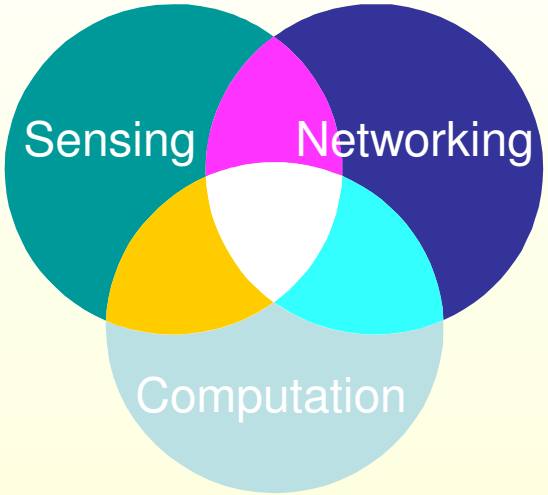
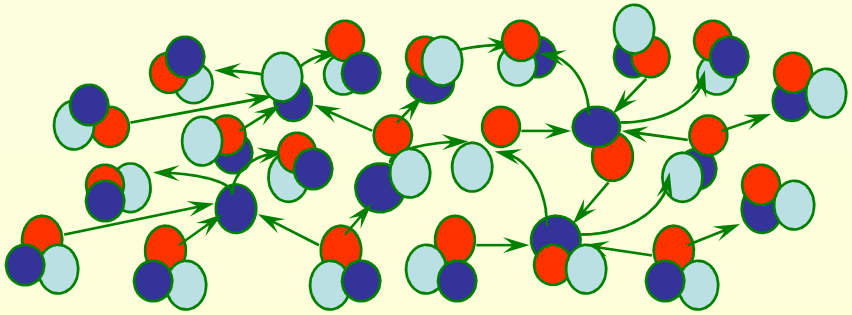
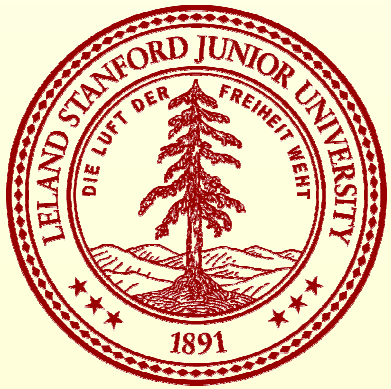
CS321: Programming Sensor Networks: nesC, TinyOS, TOSSIM





Leonidas Guibas
Brano Kusy, Primoz Skraba
Computer Science Dept.
Stanford University



Class Projects

http://graphics.stanford.edu/courses/cs321-07-fall/project.html

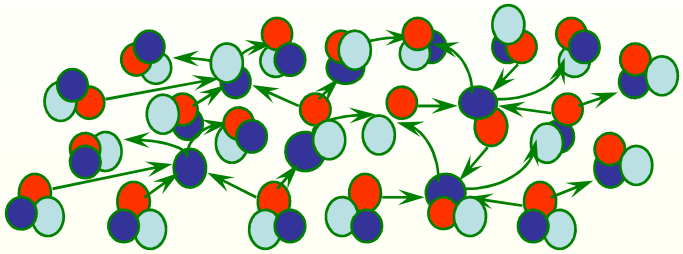
Warm-up project (5%)

- How many people?
- Everybody will get 2 iMote2s, a sensorboard, a battery board, and a debug board, come to Primoz's office to pick them up
- Due October 22nd
- The code is provided by us – majority of the work is to set up TinyOS-1.x development environment for iMote2
- Parts of the code are missing – you need to implement these
- Working applications need to be demonstrated on the due date

Today, we'll learn how to program sensor networks:

- We'll build a simple "anti-theft" application using TinyOS
  (in less than 200 lines of code with comments)
- We'll program iMote2 and run the anti-theft application

Why are sensor networks useful?



Untethered micro sensors will go anywhere and measure anything -- traffic flow, water level, number of people walking by, temperature. This is developing into something like a nervous system for the earth. -- Horst Stormer in *Business Week, 8/23-30, 1999.*

1. Provide better data resolution
   - Close to the observed phenomena
   - Mobility
2. Collect data over wide time and space intervals
3. Low cost and fast deployment

Constraints

- no wires
- run on batteries, use radio
- small size, large numbers
- low power consumption
- low cost

=> constrained size

=> constrained hardware

iMote2 platform

Processing

- PXA271 XScale® Processor at 13–416MHz
- 256kB SRAM, 32MB FLASH
- Compact Size: 36 x 48 x 9mm

Sensing

- 3-Axis Accelerometer,
- Temperature,
- Humidity, and
- Light Sensor

Communication

- Integrated 802.15.4 Radio
- Integrated 2.4GHz Antenna
- 250 kbps

Challenges

Driven by interaction with environment

- Concurrency-intensive operation
  (multiple parallel data flows, asynchronous and synchronous devices)
- Requires event-driven execution

Extremely limited resources (“3 AAA’s, 256kB of RAM”)

- Very low cost, size, and power consumption
- Primitive direct-to-device interfaces

Real-time requirements

- Some time-critical tasks (sensor acquisition and radio timing)
- Timely action: never block, complete control over app and OS

Usage diversity and constant hardware evolution

- Application specific hardware => huge hardware variety
- Modularity, hardware independent interfaces

Outline

- TinyOS and nesC overview
- Warm-up project: building a simple anti-theft application
  - The Basics
  - Networking
  - Communication with a PC
- Programming iMote2
- TOSSIM: simulating TinyOS code
- Communication with a PC
- Reflections and References

TinyOS and nesC

- TinyOS is an open source operating system designed for limited-resource sensor networks; it is a collection of software modules that can be glued together to build applications
  - TinyOS 0.4, 0.6 (2000-2001)
  - TinyOS 1.0 (2002): first nesC version
  - TinyOS 1.1 (2003): reliability improvements, many new services
  - TinyOS 2.0 (2006): complete rewrite, improved design, portability, reliability and documentation
- nesC is the language in which TinyOS modules are implemented:
  - nesC 1.0 (2002): Component-based programming
  - nesC 1.1 (2003): Concurrency support
  - nesC 1.2 (2005): Generic components, “external” types

TinyOS in a nutshell

System runs a single application

- OS services can be tailored to the application's needs

These OS services include

- timers, radio, serial port, A/D conversion, sensing, storage, LEDs, …

Application and services are built as

- a set of interacting components (as opposed to threads)
- using a strictly non-blocking execution model
  - event-driven execution

Implementation based on a set of OS abstractions

- tasks, atomic with respect to each other
- hardware abstraction architecture
- split-phase service requests
- data types for inter-operability

nesC in a seashell

C dialect

Component based

- all interaction via interfaces
- connections (“wiring”) specified at compile-time
- interfaces for code reuse, simpler programming

“External” types to simplify interoperable networking

Reduced expressivity

- no dynamic allocation
- no function pointers

Supports TinyOS’s concurrency model

- must declare code that can run in interrupts
- atomic statements to deal with data accessed by interrupts
- data race detection to detect (some) concurrency bugs

Warm-up Project

Goal:

- a simple anti-theft application which protects mote from being stolen.

