



---

# Wireless Systems Laboratory

4 November 2013

A. Cammarano, A. Caposese, D. Spenza



## Contacts

---

Cammarano: [cammarano@di.uniroma1.it](mailto:cammarano@di.uniroma1.it)

Caposese: [caposese@di.uniroma1.it](mailto:caposese@di.uniroma1.it)

Spenza: [spenza@di.uniroma1.it](mailto:spenza@di.uniroma1.it)

Tel: 06-49918430

Room: 333

Slides: <http://wwwusers.di.uniroma1.it/~spenza/lab2013.html>



# Outline

---

- The BlinkToRadio Application
- Mote-PC serial communication
- BaseStation
- SerialForwarder



# BlinkToRadio Application

---

- A counter is incremented every second
- Whenever the timer fires, the value of the counter is sent over a radio message
- Whenever a radio message is received, the three least significant bits of the counter in the message payload are displayed on the LEDs



# Exercise 1

---

- Assign an unique ID to your nodes
- Filter radio messages that are not sent to you



## Exercise 2

---

- Node A sends the value of its counter to node B
- Node B displays the three least significant bits of the counter on the LEDs, updates the value of the counter and sends it back to node A
- Node A displays the three least significant bits of the counter on the LEDs, updates the value of the counter and sends it back to node B
- ....



# Mote-Pc Serial Communication

---

- TinyOS provides high-level communication interfaces
  - Similar for radio and serial communication
- Basic interfaces:
  - Packet: Set/get payload of TinyOS message\_t packets
  - Send: Send packet by calling send() command
  - Receive: Reception of packets signaled by receive() event
- Active Message interfaces allow for multiplexing:
  - AMPacket: Provide source and destination address to packet
  - AMSend: Send packet to destination address



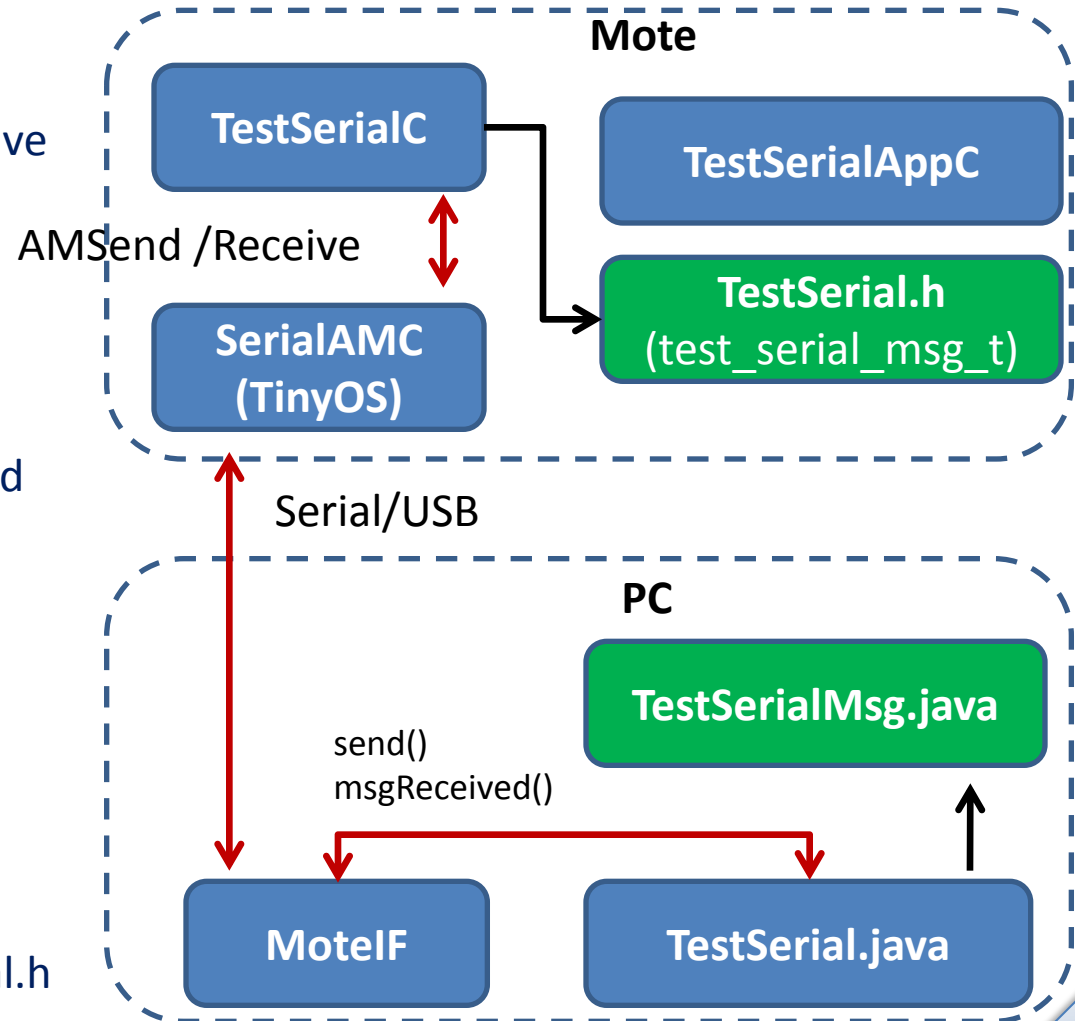
- Mote and PC components
  - Both increment counter values and send to the other
- Mote: nesC and TinyOS
  - Outputs last three bits of PC counter value to LEDs
- PC: Java and TinyOS Java libraries
  - Outputs mote counter value to stdout
- Demonstration





