WSCN 1000 Guide

Version 1.0.0

Table of Contents

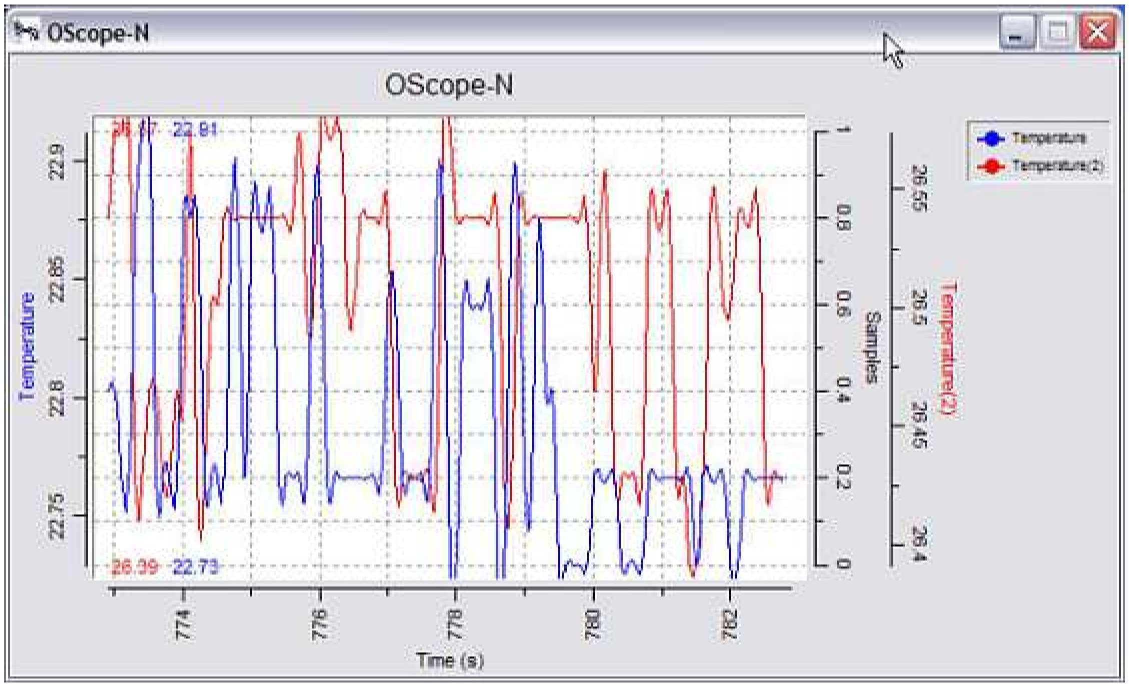
Chapter 1. TinyOS Introduction

The *TinyOS* was developed as a general and embedded operating system for wireless sensor network in the University of California at Berkeley.

The *TinyOS* system, libraries, and applications are written in *nesC*, a programming language for component-based applications. The *nesC* language is primarily intended for embedded systems such as *Wireless Sensor and Control Networks* (**WSCN**). It allows developers to build components and compose them into a concurrent embedded system. The *nesC* has a C-like syntax, but supports the *TinyOS* concurrency model, as well as mechanisms for structuring, naming, and linking together software components into robust network embedded systems.

The *nesC* expresses several important concepts in *TinyOS*. First, *nesC* applications are built out of components with well-defined, bidirectional *interfaces*. Second, *nesC* defines a concurrency model, based on *tasks* and hardware *event* handlers, and detects data races at compile time.

Below is the mote architecture diagram. MCU is the center connected by sensors, RF wireless communication, USB host connection, and/or battery.



Components

A *nesC* application consists of one or more components linked together to form an executable.

Specification

A *component* provides and uses interfaces. These *interfaces* are the only point of access to the component and are *bi-directional*. An interface declares a set of functions called *commands* and events. The interface *provider* and *user* must implement *commands* and *events* respectively. A single component may use or provide multiple interfaces and multiple instances of the same interface.

Implementation

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

NesC uses the filename extension ".nc" for all source files -- interfaces, modules, and configurations.

The reason for the distinction between *modules* and *configurations* is to allow a system designer to quickly "snap together" applications. For example, a designer could provide a configuration that simply wires together one or more modules. Likewise, another developer can provide a new set of "library" modules that can be used in a range of applications.

The convention used in the TinyOS source tree is that *Foo.nc* or *FooC.nc* represents a configuration and *FooM.nc* represents the corresponding module. This convention helps the organization of your nesC code.

Concurrency Model

TinyOS executes only one program consisting of selected system components and custom components needed for a single application. There are two threads of execution: *tasks* and hardware *event handlers*. *Tasks* are functions whose execution is deferred. Once scheduled, they run to completion and do not preempt one another.

Hardware *event* handlers are executed in response to a hardware interrupt and also run to completion, but may preempt the execution of a task or other hardware event handler. Commands and events that are executed as part of a hardware event handler must be declared with the ***async*** keyword.

Because *tasks* and hardware *event* handlers may be preempted by other asynchronous code, *nesC* programs are susceptible to certain race conditions. *Races* are avoided either by accessing shared data exclusively within tasks, or by having all accesses within *atomic* statements. The *nesC* compiler reports potential data races to the programmer at compile-time. It is possible the compiler may report a false positive. In this case a variable can be declared with the ***norace*** keyword that should be used with extreme caution.

Blink Example

Now let's look at a simple example: the test program "*Blink*" found in *apps/Blink* in the TinyOS distribution. This application simply causes the red LED on the mote to turn on and off at 1Hz.

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

The Blink.nc configuration

The *nesC* compiler, ncc, compiles a nesC application when given the file containing the top-level configuration. Typical *TinyOS* applications come with a standard Makefile that allows platform selection and invokes *ncc* with appropriate options on the application's top-level configuration.

Let's look at **Blink.nc**, the configuration for this application first:

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.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 Blink {
  }
```

simply state that this is a configuration called Blink. 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!

The actual configuration is implemented within the pair of curly bracket following key word implementation. The components line specifies the set of components that this configuration references, in this case *Main, BlinkM, SingleTimer, and LedsC.* The remainder of the implementation consists of connecting interfaces used by components to interfaces provided by others.

Main is a component that is executed first in a *TinyOS* application. To be precise, the Main.StdControl.init() command is the first command executed in TinyOS followed by Main.StdControl.start(). Therefore, a TinyOS application must have Main component in its configuration. StdControl is a common interface used to initialize and start TinyOS components. Let us have a look at **StdControl.nc** under tos/interfaces/ directory:

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

We see that StdControl defines three commands, init(), start(), and stop(). init() is called when a component is first initialized, and start() when it is started, that is, actually executed for the first time. stop() is called when the component is stopped, for example, in order to power off the device that it is controlling.

init() can be called multiple times, but will never be called after either start() or stop are called. Specifically, the valid call patterns of StdControl are init*(start | stop)*. All three of these commands have "deep" semantics; calling init() on a component must make it call init() on all of its subcomponents. The following 2 lines in Blink configuration

```
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
```

wire the StdControl interface in Main to the StdControl interface in both BlinkM and SingleTimer. SingleTimer.StdControl.init() and BlinkM.StdControl.init() will be called by Main.StdControl.init(). The same rule applies to the start() and stop() commands.

Concerning used interfaces, it is important to note that subcomponent initialization functions must be explicitly called by the using component. For example, the BlinkM module uses the interface Leds, so Leds.init() is called explicitly in BlinkM.init ().

nesC uses arrows to determine relationships between interfaces. Think of the right arrow (->) as "binds to". The left side of the arrow binds an interface to an implementation on the right side. In other words, the component that *uses an interface is on the left*, and the component *provides the interface is on the right*.

```
BlinkM.Timer -> SingleTimer.Timer;
```

is used to wire the Timer interface used by BlinkM to the Timer interface provided by SingleTimer. BlinkM.Timer on the left side of the arrow is referring to the interface called Timer (tos/interfaces/Timer.nc), while SingleTimer.Timer on the right side of the arrow is referring to the implementation of Timer (tos/lib/SingleTimer.nc). Remember that the arrow always binds interfaces (on the left) to implementations (on the right).

nesC supports multiple implementations of the same interface. The Timer interface is such a example. The SingleTimer component implements a single Timer interface while another component, TimerC, implements multiple timers using timer id as a parameter.

Wirings can also be implicit. For example,

```
BlinkM.Leds -> LedsC;
```

is really shorthand for

```
BlinkM.Leds -> LedsC.Leds;
```

If no interface name is given on the right side of the arrow, the *nesC* compiler by default tries to bind to the same interface as on the left side of the arrow.

The BlinkM.nc module

Now let's look at the module *BlinkM.nc*:

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
```

// to be continued ...

