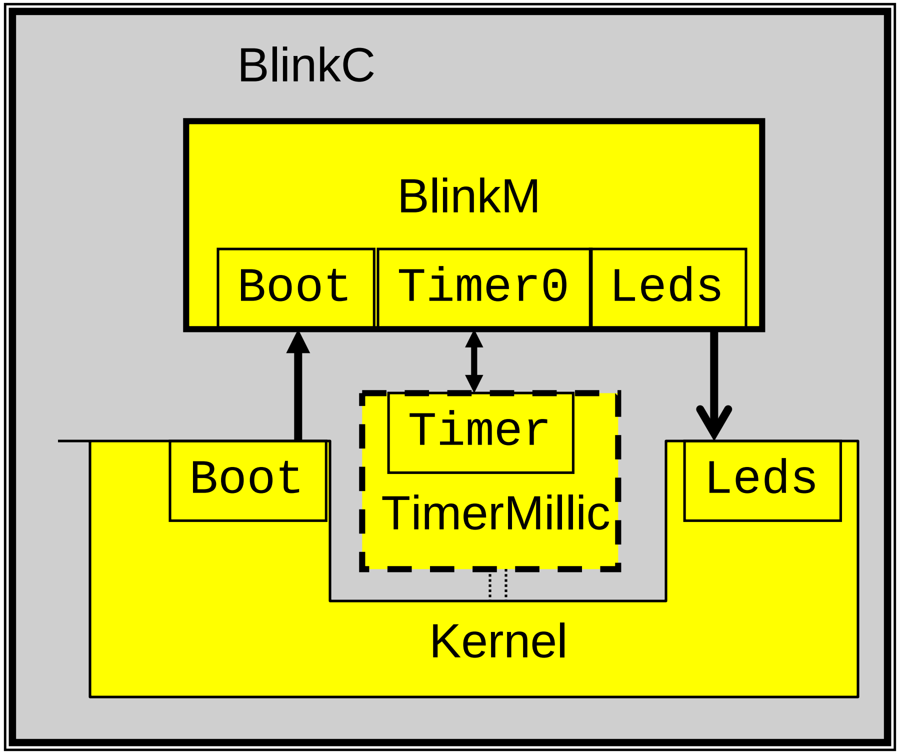



```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkM, LedsC;
  components new TimerMilliC() as Timer;

  BlinkM          -> MainC.Boot;
  BlinkM.Leds     -> LedsC;
  BlinkM.Timer0 -> Timer.Timer;
}
```

- Generic components create service instances of an underlying service. Here, a virtual timer.
- If the interface name is same in the two components, only one need be specified.

A Different Configuration

```
configuration blinkC{
}

implementation{
  components blinkM;
  components MainC;
  components Kernel;

  blinkM.Boot -> Kernel.Boot;
  blinkM.Leds -> Kernel.Leds;
  components new TimerMilliC();
  blinkM.Timer0 -> TimerMilliC.Timer;
}
```



- Same module configured to utilize a very different system substrate.

Execution Behavior

- Timer interrupt is mapped to a TinyOS event.
  - Handled in a safe context
- Performs simple operations.
- When activity stops, entire system sleeps
  - In the lowest possible sleep state
- Never wait, never spin. Automated, whole-system power management.

Execution Behavior

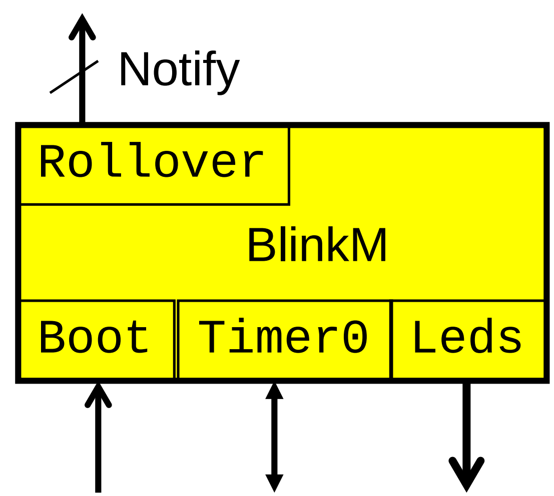
```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  users interface Boot;
}
implementation
{
  uint8_t counter = 0;

  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
  }
}
```

- Private scope
- Sharing through explicit interface only!
  - Concurrency, concurrency, concurrency!
  - Robustness, robustness, robustness!
- Static extent
- HW independent type
  - unlike int, long, char

