TinyOS Tutorial

Original slides by Greg Hackmann
Adapted by Octav Chipara

Outline

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

TinyOS Installation

- TinyOS Documentation Wiki: http://docs.tinyos.net/
  - Various installation options listed under "Getting started" section
- Best to run under linux
  - Virtualization solutions: vmware or virtualbox [free]
  - Tutorials on how to install under linux already available
  - Email me or come to office hours if you have problems
- Native OS X support is available [not officially supported]

TinyOS Directory Structure

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

make System

- `$TOSROOT/support/make` includes lots of `Makefiles` to support the build process
- Create a simple stub `Makefile` in your app directory that points to main component

```
COMPONENT=[MainComponentC]
SENSORBOARD=[boardtype] # if needed
include $(MAKERULES)
```

- `make [platform]` in app directory
  - Builds but does not install program
  - `platform`: one of the platforms defined in `$TOSROOT/tos/platforms` (`mica2`, `micaz2`, `telosb`)

make System

- `make [re]install.[node ID] [platform] [programming options]`
  - node ID: `0 - 255`
  - programming options:
    - mica2/micaz: `mib510,/dev/ttyXYZ`
    - telosb: `bsl,/dev/ttyXYZ`
- `make clean`
- `make docs [platform]`
  - Generates HTML documentation in `$TOSROOT/doc/nesdoc/[platform]`

Build Stages

```
Terminal — bash — 80×35 — ⌘1
gwh2@rooster148:/opt/tinyos-2.1.0/apps/Blink :( > make install.0 telosb bsl,/dev
/tty.usbserial-M4A5L524
mkdir -p build/telosb
    compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe  -Os -O -mdisable-hwmul -Wall -Wshadow -Wnesc-all -
target=telosb -fnesc-cfile=build/telosb/app.c -board= -DDEFINED_TOS_AM_GROUP=0x2
2 -DIDENT_APPNAME=\"BlinkAppC\" -DIDENT_USERNAME=\"gwh2\" -DIDENT_HOSTNAME=\"roo
ster148.cse.\" -DIDENT_USERHASH=0xb9e110b0L -DIDENT_TIMESTAMP=0x48c59807L -DIDEN
T_UIDHASH=0x46b3cb61L  BlinkAppC.nc -lm
    compiled BlinkAppC to build/telosb/main.exe
            2650 bytes in ROM
              55 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
    writing TOS image
tos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target ihex
build/telosb/main.ihex build/telosb/main.ihex.out-0 TOS_NODE_ID=0 ActiveMessageA
ddressC$addr=0
Could not find symbol ActiveMessageAddressC$addr in build/telosb/main.exe, ignor
ing symbol.
Could not find symbol TOS_NODE_ID in build/telosb/main.exe, ignoring symbol.
    installing telosb binary using bsl
tos-bsl --telosb -c /dev/tty.usbserial-M4A5L524 -r -e -I -p build/telosb/main.ih
ex.out-0
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
2682 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-0 build/telosb/main.ihex.out-0
gwh2@rooster148:/opt/tinyos-2.1.0/apps/Blink :) >
```

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

Set AM address and node ID in binary

Program mote

“Homework”

- Install TinyOS 2.1 and build Blink

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

How to Get Help

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

How to Get Help

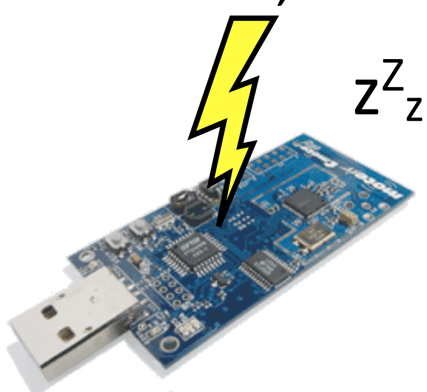
- nesdoc: annotated API for all interfaces and components in TinyOS: http://docs.tinyos.net/index.php/Source_Code_Documentation
- TinyOS Enhancement Protocols (TEP): formal documentation for TinyOS features: http://docs.tinyos.net/index.php/TEPs