The first part of the code states that this is a module called BlinkM and declares the interfaces it provides and uses. The BlinkM module provides the interface StdControl. This means that BlinkM implements the StdControl interface. As explained above, this is necessary to get the Blink component initialized and started. The BlinkM module also uses two interfaces: Leds and Timer. This means that BlinkM may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces.

The Leds interface defines several commands like redOn(), redOff(), and so forth, which turn the different LEDs (red, green, or yellow) on the mote on and off. Because BlinkM uses the Leds interface, it can invoke any of these commands. Keep in mind, however, that Leds is just an interface: the implementation is specified in the Blink.nc configuration file.

Timer.nc is a little more interesting:

```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}
```

Here we see that Timer interface defines the start() and stop() commands, and the fired() event.

The start() command is used to specify the type of the timer and the interval at which the timer will expire. The unit of the interval argument is millisecond. The valid types are TIMER_REPEAT and TIMER_ONE_SHOT. A one-shot timer ends after the specified interval, while a repeat timer goes on and on until it is stopped by the stop() command.

How does an application know that its timer has expired? The answer is when it receives an event. The Timer interface provides an event:

```
event result_t fired();
```

An ***event*** is a function that the implementation of an interface will signal when a certain event takes place. In this case, the fired() event is signaled when the specified interval has passed. This is an example of a bi-directional interface: an interface not only provides commands that can be called by users of the interface, but also signals events that call handlers in the user. Think of an event as a callback function that the implementation of an interface will invoke. A module that uses an interface must implement the events that this interface uses.

Let's look at the rest of BlinkM.nc to see how this all fits together

```
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000) ;
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

This is simple enough. As we see the BlinkM module implements the *StdControl.init()*, *StdControl.start()*, and *StdControl.stop()* commands, since it provides the StdControl interface. It also implements the Timer.fired() event, which is necessary since BlinkM must implement any event from an interface it uses.

The init() command in the implemented StdControl interface simply initializes the Leds subcomponent with the call to Leds.init(). The start() command invokes Timer.start() to create a repeat timer that expires every 1000 ms. stop() terminates the timer. Each time Timer.fired() event is triggered, the Leds.redToggle() toggles the red LED.

You can view a graphical representation of the component relationships within an application. TinyOS source files include metadata within comment blocks that ncc, the nesC compiler, uses to automatically generate html-formatted documentation. To generate the documentation, type

```
% make <platform> docs
```

from the application directory. The resulting documentation is located in docs/nesdoc/<platform>.docs/nesdoc/<platform>/index.html is the main index to all documented applications.

Compiling the Blink application

TinyOS supports multiple platforms. Each platform has its own directory in the tos/platform directory. In this tutorial, we will use the **telosb** platform as an example. Compiling the Blink application for the telosb mote is as simple as typing

```
% make telosb
```

in the apps/Blink directory.

nesC itself is invoked using the ncc command which is based on gcc. For example, you can type

```
% ncc -o main.exe -target=telosb Blink.nc
```

to compile the Blink application (from the Blink.nc top-level configuration) to main.exe, an executable file for the WSCN 1000 mote. Before you can upload the code to the mote, you use

```
msp430-objcopy --output-target=srec main.exe main.srec
```

to produce main.srec, which essentially represents the binary main.exe file in a text format that can be used for programming the mote. You then use bsl loader tool to actually upload the code to the mote. In general you will never need to invoke ncc or msp430-objcopy by hand, the Makefile does all this for you, but it's nice to see that all you need to compile a nesC application is to run ncc on the top-level configuration file for your application. ncc takes care of locating and compiling all of the different components required by your application, linking them together, and ensuring that all of the component wiring matches up.

Programming the mote and running Blink

Now that we've compiled the application it's time to program the mote and run it. This example will use the Telos platform and the USB based development board (**WSCN 1000 sinker**). To download your program onto the mote, you can insert the mote board (or mote and sensor stack) into the USB port. You can either supply a 3 volt supply to the connector on the programming board or power the node directly. If you are using batteries to power the mote, be sure the mote is switched on (the power switch should be towards the connector).

If the installation is successful you should see something like the following:

```
% make telosb install
mkdir -p build/telosb
    compiling Blink to a telosb binary
ncc -o build/telosb/main.exe -O -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d
      -Wnesc-all -target=telosb -fnesc-cfile=build/telosb/app.c
      -board= -DIDENT_PROGRAM_NAME=\"Blink\" -DIDENT_USER_ID=\"Charlie
\"
      -DIDENT_HOSTNAME=\"neatware-work\" -DIDENT_USER_HASH=0x20f3d905L
      -DIDENT_UNIX_TIME=0x440d1e88L -DIDENT_UID_HASH=0xdf695919L
      -mdisable-hwmul -I/opt/tinyos-1.x/tos/lib/CC2420Radio Blink.nc -lm
    compiled Blink to build/telosb/main.exe
            2606 bytes in ROM
              40 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe
build/telosb/main.ihex
    writing TOS image
cp build/telosb/main.ihex build/telosb/main.ihex.out
    found mote on COM6 (using bsl,auto)
    installing telosb binary using bsl
msp430-bsl --telosb -c 5 -r -e -I -p build/telosb/main.ihex.out
MSP430 Bootstrap Loader Version: 1.39-telos-7
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
2638 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out build/telosb/main.ihex.out
```

You can now test the program, the red LED should light up every second.

Typing

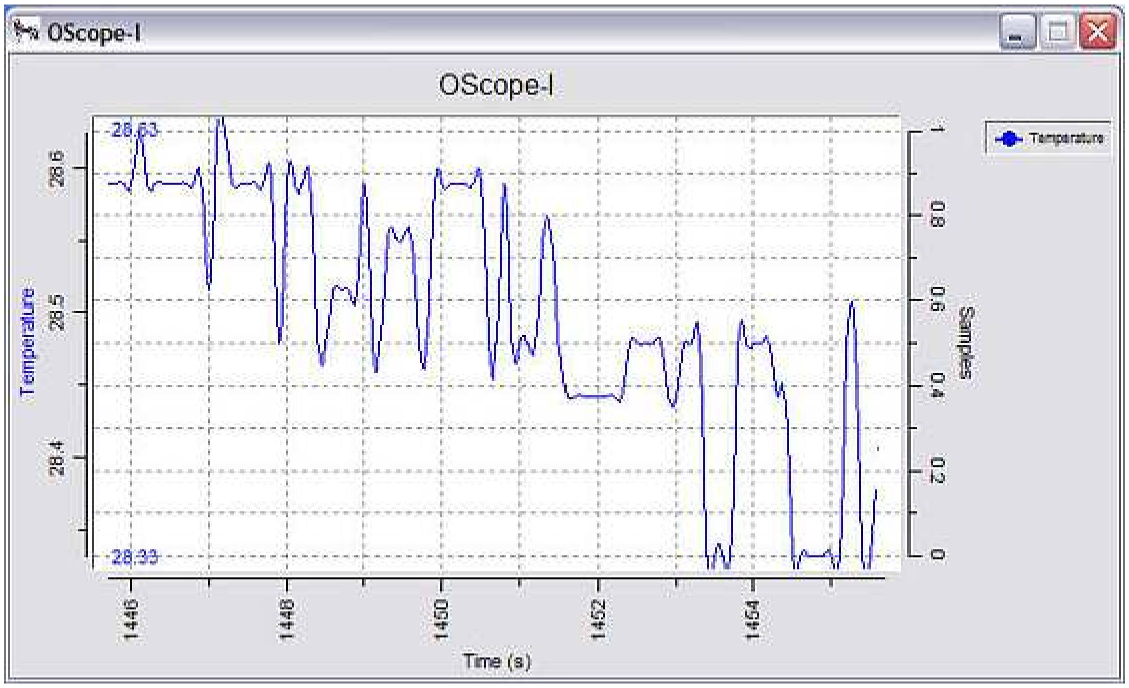
```
% make clean
```

in the Blink directory will clean up the compiled binary files.

If you are still having errors, then you need to check your *TinyOS* installation and hardware.

Chapter 2. Data Acquisition from Sensors

This chapter demonstrates a simple sensor application that takes the low 3 bits of the light intensity (from the onboard photo sensor on WSCN 1000) and displays those readings on the LEDs. This is based on the Sense application, found in apps/Sense.



The SenseM.nc component

Let's first look at the top portion of SenseM.nc:

```
module SenseM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface ADC;
    interface StdControl as ADCControl;
    interface Leds;
  }
}
// Implementation not shown ...
```

Like BlinkM, this component provides the StdControl interface and uses Timer and Leds. It also uses two more interfaces: *ADC*, which is used to access data from the analogue-to-digital converter, and StdControl, which is used to initialize the ADC component.

This application uses a new component, TimerC, in place of SingleTimer. The reason is that TimerC allows multiple instances of timers, while SingleTimer only provides a single one that only one component can use. We'll discuss more on Timer a bit later.

Note the line

```
interface StdControl as ADCControl;
```

This is saying that this component uses the StdControl interface but gives the **interface instance** the name ADCControl. In this way a component can require multiple instances of the same interface, but give them different names. For example, a component might need to use the StdControl interface to control both the ADC and the sounder components. In this case we might say:

```
interface StdControl as ADCControl;
interface StdControl as SounderControl;
```

The configuration that uses this module will be responsible for wiring each interface instance to some actual implementation.

One related note: if you don't provide an interface instance name then the instance is named the same as the interface.

That is, the line

`interface ADC;` is just shorthand for `interface ADC as ADC;`

Take a look at the StdControl and ADC interfaces (both in tos/interfaces). You'll see that StdControl is used to initialize and power a component (usually a piece of physical hardware) and ADC is used to get data from an ADC channel. ADC signals an event, dataReady(), when data is ready on the channel. Notice the ***async*** keyword is used in the ADC interface. This is declaring the commands and events as asynchronous code, i.e., code which can run in response to a hardware interrupt.

Open the file apps/Sense/SenseM.nc. You'll see the code simply calls ADC.getData() each time Timer.fired() is signaled. Likewise, when ADC.dataReady() is signaled, the internal function display() is invoked, which sets the LEDs to the low-order bits of the ADC value.

Notice the use of the function rcombine() in the implementation of StdControl.init().

```
return rcombine(call ADCControl.init(), call Leds.init());
```

The function rcombine() is a special nesC combining function which returns the logical-and of two commands who result type is ***result_t***.

The Sense.nc configuration

At this point you should be wondering, "how does the Sense application know that the ADC channel should access the light sensor?" This is exactly what the Sense.nc configuration wires up for us.

```
Sense.nc

configuration Sense {
  // this module does not provide any interface
}
implementation
{
  components Main, SenseM, LedsC, TimerC, DemoSensorC as Sensor;
  Main.StdControl -> TimerC;
  Main.StdControl -> DemoSensorC;
  Main.StdControl -> SenseM;

  SenseM.ADC -> Sensor;
  SenseM.ADCControl -> Sensor;
  SenseM.Leds -> LedsC;
  SenseM.Timer -> TimerC.Timer[unique("Timer")];
}
```

Most of this should be familiar from Blink. We're wiring up the Main.StdControl to SenseM.StdControl, as before. The Leds wirings are also similar to Blink, and we'll discuss Timer in a minute. The wiring for the ADC is:

```
SenseM.ADC -> Sensor;
SenseM.ADCControl -> Sensor;
```

All we're doing is wiring up the ADC interface (used by SenseM) to a new component called DemoSensorC. Same goes for the ADCControl interface, which you'll recall is an instance of the StdControl interface used by SenseM.

Remember that

```
SenseM.ADC -> Sensor;
```

is just shorthand for

```
SenseM.ADC -> Sensor.ADC;
```

On the other hand,

```
SenseM.ADControl -> Sensor;
```

is not shorthand for

```
SenseM.ADC -> Sensor.ADCControl;
```

What's going on here? If you look at the DemoSensorC.nc component (found in tos/sensorboards/micasb), you'll see that it provides both the ADC and StdControl interfaces -- it doesn't have an interface called ADCControl. (ADCControl was just the name that we gave to the instance of StdControl in the SenseM component.) Rather, the nesC compiler is smart enough to figure out that because SenseM.ADCControl is an instance of the StdControl interface, that it needs to be wired up to an instance of the StdControl interface provided by DemoSensorC. (If DemoSensorC were to provide two instances of StdControl, this would be an error because the wiring would be ambiguous.) In other words,

```
SenseM.ADControl -> Sensor;
```

is shorthand for

```
SenseM.ADControl -> Sensor.StdControl;
```

Timer and parameterized interfaces

Let's look at the line

```
SenseM.Timer -> TimerC.Timer[unique("Timer")];
```

This is introducing a new bit of syntax, called a parameterized interface.

A parameterized interface allows a component to provide multiple instances of an interface that are parameterized by a runtime or compile-time value. Recall that it's possible for a component to provide multiple instances of an interface and to give them different names, e.g.,

```
  provides {
    interface StdControl as fooControl;
    interface StdControl as barControl;
  }
```

This is just a generalization of the same idea. The TimerC component declares:

```
provides interface Timer[uint8_t id];
```

In other words, it provides 256 different instances of the Timer interface, one for each uint8_t value!

In this case, we want *TinyOS* applications to create and use multiple timers, each of which can be managed independently. For example, an application component might need one timer to fire at a certain rate (say, once per second) to gather sensor readings, while another component might need the a timer to fire at a different rate to manage radio transmission. By wiring the Timer interface in each of these components to a separate instance of the Timer interface provided by TimerC, each component can effectively get its own "private" timer.

When we say TimerC.Timer[someval], we are specifying that BlinkM.Timer should be wired to the instance of the Timer interface specified by the value (someval) in the square brackets. This can be any 8-bit positive number. If we were to specify a particular value here, such as 38 or 42, we might accidentally conflict with the timer being used by another component (if that component used the same value in the brackets). So, we use the compile-time constant function unique(), which generates a unique 8-bit identifier from the string given as an argument. Here,

```
unique( "Timer" )
```

generates a unique 8-bit number from a set corresponding to the string "Timer". Therefore, Every component that uses unique("Timer") is guaranteed to get a different 8-bit value as long as the string used in the argument is the same. Note that if one component uses unique("Timer") and another uses unique("MyTimer"), they might get the same 8-bit value. Therefore it's good practice to use the name of the parameterized interface as an argument to the unique() function. The compile-time constant function

```
uniqueCount( "Timer" )
```

will count the number of uses of unique("Timer").

Running the Sense application

As before, just do a **make telosb install** in the Sense directory to compile and install the application on your mote. A photo sensor is built into the WSCN 1000 and does not require a sensor board in this example.

The operation of the sensor is a bit unusual. The ADC yields a 10-bit digitized sample of the photo sensor. What we would like is for the LEDs to be off when the node is in the light, and on when the node is in the dark. Here, we are looking at the upper three bits of this data coming from the ADC, and inverting the value. Note the line:

```
display( 7 - (( data >> 7 ) & 0x7 ) );
```

in the ADC.dataReady() function of SenseM.

Chapter 3. Tasks for Data Processing

This chapter introduces the *TinyOS* notion of **tasks**, which can be used to perform general-purpose "background" processing in an application. It makes use of the SenseTask application, which is a revision of the Sense application from the previous chapter.

Task creation and scheduling

TinyOS provides a two-level scheduling hierarchy consisting of **tasks** and **hardware event handlers**. Remember that the keyword **async** declares a command or event that can be executed by a hardware event handler. This means it could be executed at any time (preempting other code), so **async** commands and events should do small amounts of work and complete quickly. Additionally, you should pay attention to the possibility of data races on all shared data accessed by an **async** command or event. **Tasks** are used to perform longer processing operations, such as background data processing, and can be preempted by hardware event handler.

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 task can be posted from within a command, an event, or even another task.

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, a task should not spin or block for long periods of time. Tasks do not preempt each other, but a task can be preempted by a hardware event handler. 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 SenseTask Application

To illustrate tasks, we have modified the Sense application from Chapter 2, which is found in apps/SenseTask. The SenseTaskM component maintains a circular data buffer, rdata, that contains recent photo sensor samples; the putdata() function is used to insert a new sample into the buffer. The dataReady() event simply deposits data into the buffer and posts a task, called processData(), for processing.

`SenseTaskM.nc`

ADC data ready event handler

```
  async event result_t ADC.dataReady( uint16_t data ) {
    putdata( data );
    post processData( );
    return SUCCESS;
  }
```

Some time after the **async** event completes (there may be other tasks pending for execution), the processData() task will run. It computes the sum of the recent ADC samples and displays the upper three bits of the sum on the LEDs.

