Mote-mote radio communication

From TinyOS Wiki

This lesson introduces radio communications in TinyOS. You will become familar with TinyOS interfaces and components that support communications and you will learn how to:

- Use `message_t`, the TinyOS 2.0 message buffer.
- Send a message buffer to the radio.
- Receive a message buffer from the radio.

Contents

- 1 Introduction
  - 1.1 Basic Communications Interfaces
  - 1.2 Active Message Interfaces
  - 1.3 Components
  - 1.4 Naming Wrappers
- 2 The TinyOS 2.0 Message Buffer
- 3 Sending a Message over the Radio
  - 3.1 Reimplementing Blink
  - 3.2 Defining a Message Structure
  - 3.3 Sending a Message
- 4 Receiving a Message over the Radio
- 5 Conclusions
- 6 Related Documentation

Introduction

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 message buffer abstraction, called `message_t`, which is implemented as a nesC struct (similar to a C struct). The `message_t` abstraction replaces the TinyOS 1.x `TOS_Msg` abstraction. Unlike TinyOS 1.x, the members of `message_t` are opaque, and therefore not accessed directly. Rather, `message_t` is an *abstract data type*, whose members are read and written using accessor and mutator functions <ref name="fn1">TEP 111: message_t (http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html) </ref>.

Basic Communications Interfaces

There are a number of interfaces and components that use `message_t` as the underlying data structure. Let's take a look at some of the interfaces that are in the `tos/interfaces` directory to familiarize ourselves with the general functionality of the communications system:

- `Packet (http://www.tinyos.net/tinyos-2.x/tos/interfaces/Packet.nc)` - 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 (http://www.tinyos.net/tinyos-2.x/tos/interfaces/Send.nc)` - Provides the basic *address-free* message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.
- `Receive (http://www.tinyos.net/tinyos-2.x/tos/interfaces/Receive.nc)` - 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.
- `PacketAcknowledgements (http://www.tinyos.net/tinyos-2.x/tos/interfaces/PacketAcknowledgements.nc)` - Provides a mechanism for requesting acknowledgements on a per-packet basis.
- `RadioTimeStamping (http://www.tinyos.net/tinyos-2.x/tos/interfaces/RadioTimeStamping.nc)` - Provides time stamping information for radio transmission and reception.

Active Message Interfaces

Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. The term "AM type" refers to the field used for multiplexing. AM types are similar in function to the Ethernet frame type field, IP protocol field, and the UDP port in that all of them are used to multiplex access to a communication service. An AM packet also includes a destination field, which stores an "AM address" to address packets to particular motes. Additional interfaces, also located in the `tos/interfaces` directory, were introduced to support the AM services:

- `AMPacket (http://www.tinyos.net/tinyos-2.x/tos/interfaces/AMPacket.nc)` - Similar to `Packet`, provides the basic AM accessors for the `message_t` abstract data type. This interface provides commands for getting a node's AM address, an AM packet's destination, and an AM packet's type. Commands are also provided for setting an AM packet's destination and type, and checking whether the destination is the local node.
- `AMSend (http://www.tinyos.net/tinyos-2.x/tos/interfaces/AMSend.nc)` - Similar to `Send`, provides the basic Active Message sending interface. The key difference between `AMSend` and `Send` is that `AMSend` takes a destination AM address in its `send` command.

The AM address of a node can be set at installation time, using the `make install.`*n* or `make reinstall.`*n* commands. It can be changed at runtime using the `ActiveMessageAddressC` component (see below).

Components

A number of components implement the basic communications and active message interfaces. Let's take a look at some of the components in the `/tos/system` directory. You should be familiar with these components because your code needs to specify both the *interfaces* your application *uses* as well as the *components* which *provide* (implement) those interfaces:

- `AMReceiverC (http://www.tinyos.net/tinyos-2.x/tos/system/AMReceiverC.nc)` - Provides the following interfaces: `Receive`, `Packet`, and `AMPacket`.
- `AMSenderC (http://www.tinyos.net/tinyos-2.x/tos/system/AMSenderC.nc)` - Provides `AMSend`, `Packet`, `AMPacket`, and `PacketAcknowledgements` as `Acks`.
- `AMSnooperC (http://www.tinyos.net/tinyos-2.x/tos/system/AMSnooperC.nc)` - Provides `Receive`, `Packet`, and `AMPacket`.
- `AMSnoopingReceiverC (http://www.tinyos.net/tinyos-2.x/tos/system/AMSnoopingReceiverC.nc)` - Provides `Receive`, `Packet`, and `AMPacket`.
- `ActiveMessageAddressC (http://www.tinyos.net/tinyos-2.x/tos/system/ActiveMessageAddressC.nc)` - Provides commands to get and set the node's active message address. This interface is not for general use and changing a node's active message address can break the network stack, so avoid using it unless you know what you are doing.