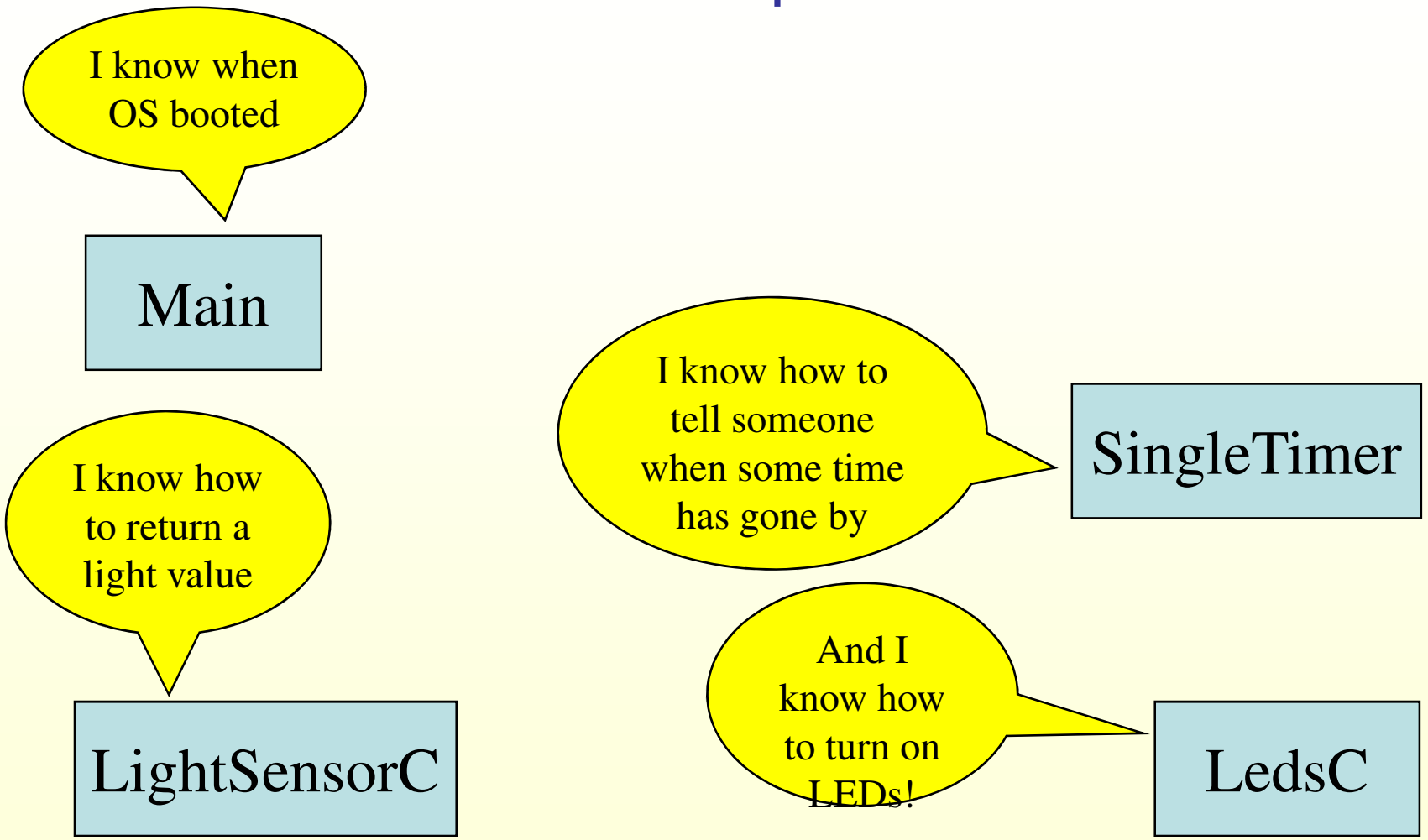
Tasks:

- Detecting theft.
  - Assume: thieves put the motes in their pockets.
  - So, a “dark” mote is a stolen mote.
  - detection algorithm: every N ms check if light sensor is below threshold.
- Reporting theft.
  - Assume: bright flashing lights deter thieves.
  - Theft reporting algorithm: light the **red LED** for a little while!

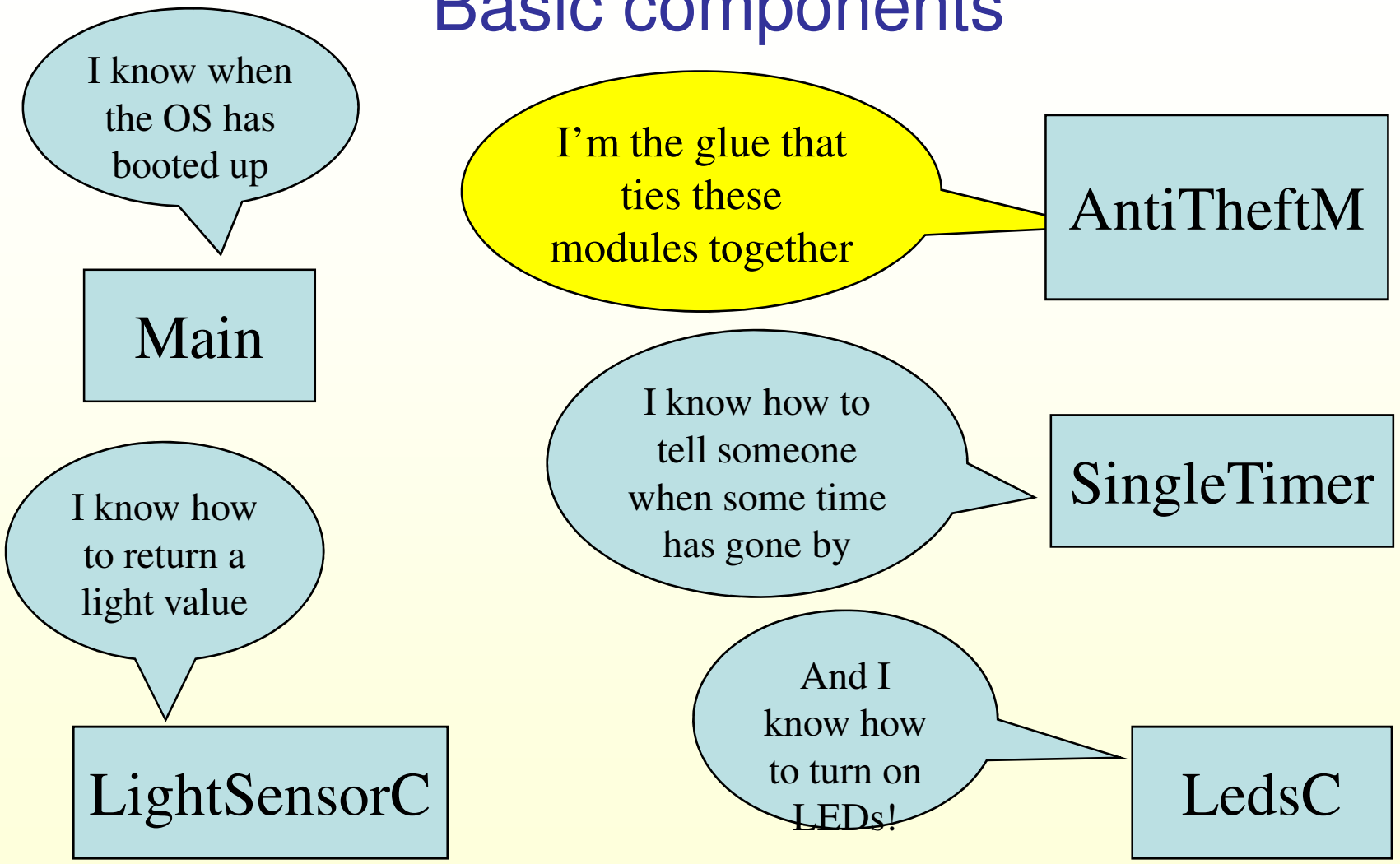
You will see

- Basic components, interfaces, wiring
- Essential system interfaces for startup, timing, sensor sampling

