Introduction to TinyOS

Outline

1. **Introduction to TinyOS and NesC**
2. NesC Syntax
3. 1st exercise : Blink
4. Radio communication
5. 2nd exercise : Radio communication

Introduktion to TinyOS

What is TinyOS?

- An operating system but Not an operating system for general purpose, it is designed for wireless embedded sensor network. (www.tinyos.net)
- An open-source development environment.
  - A programming language and model(nesC)
  - A set of services.
- It features a component-based architecture.

- Main ideology
  - HURRY UP AND SLEEP!
    - Sleep as often as possible to save power.
  - Tasks and event-based concurrency.

What are TinyOS and nesC?

TinyOS is a collection of software modules that can be glued together to build applications. Examples:

- AMSender and AMReceiver: send and receive radio packets
- TimerC: start timers and get notified when they expire
- ADC: sample light and temperature data, among others
- UART: communicate over the serial interface
- LedsC: make pretty lights blink

TinyOS is also a FIFO scheduler:

- Interrupts are handled immediately
- Background tasks are scheduled (put on a queue are executed when there's nothing more important to do)

NesC is the language in which TinyOS modules are written

- To define modules and the interfaces that connect them
- Can create configurations, which are hierarchies of glued together modules

Outline

1. Introduction to TinyOS and NesC
2. **NesC Syntax**
3. 1st exercise : Blink
4. Radio communication
5. 2nd exercise : Radio communication

