MOTES PROGRAMMING

Vahid Meghdadi

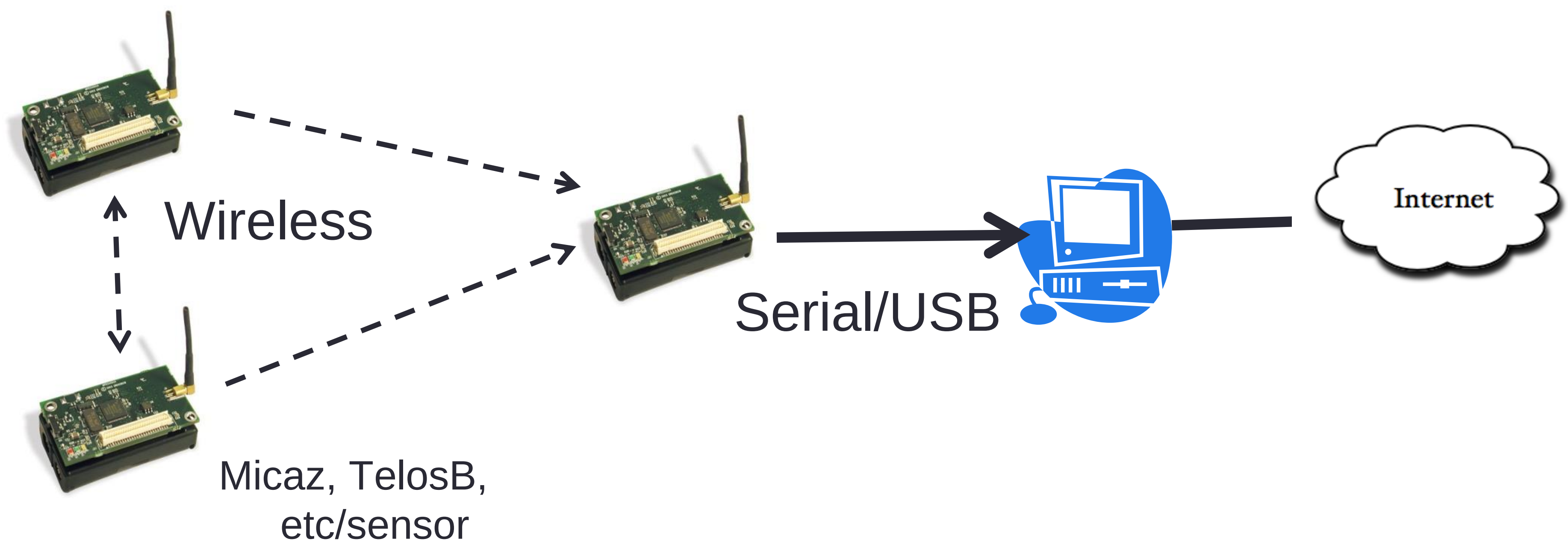
Partially based on
the tutorial, by Jun Yi
And the presentation by David E. Culler
University of California, Berkeley

Overview

Sensor code (nesC/TinyOS)

Base station code (nesC/TinyOS)

Gateway code (Java, c, …)



nesC: Network Embedded Systems

What is TinyOS?

- An operating system for low power, embedded, wireless devices
  - Wireless sensor networks (WSNs)
  - Sensor-actuator networks
  - Embedded robotics
- Open source, open developer community
- http://www.tinyos.net
- *E-book: TinyOS Programming: http://csl.stanford.edu/~pal/pubs/**tinyos-programming**.pdf*

Supported Hardware

- Platforms
  - TelosB
  - Mica2
  - MicaZ
  - EPIC
  - …
- Microcontrollers
  - Atmel ATmega128, a 8-bit RISC microcontroller
  - Texas Instruments MSP430 a 16-bit low power microcontroller
  - Intel XScale PXA271 a 32-bit RISC microcontroller
- Radio
  - CC1000 (FSK, sous 1GHz)
  - CC1100/CC2500 (ASK, FSK, GFSK, MSK sous 1GHz)
  - CC2420 (ZigBee 2.4 GHz)
  - AT86RF212/AT86RF230 (ZigBee 2.4 GHz))

TinyOS and nesC

- *Components and interfaces*
- Tasks
- Compiling and tool-chain

TinyOS Components

- TinyOS and its applications are in nesC
  - C dialect with extra features
- Basic unit of nesC code is a component
- Components connect via interfaces
  - Connections called “wiring”

Components

- A component is a file containing programs in nesC
- Modules are components that have variables and executable code
- Configurations are components that wire other components together
- Components are like objects but it must declare not only the functions that it implements but also **the functions that it calls**.
- Therefore a component has a code block that declares the functions it provides (implements) and **the functions that it uses** (calls)

components

```
module SmoothingFilterC {
        provides command uint8_t topRead(uint8_t* array, uint8_t len);
        uses command uint8_t bottomRead(uint8_t* array, uint8_t len);
}
```

- Other components can call topRead
- bottomRead is just a reference, and is implemented by another component
- However, it is rare to declare individual functions, but interfaces, which are the collections of functions.

Component Example

- BlinkAppC wires BlinkC.Timer to TimerC.Timer


```
module BlinkC {
 uses interface Timer<TMilli>
    as Timer0
 provide interface xxxx}
implementation {
  int c;
  void increment() {c++;}
  event void Timer0.fired()
  {
     call Leds.led0Toggle();
  }
}
```

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC()
              as Timer0;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Boot -> MainC.Boot;
  BlinkC.Leds -> LedsC.Leds;
}
```

Singletons and Generics

- Singleton components are unique: they exist in a global namespace
- Generics are instantiated: each instantiation is a new, independent copy

```
configuration BlinkC { … }
implementation {
  components new TimerC();
  components BlinkC;

  BlinkC.Timer -> TimerC;
}
```

Interfaces

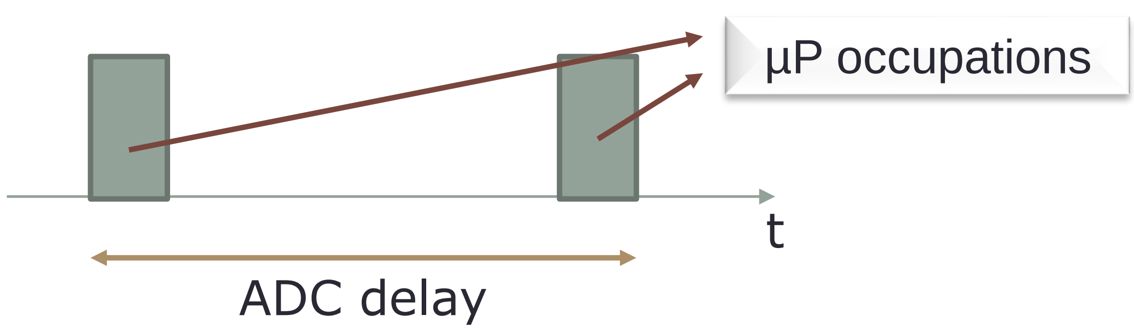
- Collections of related functions

- Define how components connect

- Interfaces are bi-directional: for A->B
  - Commands are from A to B
  - Events are from B to A

Split-phase

- Many hardware parts are working with the software
- Normally hardware is split-phase rather that blocking
- For example for a DAC reading
  - Software writes a few configuration registers to start a sample
  - When ADC done, the HW issues an interrupt
  - Then the SW reads the value out of a data register



Send is a split-phase interface

```
interface Send {
        command error_t send(message_t* msg, uint8_t len);
        event void sendDone(message_t* msg, error_t error);

        command error_t cancel(message_t* msg);
        command void* getPayload(message_t* msg);
        command uint8_t maxPayloadLength(message_t* msg);
}
```

- The command send is used to initialize a packet sending
- The sendDone event gives a report on the send.

Interface with argument

- Some interfaces can take arguments

- Example: Timer

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

- To use in your component:

```
        uses interface Timer<TMilli> ; // or T32khz, or TMicro
```

Interface with arguments

- Sometimes the arguments are a type

```
interface Read<val_t> {
        command error_t read();
        event void readDone(error_t err, val_t t);
}
```

- To use:

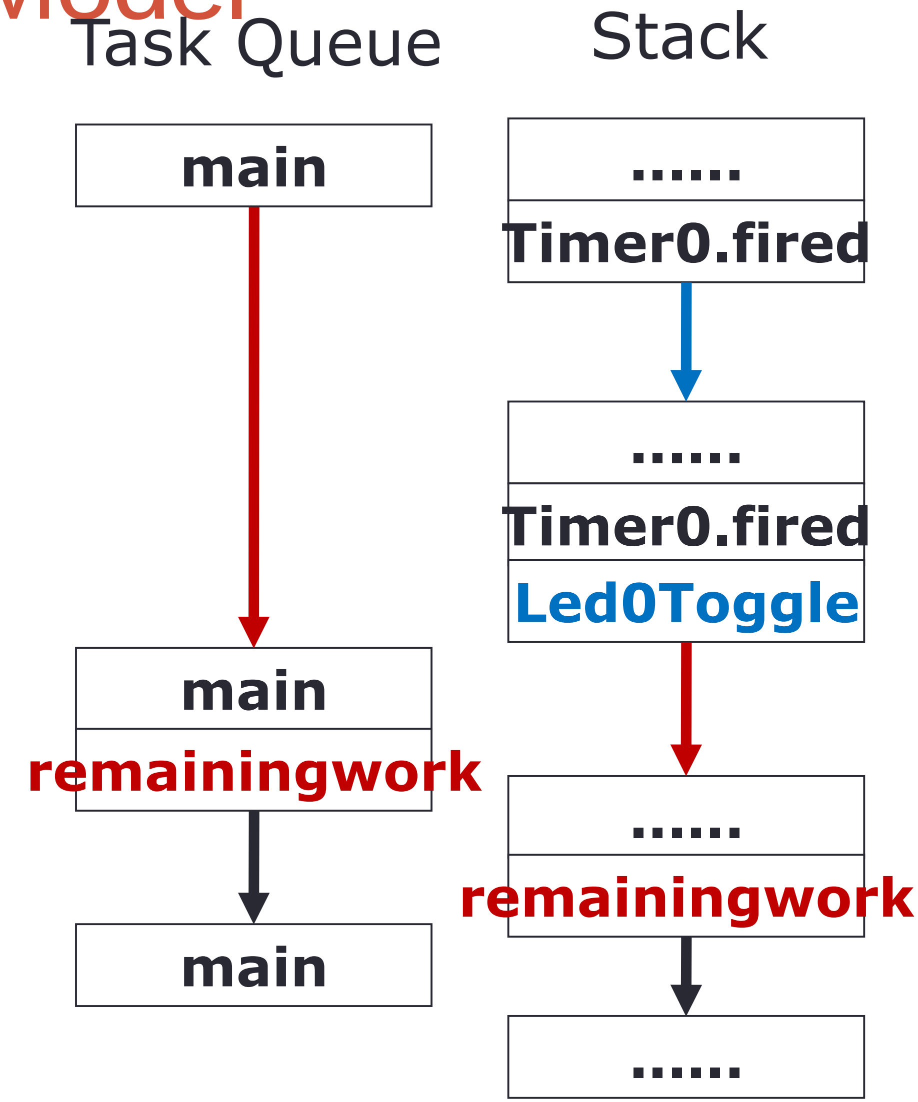
```
module PeriodicReaderC {
        uses interface Timer<TMilli>;
        uses interface Read<uint16_t>;
}
```

Tasks

- TinyOS has a single stack: long-running computation can reduce responsiveness
- Tasks: mechanism to defer computation
  - Tells TinyOS “do this later”
- Tasks run to completion
  - TinyOS scheduler runs them one by one in the order they post
  - Keep them short !

TinyOS Execution Model

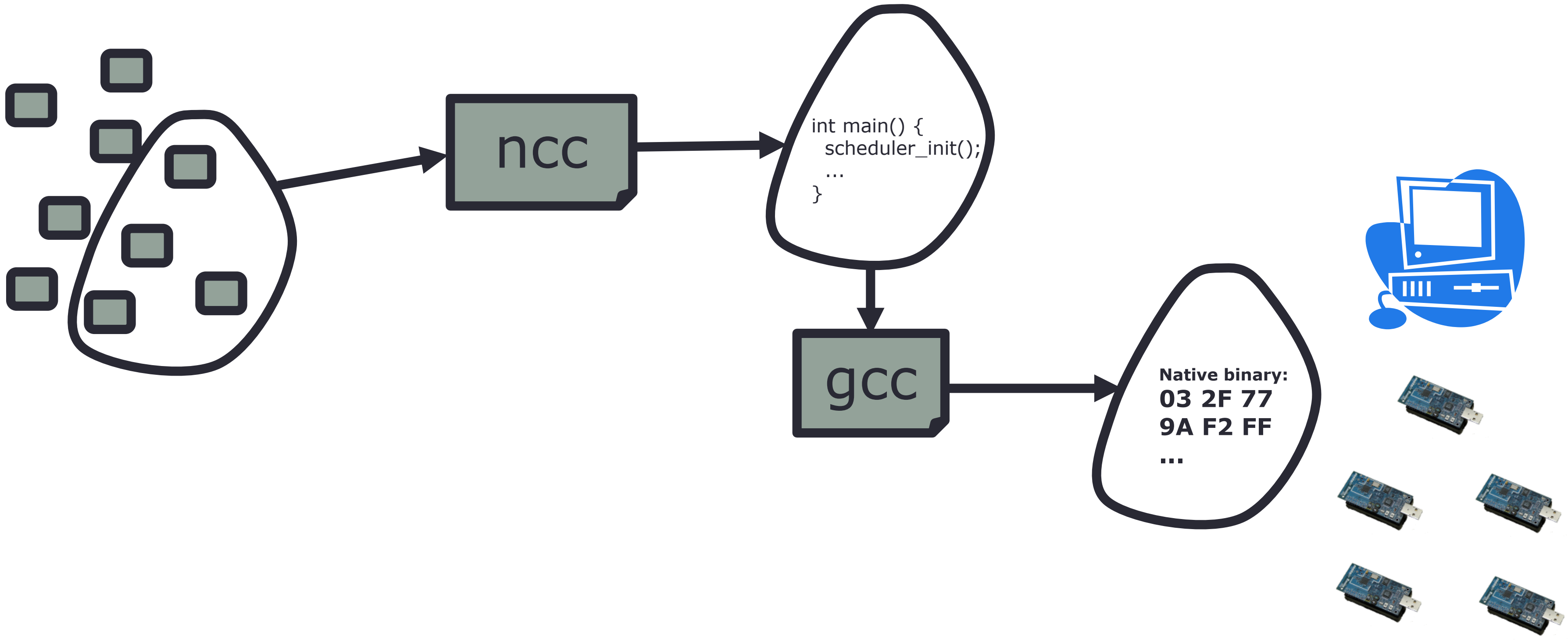
```
Xxxxxx;
event void Timer0.fired()
  {
      xxxxxx;
      xxxxxx;
      xxxxxx;
      xxxxxx;
      call Leds.led0Toggle();
      xxxxxx;
      xxxxxx;
      post remainingwork();
  }
  xxxxx;
  remainingwork(){xxxx;};
  xxxxx;
```



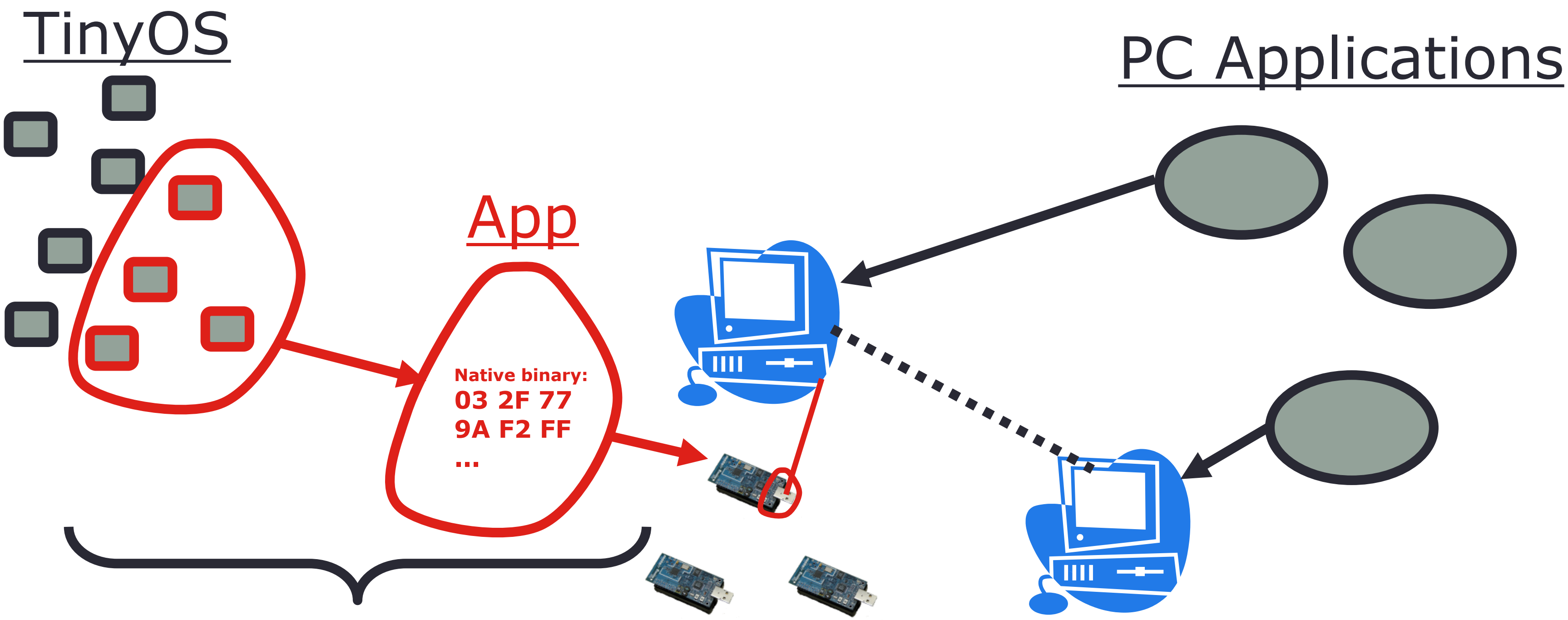
TinyOS/nesC Summary

- Components and Interfaces
  - Programs built by writing and wiring components
    - modules are components implemented in C
    - configurations are components written for assembling other components
- Execution model
  - Execution happens in a series of tasks (atomic with respect to each other) and interrupt handlers
- System services: start-up, timing, sensing
  - (Mostly) represented by instantiable generic components
    - This instantiation happens at compile-time!
  - All slow system requests are split-phase
    - A command starts an operation
    - An event signals operation completion

“Make”: The Tool Chain