Outline

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

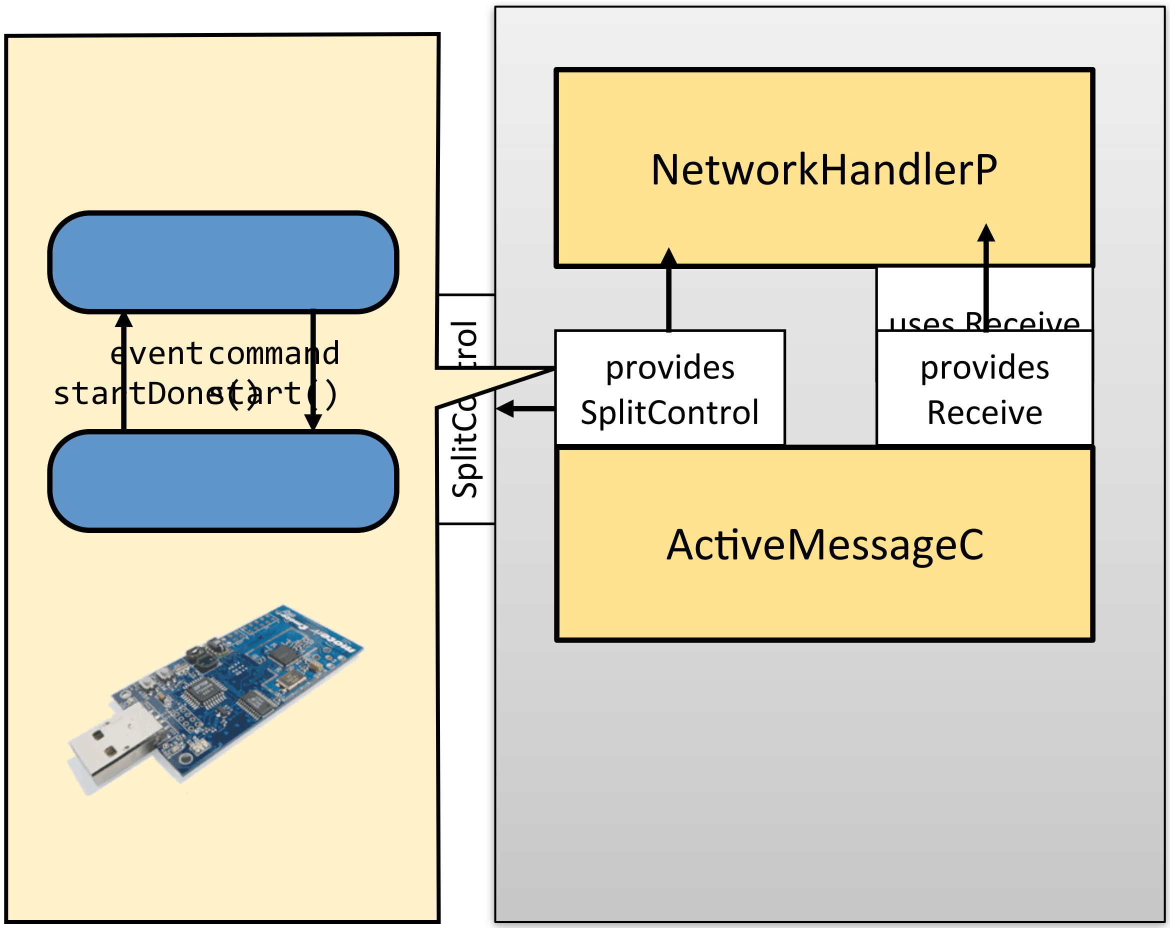
TinyOS Execution Model

- 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

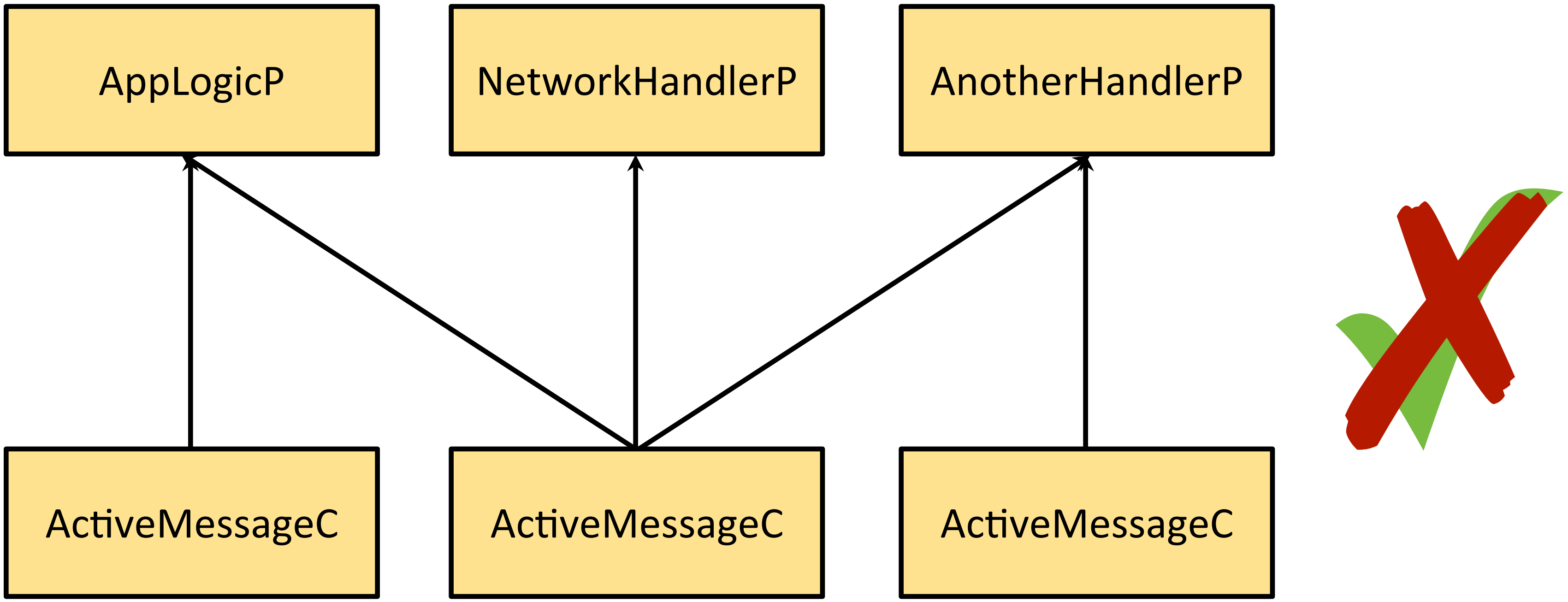


handlePacket | readSensor | sendResponse

TinyOS Component Model



Components != Objects



Interfaces

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

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

Modules

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

```
module ExampleModuleP {
  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

- `implementation` block may contain:
  - Variable declarations
  - Helper functions
  - Tasks
  - Event handlers
  - Command implementations

Modules: Variables and Functions

- Placed inside `implementation` block exactly like standard C declarations:

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

  ...

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

Modules: Tasks

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

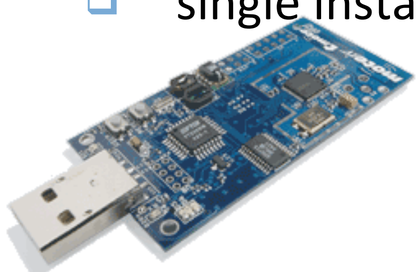
```
implementation {
  ...
  task void legalTask() {
    // OK
  }
  task bool illegalTask() {
    // Error: can’t have a return value!
  }
  task void anotherIllegalTask(bool param1) {
    // Error: can’t have parameters!
  }
}
```

Modules: Task Scheduling

- Tasks are scheduled using the `post` keyword

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

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



task1 task1 task1 ...

Modules: Commands and Events

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

```
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:

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

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

```
signal SplitControl.startDone();
// Invoke the startDone event handler on the SplitControl interface
```

Modules: Commands and Events

- A command, event handler, or function can call or signal *any* other command or event from *any* interface wired into the module:

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

Synchronous vs. Asynchronous

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

```
implementation {
  async event void Alarm.fired() {
     // Handle hardware alarm interrupt
  }
}
```

Reminder: Race Conditions

- Use `atomic` blocks to avoid race conditions

```
implementation {
  uint8_t sharedCounter;

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

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

Interrupts are disabled here -- use sparingly and make as short as practical

Reminder: Race Conditions

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

```
implementation {
  uint8_t sharedCounter;

  task void incrementCounter() { sharedCounter++; }

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

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

Task is scheduled immediately, but executes later

nesC and Race Conditions

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

```
implementation {
        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; events are mutually exclusive

Configurations

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

List interfaces that the component imports & exports

Give comma-separated list(s) of constituent components

Wire two components’ … together using … pointing from … provider

Wire external interfaces using =

Outline

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

High-Level Summary

- nesC includes a lot of complex features that try to alleviate design problems with TinyOS 1.x
- The good news: you will probably never have to **write** code that incorporates these features
- The bad news: you're almost certain to **use** code that incorporates these features

- First, an abstract look at what these features are and what their syntax means
- Second, a concrete example of how to use them to build components

Interfaces with Arguments

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

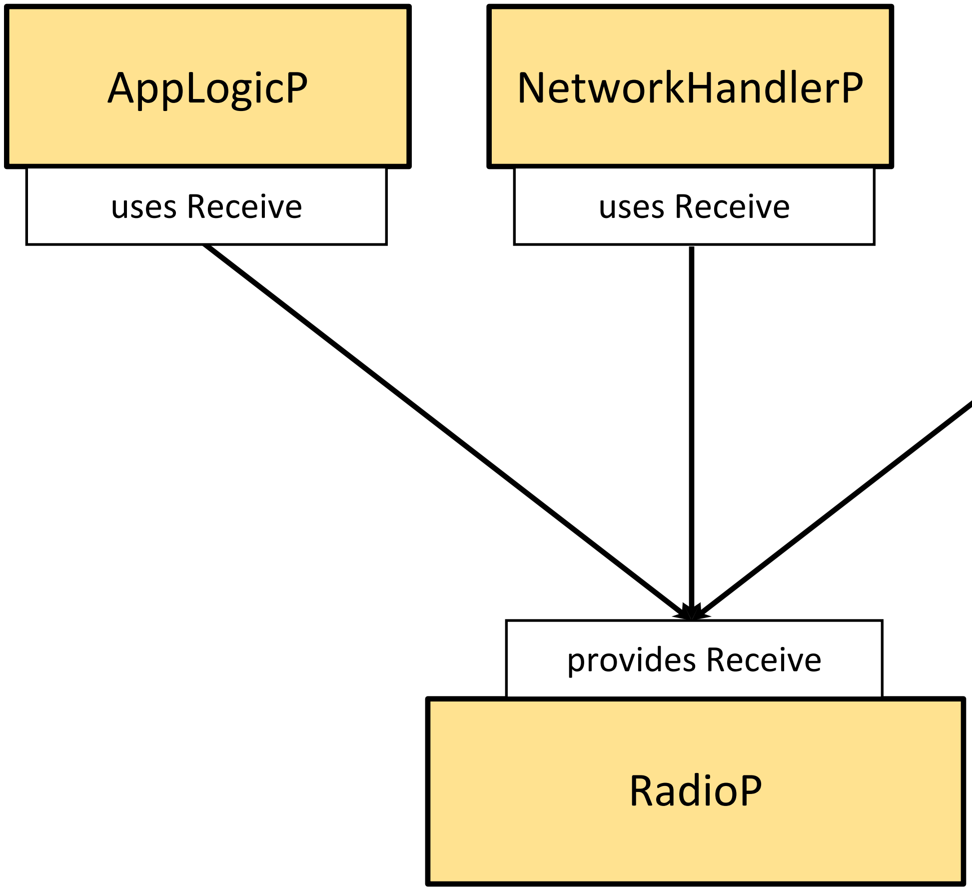
```
interface ReadUint16 {
  command error_t read();
  event void readDone(error_t error, uint16_t value);
}

