Getting Started with TinyOS

From the TinyOS Documentation Wiki (www.tinyos.net)

Components and Interfaces

TinyOS code is written in nesC, which is C with some additional language features for components and concurrency. A nesC application consists of one or more *components* assembled, or *wired*, to form an application executable. Components define two scopes: one for their specification which contains the names of their *interfaces*, and a second scope for their implementation. A component *provides* and *uses* interfaces. The provided interfaces are intended to represent the functionality that the component provides to its user in its specification; the used interfaces represent the functionality the component needs to perform its job in its implementation.

Interfaces are bidirectional: they specify a set of *commands*, which are functions to be implemented by the interface's provider, and a set of *events*, which are functions to be implemented by the interface's user. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface.

The set of interfaces which a component provides together with the set of interfaces that a component uses is considered that component's *signature*.

Configurations and Modules

There are two types of components in nesC: *modules* and *configurations*. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components inside.

Blink: An Example Application

Let's look at a concrete example: Blink in the TinyOS tree. This application displays a counter on the three mote LEDs. In actuality, it simply causes the LED0 to to turn on and off at 4Hz, LED1 to turn on and off at 2Hz, and LED2 to turn on and off at 1Hz. The effect is as if the three LEDs were displaying a binary count of zero to seven every two seconds.

Blink is composed of two **components**: a **module**, called "`BlinkC.nc`", and a **configuration**, called "`BlinkAppC.nc`". Remember that all applications require a top-level configuration file, which is typically named after the application itself. In this case `BlinkAppC.nc` is the configuration for the Blink application and the source file that the nesC compiler uses to generate an executable file. `BlinkC.nc`, on the other hand, actually provides the *implementation* of the Blink application. As you might guess, `BlinkAppC.nc` is used to wire the `BlinkC.nc` module to other components that the Blink application requires.

The reason for the distinction between modules and configurations is to allow a system designer to build applications out of existing implementations. For example, a designer could provide a configuration that

simply wires together one or more modules, none of which she actually designed. Likewise, another developer can provide a new set of library modules that can be used in a range of applications.

The BlinkAppC.nc Configuration

The nesC compiler compiles a nesC application when given the file containing the top-level configuration. Let's look at `BlinkAppC.nc`, the configuration for this application first:

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

The first thing to notice is the key word `configuration`, which indicates that this is a configuration file. The first two lines,

```
configuration BlinkAppC {
}
```

simply state that this is a configuration called `BlinkAppC`. Within the empty braces here it is possible to specify `uses` and `provides` clauses, as with a module. This is important to keep in mind: a configuration can use and provide interfaces. Said another way, not all configurations are top-level applications.

The actual configuration is implemented within the pair of curly brackets following the key word `implementation` . The `components` lines specify the set of components that this configuration references. In this case those components are `Main`, `BlinkC`, `LedsC`, and three instances of a timer component called `TimerMilliC` which will be referenced as Timer0, Timer1, and Timer2. This is accomplished via the *as* keyword which is simply an alias.

As we continue reviewing the BlinkAppC application, keep in mind that the BlinkAppC component is not the same as the BlinkC component. Rather, the BlinkAppC component is composed of the BlinkC component along with MainC, LedsC and the three timers.

The remainder of the BlinkAppC configuration consists of connecting interfaces used by components to interfaces provided by others. The `MainC.Boot` and `MainC.SoftwareInit` interfaces are part of TinyOS's boot sequence. Suffice it to say that these wirings enable the LEDs and Timers to be initialized.

The last four lines wire interfaces that the BlinkC component *uses* to interfaces that the TimerMilliC and LedsC components *provide*. To fully understand the semantics of these wirings, it is helpful to look at the BlinkC module's definition and implementation.

The BlinkC.nc Module

```
module BlinkC {
  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
{
  // implementation code omitted
}
```

The first part of the module code states that this is a module called `BlinkC` and declares the interfaces it provides and uses. The `BlinkC` module **uses** three instances of the interface `Timer<TMilli>` using the names Timer0, Timer1 and Timer2 (the `<TMilli>` syntax simply supplies the generic Timer interface with the required timer precision). Lastly, the `BlinkC` module also uses the Leds and Boot interfaces. This means that BlinkC may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces.