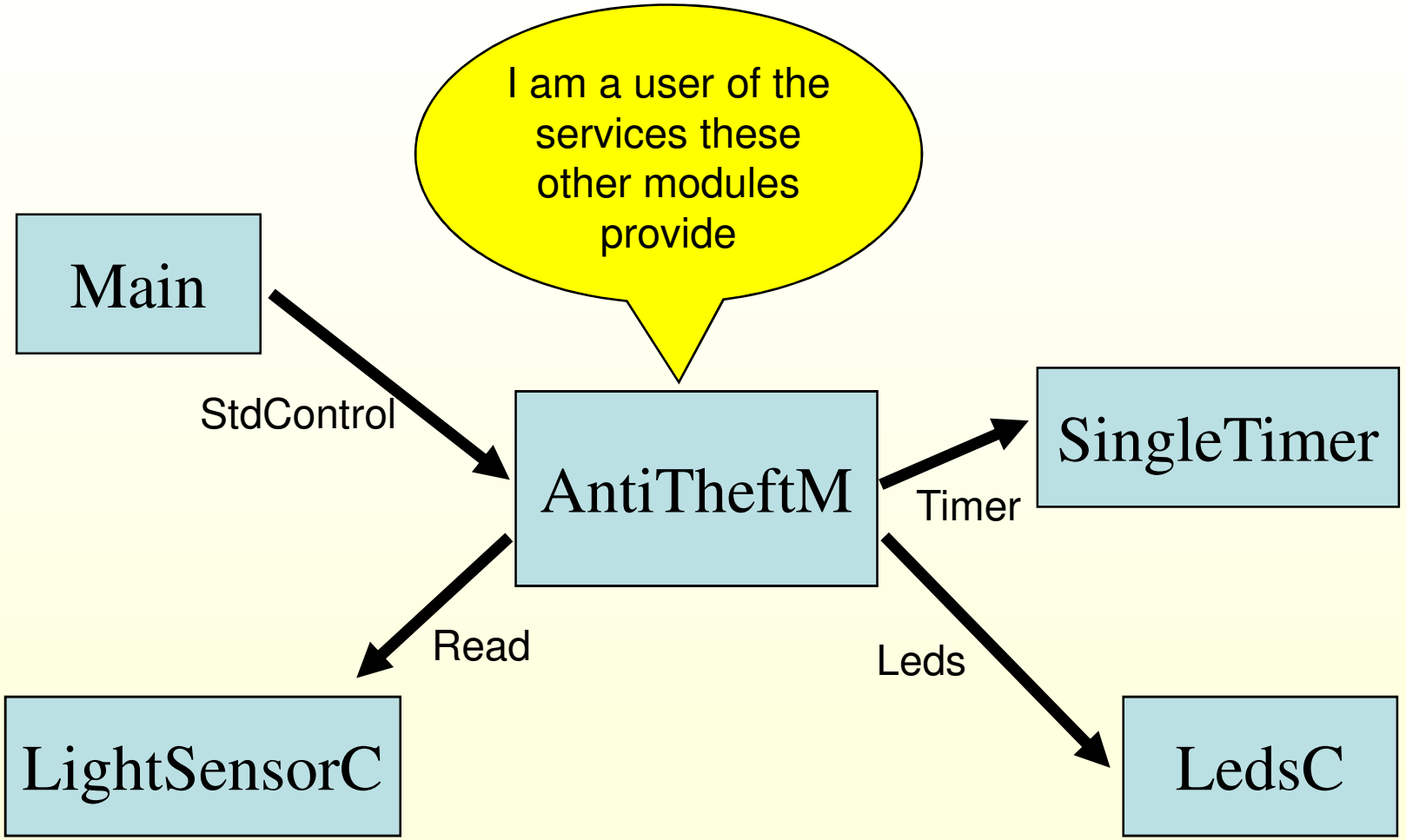
Basic components



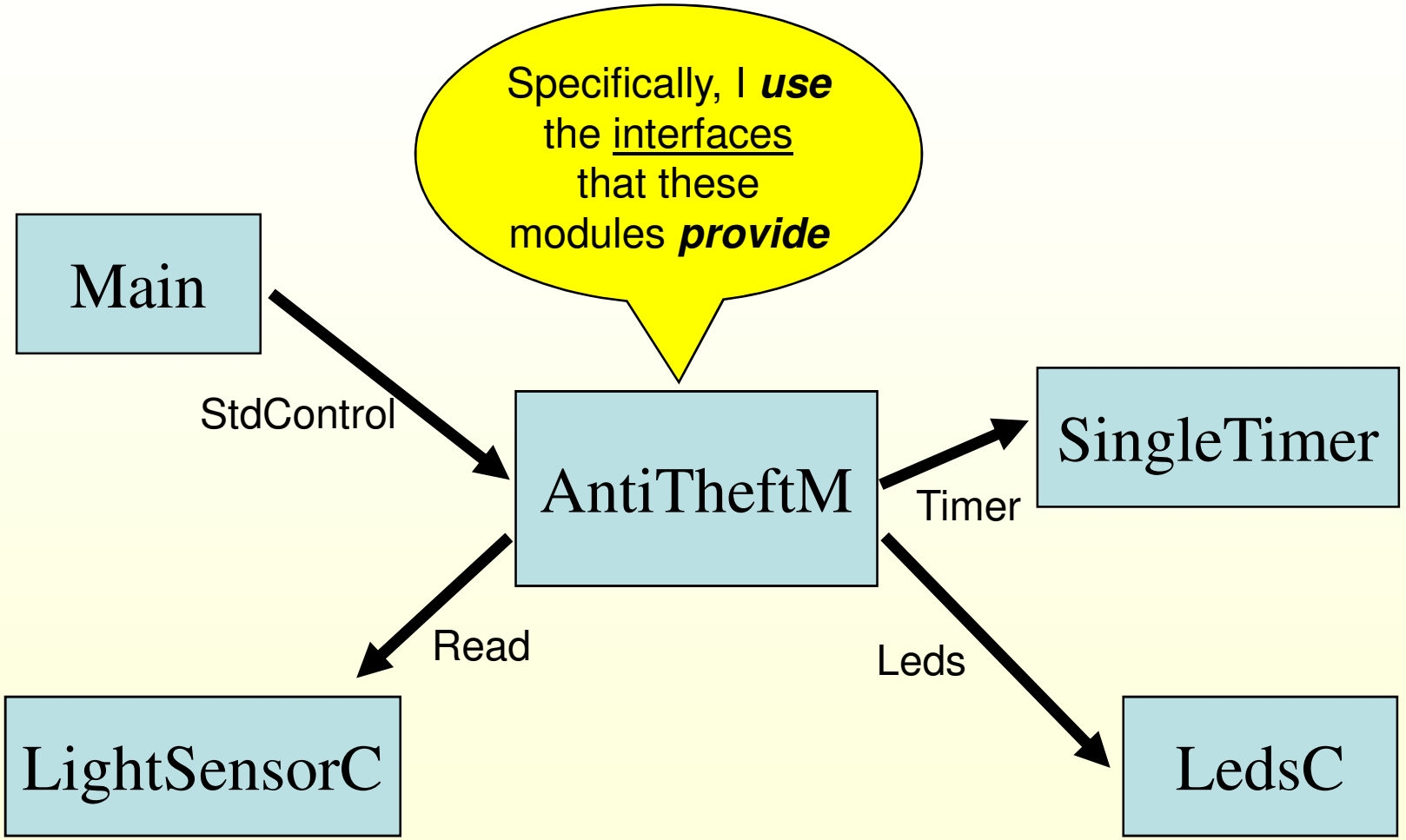
Basic components



Basic components



Basic components



Basics: Programs and Interfaces

Programs are built out of named components (think objects): modules and configurations.
A component has a name, specification, and implementation.
A module is a component implemented in C.