```
SenseTaskM.nc, continued

  task void processData( ) {
    int16_t i, sum=0;

    atomic {
      for ( i=0; i < size; i++ )
        sum += (rdata[i] >> 7);
    }
    display( sum >> log2size );
  }
```

The keyword **atomic** in the task processData() illustrates the use of nesC atomic statements. This means the code section within the atomic curly braces will execute without preemption. In this example, access to the shared buffer data is being protected. Where else should this be used?

Atomic statements delay interrupt handling which makes the system less responsive. To minimize this effect, **atomic** statements in nesC should avoid calling commands or signaling events when possible (the execution time of an external command or event will depend on what the component is wired to).

Chapter 4. RF Communication

This chapter introduces two concepts: hierarchical decomposition of component graphs, and using radio communication. The applications that we will consider are CntToLedsAndRfm and RfmToLeds. CntToLedsAndRfm is a variant of Blink that outputs the current counter value to multiple output interfaces: both the LEDs, and the radio communication stack. RfmToLeds receives data from the radio and displays it on the LEDs. Programming one mote with CntToLedsAndRfm will cause it to transmit its counter value over the radio; programming another with RfmToLeds causes it to display the received counter on its LEDs - your first distributed application!

The CntToRfmAndLeds Application

Look at CntToRfmAndLeds.nc. Note that this application only consists of a configuration; all of the component modules are located in libraries!

CntToLedsAndRfm.nc

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[ unique( "Timer" ) ];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;
}
```

The first thing to note is that a single interface requirement (such as Main.StdControl or Counter.IntOutput) can be fanned out to multiple implementations. Here we wire Main.StdControl to the Counter, IntToLeds, IntToRfm, and TimerCcomponents. (All of these components can be found in the tos/lib/Counters directory.) The names of the various components tell you what they do: Counter receives Timer.fired() events to maintain a counter. IntToLeds and IntToRfm provide the IntOutput interface, that has one command, output(), which is called with a 16-bit value, and one event, outputComplete(), which is called with result_t . IntToLeds displays the lower three bits of its value on the LEDs, and IntToRfm broadcasts the 16-bit value over the radio.

So we're wiring the Counter.Timer interface to TimerC.Timer, and Counter.IntOutput to both IntToLeds and IntToRfm. The nesC compiler generates code so that all invocations of the Counter.IntOutput.output() command will invoke the command in both IntToLeds and IntToRfm. Also note that the wiring arrow can go in either direction: the arrow always points from a used interface to a provided implementation.

IntToRfm: Sending a message

IntToRfm is a simple component that receives an output value (through the IntOutput interface) and broadcasts it over the radio. Radio communication in TinyOS follows the **Active Message (AM)** model, in which each packet on the network specifies a handler ID that will be invoked on recipient nodes. Think of the handler ID as an integer or "port number" that is carried in the header of the message. When a message is received, the receive event associated with that handler ID is signaled. Different motes can associate different receive events with the same handler ID.

In any messaging layer, there are 5 aspects involved in successful communication:

1. Specifying the message data to send;
2. Specifying which node is to receive the message;
3. Determining when the memory associated with the outgoing message can be reused;
4. Buffering the incoming message; and,
5. Processing the message on reception

In Tiny Active Messages, memory management is very constrained as you would expect from a small-scale embedded environment.

Let's look at IntToRfm.nc:

```
configuration IntToRfm
{
  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  components IntToRfmM, GenericComm as Comm;

  IntOutput = IntToRfmM;
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.StdControl -> Comm;
}
```

This component provides the IntOutput and StdControl interfaces. This is the first time that we have seen a configuration provide an interface. In the previous lessons we have always used configurations just to wire other components together; in this case, the IntToRfm configuration is itself a component that another configuration can wire to.

In the implementation section, we see:

```
components IntToRfmM, GenericComm as Comm;
```

The phrase "GenericComm as Comm" is stating that this configuration uses the GenericComm component, but gives it the (local) name Comm. The idea here is that you can easily swap in a different communication module in place of GenericComm, and only need to change this one line to do so; you don't need to change every line that wires to Comm.

We also see some new syntax here in the lines:

```
IntOutput = IntToRfmM;
StdControl = IntToRfmM;
```

The equal sign (=) is used to indicate that the IntOutput interface provided by IntToRfm is "equivalent to" the implementation in IntToRfmM. We can't use the arrow (->) here, because the arrow is used to wire a used interface to a provided implementation. In this case we are "equating" the interfaces provided by IntToRfm with the implementation found in IntToRfmM.

The last two lines of the configuration are:

```
IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
IntToRfmM.StdControl -> Comm;
```

The last line is simple; we're wiring IntToRfmM.StdControl to GenericComm.StdControl. The first line shows us another use of parameterized interfaces, in this case, wiring up the Send interface of IntToRfmM to the SendMsg interface provided by Comm.

The GenericComm component declares:

```
  provides {

    ...
    interface SendMsg[uint8_t id];

    ...
  }
```

In other words, it provides 256 different instances of the SendMsg interface, one for each uint8_t value. This is the way that Active Message handler IDs are wired together. In IntToRfm, we are wiring the SendMsg interface corresponding to the handler ID AM_INTMSG to GenericComm.SendMsg. (AM_INTMSG is a global value defined in tos/lib/Counters/IntMsg.h.) When the SendMsg command is invoked, the handler ID is provided to it, essentially as an extra argument. You can see how this works by looking at tos/system/AMStandard.nc (the implementation module for GenericComm):

```
command result_t SendMsg.send[uint8_t id]( ... ) { ... };
```

Of course, parameterized interfaces aren't strictly necessary here - the same thing could be accomplished if SendMsg.send took the handler ID as an argument. This is just an example of the use of parameterized interfaces in nesC.

IntToRfmM: Implementing Network Communication

Now we know how IntToRfm is wired up, but we don't know how message communication is implemented. Take a look at the IntOutput.output() command in IntToRfmM.nc:
IntToRfmM.nc

```
  bool pending;
  struct TOS_Msg data;

  /* ... */

  command result_t IntOutput.output( uint16_t value ) {
    IntMsg *message = ( IntMsg * )data.data;

    if ( !pending ) {
      pending = TRUE;

      message->val = value;
      atomic {
        message->src = TOS_LOCAL_ADDRESS;
      }

      if ( call Send.send( TOS_BCAST_ADDR, sizeof( IntMsg ), &data ) )
        return SUCCESS;

      pending = FALSE;
    }
    return FAIL;
  }
```

The command is using a message structure called IntMsg, declared in tos/lib/Counters/IntMsg.h. It is a simple struct with val and src fields; the first being the data value and the second being the message's source address. We assign these two fields (using the global constant TOS_LOCAL_ADDRESS for the local source address) and call Send.send() with the destination address (TOS_BCAST_ADDR is the radio broadcast address), the message size, and the message data.

The "raw" message data structure used by SendMsg.send() is struct TOS_Msg, declared in tos/system/AM.h. It contains fields for the destination address, message type (the AM handler ID), length, payload, etc. The maximum payload size is TOSH_DATA_LENGTH and is set to 29 for miza platform and to 28 for telos platform by default; you are welcome to experiment with larger data packets but some nontrivial hacking of the code may be required :-) Here we are encapsulating an IntMsg within the data payload field of the TOS_Msg structure.

The SendMsg.send() command is split-phase; it signals the SendMsg.sendDone() event when the message transmission has completed. If send() succeeds, the message is queued for transmission, and if it fails, the messaging component was unable to accept the message.