After reviewing the interfaces used by the `BlinkC` component, the semantics of the last four lines in `BlinkAppC.nc` should become clearer. The line `BlinkC.Timer0 -> Timer0` wires the three `Timer<TMilli>` interface used by `BlinkC` to the `Timer<TMilli>` interface provided the three `TimerMilliC` component. The `BlinkC.Leds -> LedsC` line wires the `Leds` interface used by the `BlinkC` component to the `Leds` interface provided by the `LedsC` component.

nesC uses arrows to bind interfaces to one another. The right arrow (`A->B`) as "A wires to B". The left side of the arrow (A) is a user of the interface, while the right side of the arrow (B) is the provider. A full wiring is `A.a->B.b`, which means "interface a of component A wires to interface b of component B." Naming the interface is important when a component uses or provides multiple instances of the same interface. For example, BlinkC uses three instances of Timer: Timer0, Timer1 and Timer2. When a component only has one instance of an interface, you can elide the interface name. For example, returning to BlinkAppC:

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

The interface name Leds does not have to be included in LedsC:

```
  BlinkC.Leds -> LedsC; // Same as BlinkC.Leds -> LedsC.Leds
```

Because BlinkC only uses one instance of the Leds interface, this line would also work:

```
  BlinkC -> LedsC.Leds; // Same as BlinkC.Leds -> LedsC.Leds
```

As the TimerMilliC components each provide a single instance of Timer, it does not have to be included in the wirings:

```
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
```

However, as BlinkC has three instances of Timer, eliding the name on the user side would be a compile-time error, as the compiler would not know which instance of Timer was being wired:

```
  BlinkC -> Timer0.Timer;  // Compile error!
```

The direction of a wiring arrow is always from a user to a provider. If the provider is on the left side, you can also use a left arrow:

```
  Timer0 <- BlinkC.Timer0; // Same as BlinkC.Timer0 -> Timer0;
```

For ease of reading, however, most wirings are left-to-right.

Interfaces, Commands, and Events

We learned that if a component uses an interface, it can call the interface's commands and must implement handlers for its events. We also saw that the BlinkC component uses the Timer, Leds, and Boot interfaces. Let's take a look at those interfaces:

```
interface Boot {
  event void booted();
}
```

```
interface Leds {

  /**
   * Turn LED n on, off, or toggle its present state.
   */
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();

  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();

  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();

  /**
   * Get/Set the current LED settings as a bitmask. Each bit corresponds to
   * whether an LED is on; bit 0 is LED 0, bit 1 is LED 1, etc.
   */
  async command uint8_t get();
  async command void set(uint8_t val);

}
```

```
interface Timer
{
  // basic interface
  command void startPeriodic( uint32_t dt );
  command void startOneShot( uint32_t dt );
  command void stop();
  event void fired();

  // extended interface omitted (all commands)
}
```

Looking over the interfaces for `Boot`, `Leds`, and `Timer`, we can see that since `BlinkC` uses those interfaces it must implement handlers for the `Boot.booted()` event, and the `Timer.fired()` event. The `Leds` interface signature does not include any events, so `BlinkC` need not implement any in order to call the Leds commands. Here, again, is `BlinkC`'s implementation of `Boot.booted()`:

```
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

`BlinkC` uses 3 instances of the TimerMilliC component, wired to the interfaces `Timer0`, `Timer1`, and `Timer2`. The `Boot.booted()` event handler starts each instance. The parameter to `startPeriodic()` specifies the period in milliseconds after which the timer will fire (it's millseconds because of the `<TMilli>` in the interface). Because the timer is started using the `startPeriodic()` command, the timer will be reset after firing such that the `fired()` event is triggered every n milliseconds.

Invoking an interface command requires the `call` keyword, and invoking an interface event requires the `signal` keyword. BlinkC does not provide any interfaces, so its code does not have any signal statements: in a later lesson, we'll look at the boot sequence, which signals the Boot.booted() event.

Next, look at the implementation of the `Timer.fired()`:

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

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
```

Because it uses three instances of the Timer interface, `BlinkC` must implement three instances of `Timer.fired()` event. When implementing or invoking an interface function, the function name is always *interface.function*. As BlinkC's three Timer instances are named `Timer0`, `Timer1`, and `Timer2`, it implements the three functions `Timer0.fired`, `Timer1.fired`, and `Timer2.fired`.

TinyOS Execution Model: Tasks

All of the code we've looked at so far is *synchronous*. It runs in a single execution context and does not have any kind of pre-emption. That is, when synchronous (sync) code starts running, it does not relinquish the CPU to other sync code until it completes. This simple mechanism allows the TinyOS scheduler to minimize its RAM consumption and keeps sync code very simple. However, it means that if one piece of sync code runs for a long time, it prevents other sync code from running, which can adversely affect system responsiveness. For example, a long-running piece of code can increase the time it takes for a mote to respond to a packet.

So far, all of the examples we've looked at have been direct function calls. System components, such as the boot sequence or timers, signal events to a component, which takes some action (perhaps calling a command) and returns. In most cases, this programming approach works well. Because sync code is non-preemptive, however, this approach does not work well for large computations. A component needs to be able to split a large computation into smaller parts, which can be executed one at a time. Also, there are times when a component needs to do something, but it's fine to do it a little later. Giving TinyOS the ability to defer the computation until later can let it deal with everything else that's waiting first.

Tasks enable components to perform general-purpose "background" processing in an application. A task is a function which a component tells TinyOS to run later, rather than now.