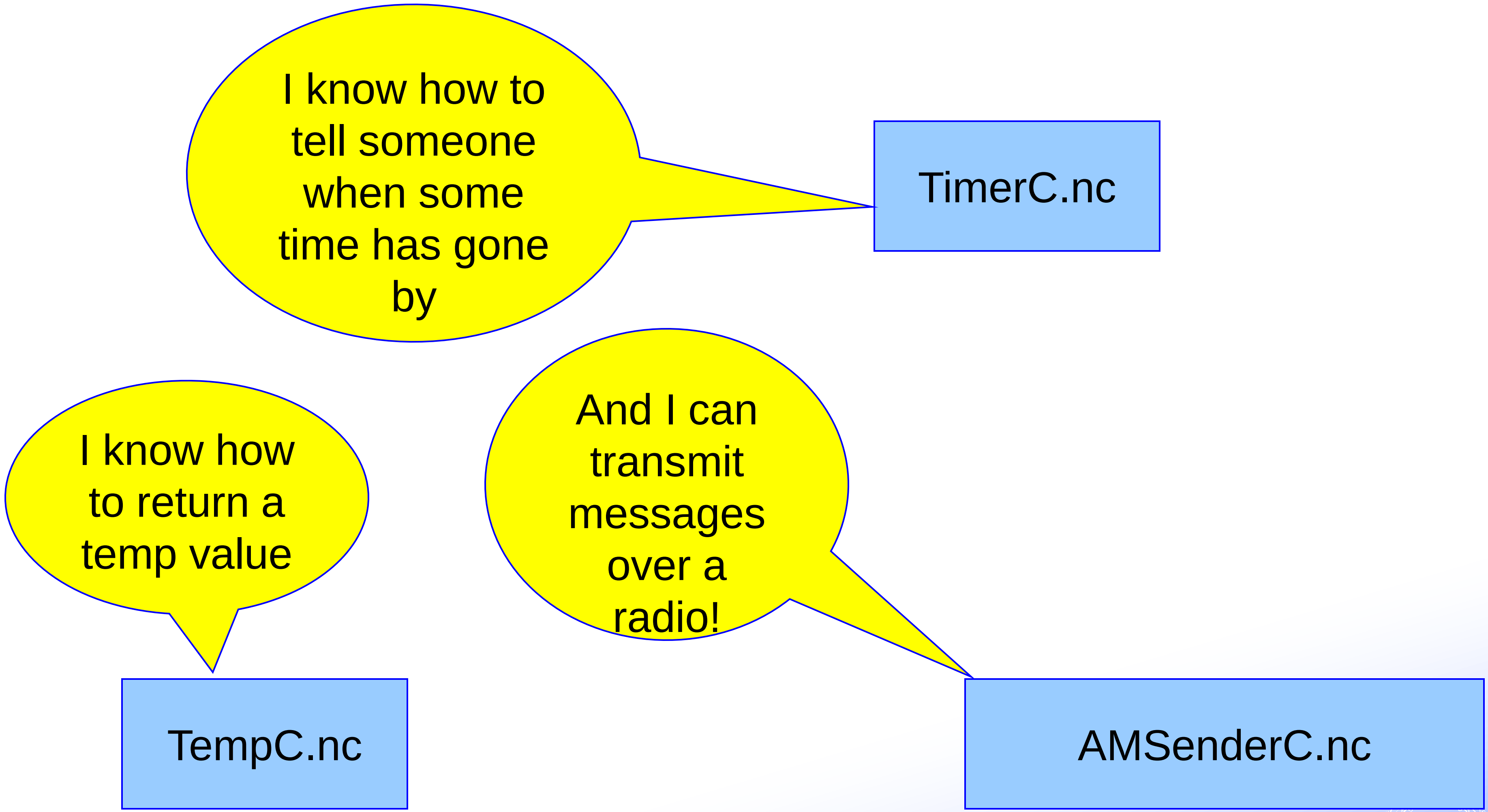
Writing programs for motes

- Write C-style programs using a language for embedded software development (e.g., nesC)
- Use a cross-compiler to build a binary image for a mote MCU (e.g., avr-gcc).
- Use a programmer to load the binary onto a mote

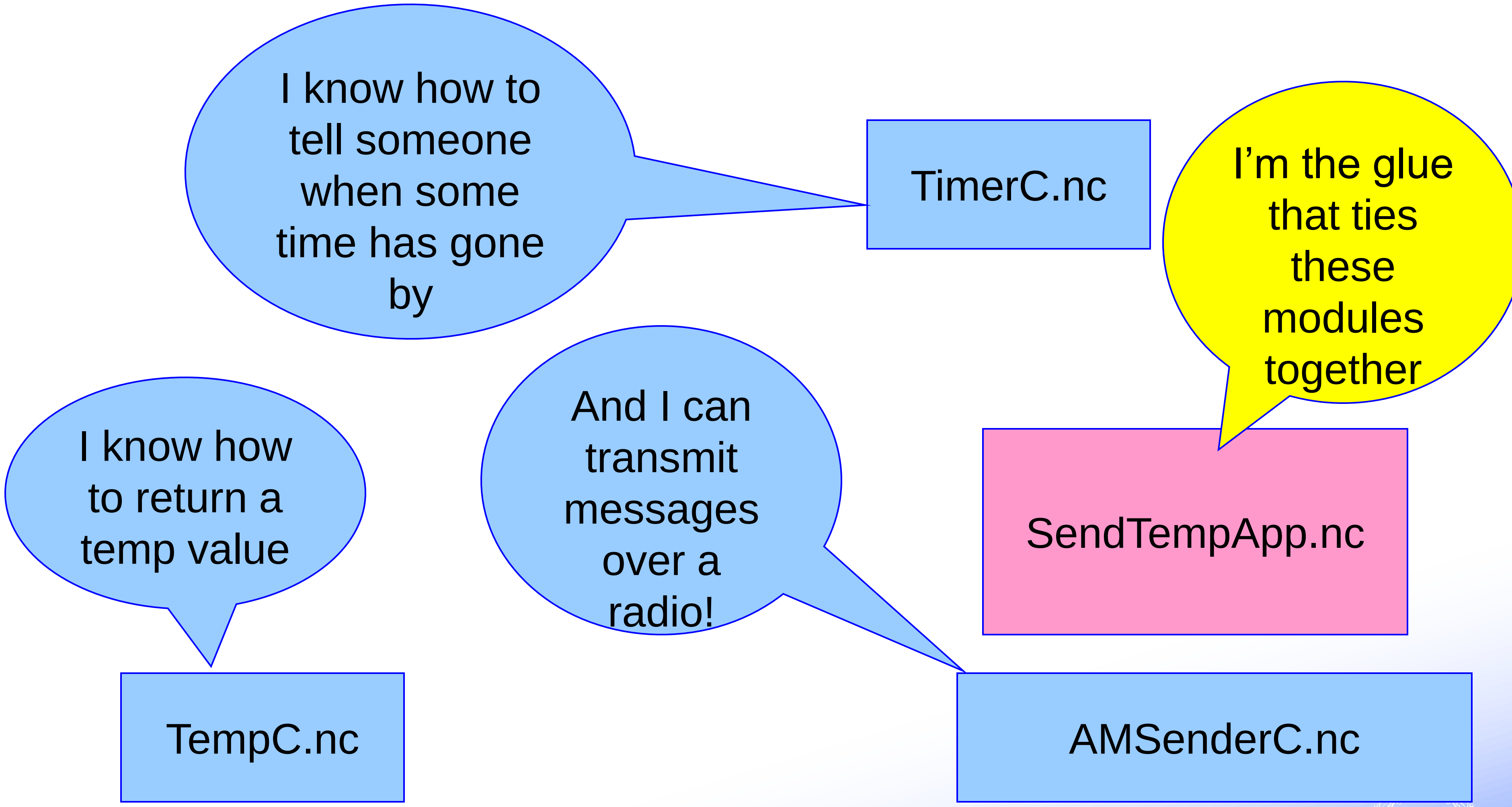
Event driven execution:

- Messages received over radio, discrete event sensors generate interrupts when they detect things, timers go off.
- We write handlers (functions) that are called in response to various events.
- We write tasks to do background processing

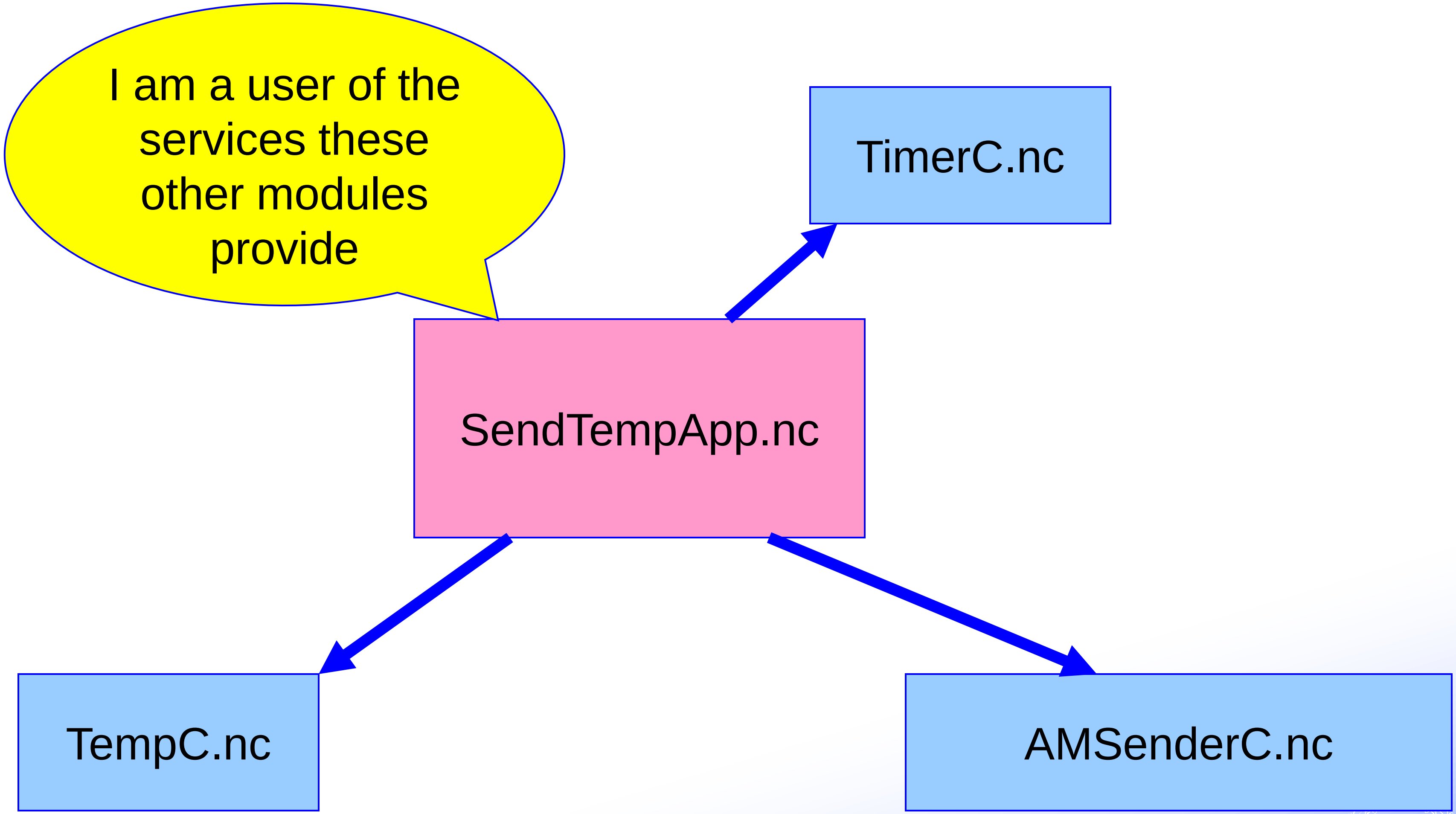
How to periodically sense temperature and transmit it