Main

Std

LightSen

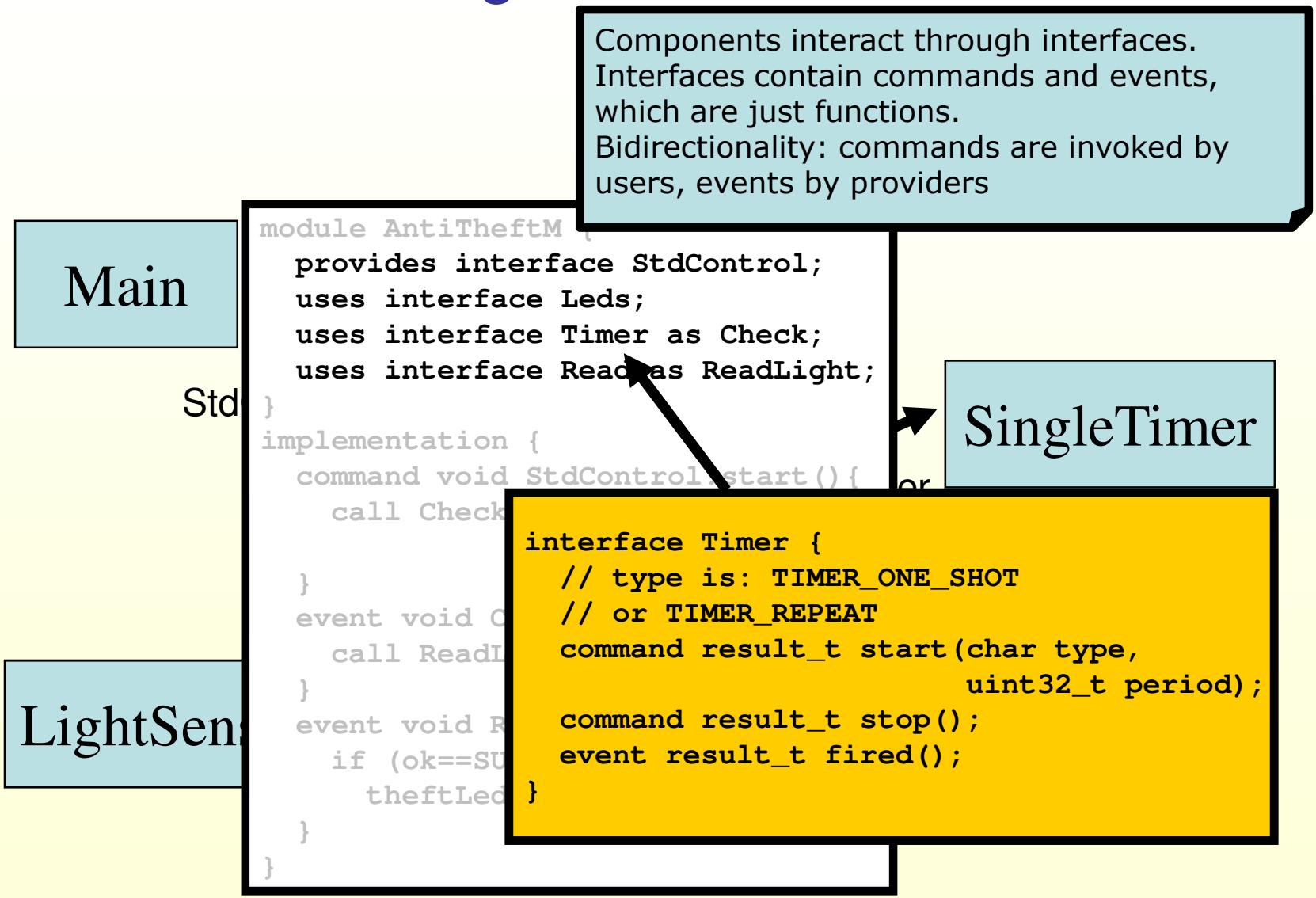
SingleTimer

er

LedsC

```
module AntiTheftM {
  provides interface StdControl;
  uses interface Leds;
  uses interface Timer as Check;
  uses interface Read as ReadLight;
}
implementation {
  command void StdControl.start(){
    call Check.start(TIMER_REPEAT,
                     1000);
  }
  event void Check.fired() {
    call ReadLight.read();
  }
  event void ReadLight.readDone(…){
    if (ok==SUCCESS && val<200)
      theftLed();
  }
}
```

Basics: Programs and Interfaces

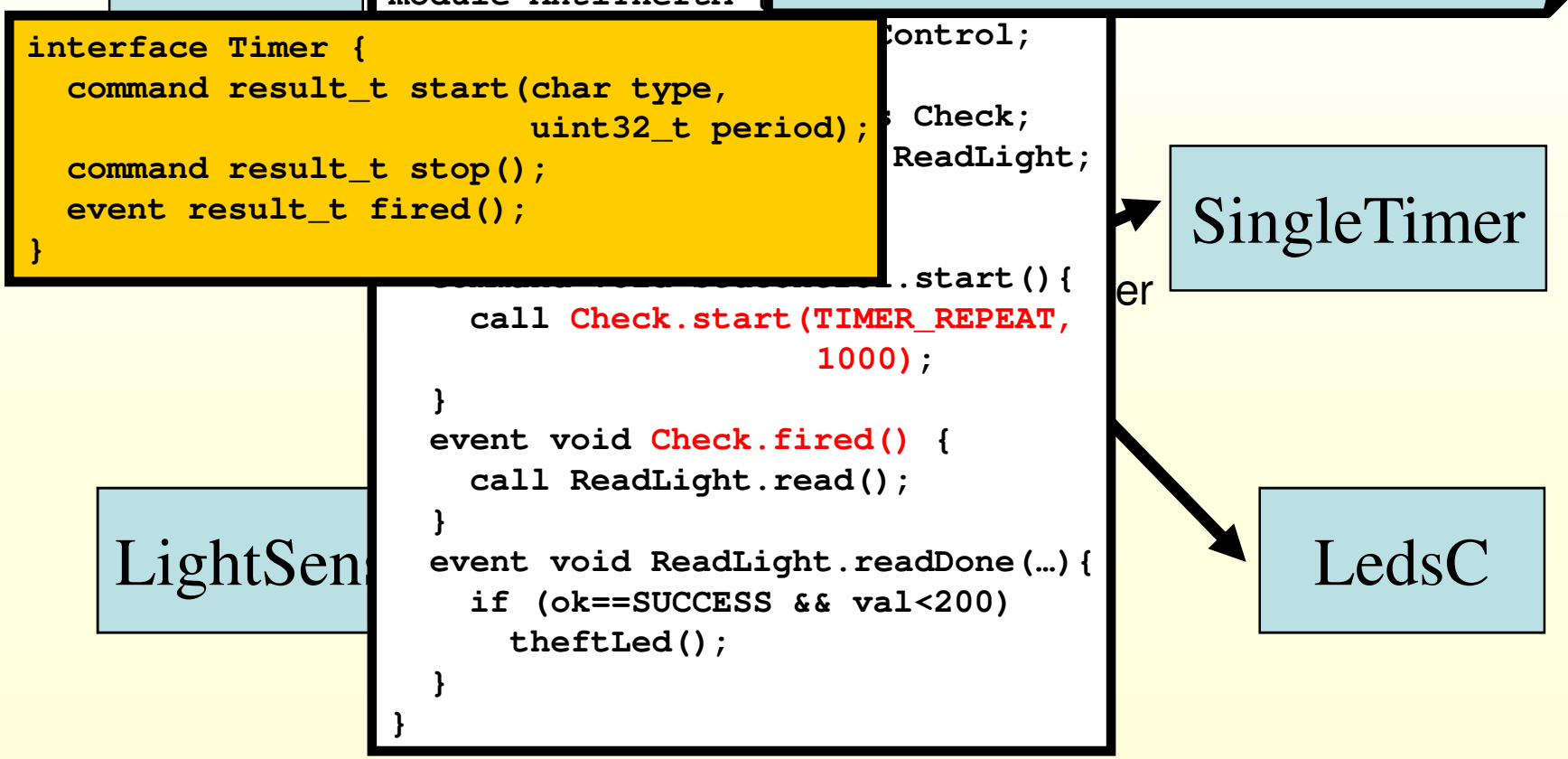


Basics: Interfaces

Components interact through interfaces.
Interfaces contain commands and events, which are just functions.
Bidirectionality: commands are invoked by users, events by providers

```
interface Timer {
  command result_t start(char type,
                         uint32_t period);
  command result_t stop();
  event result_t fired();
}
```

```
module AntiTheftM {
...ontrol;
... Check;
... ReadLight;
...
...er
...start(){
    call Check.start(TIMER_REPEAT,
                     1000);
  }
  event void Check.fired() {
    call ReadLight.read();
  }
  event void ReadLight.readDone(…){
    if (ok==SUCCESS && val<200)
      theftLed();
  }
}
```



Basics: Split Phase Interfaces

All long-running operations are split-phase:
- A command starts the op: read
- An event signals op completion: readDone

Components interact through interfaces. Interfaces contain commands and events, which are just functions.
Bidirectionality: commands are invoked by users, events by providers

```
  ace StdControl;
  uses interface Leds;
  uses interface Timer as Check;
  uses interface Read as ReadLight;
}
implementation {
  command void S
    call Check.s

  }
  event void Check.fired() {
    call ReadLight.read();
  }
  event void ReadLight.readDone(…){
    if (ok==SUCCESS && val<200)
      theftLed();
  }
}
```

```
interface Read {
  command result_t read();
  event void readDone( result_t ok,
                       uint32_t val );
}
```