The "Make" System

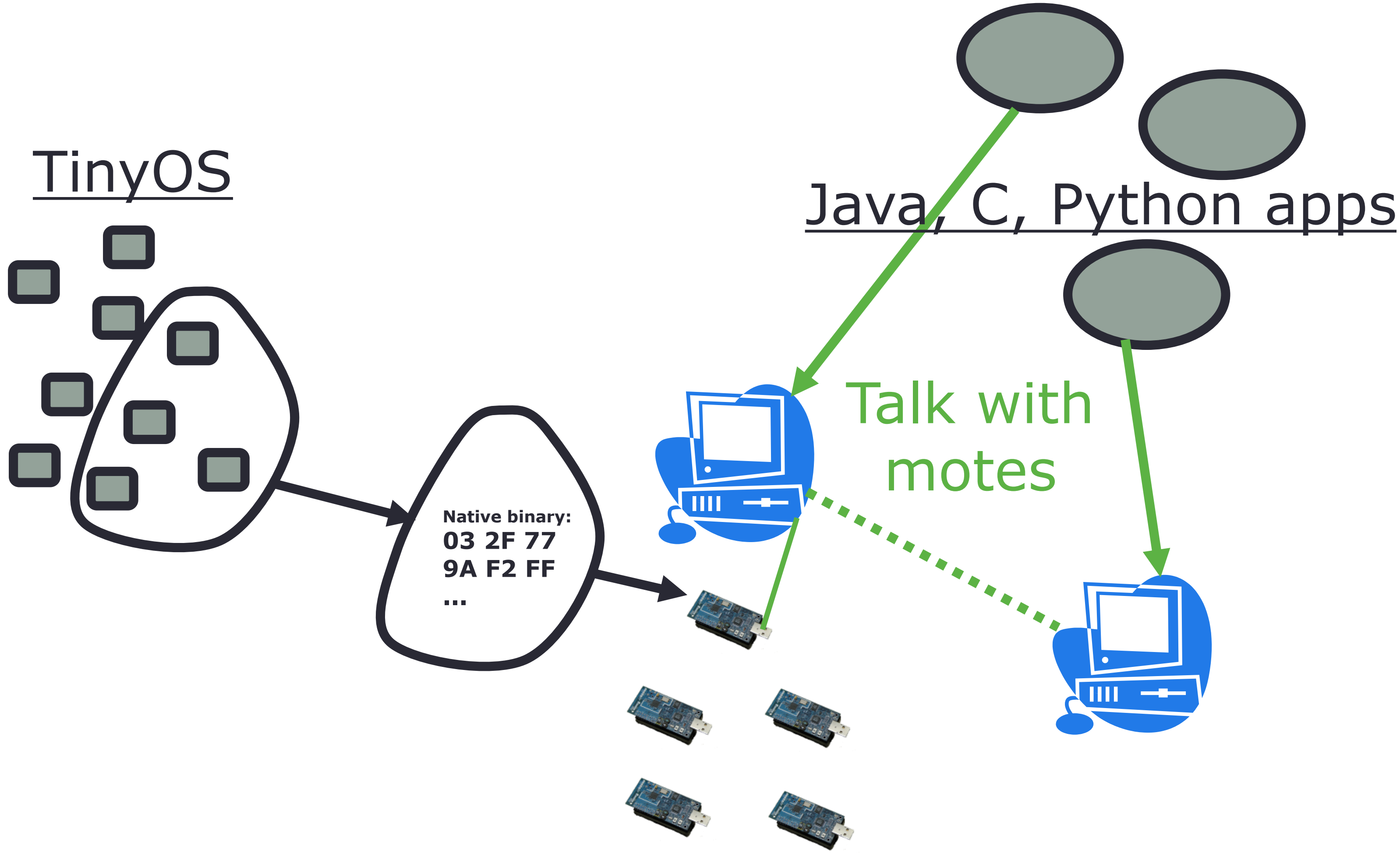


make micaz install mib520, /dev/ttyS0

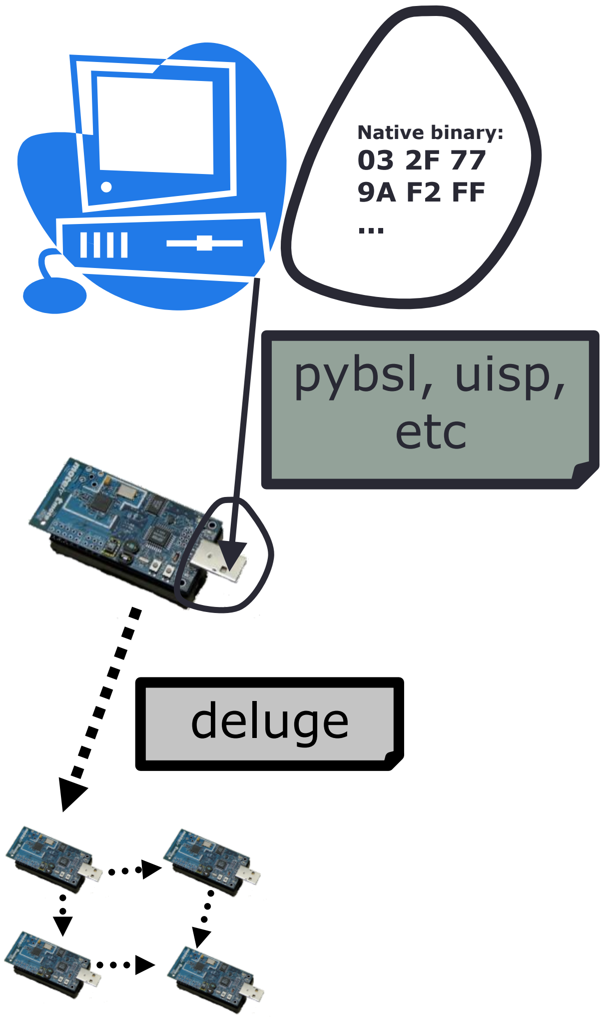
automates nesC, C compilation,
mote installation

Build PC Applications

java classname -comm serial@/dev/ttyS0:micaz

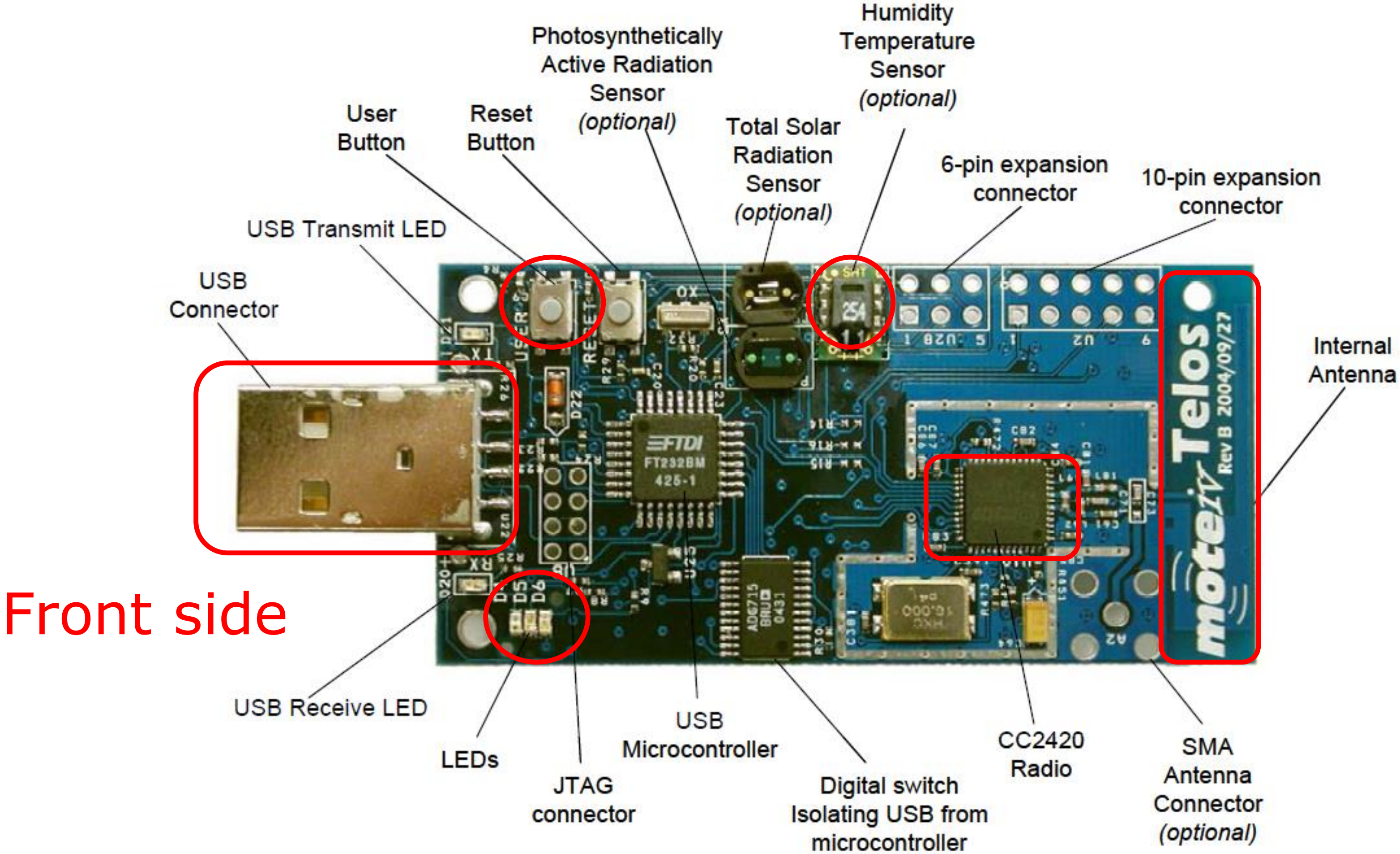


"Make": Install Applications

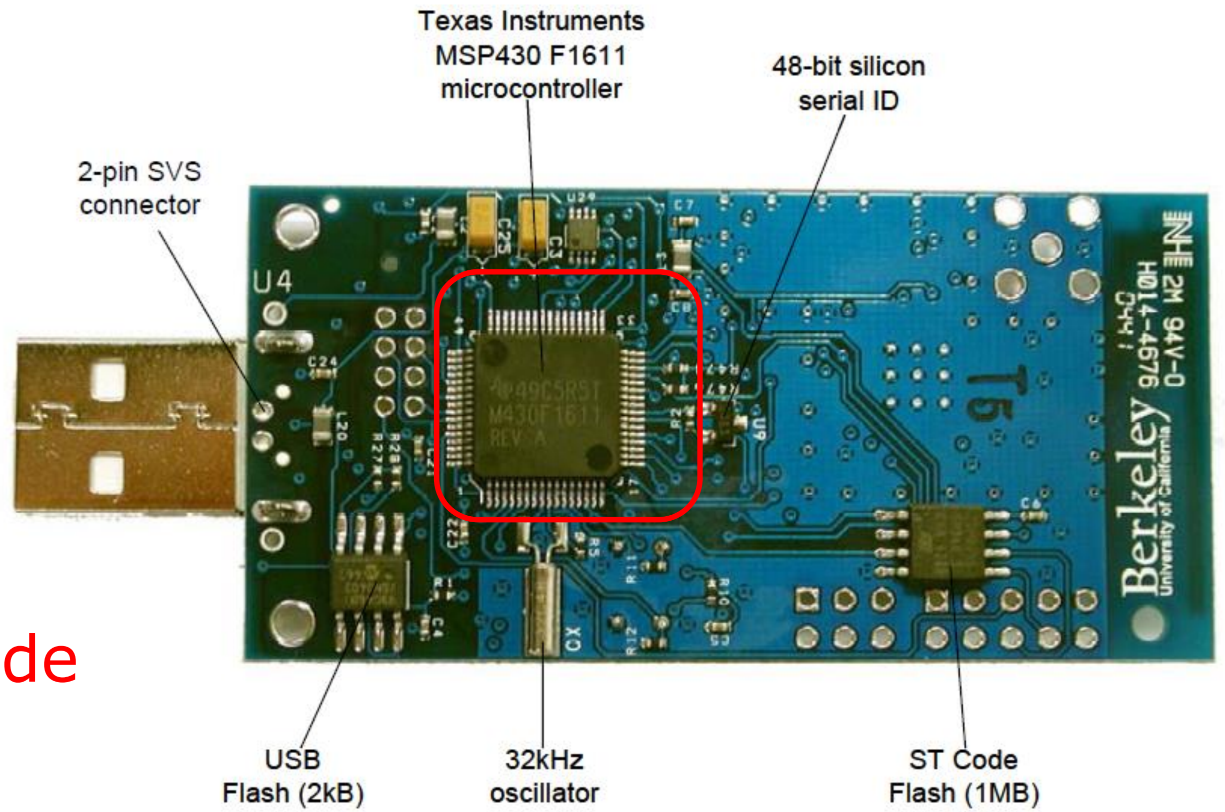


UISP : Micro-In-System-Programmer

Hardware board :TelosB



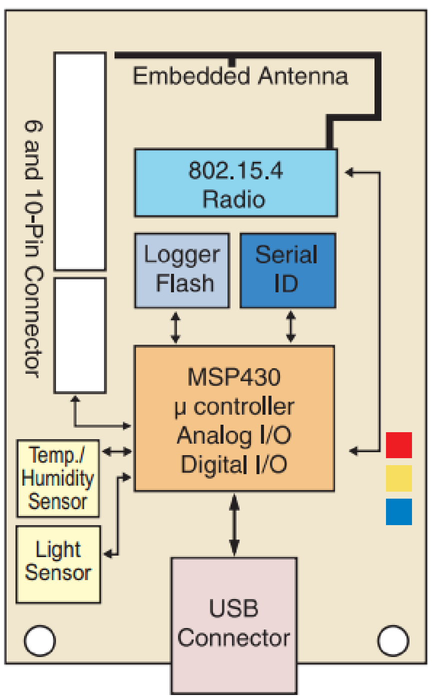
Hardware board :TelosB

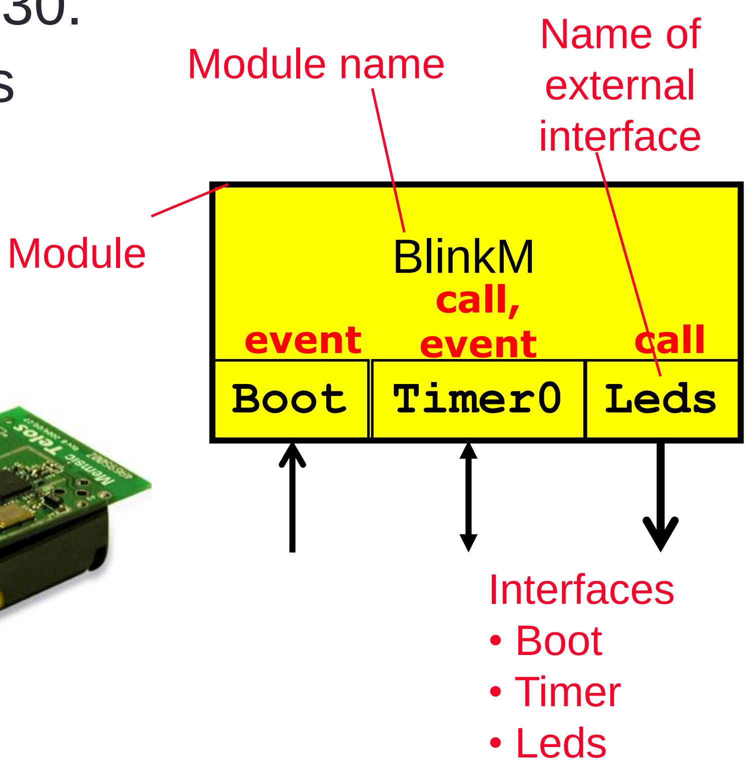


Back side

Programming skills

- Example: The objective is to make blink an LED on the TelosB board (CC2420).
- The executable is run on TI MSP430.
- Some tinyOS components such as Timer and Leds will be used,





Blink example, events

- At the beginning, the component main is executed where boot sequences are done
- Once this task is done, the event “booted” is generated that we can use to initialize our own circuit
- We can put for example the following codes to initialize a timer that gives interrupts every 250 milliseconds.

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

- Note at the syntax
- This is an event handler of the event “booted”, which calls a command provided by Timer interface. We are therefore using the Timer interface.

Blink example, on "timer fire" event

- When the timer reaches at the end, an event is generated. At this event we may want to toggle an LED on the board.

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

- The written handler calls a command provided by Leds' interface, which changes the state of the LED0
- We are therefore using the Leds interface.

Module BlinkM