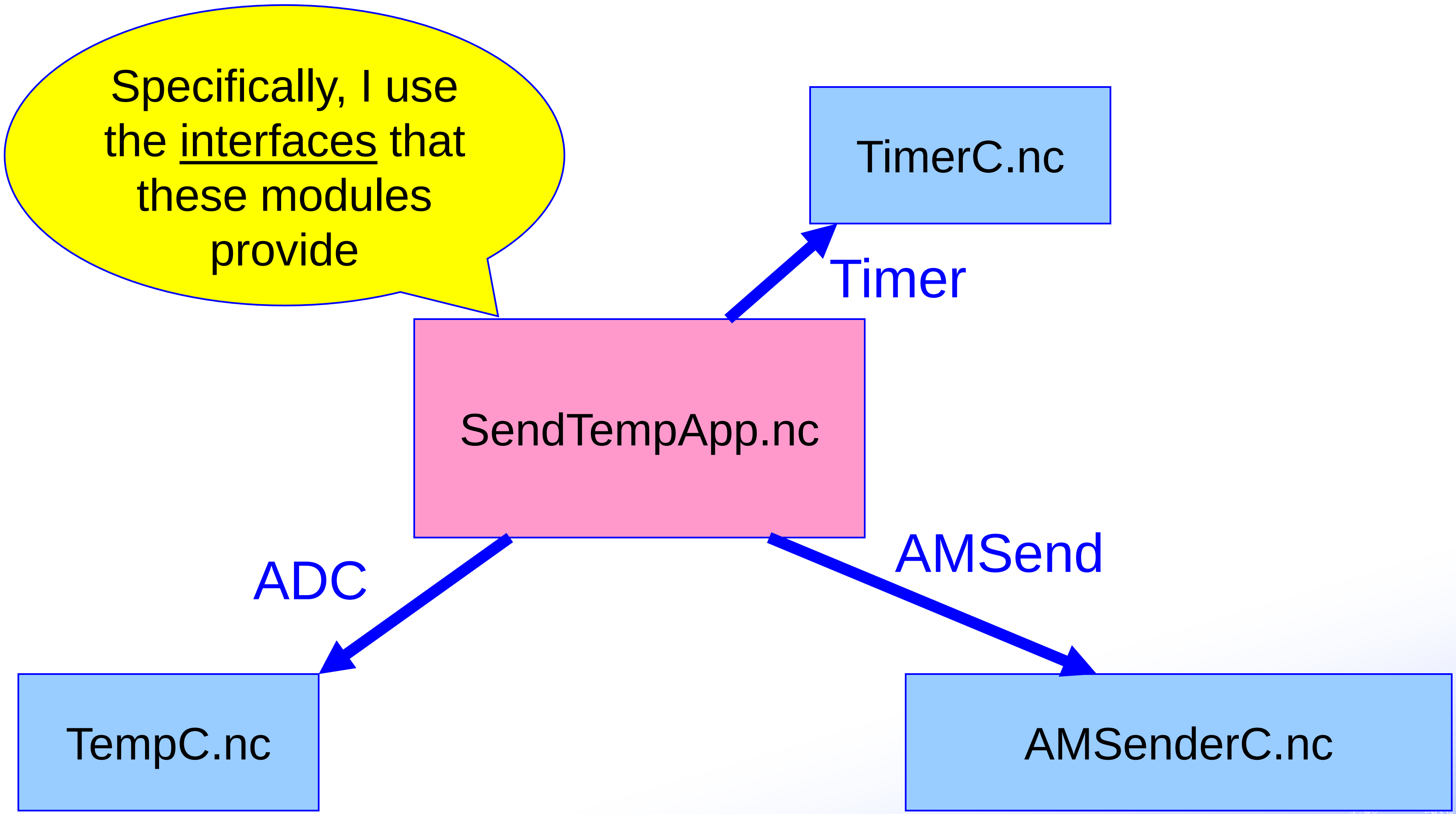
How to periodically sense temperature and transmit it



How to periodically sense temperature and transmit it



How to periodically sense temperature and transmit it



Components

- A component use and provide interfaces, commands and events.
- Components implements the events they use and the commands they provide:

Component	Command	Events
Use	Can call	Must implement
Provide	Must implement	Can signal

- There are two types of components in nesC:
  - Modules: Implements the application behaviour.
  - Configurations: Assembles other components together, called wiring.
- A component may be composed of other components.

TinyOS Commands and Events

Commands are invoked using the call keyword

```
{
….
status = call CmdName(args)
…..
}
```

```
command CmdName(args){
….
return status;
}
```

Event handlers are invoked using the signal keyword

```
event result_t EvtName(args){
….
return status;
}
```

```
{
…
status = signal EvtName(args);
…
}
```

Configuration and Module

- A configuration can bind an interface user to a provider ->
  - User.interface -> Provider.interface