Main

Std

LightSen

LedsC

Basics: Split Phase Interfaces

All long-running operations are split-phase:
- A command starts the op: read
- An event signals op completion: readDone

Errors are signalled using result_t type
- Commands only allow one request
- Events report problems during the op

Interfaces specify the interaction between two components, the *provider* and the *user*. This interaction is just a function call. commands are calls from user to provider events are calls from provider to user

Main

Std

```
  Leds;
  uses interface Timer as Check;
  uses interface Read as ReadLight;
}
implementation
  command void
    call Check.
  }
  event void Check.fired() {
    call ReadLight.read();
  }
  event void ReadLight.readDone(
       result_t ok, uint32_t val){
    if (ok==SUCCESS && val<200)
      theftLed();
  }
}
```

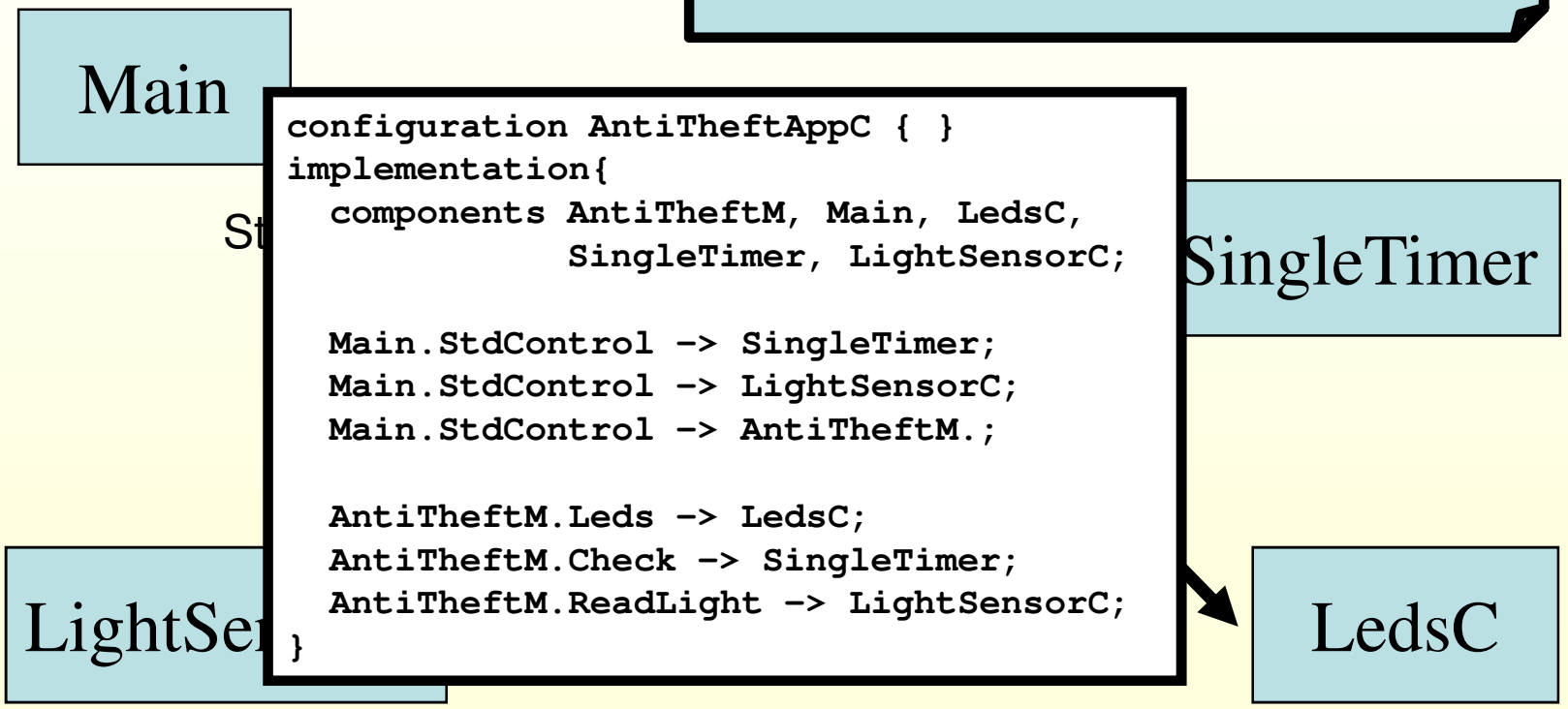
```
interface Read {
  command result_t read();
  event void readDone( result_t ok,
                       uint32_t val );
}
```

LightSen

LedsC

Basics: Configurations

A configuration is a component built out of other components.
It *wires* "used" to "provided" interfaces.
It can itself provide and use interfaces



The Basics - Summary

Components and Interfaces

- Programs built by writing and wiring components
  - modules are components implemented in C
  - configurations are components written by assembling other components
- Components interact via interfaces only

System services: startup, timing, sensing, LEDs (so far)

- Represented by named components
- All slow system requests are split-phase

Advanced AntiTheft

Let's improve our anti-theft application.

- Different anti-theft mechanisms may be appropriate for different times or places. Our perfect anti-theft system must be configurable!
- We also need to have a backup plan if the bright LED does not deter the thief: report the theft to a PC.

Assume:

- a base station (dedicated mote) is connected to a PC via USB cable and the PC is running our java application.

Tasks:

Reconfigurability

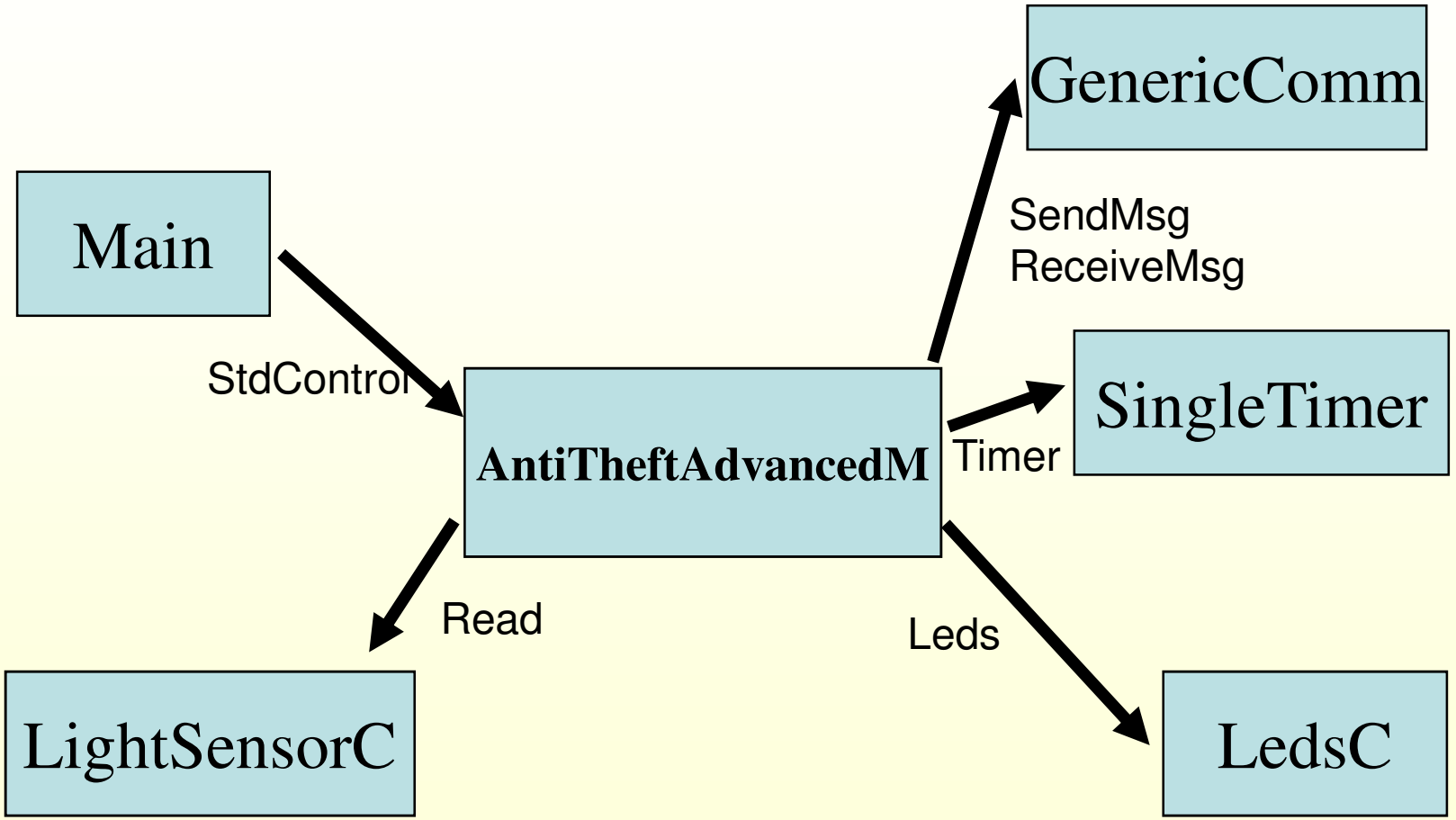
- Some settings, such as light level threshold or frequency of the theft detection algorithm can be changed and updated on the motes from the java application.