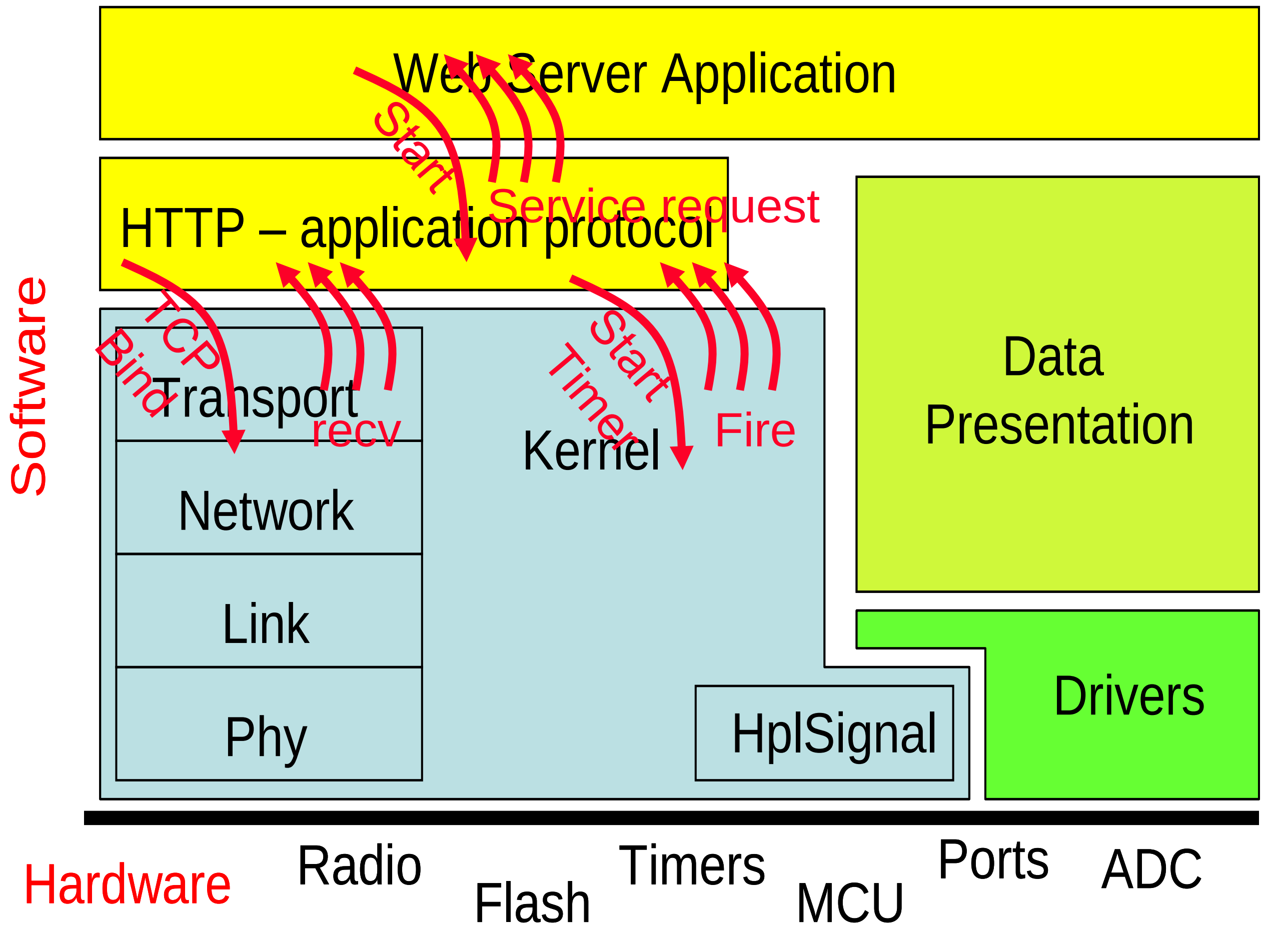
Events

```
module BlinkM {
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  uses interface Boot;
  provides interface Notify<bool> as Rollover;
}
implementation
{
  uint8_t counter = 0;

  event void Boot.booted()
  { call Timer0.startPeriodic( 250 ); }

  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
    if (!counter) signal Rollover.notify(TRUE);
  }
}
```

- Call commands
- Signal events
- Provider of interface handles calls and signals events
- User of interface calls commands and handles signals



Examples – Event-Driven Execution



Split-Phase Operations

- For potentially long latency operations
  - Don’t want to spin-wait, polling for completion
  - Don’t want blocking call - hangs till completion
  - Don’t want to sprinkle the code with explicit sleeps and yields
- Instead,
  - Want to service other concurrent activities will waiting
  - Want to go sleep if there are none, and wake up upon completion
- Split-phase operation
  - Call command to initiate action
  - Subsystem will signal event when complete
- The classic concurrent I/O problem, but also want energy efficiency.
  - Parallelism, or sleep.
  - Event-driven execution is fast and low power!

Examples

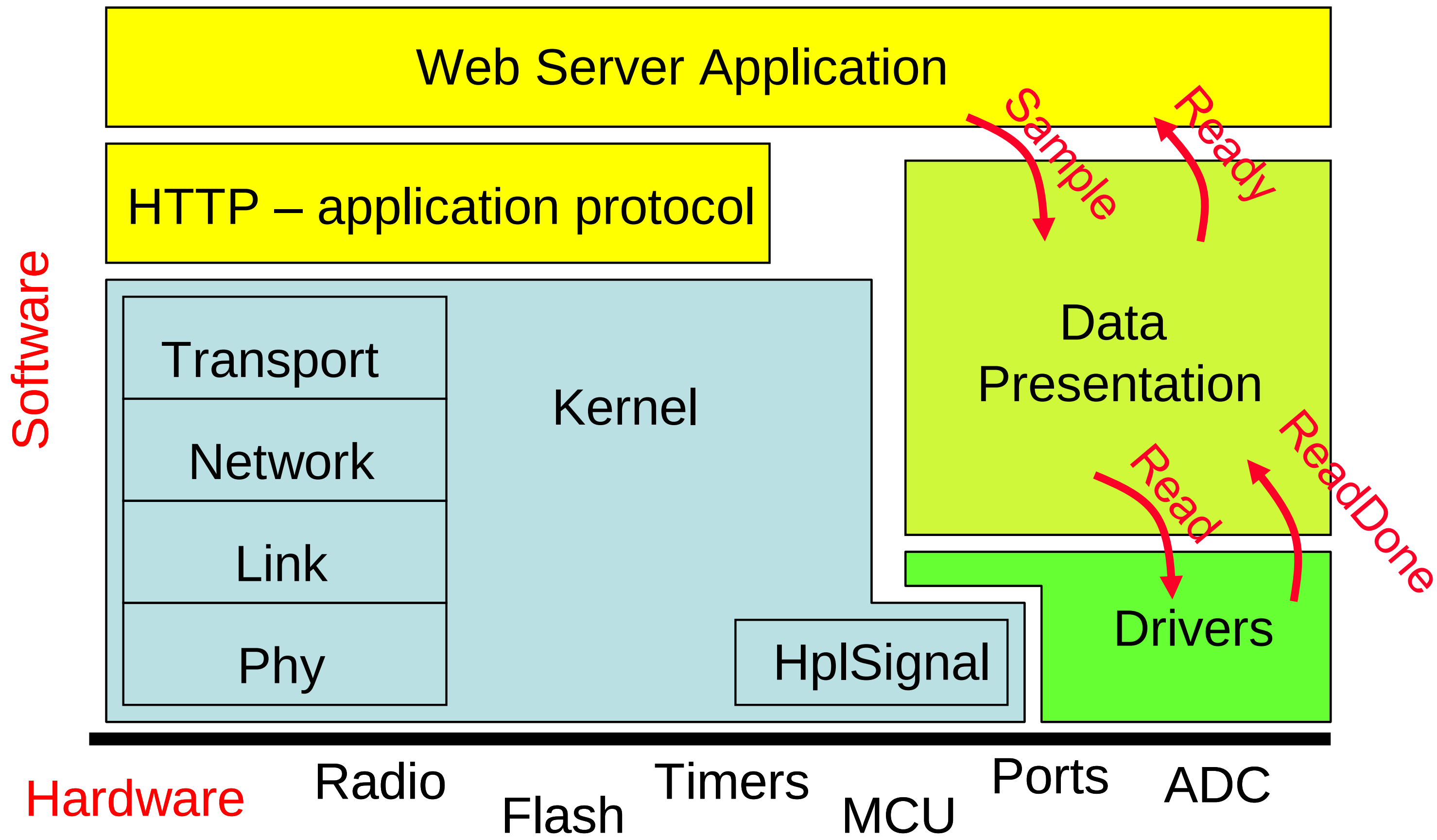
```
/* Power-hog Blocking Call */
if (send() == SUCCESS) {
  sendCount++;
}
```

```
/* Split-phase call */
// start phase
…
 call send();
…
}
//completion phase
void sendDone(error_t err) {
  if (err == SUCCESS) {
    sendCount++;
  }
}
```

```
/* Programmed delay */
state = WAITING;
op1();
sleep(500);
op2();
state = RUNNING
```

```
state = WAITING;
op1();
call Timer.startOneShot(500);

command void Timer.fired() {
  op2();
  state = RUNNING;
```