- A configuration file name usually ends with “AppC.nc”, example TempAppC.nc

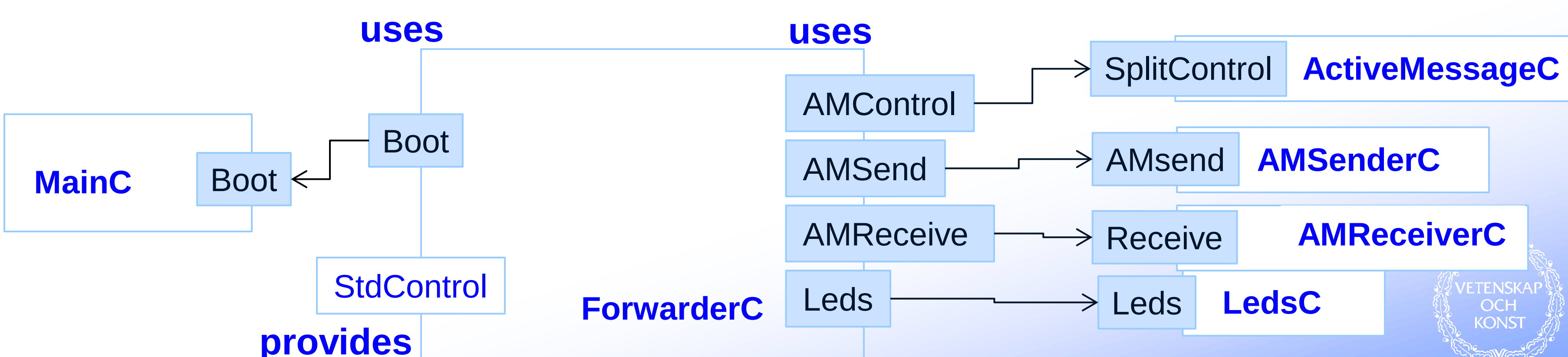
- A module file name usually ends with “C.nc”, example TempC.nc

Component Syntax - Configuration

Component selection

Wiring the components together

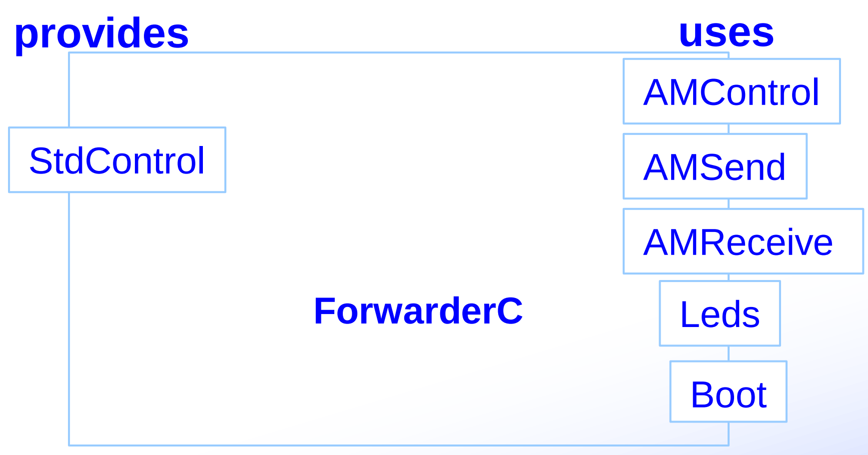
```
configuration ForwarderAppC{
}
implementation
{
  components ForwarderC;
  components MainC,LedsC;
  components ActiveMessageC;
  components new AMReceiverC(AM_PACKETMSG);
  components new AMSenderC(AM_PACKETMSG);

  ForwarderC.Boot -> MainC;
  ForwarderC.AMControl -> ActiveMessageC;
  ForwarderC.AMSend -> AMSenderC;
  ForwarderC.AMReceive -> AMReceiverC;
  ForwarderC.Leds -> LedsC;
}
```



Component Syntax - Module

- A component specifies a set of interfaces by which it is connected to other components
  - provides a set of interface to others
  - uses a set of interfaces provided by others

```
module ForwarderC {
    provides{
         interface StdControl;
    }
    uses {
        interface SplitControl as AMControl;
        interface AMSend;
        interface Receive as AMReceive;
        interface Leds;
        interface Boot;
    }
}
implementation{
// Code implementing all provided commands
//and used events. See next slide
}
```



Component implementation

Command Implementation (interface provided)

Event Implementation (interface used)

```
module ForwarderC{
                //interface declaration
}
implementation{
 command result_t StdControl.start(){
        call AMControl.start();
        return SUCCESS;
  }
 command result_t StdControl.stop(){
        call AMControl.stop();
        return SUCCESS;
  }

event message_t*  AMReceive.receive(message_t*  msg, void* payload, uint8_t len){
  call Leds.led1Toggle();
  call AMSend.send(AM_BROADCAST_ADDR, &msg, sizeof(&msg)) ;
  return msg;
 }

event void AMSend.sendDone(message_t* msg, error_t error) {
 call Leds.led2Toggle();
 }
```