Naming Wrappers

Since TinyOS supports multiple platforms, each of which might have their own implementation of the radio drivers, an additional, platform-specific, naming wrapper called `ActiveMessageC` is used to bridge these interfaces to their underlying, platform-specific implementations. `ActiveMessageC` provides most of the communication interfaces presented above. Platform-specific versions of `ActiveMessageC`, as well the underlying implementations which may be shared by multiple platforms (e.g. Telos and MicaZ) include:

- `ActiveMessageC` for the eyesIFX (http://www.tinyos.net/tinyos-2.x/tos/platforms/eyesIFX/ActiveMessageC.nc) platform is implemented by `Tda5250ActiveMessageC (http://www.tinyos.net/tinyos-2.x/tos/chips/tda5250/Tda5250ActiveMessageC.nc)` .
- `ActiveMessageC` for the intelmote2 (http://www.tinyos.net/tinyos-2.x/tos/platforms/intelmote2/ActiveMessageC.nc) , micaz (http://www.tinyos.net/tinyos-2.x/tos/platforms/micaz/ActiveMessageC.nc) , telosa (http://www.tinyos.net/tinyos-2.x/tos/platforms/telosa/ActiveMessageC.nc) , and telosb (http://www.tinyos.net/tinyos-2.x/tos/platforms/telosa/ActiveMessageC.nc) are all implemented by `CC2420ActiveMessageC (http://www.tinyos.net/tinyos-2.x/tos/chips/cc2420/CC2420ActiveMessageC.nc)` .
- `ActiveMessageC` for the mica2 (http://www.tinyos.net/tinyos-2.x/tos/platforms/mica2/ActiveMessageC.nc) platform is implemented by `CC1000ActiveMessageC (http://www.tinyos.net/tinyos-2.x/tos/chips/cc1000/CC1000ActiveMessageC.nc)` .

The TinyOS 2.0 Message Buffer

TinyOS 2.0 introduces a new message buffer abstraction called `message_t`. If you are familiar with earlier versions of TinyOS, you need to know that `message_t` replaces `TOS_Msg`. The `message_t` structure is defined in `tos/types/message.h (http://www.tinyos.net/tinyos-2.x/tos/types/message.h)` .

```
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;
```

Note: The header, footer, and metadata fields are all opaque and must not be accessed directly. It is important to access the `message_t` fields only through `Packet`, `AMPacket`, and other such interfaces, as will be demonstrated in this tutorial. The rationale for this approach is that it allows the data (payload) to be kept at a fixed offset, avoiding a copy when a message is passed between two link layers. See Section 3 in TEP 111<ref name="fn1"/> for more details.

Sending a Message over the Radio

We will now create a simple application that increments a counter, displays the counter's three least significant bits on the three LEDs, and sends a message with the counter value over the radio. Our implementation will use a single timer and a counter, in a way similar to the `BlinkSingle` example from lesson 2.

Reimplementing Blink

As a first step, we can reimplement `Blink` using a single timer and counter. Create a new directory in `apps` named `BlinkToRadio`:

```
$ cd tinyos-2.x/apps
$ mkdir BlinkToRadio
```

Inside this directory, create a file `BlinkToRadioC.nc`, which has this code:

```
#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);
  }
}
```

Let's look at a few specific lines in this program. First, notice the C preprocessor `include` directive on the first line. This directive tells the compiler to simply replace the directive with the entire contents of `Timer.h`. The compiler looks for `Timer.h` in the *standard* places. In this case, standard means the TinyOS system directories that are located in `tos` or its subdirectories. It is possible to tell the compiler to look beyond these standard directories by using the `-I` flag in the Makefile, for example, as is common when including contributed libraries located in `contrib` directory tree.

The second line of this program is also an `include` directive, but note that it uses quotes around the filename rather than angle brackets. The quotes tell the preprocessor to look in the current directory before searching through the standard directories for the particular file. In this case, the `BlinkToRadio.h` file is located in the same directory and defines some constants that are used in this program. We will take a look at `BlinkToRadio.h` in just a bit.