Examples – Split-Phase



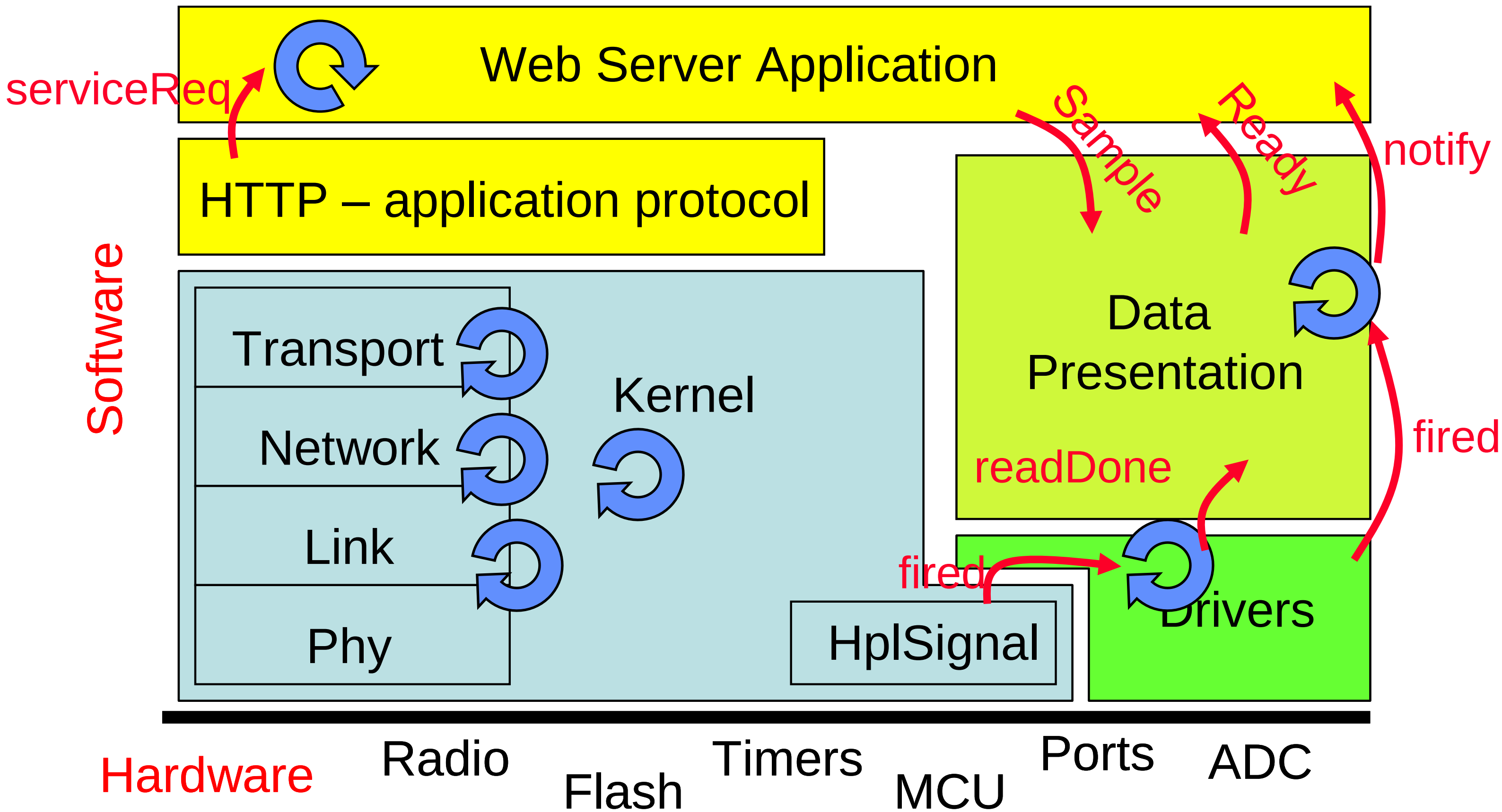
Concurrency

- Commands and event glue together concurrent activities
- Hardware units operate on parallel
  - Commands used to initiate activity
  - Events used to signal completion, etc.
- System software components are very similar
  - But they don’t have dedicated hardware to keep working on the command.
  - Tasks are used for that
- Decouple execution and leave room for juggling
  - Use lots of little tasks to keep things flowing
- Preempted by async events (interrupts)
  - Not other tasks