A task is declared in your implementation module using the syntax

```
task void taskname() { ... }
```

where `taskname()` is whatever symbolic name you want to assign to the task. Tasks must return `void` and may not take any arguments. To dispatch a task for (later) execution, use the syntax

```
post taskname();
```

A component can post a task in a command, an event, or a task. Because they are the root of a call graph, a tasks can safely both call commands and signal events. We will see later that, by convention, commands do not signal events to avoid creating recursive loops across component boundaries (e.g., if command X in component 1 signals event Y in component 2, which itself calls command X in component 1). These loops would be hard for the programmer to detect (as they depend on how the application is wired) and would lead to large stack usage.

The `post` operation places the task on an internal **task queue** which is processed in FIFO order. When a task is executed, it runs to completion before the next task is run. Therefore, and as the above examples showed, a task should not run for long periods of time. Tasks do not preempt each other, but a task can be preempted by a hardware interrupts (which we haven't seen yet). If you need to run a series of long operations, you should dispatch a separate task for each operation, rather than using one big task. The `post` operation returns an `error_t`, whose value is either `SUCCESS` or `FAIL`. A post fails if and only if the task is already pending to run (it has been posted successfully and has not been invoked yet).

Split-Phase Operations

Because nesC interfaces are wired at compile time, callbacks (events) in TinyOS are very efficient. In most C-like languages, callbacks have to be registered at run-time with a function pointer. This can prevent the compiler from being able to optimize code across callback call paths. Since they are wired statically in nesC, the compiler knows exactly what functions are called where and can optimize heavily.

The ability to optimize across component boundaries is very important in TinyOS, because it has no blocking operations. Instead, every long-running operation is **split-phase**. In a blocking system, when a program calls a long-running operation, the call does not return until the operation is complete: the program blocks. In a split-phase system, when a program calls a long-running operation, the call returns immediately, and the called abstraction issues a callback when it completes. This approach is called split-phase because it splits invocation and completion into two separate phases of execution. Here is a simple example of the difference between the two:

Blocking	Split-Phase

`if (send() == SUCCESS) {`
`  sendCount++;`
`}`
	`// start phase`
	`send();`
	
	`//completion phase`
	`void sendDone(error_t err) {`
	`  if (err == SUCCESS) {`
	`  sendCount++;`
	`  }`
	`}`

Split-phase code is often a bit more verbose and complex than sequential code. But it has several advantages. First, split-phase calls do not tie up stack memory while they are executing. Second, they keep the system responsive: there is never a situation when an application needs to take an action but all of its threads are tied up in blocking calls. Third, it tends to reduce stack utilization, as creating large variables on the stack is rarely necessary.

Split-phase interfaces enable a TinyOS component to easily start several operations at once and have them execute in parallel. Also, split-phase operations can save memory. This is because when a program calls a blocking operation, all of the state it has stored on the call stack (e.g., variables declared in functions) have to be saved. As determining the exact size of the stack is difficult, operating systems often choose a very conservative and therefore large size. Of course, if there is data that has to be kept across the call, split-phase operations still need to save it.

The command `Timer.startOneShot` is an example of a split-phase call. The user of the Timer interface calls the command, which returns immediately. Some time later (specified by the argument), the component providing Timer signals `Timer.fired`. In a system with blocking calls, a program might use `sleep()`:

Blocking	Split-phase

`state = WAITING;`
`op1();`
`sleep(500);`
`op2();`
`state = RUNNING`
	`state = WAITING;`
	`op1();`
	`call Timer.startOneShot(500);`
	
	`event void Timer.fired() {`
	`  op2();`
	`  state = RUNNING;`
	`}`

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 message buffer abstraction, called `message_t`, which is implemented as a nesC struct (similar to a C struct). `message_t` is an *abstract data type*, whose members are read and written using accessor and mutator functions [1].

```
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;
```

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` - 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. 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` - 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.

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. AM packets 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` - 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 provides for setting an AM packet's destination and type, and checking whether the destination is the local node.
- `AMSend` - 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.

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

Sending a Message 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. The following defines a message structure with a `uint16_t` node id and a `uint16_t` counter in the payload:

```
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 *network types* [2][3]. Network 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.

We will implement a new application, called `BlinkToRadioC` which periodically broadcast a counter value.Now that we have defined a message type for our application, `BlinkToRadioMsg`, we will next see how to send the message over the radio.

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

1. 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. We need to start the radio using the `ActiveMessageC.SplitControl` interface.

```
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.

2. We need a `message_t` to hold our data for transmission. 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.

3. 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,
NULL));
    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.

4. 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.

5. 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
};
```

6. 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.

1. We will use the `Receive` interface to receive packets.

```
module BlinkToRadioC {
  ...
```

```
  uses interface Receive;
}
```

2. 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.

3. 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.

4. Update the wiring by insert the following line just before the closing brace of the `implementation` block in BlinkToRadioAppC:

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