Next, the call to `Leds.set` directly sets the three LEDs to the three low-order bits of the counter.

Finally, note the `call Timer0.startPeriodic(TIMER_PERIOD_MILLI)` line in the `Boot.booted` function. The value of `TIMER_PERIOD_MILLI` is defined in the `BlinkToRadio.h` header file:

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H

enum {
  TIMER_PERIOD_MILLI = 250
};

#endif
```

`BlinkToRadio.h` is a pretty standard header file but there are two things to note here. First, notice the use of the `ifndef`, `define`, and `endif` directives. These directives are used to ensure that the definitions in each header file is not included multiple times because the compiler would complain about multiply-defined objects. By convention, the literal used for these directives is an all-caps version of the filename with any periods converted to underscores. The other important thing to note is the use of an `enum` declaration for defining the constant

TIMER_PERIOD_MILLI. Using enum for defining constants is preferred over using define because enum does not indiscriminantly replace every occurence of the defined literal, regardless of where it appears in the source. As a result, enums provide better scoping as well.

BlinkToRadioC.nc provides the *implementation* logic of the program and BlinkToRadio.h defines constants and/or data structures. A third file is needed to *wire* the interfaces that the implementation uses to the actual components which provide these interfaces. The BlinkToRadioAppC.nc provides the needed wiring:

```
#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;
}
```

The BlinkToRadioAppC should look familiar to you since it is essentially a subset of the Blink application/configuration from an earlier lesson.

These three files constitute all of the application code: the only other thing it needs is a Makefile. Create a file named Makefile. For an application as simple as this one, the Makefile is very short:

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

The first line tells the TinyOS make system that the top-level application component is BlinkToRadioAppC. The second line loads in the TinyOS build system, which has all of the rules for building and installing on different platforms.

Defining a Message Structure

Now that Blink has been reimplemented using a single timer and counter, we can now turn our attention to defining a message format to send data over the radio. Our message will send both the node id and the counter value over the radio. Rather than directly writing and reading the payload area of the message_t with this data, we will use a structure to hold them and then use structure assignment to copy the data into the message payload area. Using a structure allows reading and writing the message payload more conveniently when your message has multiple fields or multi-byte fields (like uint16_t or uint32_t) because you can avoid reading and writing bytes from/to the payload using indices and then shifting and adding (e.g. uint16_t x = data[0] << 8 + data[1]). Even for a message with a single field, you should get used to using a structure because if you ever add more fields to your message or move any of the fields around, you will need to manually update all of the payload position indices if you read and write the payload at a byte level. Using structures is straightforward. To define a message structure with a uint16_t node id and a uint16_t counter in the payload, we add the following lines to BlinkToRadio.h, just before the endif directive:

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

If this code doesn't look even vaguely familiar, you should spend a few minutes reading up on C structures. If you are familiar with C structures, this syntax should look familar but the `nx_` prefix on the keywords `struct` and `uint16_t` should stand out. The `nx_` prefix is specific to the nesC language and signifies that the `struct` and `uint16_t` are *external types* <ref name="fn3">**Programming Hint 15:**Always use platform independent types when defining message formats. From Phil Levis' *TinyOS Programming* (http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf) </ref><ref name="fn4">**Programming Hint 16:**If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea. From Phil Levis' *TinyOS Programming* (http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf) </ref>. External types have the same representation on all platforms. The nesC compiler generates code that transparently reorders access to `nx_` data types and eliminates the need to manually address endianness and alignment (extra padding in structs present on some platforms) issues. So what is endianness? Read on...

Different processors represent numbers in different ways in their memory: some processors use a "big endian" representation which means that the most significant byte of a multi-byte (e.g. 16- or 32-bit) number is located at a lower memory address than the least significant byte, while "little endian" stores data in exactly the opposite order. A problem arises when data is serialized and sent over the network because different processors will decode the same set of bytes in different ways, depending on their "endianness." The main difficulty endianness presents is that it requires operations to rearrange byte orders to match the network protocol specification or the processor architecture -- an annoying and error-prone process. The `htons`, `htonl`, `ntohs`, and `ntohl` calls used with the sockets API are an example of platform-specific calls that convert between network and host byte orders, but you have to remember to use them. The nesC programming language takes a different approach to the problem and defines *external types* which allow the programmer to avoid dealing with byte reordering. In particular, the `nx_` prefix on a type (e.g. `nx_uint16_t`) indicates the field is to be serialized in big endian format. In contrast, the `nxle_` prefix signifies that the field is serialized in little endian format.

Sending a Message

Now that we have defined a message type for our application, `BlinkToRadioMsg`, we will next see how to send the message over the radio. Before beginning, let's review the purpose of the application. We want a timer-driven system in which every firing of the timer results in (i) incrementing a counter, (ii) displaying the three lowest bits of the counter on the LEDs, and (iii) transmitting the node's id and counter value over the radio. To implement this program, we follow a number of simple steps, as described in the next paragraph.

First, we need to identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type. Second, we must update the `module` block in the `BlinkToRadioC.nc` by adding `uses` statements for the interfaces we need. Third, we need to declare new variables and add any initialization and start/stop code that is needed by the interfaces and components. Fourth, we must add any calls to the component interfaces we need for our application. Fifth, we need to implement any events specified in the interfaces we plan on using. Sixth, the `implementation` block of the application configuration file, `BlinkToRadioApp.c`, must be updated by adding a `components` statement for each component we use that provides one of the interfaces we chose earlier. Finally, we need to wire the interfaces used by the application to the components which provide those interfaces.

Let's walk through the steps, one-by-one:

1. **Identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type.**
   We will use the `AMSend` interface to send packets as well as the `Packet` and `AMPacket` interfaces to access the `message_t` abstract data type. Although it is possible to wire directly to the `ActiveMessageC` component, we will instead use the `AMSenderC` component. However, we still need to start the radio using the `ActiveMessageC.SplitControl` interface.The reason for using `AMSenderC` is because it provides a virtualized abstraction. Earlier versions of TinyOS did not virtualize access to the radio, so it was possible for two components that were sharing the radio to interfere with each other. It was not at all

uncommon for one component to discover the radio was busy because some other component, unknown to the first component, was accessing the active message layer. Radio virtualization was introduced in TinyOS 2.0 to address this interference and `AMSenderC` was written to provide this virtualization. Every user of `AMSenderC` is provided with a 1-deep queue and the queues of all users are serviced in a fair manner.

2. **Update the `module` block in the `BlinkToRadioC.nc` by adding `uses` statements for the interfaces we need:**

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

Note that `SplitControl` has been renamed to `AMControl` using the `as` keyword. nesC allows interfaces to be renamed in this way for several reasons. First, it often happens that two or more components that are needed in the same module provide the same interface. The `as` keyword allows one or more such names to be changed to distinct names so that they can each be addressed individually. Second, interfaces are sometimes renamed to something more meaningful. In our case, `SplitControl` is a general interface used for starting and stopping components, but the name `AMControl` is a mnemonic to remind us that the particular instance of `SplitControl` is used to control the `ActiveMessageC` component.

3. **Declare any new variables and add any needed initialization code.**
First, we need to declare some new module-scope variables. We need a `message_t` to hold our data for transmission. We also need a flag to keep track of when the radio is busy sending. These declarations need to be added in the `implementation` block of `BlinkToRadioC.nc`:

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

Next, we need to handle the initialization of the radio. The radio needs to be started when the system is booted so we must call `AMControl.start` inside `Boot.booted`. The only complication is that in our current implementation, we start a timer inside `Boot.booted` and we are planning to use this timer to send messages over the radio but the radio can't be used until it has completed starting up. The radio signals that it has completed starting through the `AMControl.startDone` event. To ensure that we do not start using the radio before it is ready, we need to postpone starting the timer until after the radio has completed starting. We can accomplish this by moving the call to start the timer, which is now inside `Boot.booted`, to `AMControl.startDone`, giving us a new `Boot.booted` with the following body:

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

We also need to implement the `AMControl.startDone` and `AMControl.stopDone` event handlers, which have the following bodies:

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

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

If the radio is started successfully, `AMControl.startDone` will be called with the `error_t` parameter set to a value of `SUCCESS`. If the radio starts successfully, then it is appropriate to start the timer. If, however, the radio does not start successfully, then it obviously cannot be used so we try again to start it. This process continues until the radio starts, and ensures that the node software doesn't run until the key components have started successfully. If the radio doesn't start at all, a human operator might notice that the LEDs are not blinking as they are supposed to, and might try to debug the problem.

4. **Add any program logic and calls to the used interfaces we need for our application.**
   Since we want to transmit the node's id and counter value every time the timer fires, we need to add some code to the `Timer0.fired` event handler:

```
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;
    }
  }
}
```

   This code performs several operations. First, it ensures that a message transmission is not in progress by checking the busy flag. Then it gets the packet's payload portion and casts it to a pointer to the previously declared `BlinkToRadioMsg` external type. It can now use this pointer to initialise the packet's fields, and then send the packet by calling `AMSend.send`. The packet is sent to all nodes in radio range by specyfing `AM_BROADCAST_ADDR` as the destination address. Finally, the test against SUCCESS verifies that the AM layer accepted the message for transmission. If so, the busy flag is set to true. For the duration of the send attempt, the packet is owned by the radio, and user code must not access it. Note that we could have avoided using the `Packet` interface, as it's `getPayload` command is repeated within `AMSend`.

5. **Implement any (non-initialization) events specified in the interfaces we plan on using.**
   Looking through the `Packet`, `AMPacket`, and `AMSend` interfaces, we see that there is only one `event` we need to worry about, `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);
```

   This event is signaled after a message transmission attempt. In addition to signaling whether the message was transmitted successfully or not, the event also returns ownership of `msg` from `AMSend` back to the component that originally called the `AMSend.send` command. Therefore `sendDone` handler needs to clear the `busy` flag to indicate that the message buffer can be reused:

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

Note the check to ensure the message buffer that was signaled is the same as the local message buffer. This test is needed because if two components wire to the same `AMSend`, *both* will receive a `sendDone` event after *either* component issues a `send` command. Since a component writer has no way to enforce that her component will not be used in this manner, a defensive style of programming that verifies that the sent message is the same one that is being signaled is required.

6. **Update the `implementation` block of the application configuration file by adding a `components` statement for each component used that provides one of the interfaces chosen earlier.**
The following lines can be added just below the existing `components` declarations in the `implementation` block of `BlinkToRadioAppC.nc`:

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

These statements indicate that two components, `ActiveMessageC` and `AMSenderC`, will provide the needed interfaces. However, note the slight difference in their syntax. `ActiveMessageC` is a singleton component that is defined once for each type of hardware platform. `AMSenderC` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMSenderC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMSenderC`. We can extend the `enum` in the `BlinkToRadio.h` header file to incorporate the value of `AM_BLINKTORADIO`:

```
...
enum {
  AM_BLINKTORADIO = 6,
  TIMER_PERIOD_MILLI = 250
};
...
```

7. **Wire the the interfaces used by the application to the components which provide those interfaces.**
The following lines will wire the used interfaces to the providing components. These lines should be added to the bottom of the `implementation` block of `BlinkToRadioAppC.nc`:

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

Receiving a Message over the Radio

Now that we have an application that is transmitting messages, we can add some code to receive and process the messages. Let's write code that, upon receiving a message, sets the LEDs to the three least significant bits of the counter in the message. To make this application interesting, we will want to remove the line ~~`call Leds.set(counter);`~~ from the `Timer0.fired` event handler. Otherwise, both the timer events and packet receptions will update the LEDs and the resulting effect will be bizarre.

If two motes are programmed with our modified application, then each will display the other mote's counter value. If the motes go out of radio range, then the LEDs will stop changing. You can even investigate link asymmetry by trying to get one mote's LEDs to keep blinking while the other mote's LEDs stop blinking. This

would indicate that the link from the non-blinking mote to blinking mote was available but that the reverse channel was no longer available.

1. **Identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type.**
   We will use the `Receive` interface to receive packets.
2. **Update the module block in the BlinkToRadioC.nc by adding uses statements for the interfaces we need:**

```
module BlinkToRadioC {
  ...
  uses interface Receive;
}
```

3. **Declare any new variables and add any needed initialization code.**
   We will not require any new variables to receive and process messages from the radio.
4. **Add any program logic and calls to the used interfaces we need for our application.**
   Message reception is an event-driven process so we do not need to call any commands on the `Receive`.
5. **Implemement any (non-initialization) events specified in the interfaces we plan on using.**
   We need to implement the `Receive.receive` event handler:

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

   The `receive` event handler performs some simple operations. First, we need to ensure that the length of the message is what is expected. Then, the message payload is cast to a structure pointer of type `BlinkToRadioMsg*` and assigned to a local variable. Then, the counter value in the message is used to set the states of the three LEDs.Note that we can safely manipulate the `counter` variable outside of an atomic section. The reason is that receive event executes in task context rather than interrupt context (events that have the `async` keyword can execute in interrupt context). Since the TinyOS execution model allows only one task to execute at a time, if all accesses to a variable occur in task context, then no race conditions will occur for that variable. Since all accesses to `counter` occur in task context, no critical sections are needed when accessing it.