```
#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();
  }
}
```

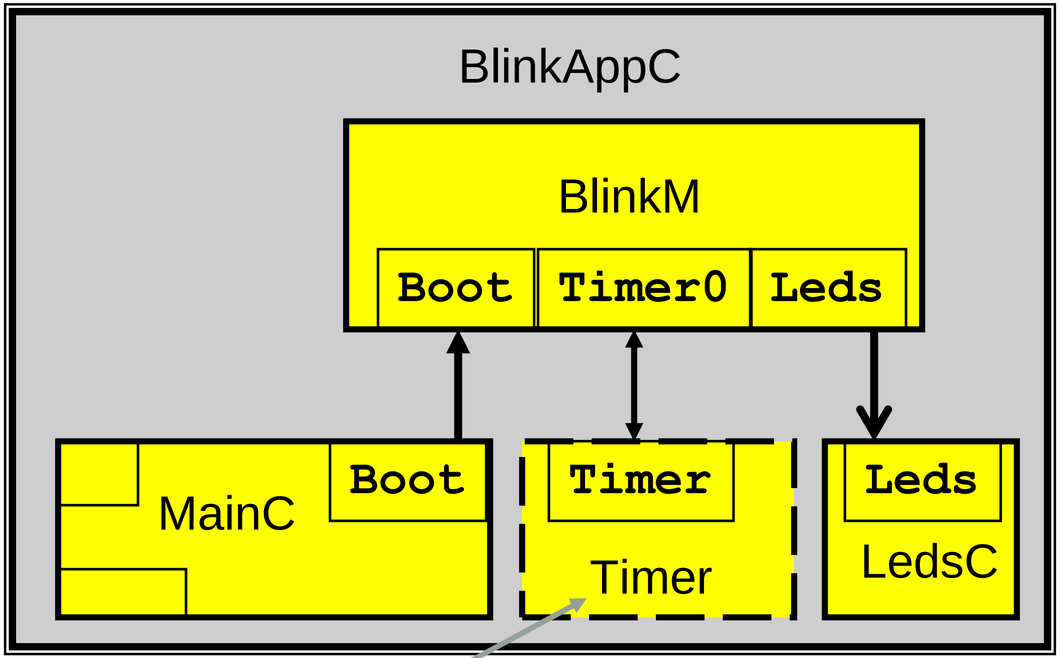
Interfaces used