# TestSerial

- TestSerialC
  - Interfaces AMSend, Receive wired to TinyOS serial component
- test\_serial\_msg\_t
  - Payload struct
- MotelF
  - TinyOS Java library to send and receive packets
- TestSerial.java
  - Prints received counter values
  - Increments and sends counter values
- TestSerialMsg.java
  - Payload encapsulation
  - Generated from TestSerial.h





# Packet Payload Format

- Usually defined in C header file (TestSerial.h)
- nx-types abstract away big/little endian hardware/communication differences
- Default payload size is 29 bytes, but can be enlarged
- Active Message type AM\_TEST\_SERIAL\_MSG
  - Integer value to distinguish between multiple packet types (multiplexing)
- TinyOS libraries convert struct to a Java class with set/get methods (TestSerialMsg.java)
  - Message Interface Generator (example in TestSerial Makefile)

```
// TestSerial.h
typedef nx_struct test_serial_msg
{
    nx_uint16_t counter;
} test_serial_msg_t;

enum { AM_TEST_SERIAL_MSG=6; }
```



# TestSerialAppC: wiring

- SerialActiveMessageC allows for multiplexing
  - Multiple packet types (e.g. sensor control and data)
  - Differentiate through AM types: AM\_TEST\_SERIAL\_MSG
  - Parameters defined in brackets []
- SerialActiveMessageC provides several interfaces
  - Wired to TestSerialC
  - SplitControl to turn on and off the UART/serial bus
  - AMSend and Receive for transmitting and receiving
  - Packet to set and get payload

```
configuration TestSerialAppC {}  
implementation {  
    components SerialActiveMessageC as AM, SplitControl;  
    ...  
    App.Receive -> AM.Receive[AM_TEST_SERIAL_MSG];  
    App.AMSend -> AM.AMSend[AM_TEST_SERIAL_MSG];  
    App.Control -> AM;  
    App.Packet -> AM;  
}
```



## TestSerialC: Booting

- When mote boots, turn on UART
- When UART is powered, start timer to send packets
- Implement `Control.stopDone()` to turn off UART

```
event void Boot.booted() {
    call Control.start();
}
event void Control.startDone(error_t err) {
    if (err == SUCCESS) {
        call MilliTimer.startPeriodic(1000);
    }
}
event void Control.stopDone(error_t err) {}
```



# TestSerialC: Sending Packets

- Timer fires, increment counter
- Get message\_t payload pointer: Packet.getPayload();
- Set payload value: rcm->counter = counter;
- Send packet: AMSend.send();
  - Provide AM destination address, message\_t packet address, payload size
- Packet sent: AMSend.sendDone();

```
event void MilliTimer.fired() {
    counter++;
    message_t packet;
    test_serial_msg_t* rcm = (test_serial_msg_t*)call
...Packet.getPayload(&packet, sizeof(test_serial_msg_t));

    rcm->counter = counter;
    call AMSend.send(AM_BROADCAST_ADDR, &packet,
...sizeof(test_serial_msg_t));
}

event void AMSend.sendDone(message_t* bufPtr, error_t error){
}
```