Advanced theft reporting

- The stolen mote will broadcast a theft message. Consequently, an alert message with the ID of the mote will be shown on a PC screen.

You will see: How to use basic networking, how to start (and stop) services, external types, and their use in networking, how to receive and parse tinyos messages on a PC (using IIB2400 interface board).

Advanced: Basic Networking



Advanced: Basic Networking

```
interface SendMsg{
  command result_t send(uint16_t address,
        uint8_t length, TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg,
        result_t success);
}

interface ReceiveMsg{
  event TOS_MsgPtr receive(TOS_MsgPtr m);
}
```

```
typedef struct TOS_Msg{
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[29];
  uint16_t crc;
} TOS_Msg;
```

SingleTimer

LightSensor

"Active Messages":

- Each active message contains the name of a user-level handler to be invoked on arrival
- Minimal buffering for messages and no blocking
- Used in large parallel systems, but matches event based programming of TinyOS well

Advanced: Basic Networking

```
module AntiTheftAdvancedM {
  uses interface SendMsg as AlertMsg;
  uses interface ReceiveMsg as RcvSettings;
}
implementation {
 settings_t settings;
 TOS_Msg alertMsg;
 event TOS_MsgPtr RcvSettings.receive(TOS_MsgPtr m){
   memcpy(&settings,m->data,sizeof(settings_t));
   post reconfigure();
   return m;
 }
 uint32_t sensedValue;
 task void theftAction(){
   if (sensedValue > settings.alertThreshold)
     return;
   if (settings.alert & ALERT_LEDS)
     TheftLeds()
   if (settings.alert & ALERT_RADIO) {
     ((alert_t*)alertMsg.data)->value = sensedValue;
     call AlertMsg.send(TOS_BCAST_ADDR,
                        sizeof(alert_t), &alertMsg);
   }
 }
 event result_t AlertMsg.sendDone(TOS_MsgPtr msg,result_t …){
   return SUCCESS;
 }
```

```
typedef struct settings {
  uint8_t alert;
  uint16_t checkInterval;
  uint16_t alertThreshold;
} settings_t;

typedef struct alert {
  uint16_t stolenId;
  uint32_t value;
} alert_t;
```

Advanced: External Types

```
module AntiTheftAdvancedM {
  uses interface SendMsg as AlertMsg;
  uses interface ReceiveMsg as RcvSettings;
}
implementation {
 settings_t settings;
 TOS_Msg alertMsg;
 event TOS_MsgPtr RcvSettings.receive(TOS_MsgPtr
   memcpy(&settings, m->data, sizeof(settings_t)
   post reconfigure();
   return m;
 }
 uint32_t sensedValue;
 task void theftAction(){
   if (sensedValue > settings.alertThreshold)
     return;
   if (settings.alert & A
     TheftLeds()
   if (settings.alert & A
     ((alert_t*)alertMsg.
     call AlertMsg.send(T

   }
 }
 event result_t AlertMsg.
   return SUCCESS;
 }
```

```
typedef nx_struct settings {
  nx_uint8_t alert;
  nx_uint16_t checkInterval;
  nx_uint16_t alertThreshold;
} settings_t;

typedef nx_struct alert {
  nx_uint16_t stolenId;
  nx_uint32_t value;
} alert_t;
```

External types (nx_...) provide C-like access, but:

- platform-independent layout and endianness gives interoperability
- no alignment restrictions means they can easily be used in network buffers
- compiled to individual byte read/writes

Advanced: Tasks

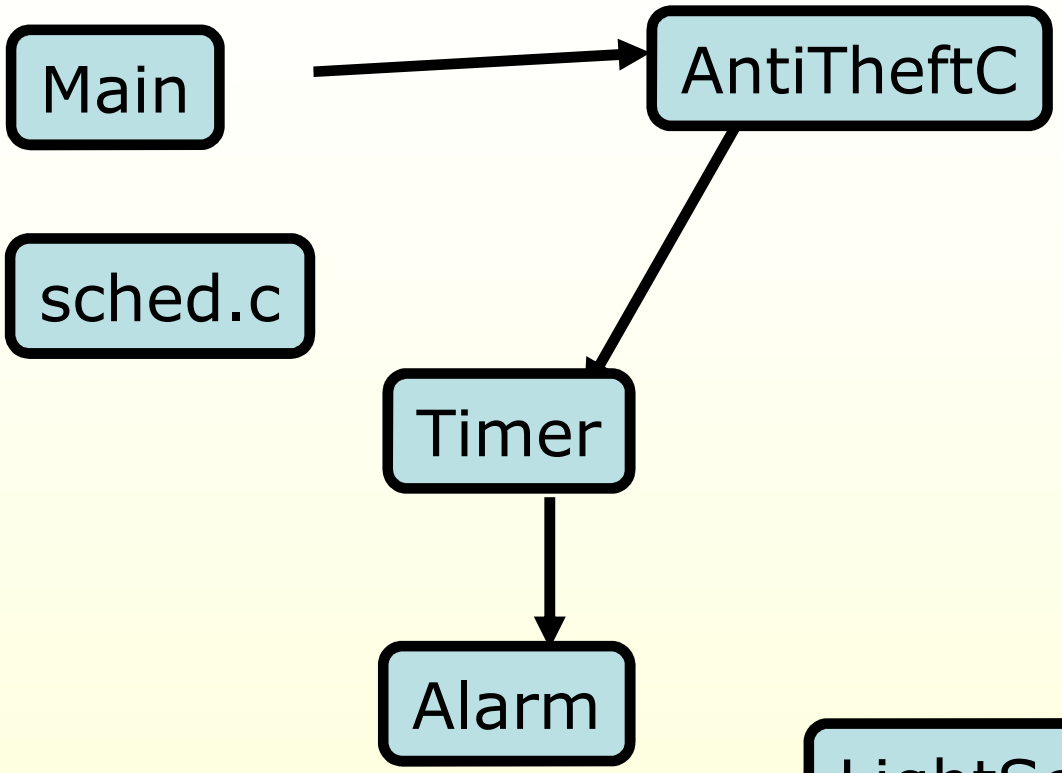
```
TOS_Msg alertMsg;

uint32_t sensedValue;
task void theftAction(){
  if (sensedValue > settings.alertThreshold)
    return;
  if (settings.alert & ALERT_
    TheftLeds();
  if (settings.alert & ALERT_
    ((alert_t*)alertMsg.data)
    call AlertMsg.send(TOS_BC
                       sizeof(alert_t), &alertMsg);
  }
}

event void ReadLight.readDone(

  sensedValue = val;
  post theftAction();
}
```

In readDone, we have multiple choices: blink LEDs or report a theft by radio. We defer this "computationally-intensive" operation to a separate *task*, using post.
We then report theft from the task.