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:

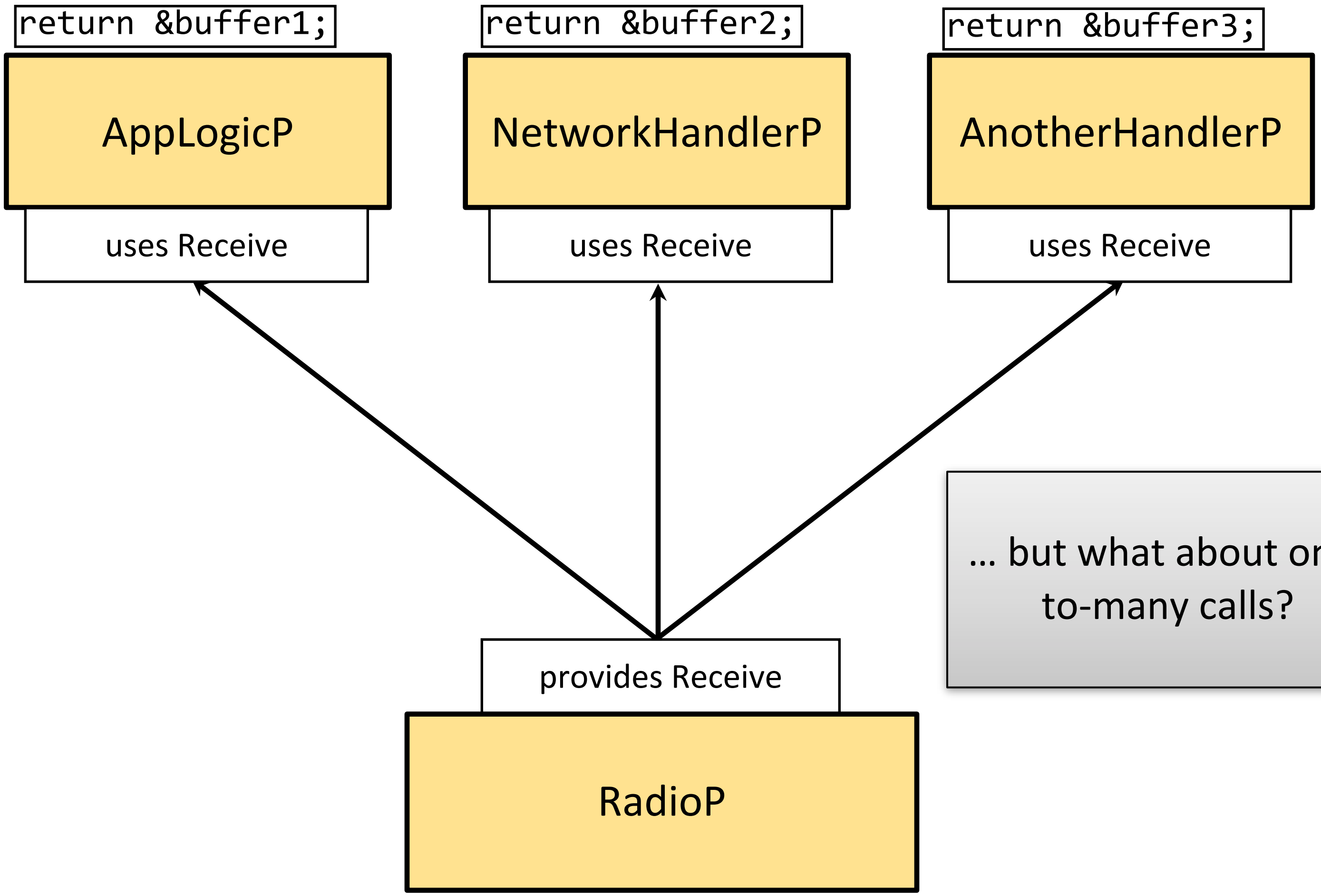
```
interface Read<type> {
  command error_t read();
  event void readDone(error_t error, type value);
}

module SixteenBitSensorP {
  provides interface Read<uint16_t>;
}

module BooleanSensorP {
  provides interface Read<bool>;
}
```

Fan-In: No Big Deal



Many-to-one calls work like you'd expect ...

Fan-Out: Bad Things Happen



… but what about one-to-many calls?

Fan-Out: What Bad Things Happen?

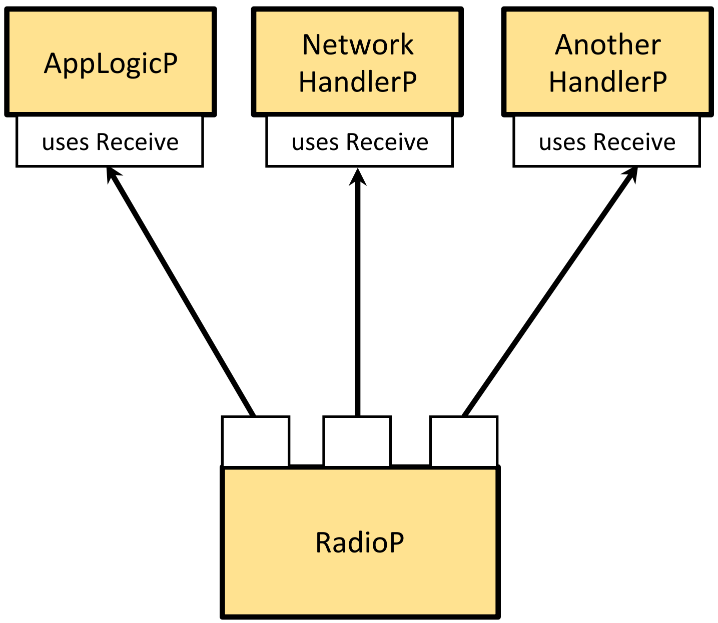
- If different return values come back, nesC may not be able to make sense of the contradiction and will *arbitrarily* pick one

- Avoid designs where this is possible
- If you can't avoid it, see TinyOS Programming Guide 5.2 for more info on combining return values

Parameterized Wiring

- Consider the following way to avoid fan-out:

```
module RadioP {
  provides interface Receive as Receive0;
  provides interface Receive as Receive1;
  provides interface Receive as Receive2;
  uses interface LowLevelRadio;
  ...
}
implementation {
  event void LowLevelRadio.packetReceived(
    uint8_t * rawPacket) {
    ...
    uint8_t type = decodeType(rawPacket);
    if(type == 0)
      signal Receive0.receive(...);
    else if(type == 1)
      signal Receive1.receive(...);
    ...
  }
  ...
}
```



Parameterized Wiring

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

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

Using Parameterized Wiring

- You can wire parameterized interfaces like so:

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

AnotherHandlerP -> RadioP.Receive[2];
```

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

Unique Parameters

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

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

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

AnotherHandlerP -> RadioP.Receive[unique(“RaadioP”)];
// unique(“RaadioP”) expands to 0 (oops)
...
```

uniqueCount()

- What if your component needs to store different state for each unique parameter?