```
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.
*/
  event void booted();
}

interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();
  ...
  async command uint8_t get();
  async command void set(uint8_t val);
}
```

Timer interface

```
interface Timer<precision_tag> // TMilli, TMicro,
{
  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();
}
```

Configuration




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

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

Multi timer example

```
#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();
  }
}
```

Types

- Common numeric types

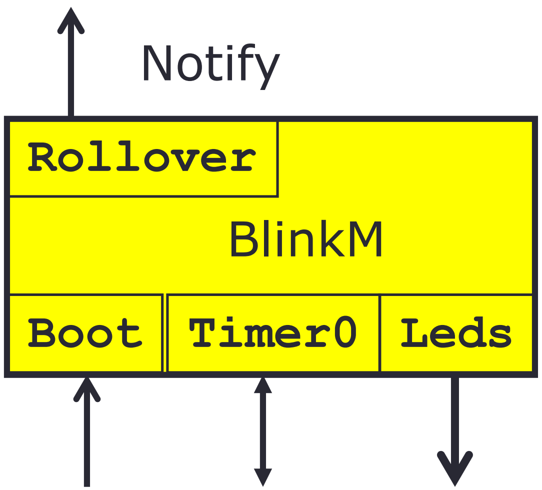
	8 bits	16 bits	32 bits	64 bits
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

- Bool, …

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

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

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

Event

- In an event implementation we can
  - Call commands
  - Signal events