## TestSerialC: Receiving Packets

- Packet received: `Receive.receive()`;
  - Provides `message_t` packet, payload pointer, and payload size
  - Get payload: cast from `void*` to `test_serial_msg_t*`
  - Set LEDs according to value of last 3 bits

```
event message_t* Receive.receive(message_t* bufPtr, void* payload, uint8_t
...len) {
    test_serial_msg_t* rcm = (test_serial_msg_t*)payload;
    if (rcm->counter & 0x1) {
        call Leds.led0On();
    }
    // turn on other LEDs accordingly
    ...
    return bufPtr;
}
```



# PC: TestSerial.java

- Initialization
  - Create packet source from args[]: “-comm serial@\dev\ttyUSB0:telosb”
  - Register packet listener for TestSerialMsg and source
- Send packets

```
public class TestSerial implements MessageListener {
    private MoteIF moteIF;

    public static void main(String[] args) throws Exception {
        ...
        String source = args[1]; PhoenixSource phoenix =
...BuildSource.makePhoenix(source, PrintStreamMessenger.err);
        MoteIF mif = new MoteIF(phoenix);
        TestSerial serial = new TestSerial(mif);
        serial.sendPackets();
    }

    public TestSerial(MoteIF moteIF) {
        this.moteIF = moteIF;
        this.moteIF.registerListener(new TestSerialMsg(), this);
    }
}
```



## TestSerial.java: send packets

- Initialize counter and create TestSerialMsg payload
- While loop
  - Increment counter and sleep for some period
  - Set payload counter: `payload.set_counter();`
  - Send packet: `moteIF.send();` with destination address 0

```
public void sendPackets() {
    int counter = 0;
    TestSerialMsg payload = new TestSerialMsg();
    ...
    while (true) {
        ...
        // increment counter and wait for some amount of time before sending
        System.out.println("Sending packet " + counter);
        payload.set_counter(counter);
        moteIF.send(0, payload);
    }
}
```





## TestSerial.java: receiving packets

---

- TestSerial.messageReceived() triggered by incoming packet while listener is registered
  - Provides AM destination address and abstract class Message
- Cast message to TestSerialMsg
- Retrieve counter: msg.get\_counter();

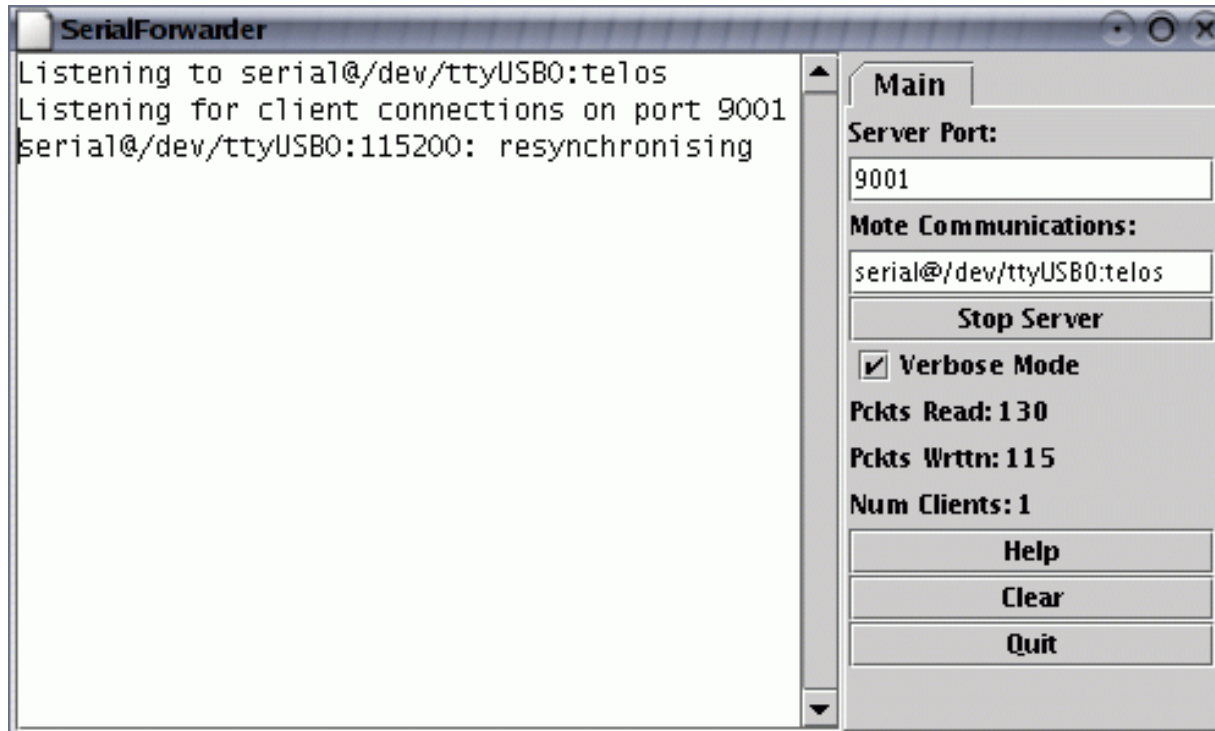
```
public void messageReceived(int to, Message message) {  
    TestSerialMsg msg = (TestSerialMsg)message;  
    System.out.println("Received packet sequence number " +  
...msg.get_counter());  
}
```