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

  ...
}
```

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

Defaults

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

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

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

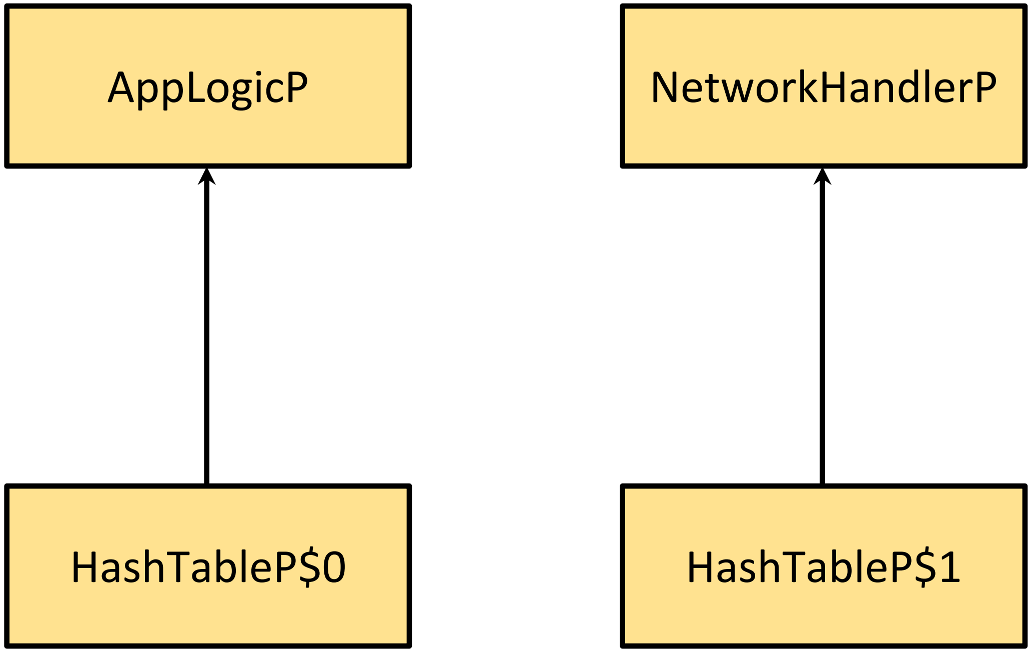
Generic Components

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

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

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

Generic Components

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



Generic Components with Parameters

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

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

components new ListP(bool, 16);
```

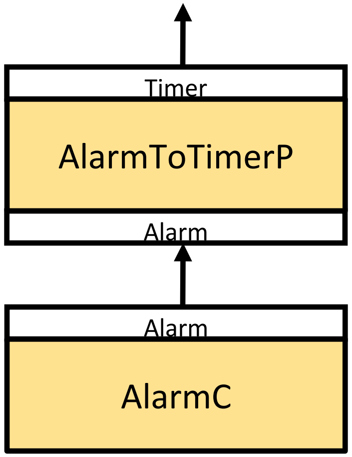
Putting It All Together: Building a Timer

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

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

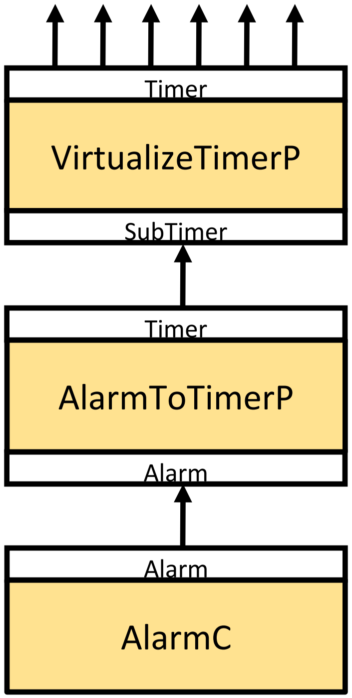
- We want to create a high-level timer component that:
  - Runs outside of the asynchronous context
  - Can be hooked into multiple components
  - Each consumer can choose a custom firing interval (every *n* ticks)
  - Can be transformed into lower frequencies (16 KHz, 1 Hz, etc.)

Step 1: Get Out of Asynchronous Context



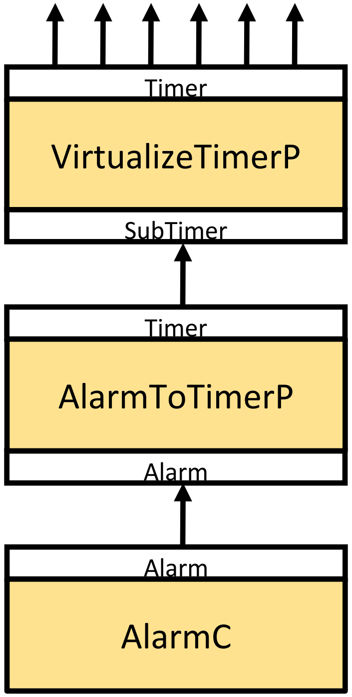
Step 2: Virtualize the Timer

```
module VirtualizeTimerP {
  uses interface Timer as SubTimer;
  provides interface Timer[uint8_t id];
}
implementation {




}
```



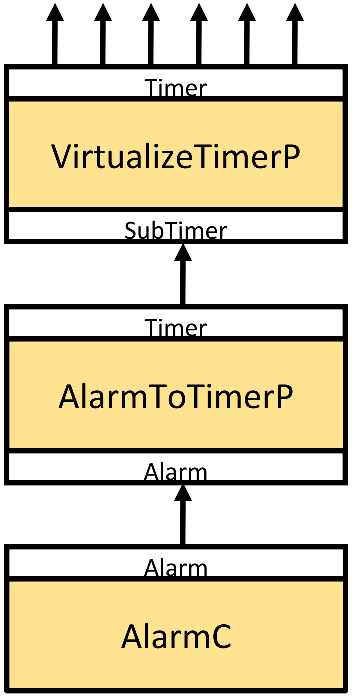
Step 3: Reprogram the Timer

```
interface Timer /* v. 2 */ {
  event void fired();
  command void startPeriodic(uint16_t interval);
}
```