Concurrency model:

- Interrupt Handlers vs Background Tasks
- Posting a task tells the scheduler to put a task on a queue and to call it when the MCU is idle
- Interrupts can preempt both interrupts and tasks
- Tasks cannot preempt other tasks
- No local state is associated with tasks

TinyOS Execution Model



Stack

Main
AntiTheftC
Timer
Alarm

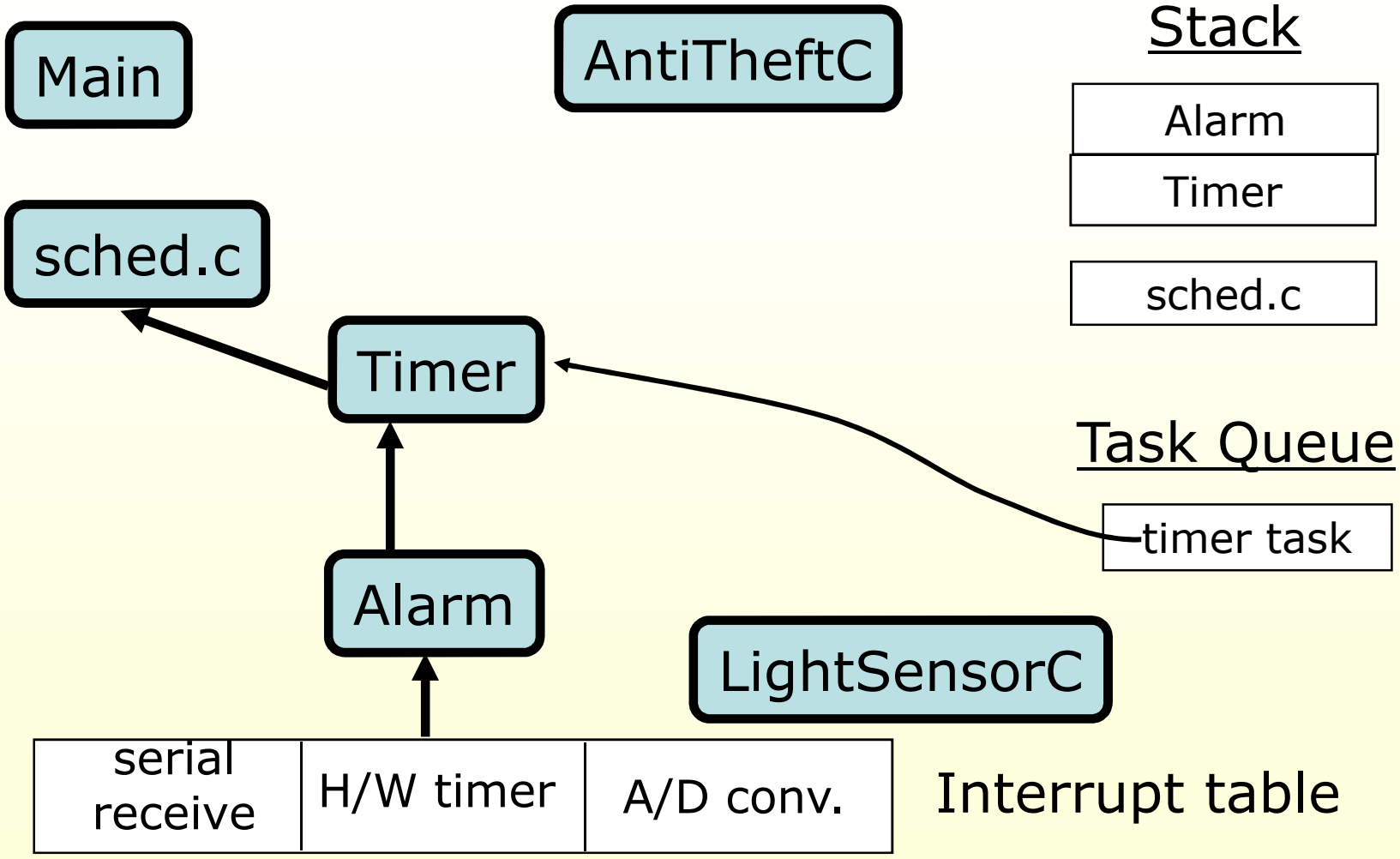
Task Queue

LightSensorC

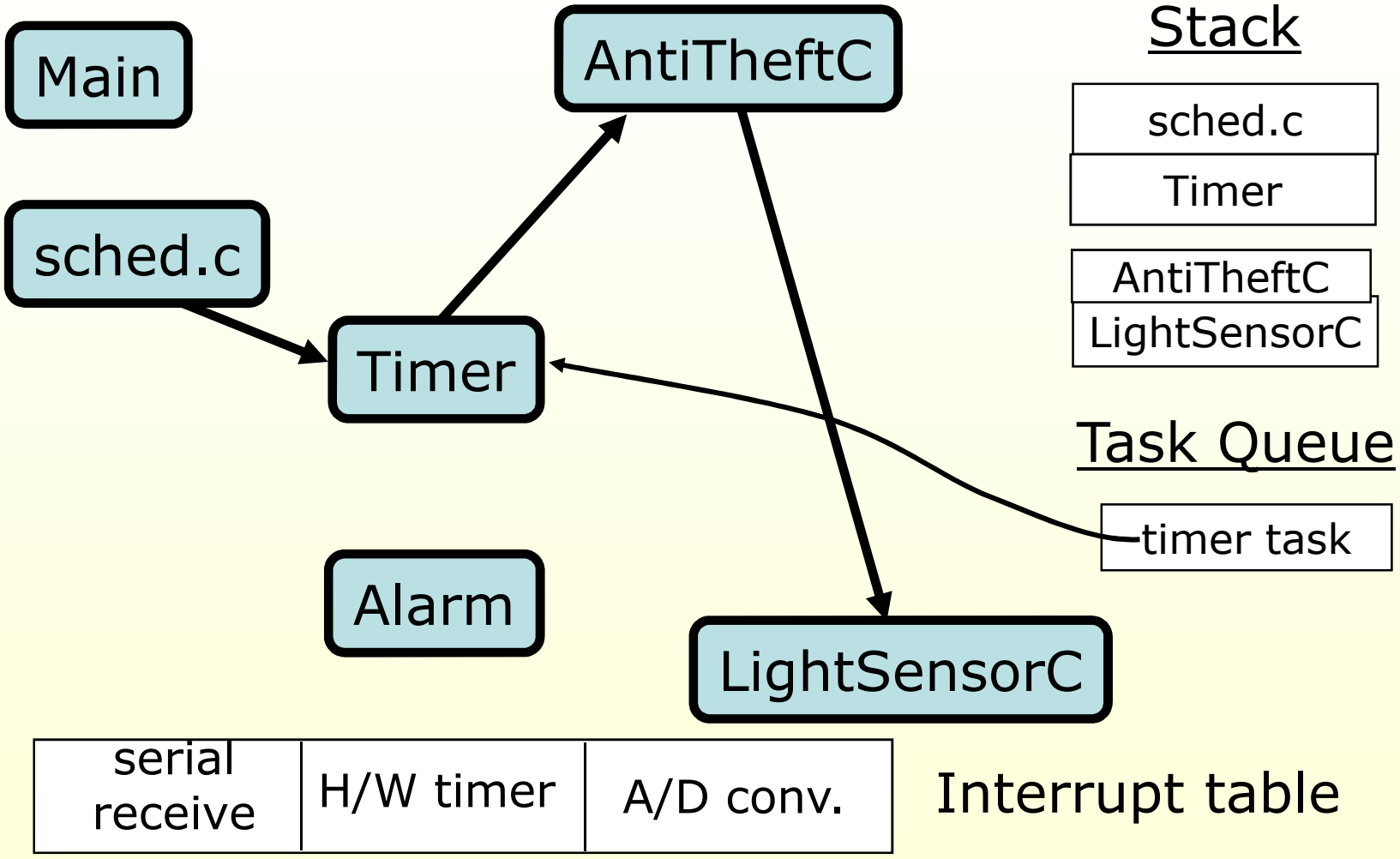
serial receive	H/W timer	A/D conv.

Interrupt table

TinyOS Execution Model



TinyOS Execution Model



Summary

Components and Interfaces

- Programs built by writing and wiring components
  - modules are components implemented in C
  - configurations are components written by assembling other components
- Components interact via interfaces only

Execution model

- Execution happens in a series of tasks (atomic with respect to each other) and interrupt handlers
- No threads

System services: startup, timing, sensing, LEDs, radio

- All slow system requests are split-phase
- External Types

So, how do we build and load this program?