# Serial Forwarder

Instead of connecting to the serial port directly, applications connect to the SerialForwarder, which acts as a proxy to read and write packets

```
java net.tinyos.sf.SerialForwarder -comm serial@/dev/ttyUSB0:telosb
```





# Base Station

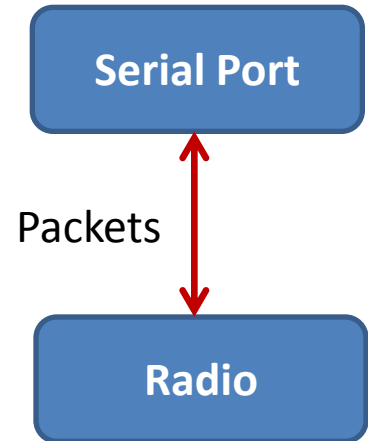
It is a basic TinyOS utility application. It acts as a bridge between the serial port and radio network.

## BlinkToRadio

- **Destination address** (2 bytes)
- **Link source address** (2 bytes)
- **Message length** (1 byte)
- **Group ID** (1 byte)
- **Active Message handler type** (1 byte)
- **Payload** (up to 28 bytes):
  - **source mote ID** (2 bytes)
  - **sample counter** (2 bytes)



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



\$ java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB0:telosb

dest addr	link source addr	msg len	groupID	handlerID	source addr	counter
ff ff	00 00	04	22	06	00 02	00 0B



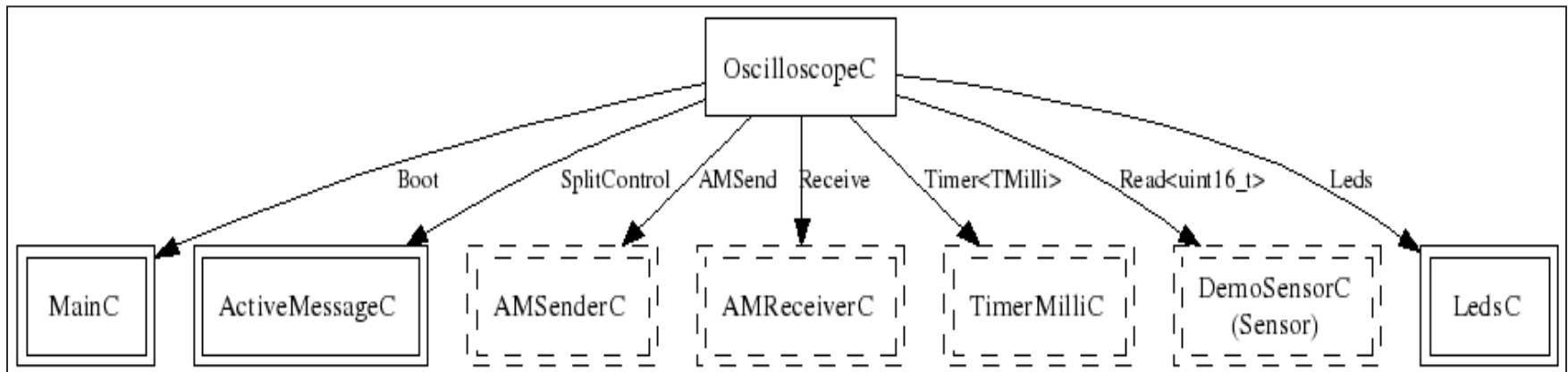
# Oscilloscope

---

- Periodically samples the default sensor via (DemoSensorC) and broadcasts a message with 10 accumulated readings over the radio.
- Application Components
  - *OscilloscopeAppC (Configuration), OscilloscopeC (Module)*
    - *BaseStation*
- System Components
  - *MainC, LedsC, TimerMilliC, DemoSensorC*
  - *AMReceiverC, AMSenderC, ActiveMessageC*