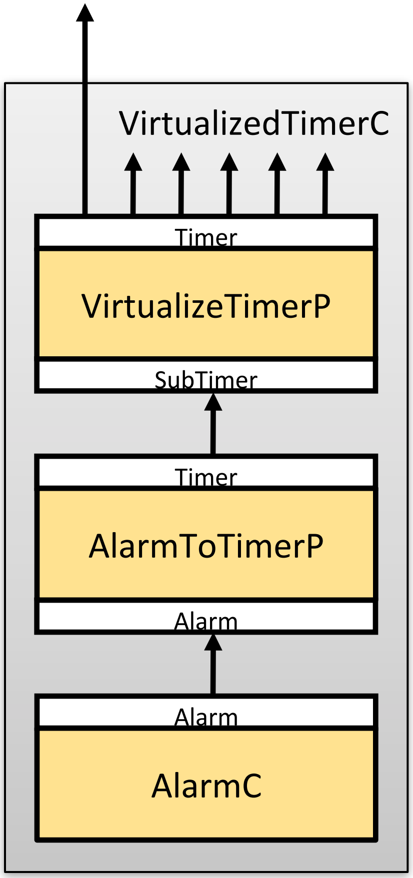


Step 3: Reprogram the Timer

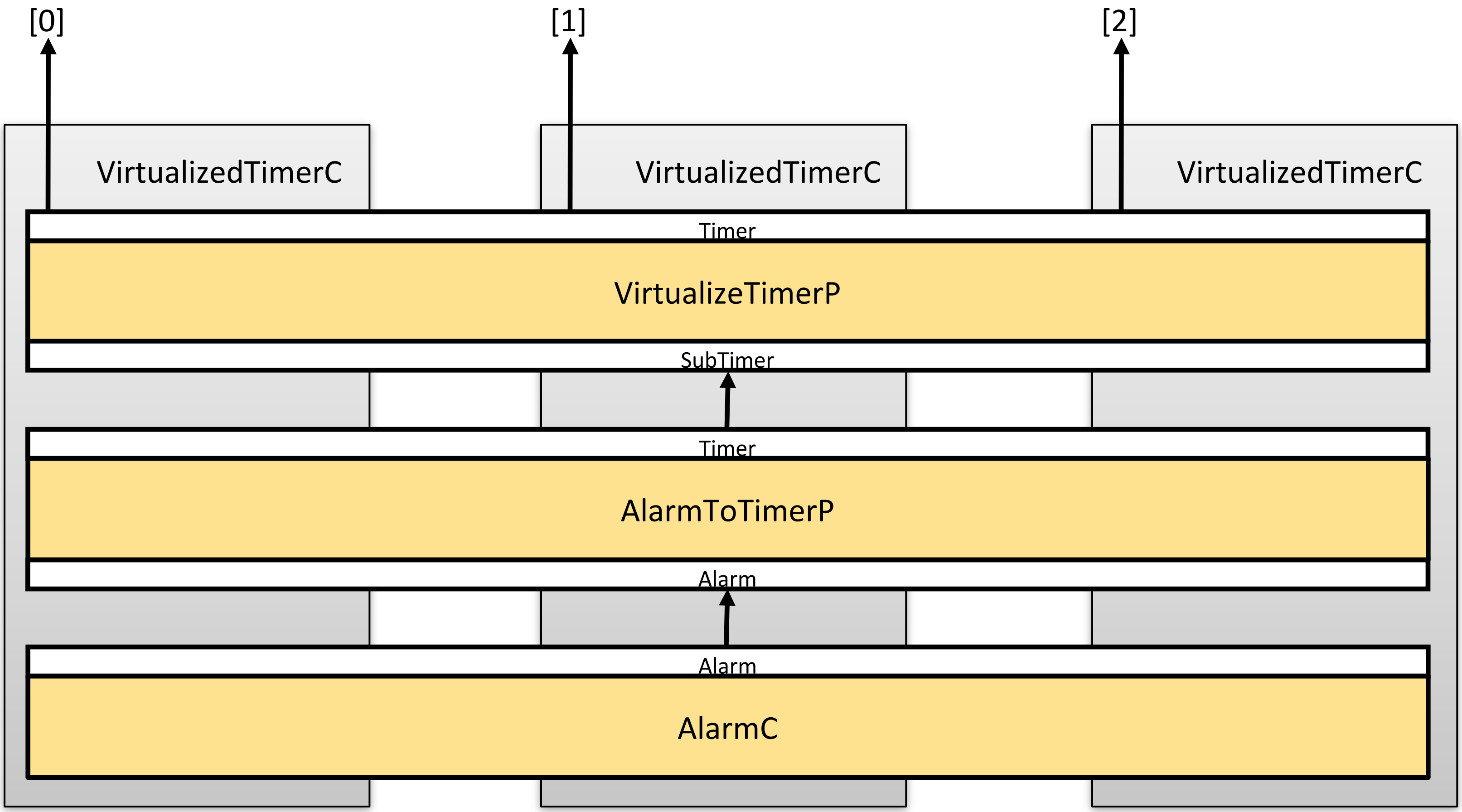
```
module VirtualizeTimerP /* v. 2 */ {
  ...
}
implementation {
  uint16_t currentTime = 0;
  uint16_t nextTimeToFire[255];
  uint16_t intervals[255];

  event void SubTimer.fired() {
    uint8_t i;
    for(i = 0; i < 255; i++) {
      if(nextTimeToFire[i] == currentTime) {
        signal Timer.fired[i]();
        nextTimeToFire[i] += intervals[i];
      }
    }
    currentTime++;
  }

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



Step 3.5: Tidy Up the Wiring

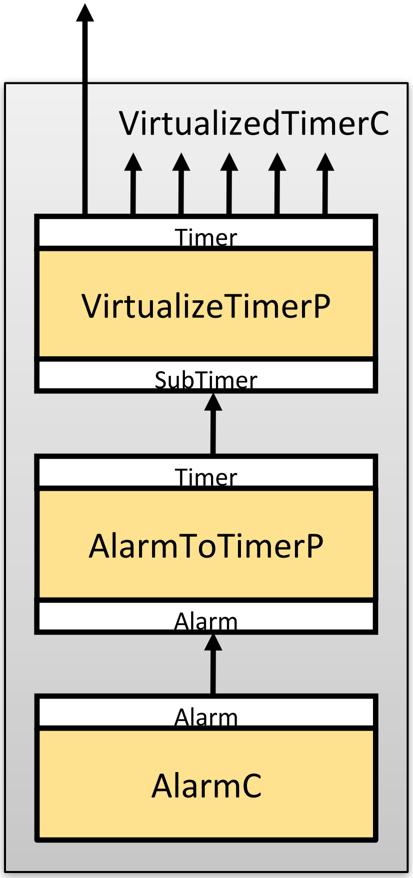


Step 3.5: Tidy Up the Wiring

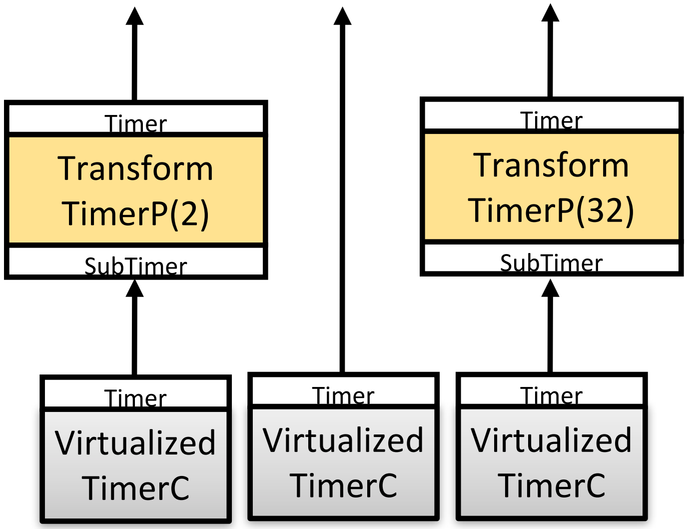


Step 3.5: Tidy Up the Wiring

```
module VirtualizeTimerP /* v. 2.5 */ {
  ...
}
implementation {
  enum {
    NUM_SLOTS =
      uniqueCount("VirtualizedTimerC");
  }
  uint16_t currentTime = 0;
  uint16_t nextTimeToFire[NUM_SLOTS];
  uint16_t intervals[NUM_SLOTS];

  event void SubTimer.fired() {
    uint8_t i;
    for(i = 0; i < NUM_SLOTS; i++) {
  ...
```



Step 4: Transform the Timer's Frequency

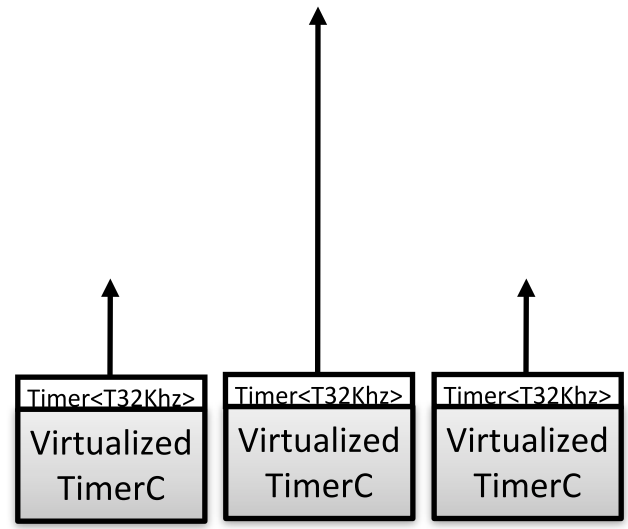


Step 5: Add Type Safety

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

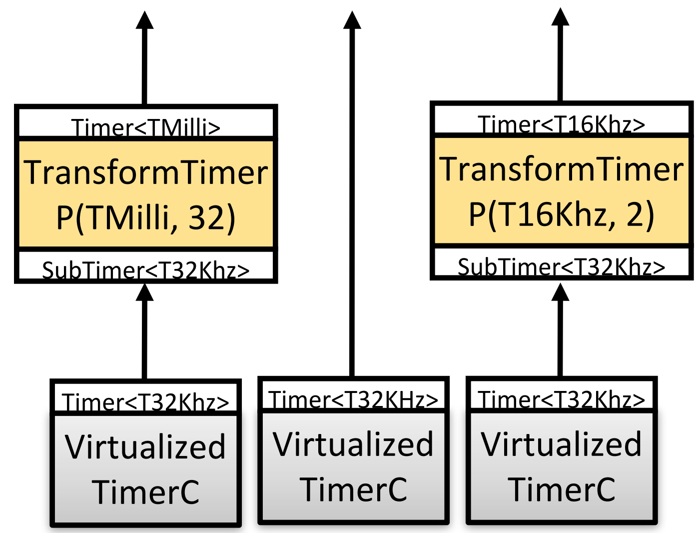
```
typedef struct {
  bool unused;
} T32Khz;

typedef struct {
  bool unused;
} T16Khz;

typedef struct {
  bool unused;
} TMilli;
```

```
enum {
   T32Khz,
   T16Khz,
   …
   TMilli,
};
```



Step 5: Add Type Safety