Tasks – Crossing the Asynch / Synch Boundary

```
module UserP {
  provides interface Button;
  uses interface Boot;
  uses interface Msp430Port as Pin;
  uses interface Msp430PortInterrupt as PinInt;
}

implementation {
  event void Boot.booted() {
    call Pin.setDirection(0); /* Input */
    call PinInt.edge(1);      /* Rising edge, button release */
    call PinInt.enable(1);    /* Enable interrupts */
  }

  task void fire() {
    signal Button.pressed();  /* Signal event to upper layers */
  }

  async event void PinInt.fired() {
    post fire();
  }
}
```

Example - Tasks

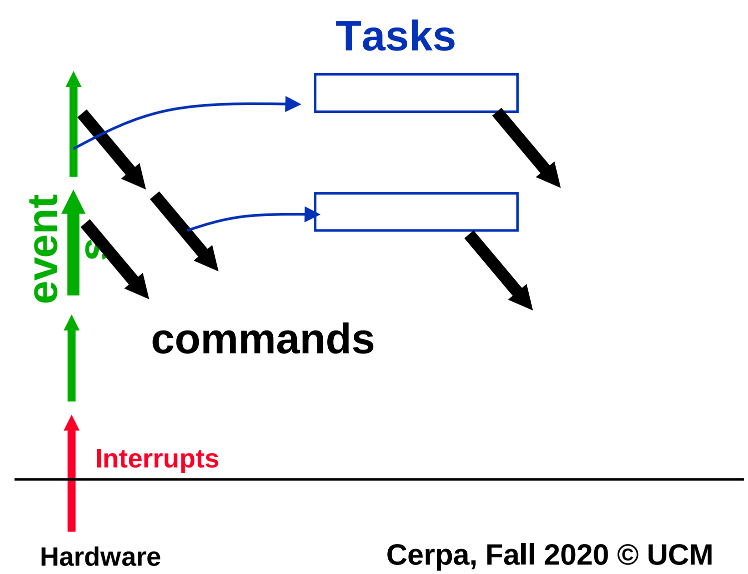


Tasks

```
/* BAD TIMER EVENT HANDLER */
event void Timer0.fired() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {
    call Leds.led0Toggle();
  }
}
```

```
/* Better way to do a silly thing */
task void computeTask() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {}
}

event void Timer0.fired() {
  call Leds.led0Toggle();
  post computeTask();
}
```

- Need to juggle many potentially bursty events.
- If you cannot get the job done quickly, record the parameters locally and post a task to do it later.
- Tasks are preempted by lower level (async) events.
  - Allow other parts of the system to get the processor.
  - Without complex critical semaphores, critical sections, priority inversion, schedulers, etc.



Uses of tasks (???)

- High speed sampling
- Filtering
- Queueing
- Smoothing
- Detection
- Classification
- …

What is TOSSIM?

- Discrete even simulator
  - Like ns2 (a well-known network simulator)

Alternatives

- Cycle-accurate simulators
  - Like Avrora, MSPSim, etc.

Two Directions

- Port
  - Make PC a supported platform
  - This is TOSSIM in tinyos-1.x
- Virtualize
  - Simulate on the supported platforms
  - This is TOSSIM in tinyos-2.x

Features

- Simulates a MicaZ mote
  - ATmega128L (128KB ROM, 4KB RAM)
  - CC2420
- Uses CPM to model the radio noise
- Supports two programming interfaces:
  - Python
  - C++

Anatomy

TOSSIM

```
tos/lib/tossim
tos/chips/atm128/sim
tos/chips/atm128/pins/sim
tos/chips/atm128/timer/sim
tos/chips/atm128/spi/sim
tos/platforms/mica/sim
tos/platforms/micaz/sim
tos/platforms/micaz/chips/cc2420/sim
```

