CSC2026
Mote-to-Mote radio communication

Daniel Nesbitt
daniel.nesbitt@ncl.ac.uk

Recap

- The motes used in this course run TinyOS.
- nesC is the programming language used to develop software on the motes.
  - nesC is a dialect of the C language.
  - A **component** is the basic unit of a nesC program.
  - Components provide and use **interfaces**
  - A **configuration** wires other components together.
  - A **module** contains executable code.
- A **Task** is a function that can be ran at a later time.
  - This is useful for large computational operations, as a synchronous/concurrent operations could result in deadlock.

Sending data with a mote

- The fundamental building block of mote communication is the common message buffer abstraction, `message_t`

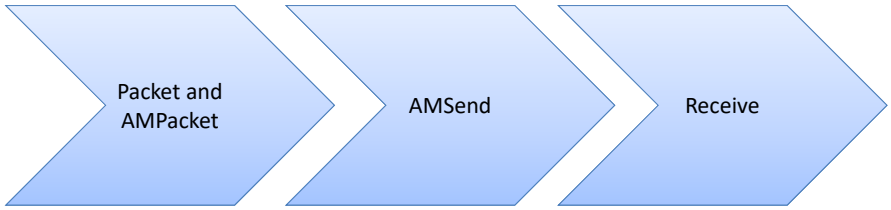
```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

- The message fields are not directly manipulated, instead they are modified through external interfaces.

Sending data with a mote

- TinyOS provides a number of abstractions of the mote's network interface.
- The most basic abstraction is the *Active Message (AM).*
- The AM layer multiplexes access to the radio, allowing for multiple services to use a single radio.
- AM types provide functionality similar to Ethernet frame type field and IP type field.

Sending data with a mote



Provides accessors to message_t, also provides functions to clear the message's contents, get the payload length etc.

Provides commands for sending a message and cancelling a pending message. The interface also provides an event to indicate if a message was sent successfully.

Provides and event for receiving messages.

Getting started

- First we rename the Blink program to BlinkToRadio from last week's tutorial, then extend it by defining `TIMER_PERIOD_MILLI` in a header file:

```
//BlinkToRadioC.nc

#include <Timer.h>
#include "BlinkToRadio.h"

 module BlinkToRadioC {
   uses interface Boot;
   uses interface Leds;
   uses interface Timer<TMilli> as Timer0;
 }
 implementation {
   uint16_t counter = 0;

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

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

```
//BlinkToRadio.h

#ifndef BLINKTORADIO_H
 #define BLINKTORADIO_H

 enum {
   TIMER_PERIOD_MILLI = 1000
 };

 #endif
```

- Here `BlinkToRadioC.nc` provides the implementation logic for our file and `BlinkToRadio.h` defines constants and data structures.

Getting started

- Next we need to wire the interfaces:

```
//BlinkToRadioAppC.nc

#include <Timer.h>
#include "BlinkToRadio.h"

configuration BlinkToRadioAppC {
}
implementation {
  components MainC;
  components LedsC;
  components BlinkToRadioC as App;
  components new TimerMilliC() as Timer0;

  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
}
```

Defining a message structure

- At this stage, we can define our message stricture.
  - For this tutorial we'll use a `uinit16_t` for the node id and a `uinit16_t` for the counter:

```
//BlinkToRadio.h
...

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

...
endif
```

- The `nx` prefix indicates that the `struct` and the `int`s are external types, which are not part of the standard C language.
  - Can anyone remember why this is done?

Sending a message

- Now that we have implemented the basic structure of our message, we now need to provide the implantation.
- We need:
  - `Packet` and `AMPacket` interfaces to access the `message_t` structure.
  - The `AMSend` interface to specify how we send packets.
  - We additionally need the `SplitControl` interface to allow the start up of the radio.