```
generic module TransformTimerP(

  uint16_t multiplier) {
  uses interface Timer<T32Khz> as SubTimer;
  provides interface Timer<              ;
}
implementation {
  event void SubTimer.fired() {
    signal Timer.fired();
  }

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



The Good News

- This is just an example! It's already been implemented for you
- `TimerMilliC` component provides `Timer<TMilli>` interface

Outline

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

Slight Diversion: App Bootstrapping

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

```
interface Boot {
  event void booted();
}
```

Slight Diversion: App Bootstrapping

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

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

```
module MyAppP {
  uses interface Boot;
}
implementation {
  event void Boot.booted() {
    // Initialize app here
  }
  ...
}
```

Slight Diversion #2: `error_t` Data Type

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

Message Addressing

- Each node has a unique 16-bit address (`am_addr_t`) specified by the make command
`make install.[address] platform`
- Two special address constants:
  - TOS_BCAST_ADDR (0xFFFF) is reserved for broadcast traffic
  - TOS_NODE_ID always refers to the node's own address
- Each message also has an 8-bit Active Message ID (`am_id_t`) analogous to TCP ports
  - Determines how host should handle received packets, not which host receives it

TinyOS Active Messages

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

```
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;
```

Header	Payload (TOSH_DATA_LENGTH)	Footer	Metadata

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 ...tions split-phase
  - Application issues ...
  - An event is signal...

```
interface SplitContr...
  command error_t start();
  event void startDone(error_t error);

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

Error code here indicat...
whether TinyOS...
processing req...

Error code here indicates
whether TinyOS could
complete processing request

Active Messaging Interfaces

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

interface Receive {
  event message_t* receive(message_t * msg, void *
    payload, uint8_t len);
}
```

Other Networking Interfaces

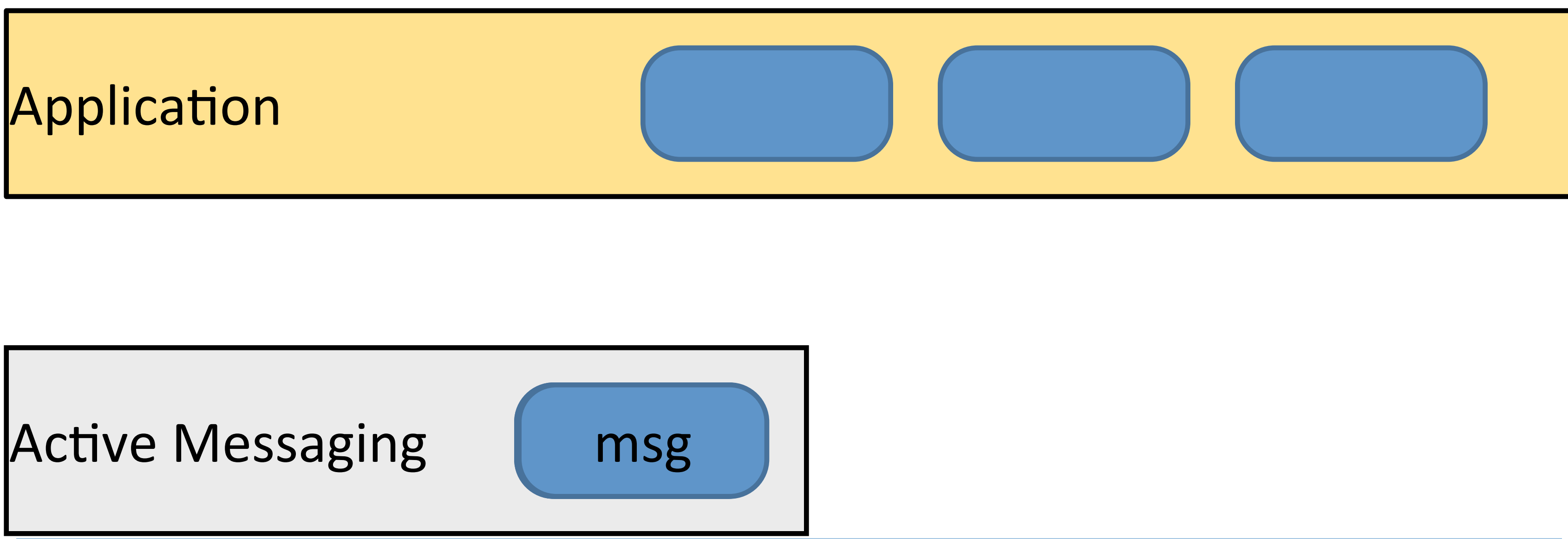
```
interface Packet {
  command void clear(message_t * msg);

  command void* getPayload(message_t * msg, uint8_t
    len);

  command uint8_t payloadLength(message_t * msg);
  command void setPayLoadLength(message_t * msg, uint8_t
    len);

  command uint8_t maxPayloadLength();
}
```

Other Networking Interfaces

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

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

  command am_id_t type(message_t* amsg);
}
```

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



Network Types

- Radio standards like 802.15.4 mean that you could have communication among different types of motes with different CPUs
- nesC defines network types (`nx_uint16_t`, `nx_int8_t`, etc.) that transparently deal with endian issues for you
- nesC also defines an `nx_struct` analogous to C `structs`

```
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)

```
enum {
  AM_SENSORREADING = 240,
};

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

Sending a Message

- Declare a `message_t` variable in your module to store the packet's contents
- Get the packet's payload using the `Packet` interface; cast it to your message type; and store whatever you want to send

```
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;
    ...
  }
}
```

Sending a Message

- Finally, use the AMSend interface to send the packet