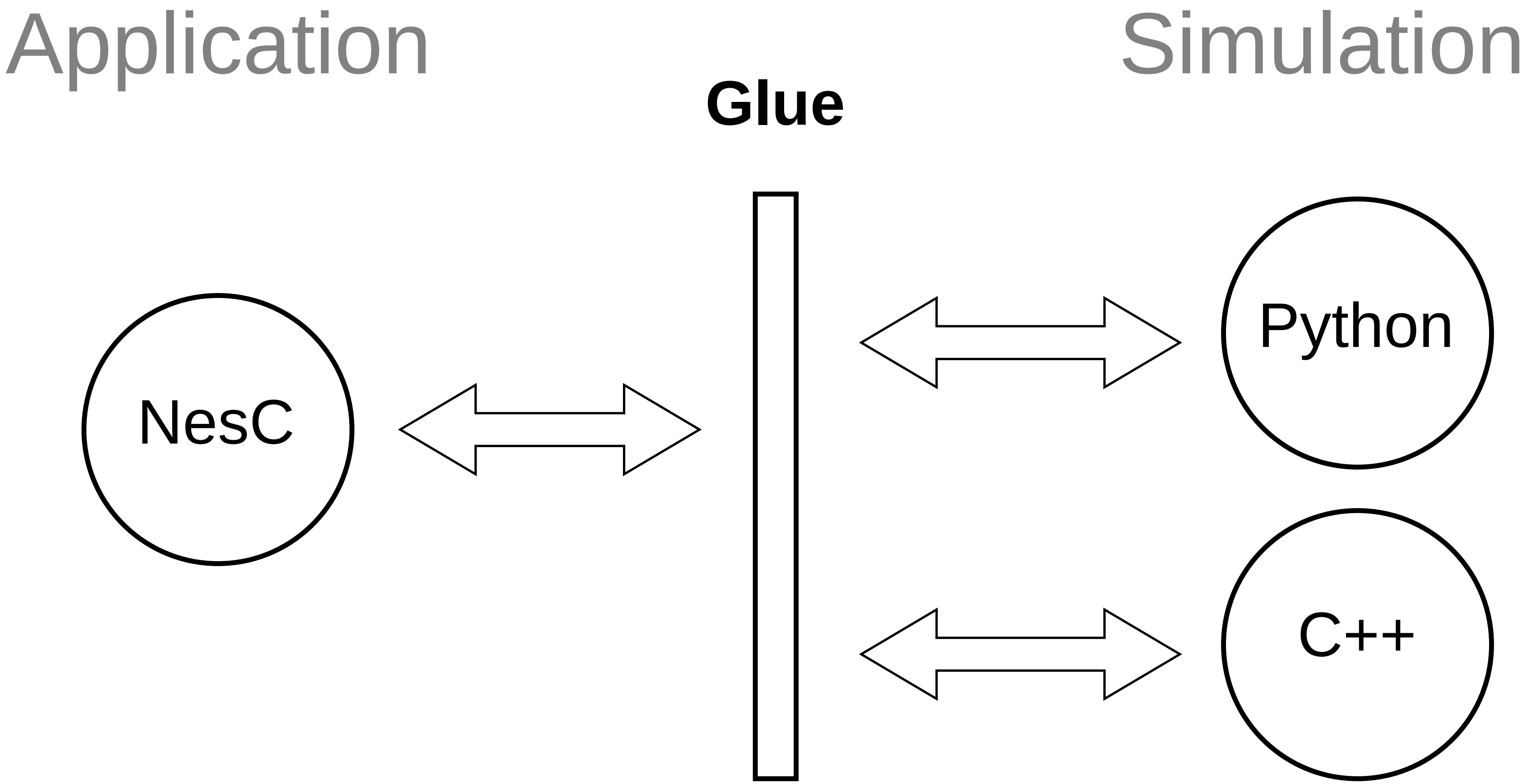
Application

```
Makefile
*.nc
*.h
```

Simulation Driver

```
*.py | *.cc
```

Anatomy



The Building Process

```
$ make micaz sim
```

1. Generate an XML schema — *app.xml*
2. Compile the application — *sim.o*
3. Compile the Python support — *pytossim.o*, *tossim.o*, *c-support.o*
4. Build a share object — *_TOSSIMmodule.o*
5. Copying the Python support — *TOSSIM.py*

```
$ ./sim.py
```

TOSSIM.py

Tossim

Radio

Mote

Packet

Mac

TOSSIM.Tossim

.getNode() → TOSSIM.Mote

.radio() → TOSSIM.Radio

.newPacket() → TOSSIM.Packet

.mac() → TOSSIM.Mac

.runNextEvent()

.ticksPerSecond()

.time()

10 seconds

```
from TOSSIM import *

t = Tossim([])

  ...

while t.time() < 10*t.ticksPerSecond():

    t.runNextEvent()
```

dbg

Syntax

dbg(*tag, format, arg1, arg2, ...*);

Example

dbg(“Flooding”, “Starting time with time %u.\n”, timerVal);

Python

t = Tossim([])
t.addChannel(“Flooding”, sys.stdout)

Useful Functions

*char**	sim_time_string()
sim_time_t	sim_time()
int	sim_random()
sim_time_t	sim_ticks_per_sec()

typedef *long long int sim_time_t*;

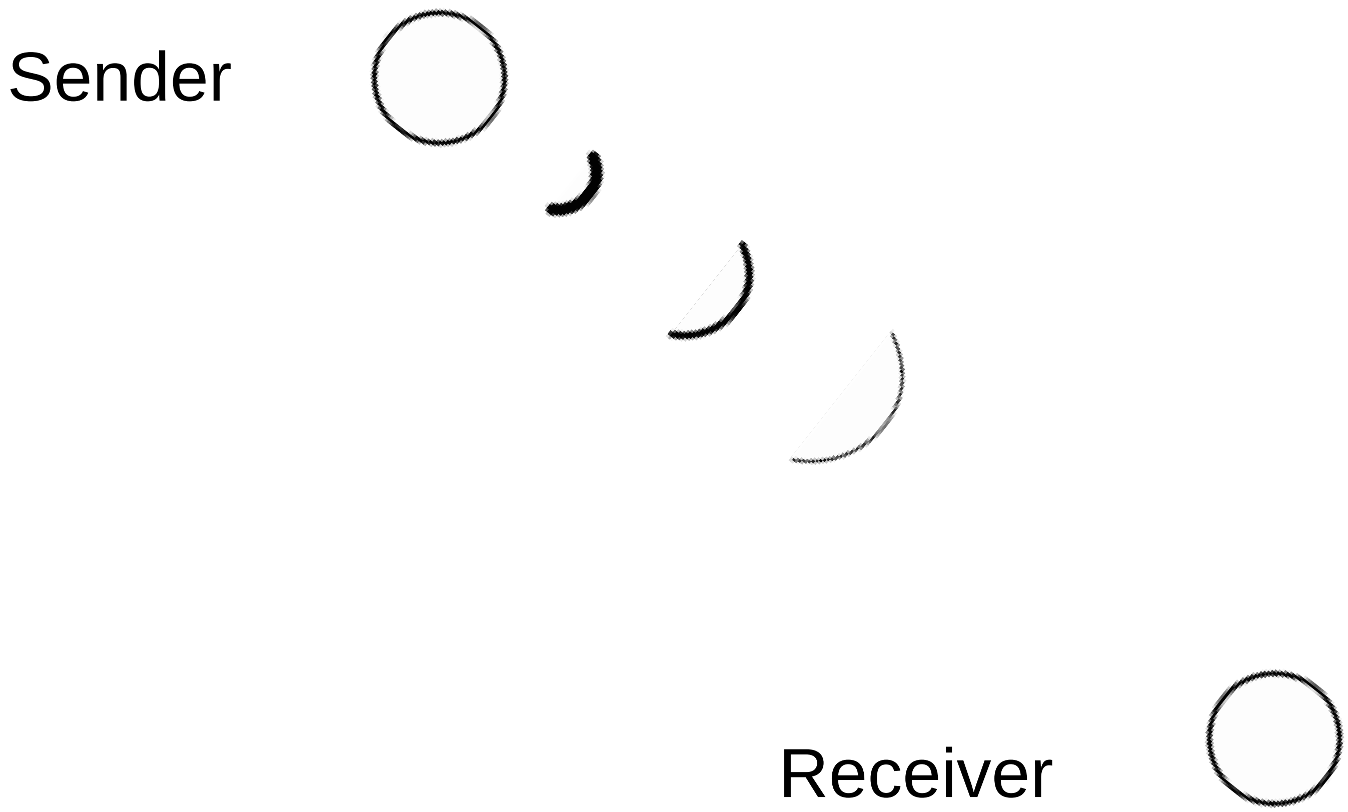
Closest-fit Pattern Matching (CPM)