TinyOS Active Message buffers follow a strict alternating ownership protocol to avoid expensive memory management, while still allowing concurrent operation. If the message layer accepts the send() command, it owns the send buffer and the requesting component should not modify the buffer until the send is complete (as indicated by the sendDone() event).

IntToRfmM uses a pending flag to keep track of the status of the buffer. If the previous message is still being sent, we cannot modify the buffer, so we drop the output() operation and return FAIL. If the send buffer is available, we can fill in the buffer and send a message.

The GenericComm Network Stack

Recall that IntToRfm's SendMsg interface is wired to GenericComm, a "generic" TinyOS network stack implementation (found in tos/system/GenericComm.nc). If you look at the GenricComm.nc, you'll see that it makes use of a number of low-level interfaces to implement communication: AMStandard to implement Active Message sending and reception, UARTNoCRCPacket to communicate over the mote's serial port, RadioCRCPacket to communicate over the radio, and so forth. You don't need to understand all of the details of these modules but you should be able to follow the GenericComm.nc wiring configuration by now.

If you're really curious, check out AMStandard.nc for some details on how the ActiveMessage layer is built. For example, it implements SendMsg.send() by posting a task to take the message buffer and send it over the serial port (if the destination address is **TOS_UART_ADDR** or the radio radio (if the destination is anything else). You can dig down through the various layers of code until you see the mechanism that actually transmits a byte over the radio or UART.

Receiving Messages with RfmToLeds

The RfmToLeds application is defined by a simple configuration that uses the RfmToInt component to receive a message, and the IntToLeds component to display the received value on the LEDs. Like IntToRfm, the RfmToInt component uses GenericComm to receive messages. Most of RfmToInt.nc should be familiar to you by now, but look at the line:

```
RfmToIntM.ReceiveIntMsg -> GenericComm.ReceiveMsg[AM_INTMSG];
```

This is how we specify that Active Messages received with the AM_INTMSG handler ID should be wired to the RfmToIntM.ReceiveMsg interface. The direction of the arrow might be a little confusing here. The ReceiveMsg interface (found in tos/interfaces/ReceiveMsg.nc)only declares an event: receive(), which is signaled with a pointer to the received message. So RfmToIntM uses the ReceiveMsg interface, although that interface does not have any commands to call -- just an event that can be signaled.

Memory management for incoming messages is inherently dynamic. A message arrives and fills a buffer, and the Active Message layer decodes the handler type and dispatches it. The buffer is handed to the application component (through the ReceiveMsg.receive() event), but, critically, the application component must return a pointer to a buffer upon completion.

For example, looking at RfmToIntM.nc,

```
  /* ... */
  event TOS_MsgPtr ReceiveIntMsg.receive( TOS_MsgPtr m ) {
    IntMsg *message = ( IntMsg * )m->data;
    call IntOutput.output( message->val );

    return m;
  }
```

Note that the last line returns the original message buffer, since the application is done with it. If your component needs to save the message contents for later use, it needs to copy the message to a new buffer, or return a new (free) message buffer for use by the network stack.

Underlying Details

TinyOS messages contain a "group ID" in the header, which allows multiple distinct groups of motes to share the same radio channel. If you have multiple groups of motes in your lab, you should set the group ID to a unique 8-bit value to avoid receiving messages for other groups. The default group ID is **0x7D**. You can set the group ID by defining the preprocessor symbol **DEFAULT_LOCAL_GROUP**.

```
DEFAULT_LOCAL_GROUP = 0x42    # for example...
```

Use the *Makelocal* file to set the group ID for all your applications.

In addition, the message header carries a 16-bit destination node address. Each communicating node within a group is given a unique address assigned at compile time. Two common reserved destination addresses we've introduced thus far are **TOS_BCAST_ADDR** (0xfff) to broadcast to all nodes or **TOS_UART_ADDR** (0x007e) to send to the serial port.

The node address may be any value EXCEPT the two reserved values described above. To specify the local address of your mote, use the following install syntax:

```
make telosb install,<addr>
```

where <addr> is the local node ID that you wish to program into the mote. For example,

```
make telosb install,311
```

compiles the application for a telosb and programs the mote with ID 311.

5. Connect to Hosts

The goal of this chapter is to integrate the sensor network with a PC, allowing us to display sensor readings on the PC as well as to communicate from the PC back to the motes. First, we'll introduce the basic tools used to read sensor network data on a desktop over the serial port. Next we'll demonstrate a Java application that displays sensor readings graphically. Finally, we'll close the communication loop by showing how to send data back to the motes.

The Oscilloscope application

The mote application we use in this chapter is found in apps/Oscilloscope. It consists of a single module that reads data from the photo sensor. For each 10 sensor readings, the module sends a packet to the serial port containing those readings. The mote only sends the packets over the serial port. (To see how the data can be sent over the radio see apps/OscilloscopeRF.)

- Insert WSCN 1000 node into USB port.
- Compile and install the Oscilloscope application on a mote linked to COM3.

  ```
  make telosb install.164 bsl,2
  ```

- When the Oscilloscope application is running, the red LED lights when the sensor reading is over some threshold (set to 0x0300) by default in the code - you might want to change this to a higher value if it never seems to go off in the dark).
- The yellow LED is toggled whenever a packet is sent to the serial port.

The 'listen' tool: displaying raw packet data

After programming your mote with the Oscilloscope code, cd to the tools/java directory, and type

```
% make

% export MOTECOM=serial@serialport:telos
```

The environment variable MOTECOM tells the java Listen tool (and most other tools too) which packets it should listen to. Here serial@serialport:telos says to listen to a mote connected to a serial port, where serialport is the serial port that you have connected the programming board to, and baudrate is the specific baudrate of the mote. For the WSCN 1000 motes, the baud rate is 57600 baud. Use motelist to find COM port. So you could do any of:

```
% export MOTECOM=serial@COM3:telos
```

Set MOTECOM appropriately, then run

```
% java net.tinyos.tools.Listen
```

You should see some output resembling the following:

```
% java net.tinyos.tools.Listen

serial@COM3:19200: resynchronising
1A 00 00 00 00 00 7E 00 0A 7D 03 00 72 01 01 00 37 0B 5A 96 03 97 03 97
03 97 03 97 03 97
1A 00 00 00 00 00 7E 00 0A 7D 03 00 72 01 01 00 37 0B 5A 96 03 97 03 97
03 97 03 97 03 97
1A 00 00 00 00 00 7E 00 0A 7D 03 00 72 01 01 00 37 0B 5A 96 03 97 03 97
```

```
03 97 03 97 03 97
...
```

The program is simply printing the raw data of each packet received from the serial port.

Before continuing, execute

```
% unset MOTECOM
```

to avoid forcing all java applications to use the serial port to get packets.

Now what are you seeing?

The application that you are running is simply printing out the packets that are coming from the mote. Each data packet that comes out of the mote contains several fields of data. Some of these fields are generic Active Message fields, and are defined in tos/types/AM.h. The data payload of the message, which is defined by the application, is defined in apps/Oscilloscope/OscopeMsg.h. The overall message format for the Oscilloscope application is as follows:

Length (1 byte)	1A
FCFHI - Frame Control Flow (1 byte)	00
FCFLO - Frame Control Flow (1 byte)	00
DSN (1 byte)	00
Dest PAN (2 bytes)	00 00
Address (2 bytes)	7E 00
Type (1 byte)	0A
Group ID (1 byte)	7D
Payload (up to 28 bytes)	03 00 72 01 ...

This format is used for IEEE 802.15.4 based sensor nodes of Telos motes. The fields like FCF, DSN, Dest and Group ID are Zigbee MAC layer specific. Note that the data is sent by the mote in little-endian format; so, for example, the two bytes 96 03 represent a single sensor reading with most-significant-byte 0x03 and least-significant-byte 0x96. That is, 0x0396 or 918 decimal.