```
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

```
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:

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

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

Networking Components

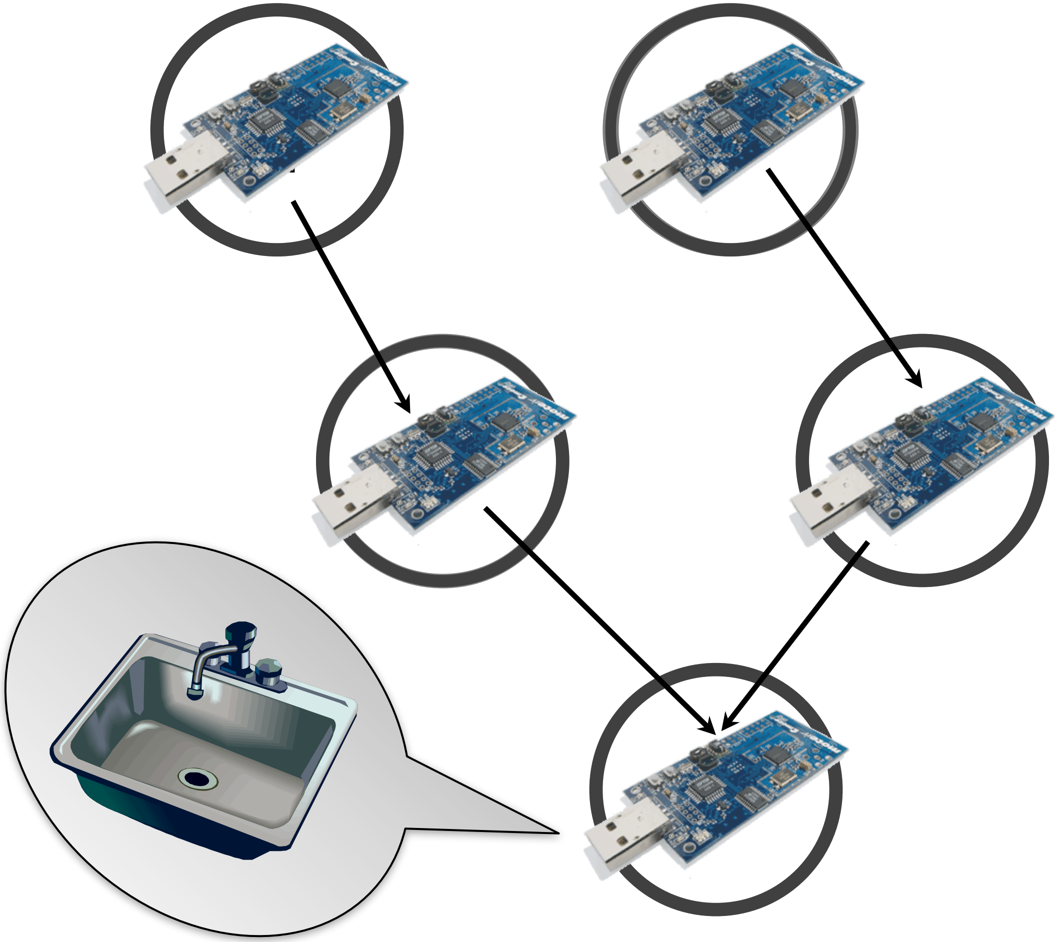
- Before you can send/receive, you need to turn the radio on
- `ActiveMessageC` component provides a `SplitControl` interface to control the radio's power state

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

What About Multi-Hop?

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

Collection Tree Protocol (CTP)



Dissemination

For More Information

- CTP & Dissemination APIs are beyond the scope of this talk

- For more information, see:
  - TinyOS Tutorial 12: Network Protocols (http://**docs**.tinyos.net/index.php/Network_Protocols)
  - TEP 123: Collection Tree Protocol (http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html)
  - TEP 118: Dissemination (http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html)

Sending Data to a PC

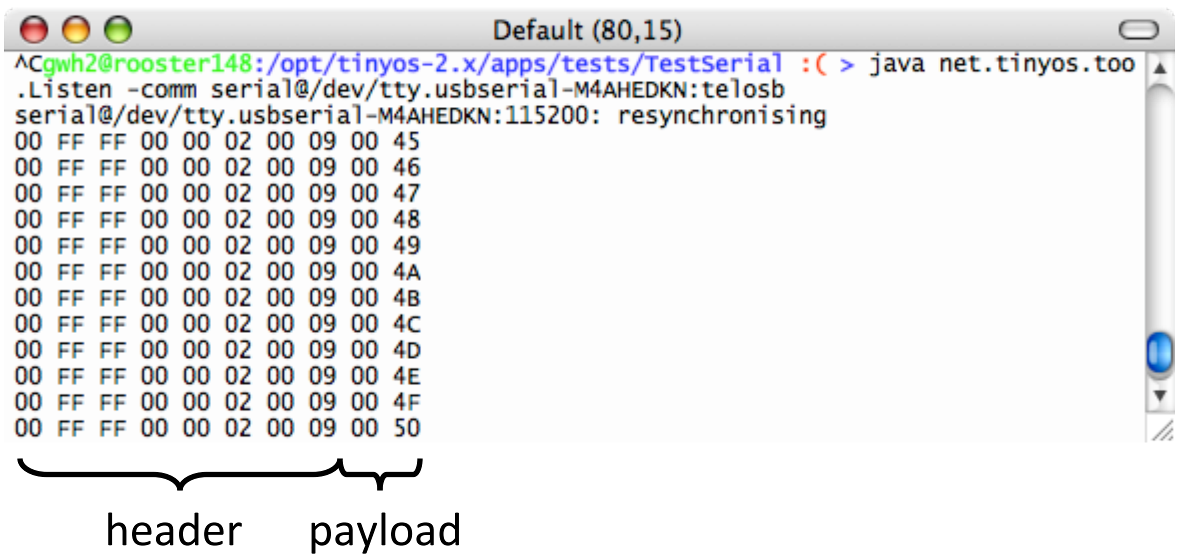
- TinyOS apps can also send or receive data over the serial/ USB connection to an attached PC
- The SerialActiveMessageC component provides an Active Messaging interface to the serial port:

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

Displaying Received Data

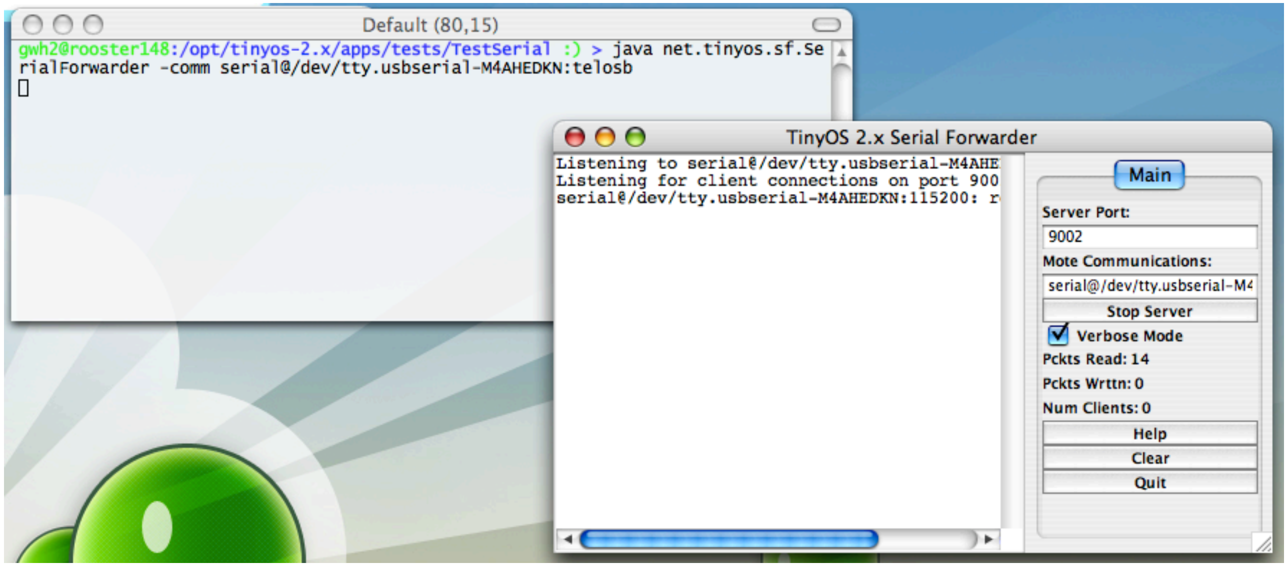
- Java application: `net.tinyos.tools.Listen`
- To specify which mote to read from, use the command-line parameter

  `-comm serial@[port]:[platform]`



Disseminating Received Data

- Java application: `net.tinyos.sf.SerialForwarder`
- Other PCs on the network can connect to the Serial Forwarder to access the sensor data



Java PC-to-Mote Interface

- MIG: Message Interface Generator
  - Generates a Java class representing a TOS message
  - Usage:
    ```
    mig java -java-classname=[classname] [header.h]
    [message-name] –o [classname].java
    ```
- TinyOS Java SDK includes a `net.tinyos.message.MoteIF` class for interfacing with motes using Java
  - See `$TOSROOT/apps/tests/TestSerial/TestSerial.java` for an example

PC-to-Mote Interface in Other Languages

- C/C++: thorough but not well-documented
  - 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

- Python: fairly good support, with one catch
  - Python classes in `$TOSROOT/support/sdk/c/python` closely mirror Java SDK
  - Curiously, code to interface directly with serial ports is missing
  - See `tinyos/message/MoteIF.py` to get started

Outline

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

Obtaining Sensor Data

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