Write C-style programs using nescC language

- Configuration file: AntiTheftAppC.nc
- Module file: AntiTheftM.nc

Use a cross-compiler to build a binary image for a mote MCU (e.g., xscale-elf)

- “make imote2 debug”
- This command calls ‘xscale-elf-gcc’ cross compiler for XScale architecture and the nesC compiler, ‘ncc’
- Binary image is stored in ./build/imote2 directory

Use a programmer (e.g., xflash) to load the binary onto a mote

- USBLoaderHost.exe –p ./build/imote2/main.bin.out
- This software, provided by xbow, let’s us program imote2 motes through USB, without using JTAG

Outline

- TinyOS and nesC overview
- Warm-up project: building a simple anti-theft application
  - The Basics
  - Networking
  - Communication with a PC
- Programming iMote2
- **TOSSIM: simulating TinyOS code**
- **Communication with a PC**
- **Reflections and References**

TOSSIM

- Simulator for TinyOS
- Useful
  - Debugging
  - Running simulations
- Several variations
  - TinyViz
  - Tython
  - PowerTossim

Features & Limitations

- Features
  - Repeatable experiments
  - Can use debug statements
  - Can use gdb
- Limitations
  - One code image only
  - Can take some work to set up scenarios
  - Doesn’t support iMote2

Compiling

- make pc
- make pc sim,micaz

Running

Usage: binpc/main [options] <num_nodes>
[options] are:
-h, --help Display this message.
-k <kb>Set radio speed to <kb> Kbits/s. Valid values: 10, 25, 50.
-r specifies a radio model (simple is default)
-l invocation logging: will block for connect on port 10583
-e <file>use <file> for eeprom; otherwise anonymous file is used-p <sec>pause <sec> seconds on each system clock interrupt
<num_nodes>number of nodes to simulate

Known dbg modes: all, boot, clock, task, sched, sensor, led, route, am, crc, packet, encode, radio, logger, adc, i2c, uart, prog, sim, queue, simradio, hardware, simmem, usr1, usr2, usr3, temp, error, none

Command Line

- Example: Simulation with 20 Nodes

  build/pc/main.exe 20

- export DBG=all,usr1,usr2,…
  - Sets which debug messages are printed

  dbg(<mode>, const char* format, ...);

  dbg(DBG_TEMP|DBG_USR1, "Counter: Value is %i\n", (int)state);

- gdb main.exe

GDB

- Like standard debugging
- gdb main.exe
  - Type help

- Important commands
  - B *component_name$interface_name$command_name
  - set args
  - set environment
  - print component_name$variable_name

Tython

- Based on Python – all python libraries apply
- Example
  - Begin TOSSIM simulation in one window with –gui option
  - In another window: java net.tinyos.sim.SimDriver –console

java net.tinyos.sim.SimDriver -noconsole -gui -run main.exe 20

- Scripting
  - from simcore import *
  - sim.exec("build/pc/main.exe", 10)
  - http://www.tinyos.net/tinyos-1.x/doc/tython/javadoc/index.html

Advice

- No pre-emption in TOSSIM
- Set up simple simulations first
- Lots of online examples
- Use Tython (TinyViz is unstable)

Communication with PC

IIB2400 iMote2 Interface board

- JTAG interface for debugging
  (we won't use it, hopefully)
- Dual USB serial port provided when
  connected to USB port of a PC

Mote can send data to TOS_UART_ADDR address (using GenericComm).
We can access the data by connecting to the serial port.

TinyOS provides java toolchain to connect to a mote and parse tinyos messages received over the serial port.

Java Toolchain for TinyOS Messages

MIG generates java classes corresponding to tinyos message structs:

```
mig -target=imote2 -java-classname=AlertMsg java $(ANTITHEFT_H) alert -o $@
```

MoteIF class provides an application-level Java interface for receiving messages from, and sending messages to, a mote through a serial port.

```
public class AntiTheftGui implements MessageListener, Messenger {
  MoteIF mote;
  mote = new MoteIF(this);
  mote.registerListener(new AlertMsg(), this);
}
```

Send & Receive TinyOS messages to a mote connected to the USB.

```
public void messageReceived(int dest_addr, Message msg) {
  if (msg instanceof AlertMsg)
    System.out.println( ((AlertMsg)msg).get_stolenId());
}
…
SettingsMsg smsg = new SettingsMsg();
smsg.set_alertThreshold(200);
mote.send(MoteIF.TOS_BCAST_ADDR, smsg);
```

Reflection – Components vs Threads

TinyOS has no thread support

- Execution examples earlier show execution of tasks, interrupt handlers
- This execution crosses component boundaries
- Each component encompasses activities initiated in different places, these could be viewed as independent “threads”. We saw:
  - booted event initiated in system setup, timer event initiated in timer subsystem, and light completion events, requested from within AntiTheftM

However, it’s not always clear exactly what a “thread” of control is

- Is the TheftAction task part of the thread initiated by the timer, by the light sensor, or by the anti-theft application?

A more productive view is to consider the system as a set of interacting components

- A component maintains its state information, makes requests for actions from other components, and responds to commands and events from other components.

Reflection – Static Allocation

TinyOS/nesC use static rather than dynamic allocation. Why?

- Ensure that resources are available for the worst case
  - ex: each message source gets one queue slot
- Simplify debugging (by removing a major source of bugs)

But where did that static allocation happen?

- AntiTheftAppC allocated some variables for message buffers, configuration settings
- Instantiation of components implicitly allocates state
  - instantiating a module creates a new set of variables

Reflection – TinyOS Goals Revisited

Operate with limited resources

- execution model allows single-stack execution

Allow high concurrency

- execution model allows direct reaction to events
- many execution contexts in limited resources

Adapt to hardware evolution

- component, execution model allow hardware / software substitution

Support a wide range of applications

- tailoring OS services to application needs

Be robust

- limited component interactions, static allocation

Support a diverse set of platforms

- OS services should reflect portable services

References

TinyOS tutorials

- www.tinyos.net
- http://www.tinyos.net/tinyos-1.x/doc/tutorial/
- D. Gay, TinyOS 2.0: A wireless sensor network operating system
- B. Greenstein, An Introduction to nesC and TinyOS, or, A really complicated way to build very simple applications

Hardware

- iMote2, Crossbow

TOSSIM

- http://www.tinyos.net/nest/doc/tutorial/tossim-lesson.html
- http://www.tinyos.net/tinyos-1.x/doc/tython/manual.html
- http://www.cs.berkeley.edu/~pal/research/tython-manual.pdf

nesC support for concurrency

- nesC does three things to simplify dealing with interrupt-related concurrency:
  - requires the use of *async* on commands and events called from interrupt handlers
  - runs a simple data-race detector to identify variables accessed from interrupt handlers
  - provides an atomic statement to guarantee the atomic execution of one or more statements

Concurrency example

- uint8_t resQ[SIZE];

- **async** command error_t Queue.enqueue(uint8_t id) {
- if (!(resQ[id / 8] & (1 << (id % 8))) { // ← concurrent access!
- resQ[id / 8] |= 1 << (id % 8); // ← concurrent access!
- return SUCCESS;
- }
- return EBUSY;
- }

- If an interrupt occurs during the if, and the interrupt handler also calls Queue.enqueue then:
  - An available slot may be ignored (probably not a problem)
  - The same slot may be given twice (oops!)
- If an interrupt happens during the 2nd concurrent access (write):
  - The interrupt handler's write of resQ will probably be lost

Data race fixed

- uint8_t resQ[SIZE];

- **async** command error_t Queue.enqueue(uint8_t id) {
- **atomic** {
- if (!(resQ[id / 8] & (1 << (id % 8))) {
- resQ[id / 8] |= 1 << (id % 8);
- return SUCCESS;
- }
- return EBUSY;
- }

- Atomic execution ensured by simply disabling interrupts...
  - Long atomic sections can cause problems! E.g.:
    - limit maximum sampling frequency
    - cause lost packets