Improving Wireless Simulation Through Noise Modeling

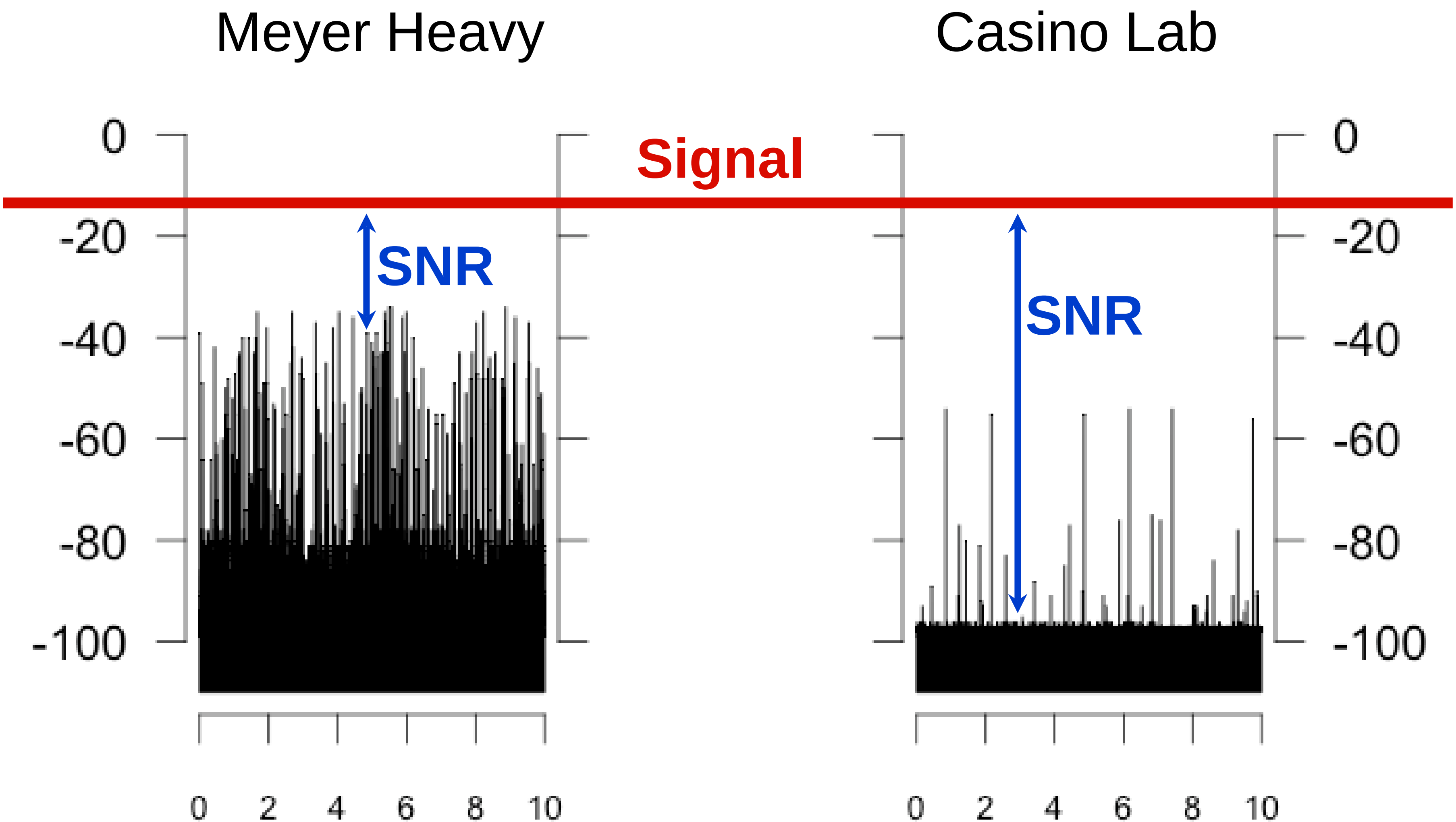
HyungJune Lee, Alberto Cerpa, and Philip Levis