```
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

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

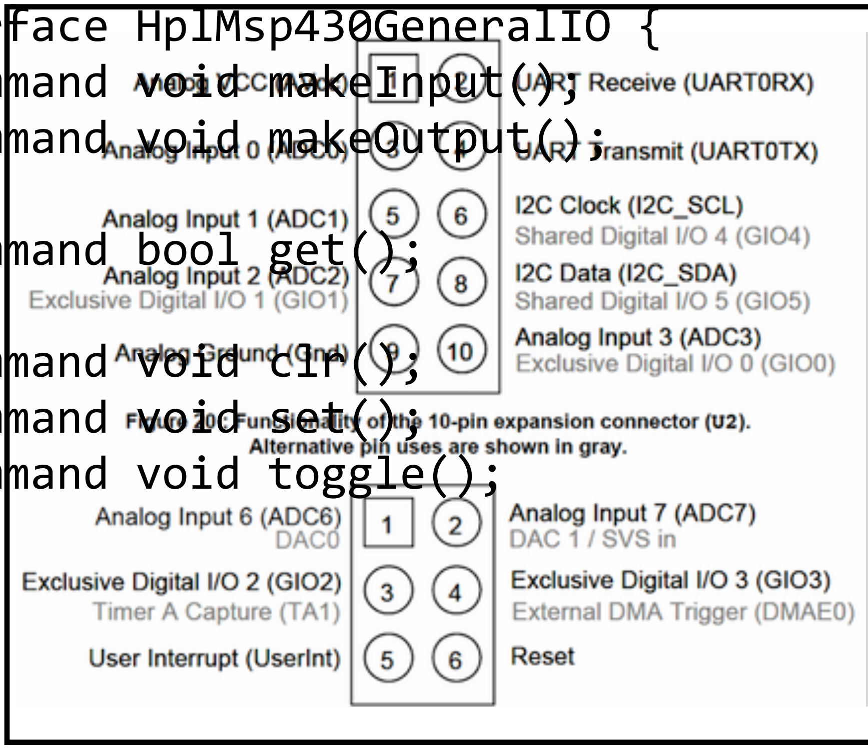
```
module MyAppP {
  uses interface Read<uint16_t> as AccelX;
  ...
}
implementation {
  ...
  task void readAccelX() {
    if(call AccelX.read() != SUCCESS)
      post readAccelX();
  }
  event void AccelX.readDone(error_t err,
    uint16_t reading) {
    if(err != SUCCESS) {
      post readAccelX();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

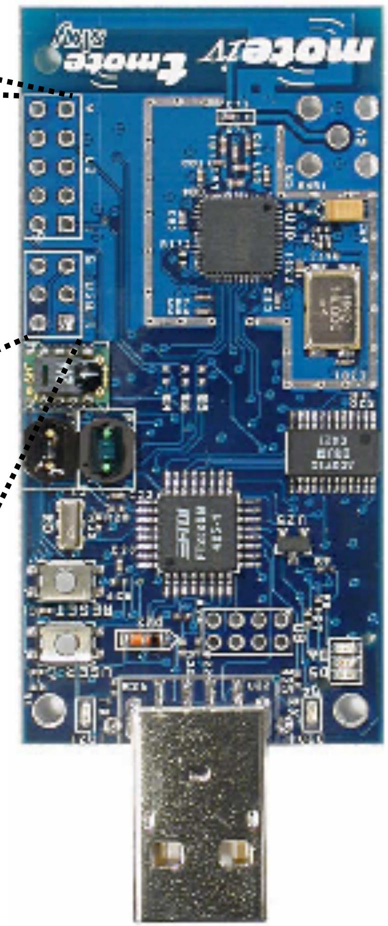
Sensor Components

- 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

External Sensors

```
interface HplMsp430GeneralIO {
  command void makeInput();
  command void makeOutput();

  command bool get();

  command void clr();
  command void set();
  command void toggle();
}
```





External Sensors

- Digital I/O: wire directly into `HplMsp430GeneralIOC` component

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

- Analog I/O: read TEP 101 (Analog-to-Digital Converters)

Outline

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

Hard-Learned Lessons

- Be sure to check return values -- don't assume `SUCCESS`!
  - At the very least, set an LED when something goes wrong
- The TinyOS toolchain doesn't always warn about overflowing integers

```
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:

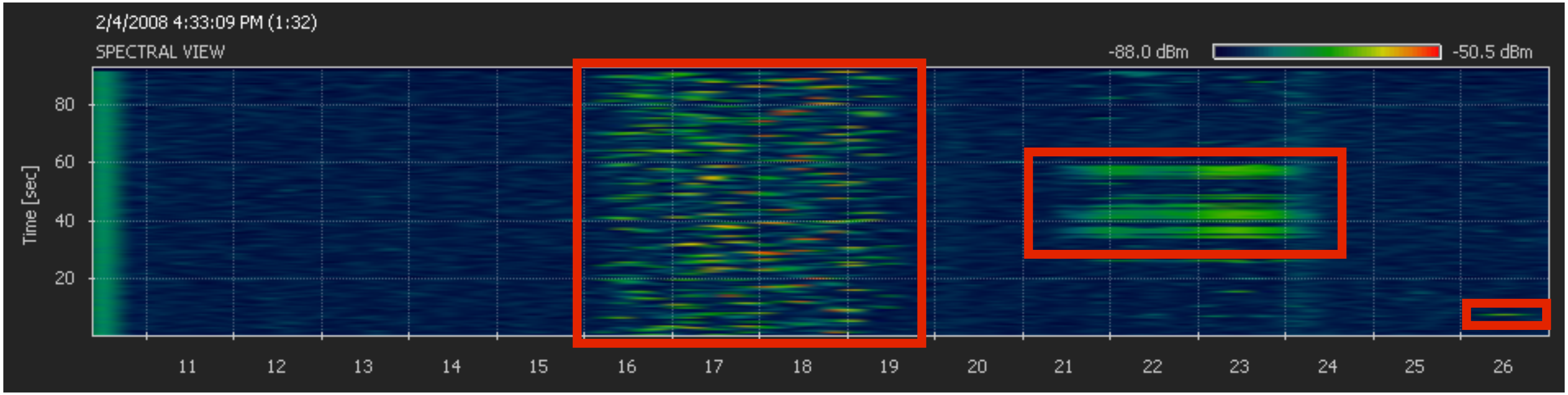
```
/opt/tinyos-2.x/tos/interfaces/TaskBasic.nc: In function `SchedulerBasicP$TaskBasic
$runTask':
/opt/tinyos-2.x/tos/interfaces/TaskBasic.nc:64: unable to generate reloads for:
(call_insn 732 3343 733 (set (reg:SI 15 r15)
        (call (mem:HI (symbol_ref:HI ("AsyncQueueC$1$Queue$dequeue")) [0 S2 A8])
            (const_int 0 [0x0]))) 14 {*call_value_insn} (nil)
    (nil)
    (nil))
/opt/tinyos-2.x/tos/interfaces/TaskBasic.nc:64: Internal compiler error in
find_reloads, at reload.c:3590
```

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

```
typedef nx_struct my_msg
{
     nx_uint8_t field1;
     nx_uint8_t pad;
     nx_uint16_t field2;
} my_msg_t;
```

802.15.4 Radio Channels

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



802.15.4 Radio Channels

- If you're seeing weird network behavior, set your CC2420 channel to something else:
  - Defaults to 26
  - Command-line: `CC2420_CHANNEL=xx make ...`
  - Makefile: `PFLAGS = -DCC2420_DEF_CHANNEL=xx`

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:

```
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
  - The messages are sent in a `printf_msg` structure (`$TOSROOT/tos/lib/printf/printf.h`)
- Though `printf()` ships with TinyOS, its components are not automatically located by the included `Makefile` stubs
- To force make to locate the `printf()`-related components, add the following line to your `Makefile`:
  `CFLAGS += -I$(TOSDIR)/lib/printf`
- Note: adding this flag automatically turns on `SerialActiveMessageC` subsystem

`BaseStation`

- The `BaseStation` app in `$TOSROOT/apps/BaseStation` will sniff all wireless traffic and forward it to the serial port
- Extremely helpful for figuring out what data is being sent!

TOSSIM

- Special target: `make micaz sim`
- Compiles application to native C code for your own machine, which can be loaded into Python or C++ simulator ("TOSSIM")
- Upshot: use your favorite Python or C++ debugger to trace through your app's execution
- Unfortunately somewhat complex and beyond the scope of this talk; see TinyOS Tutorial 11
  - http://docs.tinyos.net/index.php/TOSSIM

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

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/

Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr, "Efficient Memory Safety for TinyOS," Proceedings of 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007), 2007.