Here is an excerpt from OscilloscopeM.nc showing the data being written to the packet:

```
OscilloscopeM.nc

async event result_t ADC.dataReady( uint16_t data ) {
    struct OscopeMsg *pack;
    atomic {
      pack = ( struct OscopeMsg * )msg[currentMsg].data;
```

add the new sensor reading to the packet and update the number of bytes in this packet

```
      pack->data[ packetReadingNumber++ ] = data;
      readingNumber++; // increment total number of bytes
      dbg( DBG_USR1, "data_event\n" );
```

if the packet is full, send the packet out

```
      if ( packetReadingNumber == BUFFER_SIZE ) {
        post dataTask();
      }
    }
```

if the value of data is over 0x0300 turn the red light.

```
    if ( data > 0x0300 )
      call Leds.redOn();
    else
      call Leds.redOff();

    return SUCCESS;
  }
```

data task

```
  task void dataTask() {
    struct OscopeMsg *pack;
    atomic {
      pack = ( struct OscopeMsg * )msg[currentMsg].data;
      packetReadingNumber = 0;
      pack->lastSampleNumber = readingNumber;
    }

    pack->channel = 1;
    pack->sourceMoteID = TOS_LOCAL_ADDRESS;
```

Try to send the packet. Note that this will return failure immediately if the packet could not be queued for transmission.

```
    if ( call DataMsg.send( TOS_UART_ADDR, sizeof( struct OscopeMsg ),
                            &msg[currentMsg] ) )
      {
        atomic {
          currentMsg ^= 0x1; // flip-flop between two message buffers
        }
        call Leds.yellowToggle();
      }
  }
```

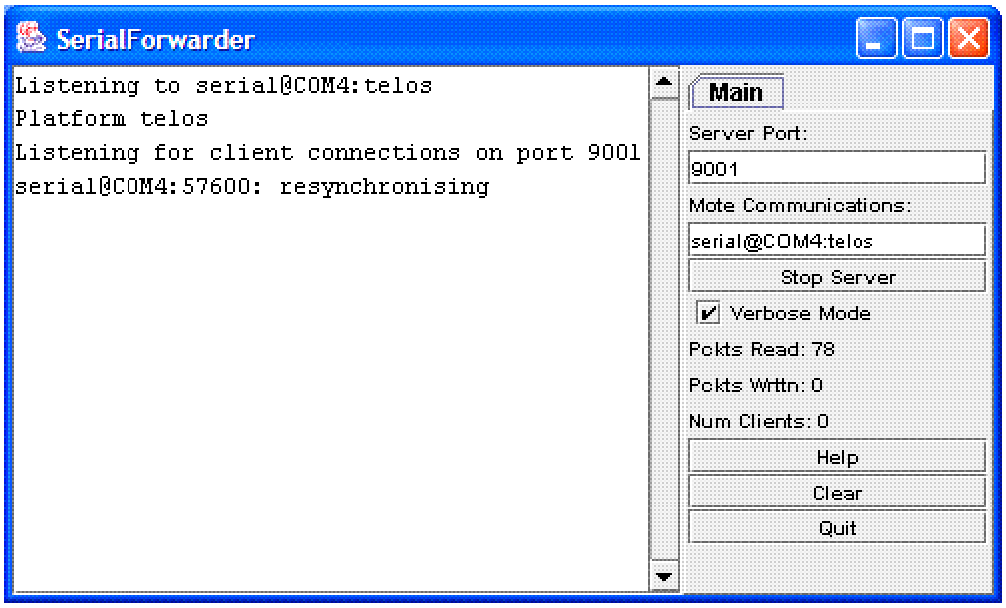
The SerialForwarder Program

The Listen program is the most basic way of communicating with the mote; it directly opens the serial port and just dumps packets to the screen. Obviously it is not easy to visualize the sensor data using this program. What we'd really like is a better way of retrieving and observing data coming from the sensor network.

The SerialForwarder program is used to read packet data from a serial port and forward it over an Internet connection, so that other programs can be written to communicate with the sensor network over the Internet. To run the serial forwarder, cd to tools/java and run the program

```
% java net.tinyos.sf.SerialForwarder -comm serial@COM3:telos &
```

This will open up a GUI window that looks similar to the following:



```
Listening to serial@COM4:telos
Platform telos
Listening for client connections on port 9001
serial@COM4:57600: resynchronising
```

The -comm argument tells SerialForwarder to communicate over serial port *COM4*. The-comm argument specifies where the packets SerialForwarder should forward come from, using the same syntax as the MOTECOM environment variable you saw above (you can run 'java net.tinyos.packet.BuildSource' to get a list of valid sources). Unlike most other programs, SerialForwarder does not pay attention to the MOTECOM environment variable; you must use the -comm argument to specify the packet source (The rationale is that you would typically set MOTECOM to specify a serial forwarder, and that serial forwader should talk to, e.g., a serial port.

SerialForwarder does not display the packet data itself, but rather updates the packet counters in the lower-right hand corner of the window. Once running, the serial forwarder listens for network client connections on a given TCP port (9001 is the default), and simply forwards *TinyOS* messages from the serial port to the network client connection, and vice versa. Note that multiple applications can connect to the serial forwarder at once, and all of them will receive a copy of the messages from the sensor network.

More information is available on SerialForwarder and packet sources is found in the SerialForwarder Documentation

Starting the Oscilloscope GUI

It is now time to graphically display the data coming from the motes. Leaving the serial forwarder running, execute the command

```
% java net.tinyos.oscope.oscilloscope
```

This will pop up a window containing a graphical display of the sensor readings from the mote (If you get an error like "port COM1 busy", you probably forgot to unset the MOTECOM environment variable at the end of the Listen example. Do that now). It connects to the serial forwarder over the network and retrieves packet data, parses the sensor readings from each packet, and draws it on the graph:

The x-axis of the graph is the packet counter number and the y-axis is the sensor light reading. If the mote has been running for a while, its packet counter might be quite large, so the readings might not appear on the graph; just power-cycle the mote to reset its packet counter to 0. If you don't see any light readings on the display, try zooming out on the Y axis (the values might be out of the displayed range) or selecting the "Scrolling" push button (the sample number may be out of the displayed range, "Scrolling" automatically scrolls to the most recent values).

Using MIG to communicate with motes

MIG (Message Interface Generator) is a tool that is used to automatically generate Java classes that correspond to Active Message types used in your mote applications. MIG reads in the nesC struct definitions for message types in your mote application and generates a Java class for each message type that takes care of the gritty details of packing and unpacking fields in the message's byte format. Using MIG saves you from the trouble of parsing message formats in your Java application.

MIG is used in conjunction with the net.tinyos.message package, which provides a number of routines for sending and receiving messages through the MIG-generated message classes. *NCG* (nesC Constant Generator) is a tool to extract constants from nesC files for use with other applications and is typically used in conjunction with MIG.

Let's look at the code from tools/java/net/tinyos/oscope/GraphPanel.java (part of the oscilloscope program) that communicates with the serial forwarder. First, the program connects to the serial forwarder and registers a handler to be invoked when a packet arrives. All of this is done through the net.tinyos.message.MoteIF interface:
GraphPanel.java

```
// OK, connect to the serial forwarder and start receiving data
mote = new MoteIF( PrintStreamMessenger.err, oscilloscope.group_id );
mote.registerListener( new OscopeMsg(), this );
```

MoteIF represents a Java interface for sending and receiving messages to and from motes. The host and port number of the serial forwarder are obtained from the environment variable MOTECOM. You initialize MoteIF with PrintStreamMessenger which indicates where to send status messages (System.err), as well as an (optional) Active Message group ID. This group ID must correspond to the group ID used by your motes.

We register a message listener (this) for the message type OscopeMsg.OscopeMsg is automatically generated by MIG from the nesC definition for struct OscopeMsg, which we saw earlier in OscopeMsg.h. Look at tools/java/net/tinyos/oscope/Makefile for an example of how this class is generated. You will see:

```
OscopeMsg.java:
  $(MIG) -java-classname=$(PACKAGE).OscopeMsg $(APP)/OscopeMsg.h OscopeMsg -o $@
```

Essentially, this generates OscopeMsg.java from the message type struct OscopeMsg in the header file apps/Oscilloscope/OscopeMsg.h.

GraphPanel implements the MessageListener interface, which defines the interface for receiving messages from the serial forwarder. Each time a message of the appropriate type is received, the messageReceived() method is invoked in GraphPanel. It looks like this:
GraphPanel.java

```
public void messageReceived(int dest_addr, Message msg) {
      if (msg instanceof OscopeMsg) {
            oscopeReceived( dest_addr, (OscopeMsg)msg);
      } else {
            throw new RuntimeException(
                  "messageReceived: Got bad message type: "+msg);
      }
  }

public void oscopeReceived(int dest_addr, OscopeMsg omsg) {
      boolean foundPlot = false;
      int moteID, packetNum, channelID, channel = -1, i;

      moteID = omsg.get_sourceMoteID();
      channelID = omsg.get_channel();
      packetNum = omsg.get_lastSampleNumber();

      /* ... */
```

messageReceived() is called with two arguments: the destination address of the packet, and the message itself (net.tinyos.message.Message).Message is just the base class for the application-defined message types; in this case we want to cast it to OscopeMsg which represents the actual message format we are using here.