IPSN 2007

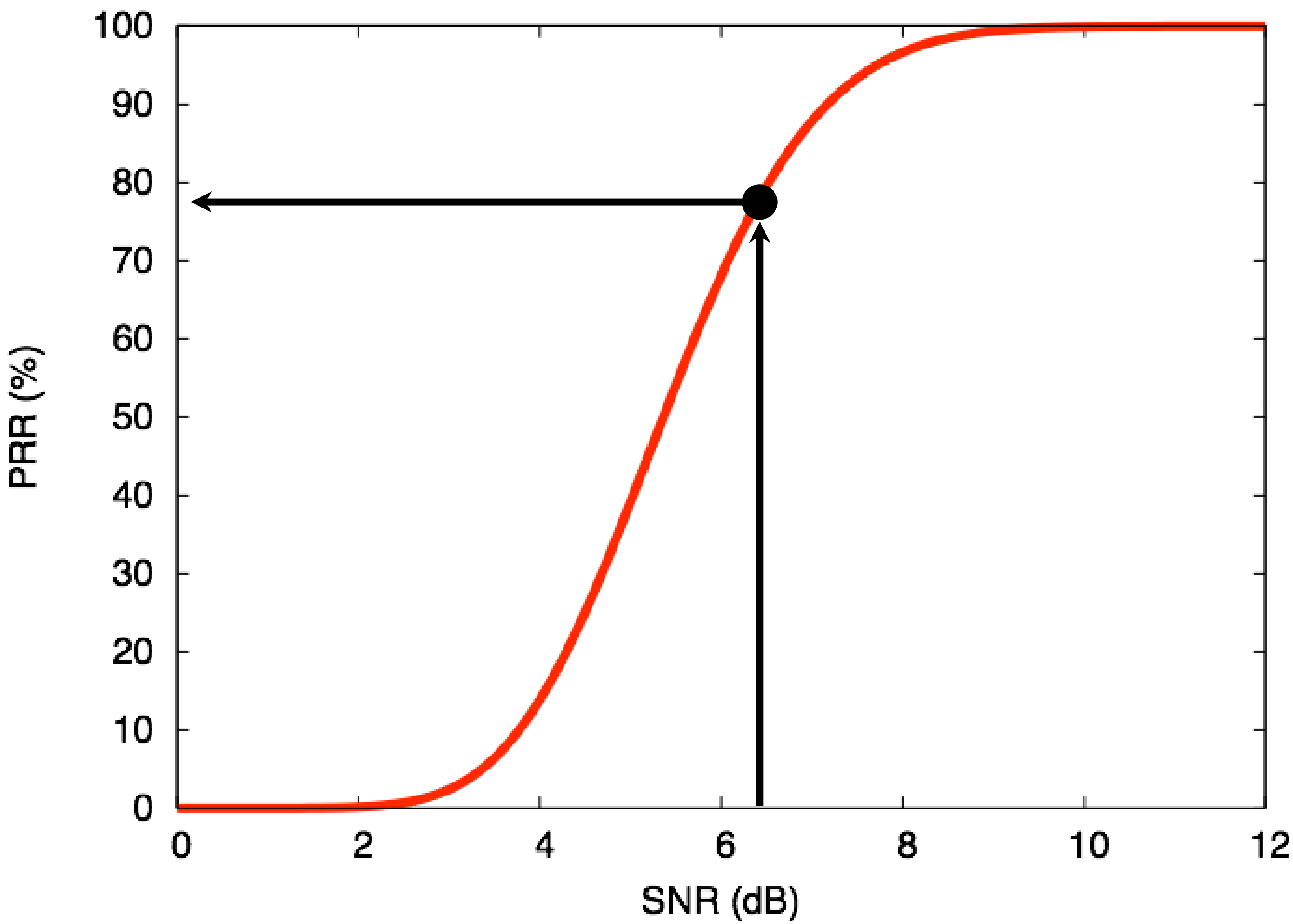
Radio Model (2)



Noise Level


Meyer Heavy
Casino Lab
Signal
SNR
SNR
0
-20
-40
-60
-80
-100
0 2 4 6 8 10
0 2 4 6 8 10

CC2420 SNR/PRR



TOSSIM.Radio

.add(*source, destination, gain*)

.connected(*source, destination*) → True/False

.gain(*source, destination*)

TOSSIM.Mote

.bootAtTime(*time*)

.addNoiseTraceReading(*noise*)

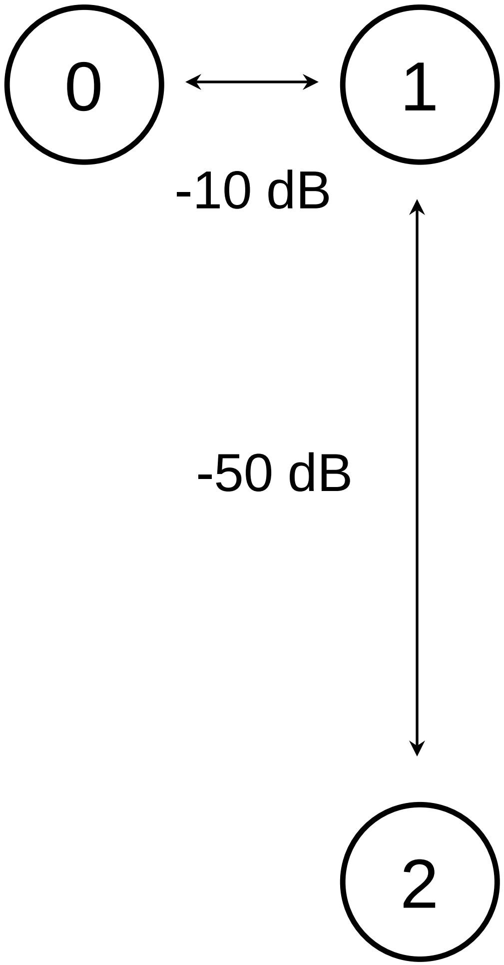
.createNoiseModel()

.isOn() → True/False

.turnOn()/.turnOff()