```
module BlinkM {
  uses interface Boot; etc
  provides interface Notify<bool> as Rollover;
}
implementation
{
  uint8_t counter = 0;
...
  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
    if (!counter) signal Rollover.notify(TRUE);
  }
}
```

Split-phase operation 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;
```

Sensor Reading

- Sensors are embedded I/O devices
  - Analog, digital, … many forms with many interfaces
- To obtain a reading
  - configure the sensor
  - and the hardware module it is attached to,
    - ADC and associated analog electronics
    - SPI bus, I2C, UART
  - Read the sensor data
- TinyOS 2.x allows applications to do this in a platform-independent manner

Read interface

```
interface Read<val_t> {
  /*  Initiates a read of the value.
   *  @return SUCCESS if a readDone() event will eventually come back.
   */
  command error_t read();

  /**
   * Signals the completion of the read().
   *
   * @param result SUCCESS if the read() was successful
   * @param val the value that has been read
   */
  event void readDone( error_t result, val_t val );
}
```

- Split-Phase data acquisition of typed values

Example

```
#include "Timer.h"
module SenseM
{
  uses {
    interface Boot;    interface Leds;    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
implementation
{
  #define SAMPLING_PERIOD 100
  event void Boot.booted() {
    call Timer.startPeriodic(SAMPLING_PERIOD);   }

  event void Timer.fired()
  {    call Read.read();    }

  event void Read.readDone(error_t result, uint16_t data)
  {
    if (result == SUCCESS){ call Leds.set(data & 0x07);}
  }
}
```

Exemple

- Here we will define another implementation of read and read-done, which exist normally in the « Read » interface.
- We define a new module, vahidC, which provides the Read interface. Add the line

```
provides interface Read<uint16_t>
```

- Then we define a command for Read.read function in the implementation part of vahidC.nc as:

```
command error_t Read.read(){
    number = rand();
    signal Read.readDone(TRUE, number);
    return TRUE;
}
```

- In the configuration

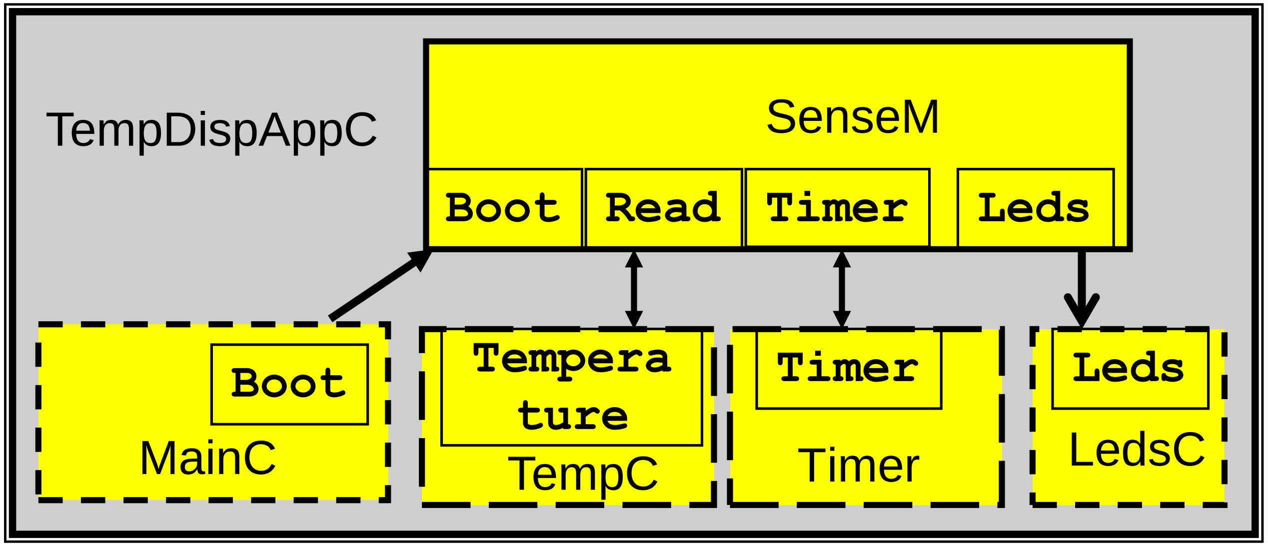
```
components VahidC;
App.Read -> VahidC.Read;
```

- For using in the module:

```
event void Read.readDone(error_t result, uint16_t val){
    printf("%d\n",val);
}
```

Configuration

```
configuration TempDispAppC {
}
implementation {
  components SenseM, MainC, LedsC, new TimerMilliC() as Timer;
  components new SensirionSht11C() as TempC ;

  SenseM.Boot  -> MainC.Boot;
  SenseM.Leds  -> LedsC.Leds;
  SenseM.Timer -> Timer.Timer
  SenseM.Read  -> TempC.Temperature;
}
```



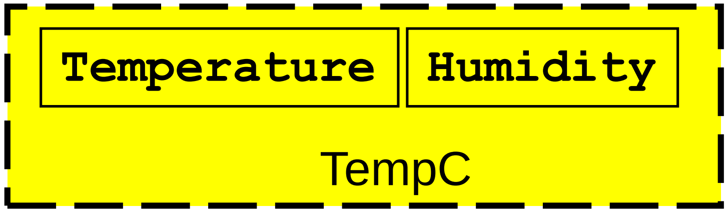
Read several sensors (1)

- Repeat the declaration of “Read interface for all the sensors

```
uses interface Read<uint16_t> as Temperature;
uses interface Read<uint16_t> as Humidity;
uses interface Read<uint16_t> as Light;   (another component)
```

- Let’s define a timer first:

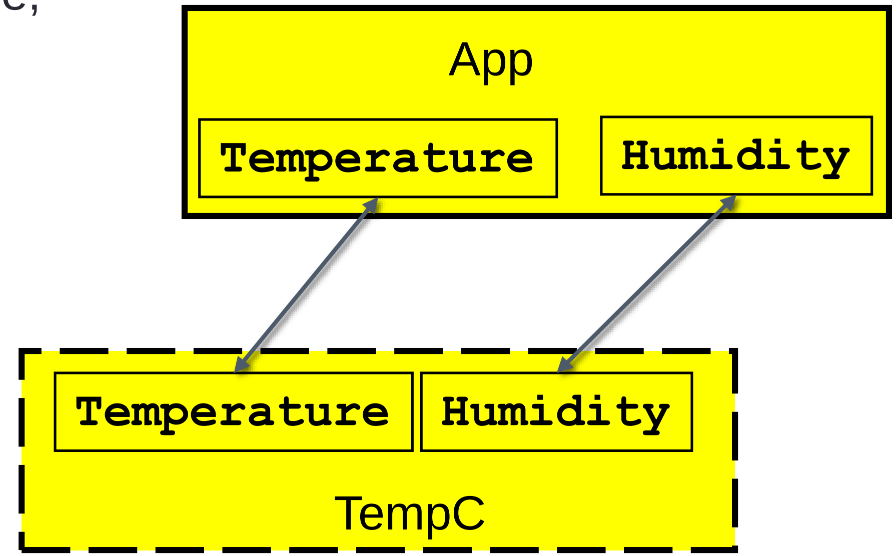
```
uses interface Timer<TMilli>  as SampleTimer;
…
event void Boot.booted() {
      call SampleTimer.startPeriodic(DEFAULT_TIMER);
       ….
}
```



The configuration

components new SensirionSht11C() as TempC ;
components new HamamatsuS10871TsrC() as LightSensor;

App.Temperature -> TempC.Temperature;
App.Humidity -> TempC.Humidity;
App.Light -> LightSensor.Read;



Read several sensors (2)

- At each event of the timer we trigger the read:

```
event void SampleTimer.fired() {
  call Temperature.read();
  call Humidity.read();
  call Photo.read();
}
```

- An event is generated when a « read » is completed:

```
event void Temperature.readDone(error_t result, uint16_t value)
{
     data.temperature = value;    // put data into packet
}
```

Type declarations

- One may construct a structure to hold all ten data, to define data.temperature, data.humidity, etc. We must then define it in a file .h (like declarations.h) that should be included.