Once we have the OscopeMsg we can extract fields from it using the handy get_sourceMoteID(), get_lastSampleNumber(), and get_channel() methods. If we look at struct OscopeMsg in OscopeMsg.h we'll see that each of these methods corresponds to a field in the message type: OscopeMsg.h

```
struct OscopeMsg
{
    uint16_t sourceMoteID;
    uint16_t lastSampleNumber;
    uint16_t channel;
    uint16_t data[BUFFER_SIZE];
};
```

Each field in a MIG-generated class has at least eight methods associated with it:

* isSigned_fieldname - Indicates whether or not this field is a signed quantity.
* isArray_fieldname - Indicates whether or not this field is an array.
* get_fieldname - Return the value of this field.
* set_fieldname - Set the value of this field.
* offset_fieldname - Return the offset (in bytes) for this field.
* offsetBits_fieldname - Return the offset (in bits) for this field.
* size_fieldname - Return the length (in bytes) of this field.
* sizeBits_fieldname - Return the length (in bits) of this field.

Note that additional methods are generated for fields that are arrays.

The remainder of messageReceived() pulls the sensor readings out of the message and places them on the graph.

Sending a message through MIG

It is also possible to send a message to the motes using MIG. The Oscilloscope application sends messages of type AM_OSCOPERESETMSG, which causes the mote to reset its packet counter. Looking at clear_data() in GraphPanel.java, we see how messages are sent to the motes:

GraphPanel.java

```
try {
      mote.send(MoteIF.TOS_BCAST_ADDR, new OscopeResetMsg());
} catch (IOException ioe) {
      System.err.println("Warning: Got IOException sending reset
      message: "+ioe);
      ioe.printStackTrace();
}
```

All we need to do is invoke MoteIF.send() with the destination address and the message that we wish to send. Here, MoteIF. **TOS_BCAST_ADDR** is used to represent the broadcast destination address, which is identical to **TOS_BCAST_ADDR** used in the nesC code.

Chapter 6. Wireless Broadcasting

This chapter discusses the use of simple Java tools to inject packets from a PC into the sensor network, as well as the use of a simple multi-hop broadcast protocol.

Injecting packets

We first demonstrate a simple application that receives "**command**" packets from a host through the radio and interprets them to perform a number of actions, such as turning the LEDs on and off. Program one mote with the apps/SimpleCmd application, and another called sink with apps/TOSBase. Next connect the sink to the serial port of the computer suppose to be COM4. The sink with the TOSBase will act as a gateway allowing radio communication between the PC and motes programmed with SimpleCmd.

```
SimpleCmd <--- TOSBase <--- PC Host
     |               |
SimpleCmd <-----------|
```

Now run the SerialForwarder using

```
% java net.tinyos.sf.SerialForwarder -comm serial@COM4 &
```

The Java application tools/java/net/tinyos/tools/BcastInject is used to inject a command packet into the sensor network from the PC. You can run it using

```
% java net.tinyos.tools.BcastInject <command> <arguments>
```

For command you can specify a number of options below:

- `led_on - Turn on the yellow LED`
- `led_off - Turn off the yellow LED`

By running BcastInject with the appropriate arguments you should be able to turn the yellow LED on and off. Looking at the code for SimpleCmdM.nc, you can see that the application simply waits for incoming messages to arrive and interprets one field of the message as the command type. Note that BcastInject and SimpleCmd have to agree on what the command types mean.

Multihop broadcast

In this part of the chapter, we extend the SimpleCmd application and cause it to forward command messages that it receives to other motes in the network. This is accomplished by re-broadcasting the command message once it has been processed. In this way we can form a simple multi-hop routing network, extending the motes' communication range.

This code is in apps/SimpleCmd/BcastM.nc. In addition to processing the command contained within each received message, it re-broadcasts the message if it has not been seen before. Note that in order to install the Bcast application you need to edit apps/SimpleCmd/Makefile and change the line

```
COMPONENT=SimpleCmd
```

to

```
COMPONENT=Bcast
```

BcastM.nc

```
  event TOS_MsgPtr ReceiveCmdMsg.receive(TOS_MsgPtr pmsg){
    TOS_MsgPtr ret = msg;
    result_t retval;
    struct SimpleCmdMsg *data= (struct SimpleCmdMsg *)pmsg->data;
```

Check if this is a new broadcast message

```
    call Leds.greenToggle();
    if (is_new_msg(data)) {
      remember_msg(data);
      retval = call ProcessCmd.execute(pmsg) ;
```

Return a message buffer to the lower levels, and hold on to the current buffer

```
      ret = msg;
      msg = pmsg;
    }
    return ret;
  }
```

To determine whether a given command message has been seen before, the BcastM component tracks the sequence number contained in the message. If the sequence number of the current message is within 127 of the current message, the command is accepted, processed, and forwarded. Otherwise, it's dropped.

Note that the BcastInject program maintains its sequence number across invocations (saving it in a file called tools/java/bcast.properties). If you remove this file the sequence number generated by BcastInject will be reset to 1. You will need to power-cycle the motes if you do this in order for them to interpret subsequent messages as "new".

Accepting messages with a wide range of sequence numbers (rather than just the previous sequence number plus 1) allows for cases where messages are dropped by the radio stack, for example, due to corruption. This is a fairly coarse mechanism, of course, and in a real application you might want to do something more involved, such as packet acknowledgments.

Chapter 7. Data Logger

This lesson will discuss a fairly complete application for remote data logging and collection, called SenseLightToLog. This is an extension to SimpleCmd that accepts two new commands: one that causes the mote to collect sensor readings and write them to the EEPROM, and another that causes the mote to transmit sensor readings from the EEPROM over the radio.

The SenseLightToLog Application

The high-level functionality of SenseLightToLog can be best understood by looking at the apps/SenseLightToLog/SimpleCmdM.nc component. This is an extended version of the original SimpleCmd component from the previous lesson. The cmdInterpret() task now recognizes two additional commands:

* START_SENSING: This command invokes the Sensing interface to collect a specified number of samples at a specified sampling rate, and to store these samples in mote's EEPROM. The LoggerWrite interface is used to write the data to the EEPROM.

* READ_LOG: This command will retrieve a line of data from the EEPROM and broadcast it in a radio packet.

The Sensing interface

We have abstracted the concept of taking a number of sensor readings behind the Sensing interface, which is implemented by the SenseLightToLog component. This interface provides the start() command to initiate a series of sensor readings, and signals the done() event when sensing has completed.

SenseLightToLogM.nc

```
  command result_t Sensing.start(int samples, int interval_ms) {
    nsamples = samples;
    call Timer.start(TIMER_REPEAT, interval_ms);
    return SUCCESS;
  }
```

Timer fired event

```
  event result_t Timer.fired() {
    nsamples--;
    if (nsamples== 0) {
      call Timer.stop();
      signal Sensing.done();
    }
    call Leds.redToggle();
    call ADC.getData();
    return SUCCESS;
  }
```

ADC dataReady

```
  async event result_t ADC.dataReady(uint16_t this_data){
   atomic {
    int p = head;
    bufferPtr[currentBuffer][p] = this_data;
    head = (p+1);
    if (head == maxdata) head = 0; // Wrap around circular buffer
    if (head == 0) {
        post writeTask();
     }
    }
    return SUCCESS;
  }
```

writeTask

```
  task void writeTask() {
    char* ptr;
    atomic {
      ptr = (char*)bufferPtr[currentBuffer];
      currentBuffer ^= 0x01; // Toggle between two buffers
    }
    call LoggerWrite.append(ptr);
  }
```

When start() is invoked, the timer is started to tick at the given interval. When the timer event fires, ADC.getData() is invoked to get a sensor reading. The ADC.dataReady() event causes the sensor reading to be stored in a circular buffer. When the appropriate number of samples have been collected the Sensing.done() event is signaled.