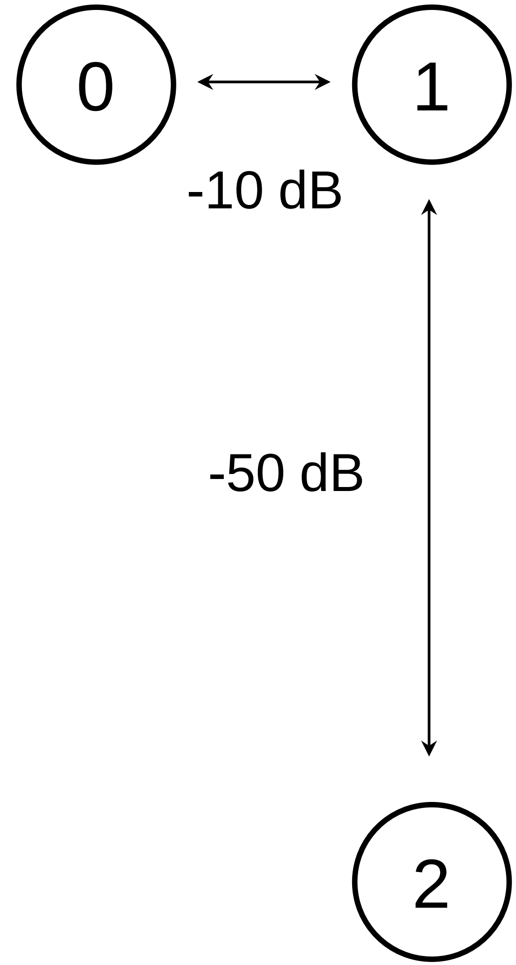
Example

```
from TOSSIM import *
t = Tossim([])
r = t.Radio()
mote0 = t.getNode(0)
mote1 = t.getNode(1)
mote2 = t.getNode(2)
r.add(0, 1, -10)
r.add(1, 0, -10)
r.add(1, 2, -50)
r.add(2, 1, -50)
```



Example (2)

```
noise = file("meyer-short.txt")
lines = noise.readlines()
for line in lines:
  str = line.strip()
  if (str != ""):
    val = int(str)
    for m in [mote0, mote1, mote2]:
      m.addNoiseTraceReading(val)

for m in [mote0, mote1, mote2]:
    m.createNoiseModel()
```

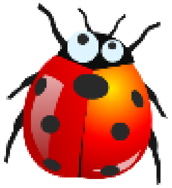


Other Features

- Injecting packets
- Inspecting internal variables
- C++ interface
- Debugging using gdb

Debugging Tips

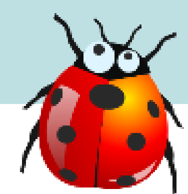
- Join and/or search TOS mailing lists
  - http://tinyos-help.10906.n7.nabble.com/
  - https://github.com/tinyos/tinyos-main/issues
- Develop apps in a private directory
  - (e.g., <tos>/broken)
- Use TOSSIM and dbg(DBG,…) statements



Potentially Nasty Bug 1

- What's wrong with the code?
  - **Symptom**: data saved in globalData is lost
- **Reason**: Race condition between two tasks
- **Solution**: Use a queue, or never rely on inter-task communication

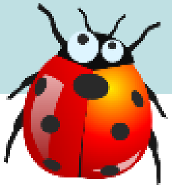
```
uint8_t globalData;

task void processData() {
  call SendData.send(globalData);
}

command result_t Foo.bar(uint8_t data) {
  globalData = data;
  post processData();
}
```



Potentially Nasty Bug 2

- What's wrong with the code?
  - **Symptom**: message is corrupt
- **Reason**: TOS_Msg is allocated in the stack, lost when function returns
- **Solution**: Declare TOS_Msg msg in component's frame.

```
command result_t Foo.bar(uint8_t data) {
  TOS_Msg msg;
  FooData* foo = (FooData*)msg.data;
  foo.data = data;
  call SendMsg.send(0x01, sizeof(FooData),
                    &msg);
}
```

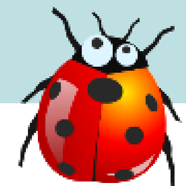


Potentially Nasty Bug 3

- What's wrong with the code?
  - Symptom: some messages are lost
- **Reason**: Race condition between two components trying to share network stack (which is split-phase)
- **Solution**: Use a queue to store pending messages

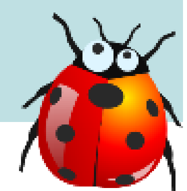
Component 1: *

```
command result_t Foo.bar(uint8_t data) {
  FooData* foo = (FooData*)msg.data;
  foo.data = data;
  call SendMsg.send(0x01, sizeof(FooData),
                    &msg);
}
```



Component 2: *

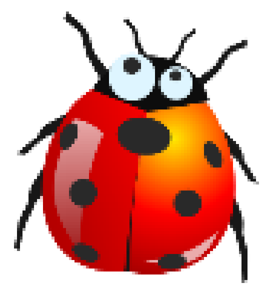
```
command result_t Goo.bar(uint8_t data) {
  GooData* goo = (GooData*)msg.data;
  goo.data = data;
  call SendMsg.send(0x02, sizeof(GooData),
                    &msg);
}
```



* Assume TOS_Msg msg is declared in component's frame.

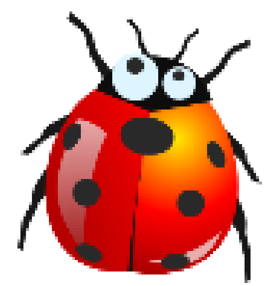
Potentially Nasty Bug 4

- **Symptom**: Some messages are *consistently* corrupt, and TOSBase is working. Your app *always* works in TOSSIM.
- **Reason**: You specified MSG_SIZE=x where x > 29 in your application but forgot to set it in TOSBase's makefile
- This is only when you develop code outside the TOSSIM simulator.



Potentially Nasty Bug 5

- **Symptom**: Your app works in TOSSIM, but never works on the mote. Compiler indicates you are using 3946 bytes of RAM.
- **Reason**: TinyOS reserves some RAM for the Stack. Your program cannot use more than 3.9K RAM.
- This is only when you develop code outside the TOSSIM simulator.



Potentially Nasty Bug 6

- **Symptom**: Messages can travel from laptop to SN but not vice versa.
- **Reason**: SW1 on the mote programming board is on. This blocks all outgoing data and is useful when reprogramming.
- This is only when you develop code outside the TOSSIM simulator.