Interface Syntax – Interface AMSend

- Look in **/opt/tinyos-main-read-only/tos/interfaces/AMSend.nc**

```
interface AMSend {
  /**
   * Send a packet with a data payload to an specified address
   */
  command error_t send(am_addr_t addr, message_t* msg, uint8_t len);

  /**
   * Cancel a requested transmission.
   */
  command error_t cancel(message_t* msg);

  /**
   * Signaled in response to an accepted send request.
   */
  event void sendDone(message_t* msg, error_t error);
……..
……..
```

- Includes both command and event.
- Split the task of sending message into two parts, send and sendDone.

Split-Phase Operations

- Split-phase interfaces enable a TinyOS component to easily start several operations at once and have them execute in parallel.
- The command Timer.startOneShot is an example of a split-phase call

Blocking

```
state = WAITING;
op1();
sleep(500);
op2();
state = RUNNING;
```

Split-phase

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

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

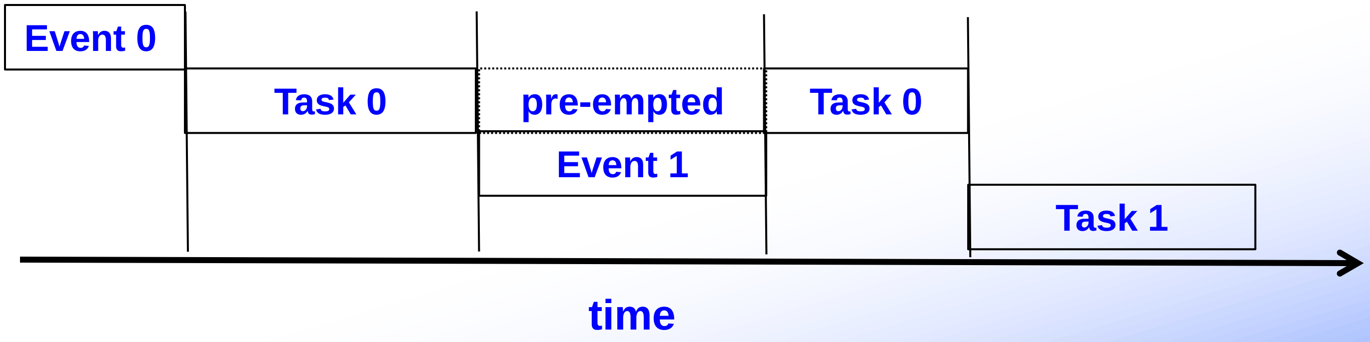
Two level scheduling

Tasks

- FIFO scheduling
- Can be pre-empted by events
- For computationally intensive work
- Every task has its own reserved slot in the task queue, and a task can only be posted once. A post fails if and only if the task has already been posted. If a component needs to post a task multiple times, it can set an internal state variable so that when the task executes, it reposts itself.