When SimpleCmd receives a READ_LOG command, it initiates an EEPROM read (through LoggerRead). When the read has completed it broadcasts the data in a packet. There are 16 bytes in each log entry, which are displayed when you run BcastInject tool (as described below).

Notice the atomic statements in ADC.dataReady() **async** event. They protect access to the shared variables head, bufferPtr, and currentBuffer. Why do you think the command LoggerWrite.append() is called from a task and not from the ADC.dataReady() async event ? That is because LoggerWrite.append() is not async, so it may not safely preempt other code and should not be called from an async event.

Logger component, interfaces, usage, and limitations

The WSCN 1000 mote has an on-board, 1MByte flash EEPROM. The EEPROM serves as a persistent storage device for the mote, and is indispensable for many applications involving data collection, such as sensor data and debugging traces. The EEPROMRead and EEPROMWrite interfaces provide a clean abstraction to this hardware. The EEPROM may be read and written in 256-byte blocks, called pages. Read and write to the EEPROM are split-phase operations: one must first initiate a read or write operation and wait for the corresponding done event before performing another operation.

STM25P80 (1MB) used in the Telosb platform are divided into 16 sectors; each sector has 256 pages; and each page includes 256 bytes. So a sector has 64KB. You can earse a sector or entire flash. Your write operations on the STM25P Flash can only flip bits from 1 to 0. If you want to flip them back to 1, you need to perform an erase that operates on 64KB sectors.

To simplify access to the EEPROM even further, the Logger component is provided (see tos/system/LoggerM.nc). Logger maintains an internal pointer to the next EEPROM line to be read or written, treating the EEPROM as a circular buffer that may be accessed sequentially. The logger does not read or write data at the beginning of the EEPROM, which is set aside to store persistent data for the mote. For example, when using network reprogramming of motes, this region stores the TOS_LOCAL_ADDRESS of the mote.

The LoggerRead and LoggerWrite interfaces are used for reading and writing, respectively. LoggerRead provides the commands:

* readNext(buffer) - Read the next line from the log
* read(line, buffer) - Read an arbitrary line from the log
* resetPointer() - Set the current line pointer to the beginning of the log
* setPointer(line) - Set the current line pointer to the given line

Likewise, LoggerWrite provides the following commands:

* append(buffer) - Append data to the log
* write(line, buffer) - Write data to the log at the given line
* resetPointer() - Set the current line pointer to the beginning of the log
* setPointer(line) - Set the current line pointer to the given line

Logging performance

The Logger component does not offer very high performance. Please see the apps/HighFrequencySampling application if you are interested in high-frequency sampling (up to around 5kHz), using the ByteEEPROM component. This lesson will eventually be updated to reflect this new component.

Data collection using SenseLightToLog

Program one mote with SenseLightToLog and another with TOSBase, as before. You may want to attach a sensor board to your mote to get meaningful photo readings.

The first step is to instruct the mote to take a number of sensor readings. Type

```
% export MOTECOM=serial@COM3:19200
```

Run

```
% java net.tinyos.tools.BcastInject start_sensing <num_samples>
<interval>
```

where num_samples is the number of samples to take (say, 8 or 16), and interval is the time in milliseconds between samples (say, 100). For example:

```
% java net.tinyos.tools.BcastInject start_sensing 16 100
```

You should see the mote's red LED blink while samples are being taken.

To get the log data back from the mote, use:

```
% java net.tinyos.tools.BcastInject read_log <mote_address>
```

where mote_address is the address of the mote to read log data from. For example:

```
% java net.tinyos.tools.BcastInject read_log 2

  Sending payload: 65 6 0 0 0 2 0 0 0 0 0
  serial@COM3:19200: resynchronizing
  Waiting for response to read_log...
  Received log message: Message <LogMsg> [sourceaddr=0x2]
  Log values: 48 1 38 1 33 1 32 1 32 1 33 1 34 1 34 1
```

The program will wait 10 seconds for a response to the read_log command; if you don't see a response, try again. If you don't get a reply, then possibly the mote didn't get the command (the green LED will toggle for each command received), or you didn't specify the right mote_address. Each subsequent read_log command you send will read the next log entry from the mote; if you want to reset the read pointer, just power-cycle the mote. Remember that the EEPROM data is persistent across power-cycles, but the current read pointer is kept in volatile memory!

Chapter 8. WSCN Security

As more systems become to use robots and sensors in industry, enterprises are going to integrate wireless MCU-based devices into their security frameworks. Financial and privacy issues also require security for home applications.

Chapter 9. Network Programming

node track
distance
Sink to host

Surge is an example application that uses MultiHop ad-hoc routing. It is designed to be used in conjunction with the Surge java tool. Each Surge node takes temperature readings and forwards them to a base station. The node can also respond to broadcast commands from the base.

The class net.tinyos.surge.MainClass processes sensor data from Surge programmed nodes via a TOSBase station. The java applet snoops the multihop headers to provide a graphical view of the logical network topology. It also permits variation of the sample rates and sending pre-defined commands to the surge nodes.

In tinyos-1.x/tools/java/net/tinyos/surge, you must recompile the java classes:

```
cd $TOSDIR/../tools/java/net/tinyos/surge
make clean
SURGE_PLATFORM=telos make
```

This creates classes from Surge.h and SurgeCmd.h in contrib/ucb/apps/Surge and Multihop.h from contrib/ucb/tos/lib/MultiHopLQI.

Now you can run SerialForwarder and Surge with

```
java net.tinyos.sf.SerialForwarder -comm serial@COM3:telos &
java net.tinyos.surge.MainClass 0x7D
```

Suppose TOSBase has been installed on the COM3 node.

Chapter 10. Database

Notes: This article is the revised and expanded document based on the TinyOS Tutorial.

Appendix A

WSCN 1000 Software Installation

1. Install TinyOS 1.1.x in CD and select a directory without space,
2. Insert a WSCN 1000 sink in a USB port.
3. Install USB serial port driver from CD.
4. Double click WSCN icon on desktop to launch Cygwin.
5. Run motelist to check the motes on serial ports.
6. cd /opt/tinyos-1.x/apps/Blink to start testing applications.
7. Type make telosb install bsl,n where n represents COM (n+1).
8. You are going to see LEDs blinking.

Appendix B

WSCN 1000 System Verification

Test TinyOS 1.x and Samples

1. Insert 2 WSCN 1000 motes in USB ports.
2. Test USB

   run motelist to find nodes in the serial ports.

   ```
   motelist
   ```

   ```
    Reference     Comm Port     Description
    ----------------------------------------------------------------
    FTOJ8HGI      COM6          USB <--> Serial Cable
   ```

3. Test MCU

   run Blink in the apps/Blink directory to watch LEDs blinking

   ```
   make telosb install
   ```

4. Test RF

   Install one mote with CntToLedsAndRfm to send packets to Leds and RF

   Install another mote with RfmToLeds that receives packets from RF

5. Test Sensor

   Install SenseToLeds that shows sensor value to leds

6. Test Java Listen

   Listen data on USB serial port by setting MOTECOM

   ```
   export MOTECOM=serial@COM3:telos
   ```

   run

   ```
   java net.tinyos.tools.Listen
   ```

   to unset MOTECOM

   ```
   unset MOTECOM
   ```

7. Test Java SerialForwarder

   Monitor data operation

   ```
   java net.tinyos.sf.SerialForwarder -comm serial@COM3:telos &
   ```

8. Test Java Oscilloscope

   To display graphs of sensor, run

```
make telosb install
```

in apps/Oscilloscope. Below command will launch the java viewer.

```
java net.tinyos.oscope.oscilloscope
```

9. Test UART Serial Port

ReverseUART is a echo-like program to check mote UART and Java comm tools.

In /opt/tinyos-1.x/apps/ReverseUART

```
make telosb install
```

to compile and install ReverseUART on mote.

```
javac testReverseUART.java
```

to generate testReverseUART.class executable.

run SerialForwarder to tap on UART

```
java net.tinyos.sf.SerialForwarder -comm serial@COM3:telos &
```

execute below command

```
java testReverseUART "hello" "world"
```

to dispaly reverse string output as

```
Send> hello
Receive> olleh
Send> world
Receive> dlrow
... done.
```