6. **Update the implementation block of the application configuration file by adding a components statement for each component used that provides one of the interfaces chosen earlier.**
   The following lines can be added just below the existing `components` declarations in the implementation block of `BlinkToRadioAppC.nc`:

```
implementation {
  ...
  components new AMReceiverC(AM_BLINKTORADIO);
  ...
}
```

   This statement means that a new instance of `AMReceiverC` will be created. `AMReceiver` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMReceiverC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMReceiverC` and is chosen to be the same as that used for the `AMSenderC` used earlier, which ensures that the same AM type is being used for both transmissions and receptions. `AM_BLINKTORADIO` is defined in the `BlinkToRadio.h` header file.

7. **Wire the the interfaces used by the application to the components which provide those interfaces.**
   Update the wiring by insert the following line just before the closing brace of the `implementation` block in BlinkToRadioAppC:

```
implementation {
  ...
  App.Receive -> AMReceiverC;
}
```

8. **Test your application!**
   Testing your application is easy. Get two motes. They can be mica2, micaz, telosa, telosb, or tmote. For this exercise, let's assume that the motes are telosb (if not, skip past the motelist part and program the mote using whatever the appropriate programmer parameters are for your hardware). Assuming you are using a telosb, first open a Cygwin or Linux shell and cd to the `apps/tutorials/BlinkToRadio` directory. Then, insert the first telosb mote into an available USB port on the PC and type `motelist` the at the Cygwin or Linux prompt ($). You should see exactly one mote listed. For example:

```
$ motelist
Reference  CommPort   Description
---------- ---------- ----------------------------------------
UCC89MXV   COM17      Telos (Rev B 2004-09-27)
```

   Now, assuming you are in the `apps/tutorials/BlinkToRadio` directory, type `make telosb install,1`. You should see a lot text scroll by that looks something like:

```
$ make telosb install,1
mkdir -p build/telosb
    compiling BlinkToRadioAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d
-Wnesc-all -target=telosb -fnesc-cfile=build/telosb/app.c -board=   BlinkToRadioAppC.nc -lm
    compiled BlinkToRadioAppC to build/telosb/main.exe
            9040 bytes in ROM
             246 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
    writing TOS image
tos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target ihex build/telosb/main.ihex
build/telosb/main.ihex.out-1 TOS_NODE_ID=1 ActiveMessageAddressC$addr=1
    found mote on COM17 (using bsl,auto)
    installing telosb binary using bsl
tos-bsl --telosb -c 16 -r -e -I -p build/telosb/main.ihex.out-1
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
9072 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-1 build/telosb/main.ihex.out-1
```

   Now, remove the first telosb from the USB port, insert the batteries, and set it aside. Insert the second telos into the USB port and once again type `motelist`. You should again see something like:

```
$ motelist
Reference  CommPort   Description
---------- ---------- ----------------------------------------
UC9VN03I   COM14      Telos (Rev B 2004-09-27)
```

   Finally, type `make telosb reinstall,2` and you should once again see something like the following scroll by:

```
$ make telosb reinstall,2
tos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target ihex build/telosb/main.ihex
build/telosb/main.ihex.out-2 TOS_NODE_ID=2 ActiveMessageAddressC$addr=2
    found mote on COM14 (using bsl,auto)
    installing telosb binary using bsl
tos-bsl --telosb -c 13 -r -e -I -p build/telosb/main.ihex.out-2
```

```
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
9072 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-2 build/telosb/main.ihex.out-2
```

At this point, both motes should be blinking their LEDs. If you press the RESET button on either telosb, then the LEDs on the *other* telosb will pause on whatever was being displayed at the moment you pressed RESET. When you release the RESET button, the paused mote will be reset and then resume counting from one.

Conclusions

This lesson has introduced radio communications in TinyOS 2.x.

Related Documentation

<references/>

< **Previous Lesson** | **Top** | **Next Lesson** >