Events

- Shorter duration (hand off to task if needed)
- Pre-empts task and events



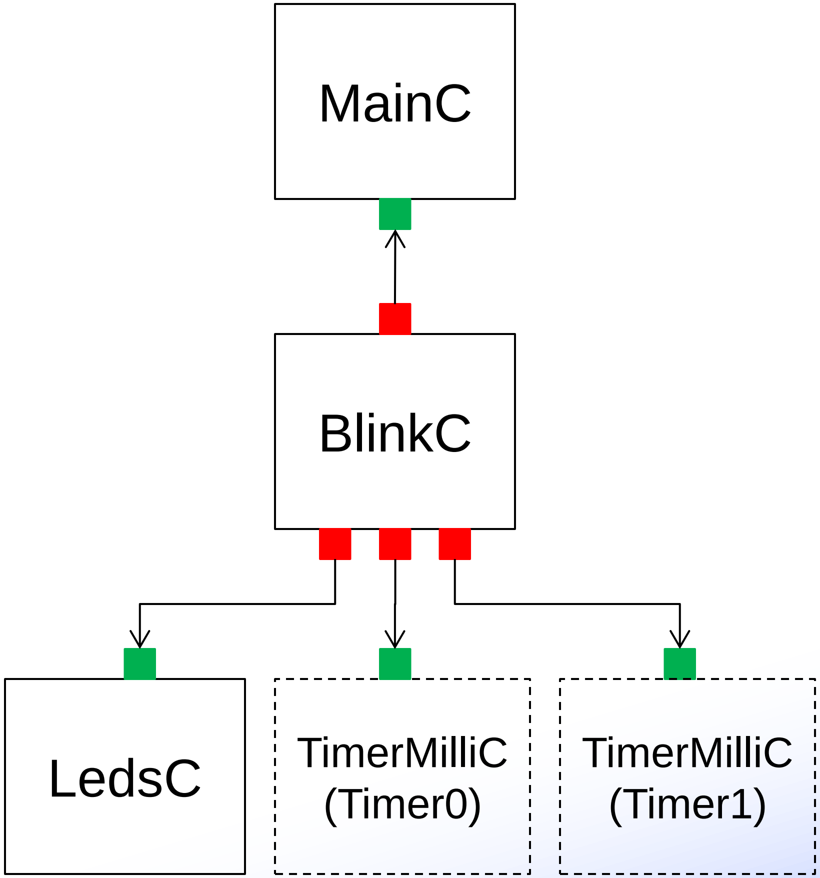
Outline

1. Introduction to TinyOS and NesC
2. NesC Syntax
3. **1st exercise : Blink**
4. Radio communication
5. 2nd exercise : Radio communication

Exercise 1 - Blink

- BlinkApp.nc – Configuration

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

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



Exercise 1 - Blink

- BlinkC.nc – Module

```
#include "Timer.h"
module BlinkC{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Leds;
  uses interface Boot;
}
implementation{
  bool state;

  event void Boot.booted(){
    call Timer0.startPeriodic(500);
    call Timer1.startPeriodic(500);
  }
```

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

event void Timer1.fired(){
    if (state){
      call Leds.led1On();
    }
    else{
      call Leds.led1Off();
    }
    state=!state;
  }
}
```

- Makefile - makefile

```
COMPONENT=BlinkAppC
include $(MAKERULES)
```

Exercise 1 - Blink

- Change the Blink application so that if uses the 3 LEDs as a 3 bits counter.

- Variables in nesC:
  - uint8_t
  - uint16_t
  - uint32_t
  - char
  - bool
  - .....

- The Leds interface “/opt/tinyos-main-read-only/tos/interfaces/Leds.nc”
  - led0On();
  - led0Off();
  - led1On();
  - …..
  - led0Toggle();
  - …….
  - …….

Getting started

• Start the VMware application and choose Advantic-tinyos and login with **user** and password **password**

• Download the source code from **datakom.haninge.kth.se/source_code.zip** to the folder **/opt/tinyos-main-read-only/** The code is under the TinyOS folder.

• Unpack the file by typing in **unzip source_code.zip** in a terminal. It will create a folder called KTH, where you find all the source code needed for this tutorial.

• Connect a node to the computer and activate the USB interface by choosing “Player -> Removable Devices ->Future Devices tmote sky” on WMware menu on the top.

• Change the directory to“/opt/tinyos-main-read-only/KTH/Blink/”

• Compiling:

- Type “**make tmote**” or to compile and build the program.
- Type “**make tmote install**” to compile and build the program and upload the program to the node.
- Type “**make tmote install,1**”

  “1” is the unique ID assigned for the node. Used for the radio communication.

Important: If you have the RED sensor nodes, use **xm1000** instead of **tmote**

Outline

1. Introduction to TinyOS and NesC
2. NesC Syntax
3. 1st exercise : Blink
4. **Radio communication**
5. 2nd exercise : Radio communication

Inter-Node Communication

General idea

- Sender

Fill message buffer with data → Specify recipients → Pass buffer to OS → Determine when message buffer can be reused

- Receiver

OS buffers, incoming message in a free buffer → Signal application with new message → OS obtains free buffer to store next message

Radio Communication

- TinyOS provides a number of interfaces to abstract the underlying communications services and a number of components that provide (implement) these interfaces.

- All of these interfaces and components use a common data type message buffer abstraction, called message_t

  - Packet - Provides the basic accessors for the message_t abstract data type. This interface provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.

  - Send - Provides the basic address-free message sending interface. This interface provides commands for sending a message and canceling a pending message send. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.

  - Receive - Provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area.

Example - Radio Communication

- Makefile

```
COMPONENT=RadioToLedsAppC
CFLAGS +=-DCC2420_DEF_CHANNEL=26
include $(MAKERULES)
```

Radio Channel
11-26

- RadioToLeds.h – Header file

```
#ifndef RADIOTOLEDS_H
#define RADIOTOLEDS_H

enum {
  AM_RADIOTOLEDS = 8, // AM type number.
};

//The message sent over the air.
//The default maximum payload is 28byte
typedef nx_struct RadioToLedsMsg {
  nx_uint16_t nodeid;  //NODE ID of the node, 2byte
  nx_uint16_t counter; //A counter of number of sent packet, 2byte
} RadioToLedsMsg;
#endif
```

Each message also has an 8-bit Active Message ID (am_id_t) analogous to TCP ports

Example – Radio Communication

- RadioToLedsAppC.nc – Configuration