- OscilloscopeAppC: components' graph





- Oscilloscope.h

```
enum
{
    NREADINGS = 10,
    DEFAULT_INTERVAL = 256,
    AM_OSCILLOSCOPE = 0x93
};

typedef nx_struct oscilloscope
{
    nx_uint16_t version;    /* Version of the interval.
*/
    nx_uint16_t interval;  /* Sampling period. */
    nx_uint16_t id;        /* id of sending mote. */
    nx_uint16_t count;     /* The readings are samples
count * NREADINGS onwards */
    nx_uint16_t readings[NREADINGS];
} oscilloscope_t;
```



- OscilloscopeAppC.nc

```
#include "Oscilloscope.h"

configuration OscilloscopeAppC { }
implementation {
    components OscilloscopeC, MainC, ActiveMessageC,
    LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE),
    new AMReceiverC(AM_OSCILLOSCOPE);

    OscilloscopeC.Boot -> MainC;
    OscilloscopeC.RadioControl -> ActiveMessageC;
    OscilloscopeC.AMSend -> AMSenderC;
    OscilloscopeC.Receive -> AMReceiverC;
    OscilloscopeC.Timer -> TimerMilliC;
    OscilloscopeC.Read -> Sensor;
    OscilloscopeC.Leds -> LedsC;
}
```



- OscilloscopeC.nc

```
#include "Timer.h"
#include "Oscilloscope.h"
module OscilloscopeC{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Leds;}
}
implementation{
  message_t sendBuf;
  bool sendBusy;
  oscilloscope_t local;
  uint8_t reading;
  bool suppressCountChange;
```





- OscilloscopeC.nc

```
// Utility C-style functions
void report_problem() { call Leds.led0Toggle(); }
void report_sent() { call Leds.led1Toggle(); }
void report_received() { call Leds.led2Toggle(); }
void startTimer()
{call Timer.startPeriodic(local.interval); reading =
0;}

event void Boot.booted() {
  local.interval = DEFAULT_INTERVAL;
  local.id = TOS_NODE_ID;
  if (call RadioControl.start() != SUCCESS)
    report_problem();
}

event void RadioControl.startDone(error_t error) {
  startTimer();
}
event void RadioControl.stopDone(error_t error) {}
```



- OscilloscopeC.nc

```
event void Timer.fired(){
    if (reading == NREADINGS)
    {
        if (!sendBusy&&sizeof local <= call
AMSend.maxPayloadLength())
        {
            memcpy(call AMSend.getPayload(&sendBuf,
                sizeof(local)), &local, sizeof local);
            if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf,
                sizeof local) == SUCCESS) sendBusy = TRUE;
        }

        if (!sendBusy) report_problem();
        reading = 0;
        if (!suppressCountChange) local.count++;
        suppressCountChange = FALSE;
    }
    if (call Read.read() != SUCCESS) report_problem();
}
```



- OscilloscopeC.nc

```
event void Read.readDone(error_t result, uint16_t
data)
{
    if (result != SUCCESS)
    {
        data = 0xffff;
        report_problem();
    }
    local.readings[reading++] = data;
}
}
```



- OscilloscopeC.nc

```
event void AMSend.sendDone(message_t* msg, error_t error)
{ if (error == SUCCESS) report_sent();
  else report_problem();
  sendBusy = FALSE; }

event message_t*
Receive.receive(message_t* msg, void* payload, uint8_t
len)
{ oscilloscope_t *ormsg = payload;
  report_received();

  if (ormsg->version > local.version)
    {local.version = ormsg->version; local.interval = ormsg-
>interval;
    startTimer(); }
  if (ormsg->count > local.count)
    {local.count = ormsg->count; suppressCountChange =
TRUE; }
  return msg;
}
```