- The file declarations.h contains the following type

```
typedef nx_struct THL_msg {
     nx_uint16_t temperature;
     nx_uint16_t humidity;
     nx_uint16_t photo;
 } THL_msg_t;
```

- Then we define a data with the above type:

```
THL_msg_t data;
```

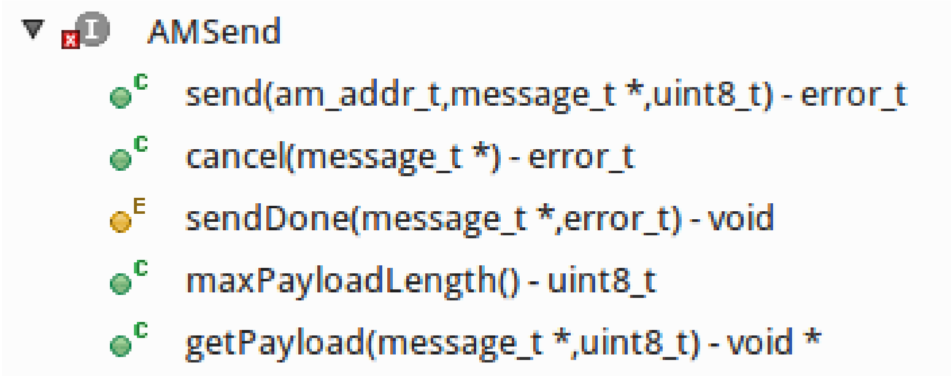
USING RADIO TO SEND PACKETS

Steps to send a packet

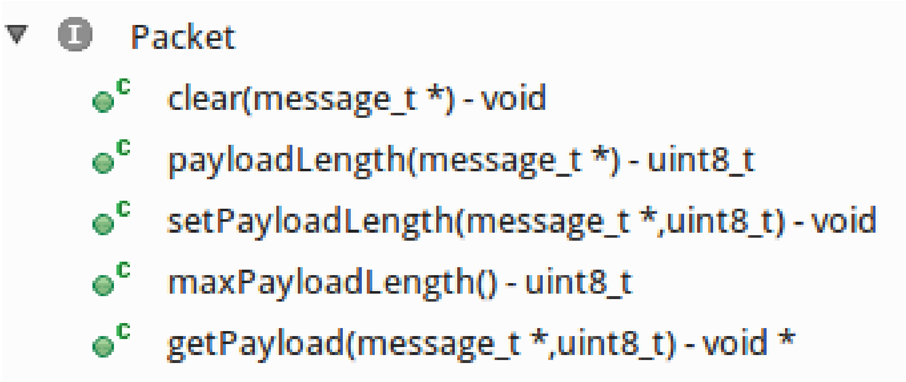
- Identify the interfaces (and components) that provides access to radio and allow us to manipulate the packet.
- Make a timer to call regularly a send function
- Prepare the whole packet
- Ask to send
- Wait for send done

Interfaces and components

- AMSend interface with this interface we can send a packet.



- Packet, and AMPacket can be used to deal with the packet.



Radio communication, packet structure

- The packet that is sent or received has a special structure that respects the underlying PHY layer standard (done in the implementation and transparent for the user),

- It is a buffer in the memory, with the payload, but also with all the required headers and addresses.

- The type used in tinyOS is : **message_t**

Dest Addr	Link Src Addr	Msg len	Grp ID	Hndlr ID	Payload
2	2	1	1	1	Max 28

Radio communications

- We go through an example.
- First step: define the message type
- Always use a structure with different fields to hold the message:

```
typedef nx_struct BlinkToRadioMsg {
      nx_uint16_t nodeid;
      nx_uint16_t counter;
} BlinkToRadioMsg;
```

- In this example we have just two values to transmit, the source identity and a 16-bit value
- The next step is to identify the component to use.

Type declaration for payload

- We may put all of our declarations in a .h file (to be included)
- The file declarations.h contains the following type

```
typedef nx_struct THL_msg {
     nx_uint16_t NodeId;
     nx_uint16_t temperature;
     nx_uint16_t humidity;
     nx_uint16_t photo;
 } THL_msg_t;
```

- Then we define a data with the above type:

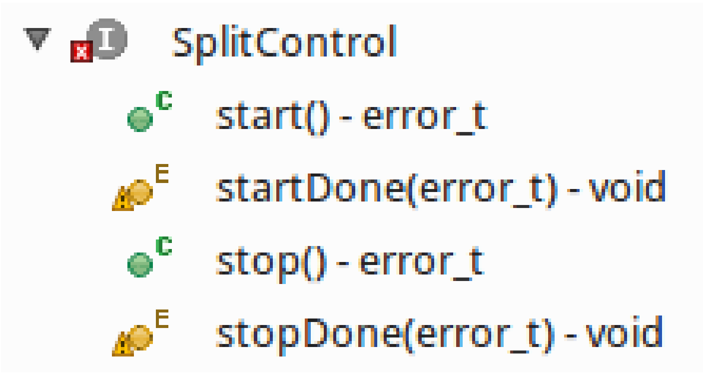
```
THL_msg_t data;
```

Radio communications

- The interfaces that can be used: Packet, AMPacket, AMSend and also SplitControl.

```
uses interface Packet;
uses interface AMPacket;
uses interface AMSend;
uses interface SplitControl as AMControl; // renamed
```

- ”SplitControl” is a general interface used for starting and stopping processes,

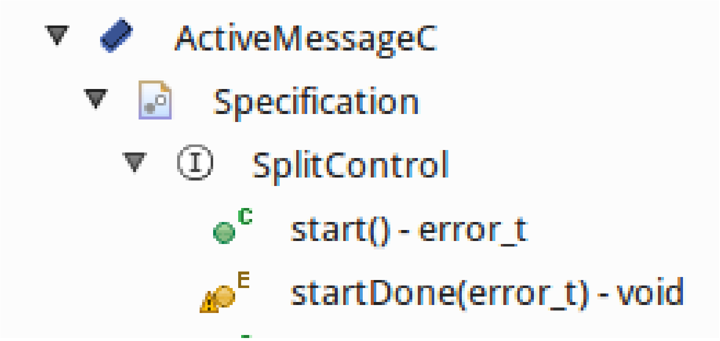


- We need a message_t to hold our data for transmission.

Turn on the raido

- At the beginning we need to turn on the radio.
- The splitControl start function can be implemented by a proper component to turn on the radio.
- So to initialize the radio we use the command:

```
AMControl.start()
```



- We can do it inside “booted” event

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

Timer configuration

- If the radio initialization is completed, we initialize the timer

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

- With already declared in .h. For example:

```
enum { TIMER_PERIOD_MILLI = 5000 };
```

- Suppose we want to send a packet at each timer event.

Packet preparation

- Now we create a pointer to point to the message. The packet is created by

```
message_t pkt;
```

- The type "message_t" is the abstraction of the buffer containing the packet structure.

- The user cannot access directly to its members. He must use the accessors to access different field of the packet. These accessors are provided by the interface AMPacket.