```
#include <Timer.h>
#include "RadioToLeds.h"

configuration RadioToLedsAppC {
}
implementation {
  components MainC,
    LedsC, ActiveMessageC,
    RadioToLedsC as App,
    new TimerMilliC() as Timer0,
    new AMSenderC(AM_RADIOTOLEDS),
    new AMReceiverC(AM_RADIOTOLEDS);

  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
  App.Packet -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.Receive -> AMReceiverC;
  App.AMControl -> ActiveMessageC;
 }
```

VETENSKAP OCH KONST KTH

• RadioToLedsC.nc – Module

```
#include <Timer.h>
#include "RadioToLeds.h"

module RadioToLedsC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
  uses interface SplitControl as AMControl;
  uses interface Packet;
  uses interface AMSend;
  uses interface Receive;
}

implementation {
  uint16_t counter = 0;
  bool busy = FALSE;
  message_t pkt; //Std message buffer

  event void Boot.booted() {
    call AMControl.start();
  }
```

```
  event void AMControl.startDone(error_t err)
{
    if (err == SUCCESS) {
       call Timer0.startPeriodic(500);
    }
    else {
      call AMControl.start();
    }
  }

  event void AMControl.stopDone(error_t err) {
  }
```

The code continues on next page

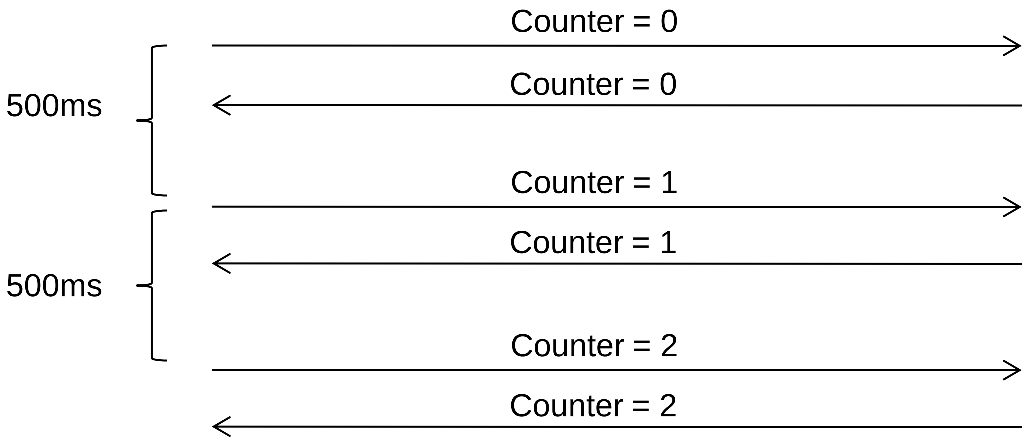
```
event void Timer0.fired() {
   counter++;
   if (!busy) {
     RadioToLedsMsg* btrpkt = (RadioToLedsMsg*)(call Packet.getPayload(&pkt,
                                                               sizeof(RadioToLedsMsg));
     btrpkt->nodeid = TOS_NODE_ID;
     btrpkt->counter = counter;
     if (call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(RadioToLedsMsg))==SUCCESS){
               busy = TRUE;
     }
   }
 }

event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
    if (len == sizeof(RadioToLedsMsg)) {
       RadioToLedsMsg* btrpkt = (RadioToLedsMsg*)payload;
       call Leds.set(btrpkt->counter);
     }
     return msg;
 }

 event void AMSend.sendDone(message_t* msg, error_t error) {
     if (&pkt == msg) {
       busy = FALSE;
     }
  }
}
```

Outline

1. Introduction to TinyOS and NesC
2. NesC Syntax
3. 1st exercise: Blink
4. Radio communication
5. **2nd exercise: Radio communication**

Exercise 2 – Radio Communication


Master




Slave

- The master node
  - should send the counter value to the slave node every 500ms starting with the value 0 and end with 7 (3-bits counter) and repeat it.
  - should show the counter value on the LEDs received from the slave node.

- The slave node
  - should receive the counter from the master node and show the value on the LEDs.
  - should immediately send the same counter value back to the master node.

IMPORTANT: DON'T FORGET TO CHANGE THE ACTIVE MESSAGE ID(Slide 28)

Remove the Project folder

Remove only the KTH folder from your computer before you leave the classroom.

References

- David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, David Culler, “The nesC Language: A Holistic Approach to Networked Embedded Systems”
- TinyOS Tutorials, http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Tutorials
    - Lesson 1: Getting Started with TinyOS
    - Lesson 2: Modules and the TinyOS Execution Model
    - Lesson 3: Mote-mote radio communication