Sending a message

- Now that we know which interfaces to use, they need to be added to the module block of BlinkToRadioC.nc:

```
module BlinkToRadioC {
  ...
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface SplitControl as AMControl;
}
```

- In our example, `SplitControl` is renamed to `AMControl` using the `as` keyword.
- Next we can add additional module-scope variables. We need a `message_t` to hold the data for transmission. We also need a bool flag to indicate when the radio is busy sending.

```
//BlinkToRadioC.nc

implementation {
  bool busy = FALSE;
  message_t pkt;
  ...
}
```

Sending a message

- Now we need to handle initialisation of the radio. The radio is started by calling `AMControl.start` inside `Boot.booted`. The timer will be used to send the messages, but should be only initialised when the radio is ready. This can be done by placing the timer code in the `AMControl.startDone` event.

```
//BlinkToRadioC.nc
event void Boot.booted() {
//Removed Timer0.startPeriodic(TIMER_PERIOD_MILLI);
call AMControl.start();
  }

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

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

If the radio starts successfully, the `error_t` parameter will be set to SUCESS

We also need to implement the event `AmControl.stopDone`, however we can leave it as a stub.

Sending a message

- Since we want to transmit the node ID and the counter when the timer fires, we need to add the following to `Timer0.fired` event handler:

```
//BlinkToRadioC.nc
event void Timer0.fired() {
  ...
1 if (!busy) {
    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call Packet.getPayload(&pkt, sizeof (BlinkToRadioMsg))); 2
  3 btrpkt->nodeid = TOS_NODE_ID;
    btrpkt->counter = counter;
    if (call AMSend.send(2, &pkt, sizeof(BlinkToRadioMsg)) == SUCCESS) { 4
      busy = TRUE;
    }
  }
}
```

1. Check that a message is not already in transmission

2. Get the Packet's payload portion and cast it as a pointer to `BlinkToRadioMsg` external type (as defined in `BlinkToRadio.h`)

3. Use the pointer to initialise the Packet's fields.

4. Send the packet using `AMSend.send` to the node with ID 2, and verify that the AM layer has accepted the message.

Implement any events

- Looking at the `Packet`, `AMPacket` and `AMSend` files, we can see that we only need to implement one event, `AMSend.sendDone`:

```
/**
   * 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
   * @see send
   * @see cancel
   */
  event void sendDone(message_t* msg, error_t error);
```

- The event is signalled after a transmission attempt. It also returns ownership of `msg` from `AMSend` back to the component that originally called the `AMSend.send` command. Therefore in this event, we need to set the `busy` flag to FALSE so the message buffer can be re-used.

Implementing the sendDone event

```
/**
   * 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
   * @see send
   * @see cancel
   */
  event void sendDone(message_t* msg, error_t error);
```

Why is it a good idea to check that the message buffer is the same as the local message buffer?

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

Specifying the components

- We now need to specify the components we have used in BlinkToRadioAppc.nc:

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

- The `AMSenderC` is a generic, parameterised component. The new keyword creates a new instance of `AmSenderC`, however we need to specify the AM Type using a parameter (AM_BLINKTOREADIO):

```
//BlinkToRadio.h
...
enum {
  AM_BLINKTORADIO = 6,
  TIMER_PERIOD_MILLI = 1000
};
...
```

Wiring the components

- Now we need to wire together the interfaces we use to the providing components:

```
//BlinkToRadioAppC.nc
implementation {
  ...
  App.Packet -> AMSenderC;
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.AMControl -> ActiveMessageC;
}
```

- At this stage we should check our program compiles by typing in the terminal: `make gnode install,number`
- `number` specifies the AM address of the node.

Receiving messages

- Now that our mote can send messages, we can now extend the BlinkToRadio program to receive messages back.
- In the practicals, the demonstrators will have a Echo mote with a program that receives any radio message. It will then echo back the message to the mote using the sender's `TOS_NODE_ID` as the destination.

Receiving messages – extending BlinkToRadio

- For our program, we'll set the LEDs to blink when the mote receives a message.
- So before we implement anything, the line **`call Leds.set(counter);`** from the **`Timer0.fired`** event handler should be removed.
- Next, we'll specify the Receive interface in BlinkToRadioAppC.nc:

```
//BlinkToRadioAppC.nc
module BlinkToRadioC {
  ...
  uses interface Receive;
}
```

Receiving messages - implementation

- Next we need to implement the `Receive.receive` event handler:

```
//BlinkToRadioC.nc
...
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
1  if ((len == sizeof(BlinkToRadioMsg)) && (call AMPacket.destination(msg) ==
TOS_NODE_ID)) {
     BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;  2
3    call Leds.set(btrpkt->counter*2);
   }
   return msg;
}
...
```

1. Check the message length is what is expected and it is not a broadcast message.

2. Cast the payload message to a structure pointer of type `BlinkToRadioMsg`

3. Set the three LEDs using the counter value contained in the message.

Receiving messages – wiring interfaces

- To finish, we need to specify the components and wire them together:

AMReceverC is initialised in the same manor as AmSenderC

```
//BlinkToRadioAppC.nc
implementation {
  ...
  components new AMReceiverC(AM_BLINKTORADIO);
  ...

  App.Receive -> AMReceiverC;

}
```

- Now we can run the program by typing make `gnode install,number`

Mote-PC serial communication

- So far the only way we can tell if our program works is by the blinking LEDs.
- As part of the TinyOS installation, there is a Java based utility for listening in on the communication between motes.
- This is extremely useful for debugging *and possibly the coursework!*
- First we need to add some extra lines to our program:

```
//BlinkToRadioAppC.nc
components SerialActiveMessageC; //new
components new SerialAMSenderC(AM_BLINKTORADIO); //new

App.SerialControl -> SerialActiveMessageC; //new
App.SerialSend -> SerialAMSenderC; //new
```

Mote-PC serial communication

- Next we need to declare the interfaces and define the required event handlers:

```
//BlinkToRadioC.nc
Module BlinkToRadioC
{
      …
     uses interface SplitControl as SerialControl; //new
     uses interface AMSend as SerialSend; //new
}

event void SerialControl.startDone(error_t err) {
 if (err == SUCCESS) {
  } else {
    call SerialControl.start();
  }
}
event void SerialControl.stopDone(error_t err) {}
event void SerialSend.sendDone(message_t* msg, error_t error) {}
```

Mote-PC serial communication

- Next we need to initialise the serial interface component:

```
//BlinkToRadioC.nc
...

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

- Finally we need to send a message to the PC after the message is received from the Echo:

```
//BlinkToRadioC.nc
...
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
   if ((len == sizeof(BlinkToRadioMsg)) && (call AMPacket.destination(msg) == TOS_NODE_ID)) {
     BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
     call SerialSend.send(TOS_NODE_ID, msg, sizeof(BlinkToRadioMsg)); // new
     call Leds.set(btrpkt->counter);
   }
   return msg;
 }
```

Seeing the messages

- Make sure the program is compiled and running: `make gnode install,number`
- To see the messages being sent to your PC, enter the following commands into the VM terminal:

```
cd /home/tinyos/tinyos-2.x/support/sdk/java
java -cp tinyos.jar net.tinyos.tools.Listen -comm serial@/dev/ttyUSB0:57600
```

- After a short pause, you should start to see a series of lines of hexadecimal numbers. Each line represents a return message from the Echo:

1	2	3	4	5
00	00 0C	00 00 02	01 06	00 0C 00 0B
00	00 0C	00 00 02	01 06	00 0C 00 0C
00	00 0C	00 00 02	01 06	00 0C 00 0D
00	00 0C	00 00 02	01 06	00 0C 00 0E
00	00 0C	00 00 02	01 06	00 0C 00 0F
00	00 0C	00 00 02	01 06	00 0C 00 10
00	00 0C	00 00 02	01 06	00 0C 00 11

1. The first byte is always zero (00)
2. Two bytes represent the source node for the message, most significant byte first (00 0C). This should be the HEX representation of your unique ID.
3. Two bytes represent the destination node for the message, most significant byte first (00 02). (Echo mote ID)
4. 1 byte represents the AM message number defined in the BlinkToRadio.h header file (06).
5. The last 4 bytes represent the payload. In this case btrpkt->nodeid (00 0C) and btrpkt->counter (00 0B).