Packet manipulations

- Using the command Packet.getPayload:

```
call Packet.getPayload(&pkt, NULL);
```

The pointer that is returned point to the payload and we cast it to the real type of the payload:

```
(BlinkToRadioMsg*)(call Packet.getPayload(&pkt, NULL));
```

- Now create the pointer:

```
BlinkToRadioMsg* btrpkt =
(BlinkToRadioMsg*)(call Packet.getPayload(&pkt, NULL));
```

Packet preparation

- Different fields of the message will be assigned:

```
btrpkt->nodeid = TOS_NODE_ID;
btrpkt->counter = counter;
```

- The memory is pointed by “btrpkt” of type “BlinkToRadioMsg” and also by pkt of standard type “message_t”

- Now send the message pointed by pkt using AMSend:

```
        call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof
(BlinkToRadioMsg))
```

- The call will return the status, if it is ok, we should make the busy flag “TRUE” (see next slide)

Packet preparation

```
if (call AMSend.send(AM_BROADCAST_ADDR, &pkt,
sizeof(BlinkToRadioMsg)) == SUCCESS) { busy = TRUE; }
```

- Using the destination address as AM_BROADCAST_ADDR, causes a broadcast to all nodes in the range.
- When the packet is sent the sendDone event is triggered.

Packet preparation

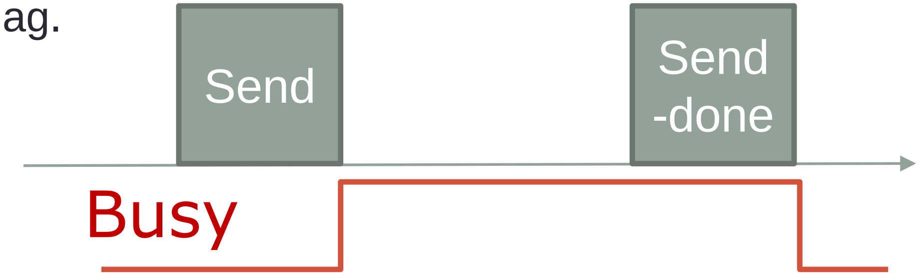
```
event void Timer0.fired() {
    ...
    if (!busy) {
          BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call
Packet.getPayload(&pkt, sizeof(BlinkToRadioMsg));
          btrpkt->nodeid = TOS_NODE_ID;
          btrpkt->counter = counter;
          if (call AMSend.send(AM_BROADCAST_ADDR, &pkt,
sizeof(BlinkToRadioMsg))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

- The first line checks and goes through if the transmitter is not busy

Busy flag trick

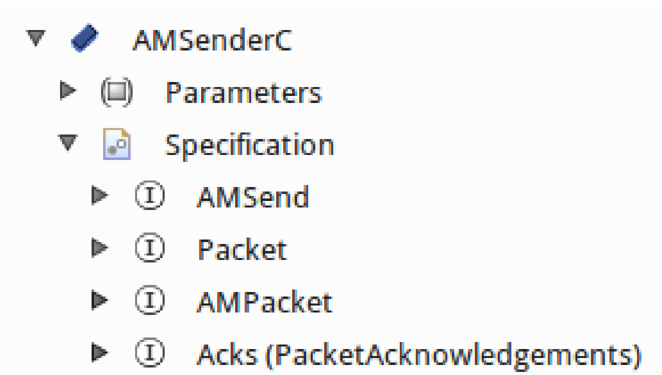
```
/**
* Signaled in response to an accepted send request. msg is
* the message buffer sent, and error indicates whether
* the send was successful.
*
* @param msg the packet which was submitted as a send request
* @param error SUCCESS if it was sent successfully, FAIL if it was not,
• ECANCEL if it was cancelled
*/
event void sendDone(message_t* msg, error_t error);
```

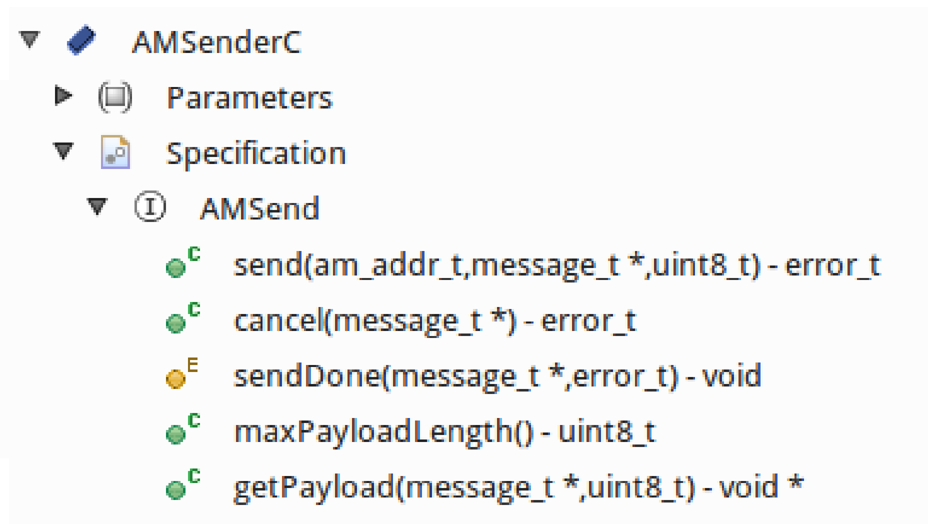
- We use this event to clear the busy flag.



Components

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






```
implementation {
      ...
      components ActiveMessageC;
      components new AMSenderC(AM_BLINKTORADIO);
      ...
}
```

Connection in configuration file

```
implementation {
    ...
    App.Packet -> AMSenderC;
    App.AMPacket -> AMSenderC;
    App.AMSend -> AMSenderC;
    App.AMControl -> ActiveMessageC.splitControl;
}
```

RADIO RECEPTION

Reception

- The reception is done by event.

- We need the components:
    - ActiveMessageC
    - AMReceiverC(x), with x (AM Type) the same number as the TX used in AMSenderC(x)

- In the module we need two interfaces
    - SplitControl
    - Receive

The event is Receive.receive where it returns a pointer that points to the data structure (payload). We must cast the type to the same structure that is created in both TX and RX sides.

Destination ID

- If the data is not addressed to the mote, the event is not generated
- Therefore the µP will not informed about the packet
- Use AMPacket interface implemented by ActiveMessageC component to fix the destination address.

- AMPacket
  - address() - am_addr_t
  - destination(message_t *) - am_addr_t
  - source(message_t *) - am_addr_t
  - setDestination(message_t *,am_addr_t) - void
  - setSource(message_t *,am_addr_t) - void
  - isForMe(message_t *) - bool
  - type(message_t *) - am_id_t
  - setType(message_t *,am_id_t) - void
  - group(message_t *) - am_group_t
  - setGroup(message_t *,am_group_t) - void
  - localGroup() - am_group_t

Dest Addr	Link Src Addr	Msg len	Grp ID	Hndlr ID	Payload
2	2	1	1	1	Max 28

Conclusion

- You should have a good understanding about WSN implementation, and the tools
- We focused on the high level programming, however the engineer are supposed to go to lower levels.
- You are able to send and receive packets
- Now you are required to do more, such as routing.

Writing on the console

- Several things to do

1. add the following line in the configuration
   - components SerialPrintfC;
2. add in the module the following lines
   - #include <stdio.h>
   - #include <string.h>
3. Add the following line in Makefile:
   - PFLAGS += -I$(TOSDIR)/lib/printf
4. Now you can use
   - Printf(« %d », val);
5. To be able to receive the characters on the PC you need a serial port terminal. Use in ubuntu: Applications -> Accessories -> Terminal

Board connection

- Enter : motelist
  - You should see the answer of the system as /dev/ttyUSB0 exists
- Give the write permission by
  - sudo chmod 666 /dev/ttyUSB0
  - Then it asks you for your password

Some components and their interfaces