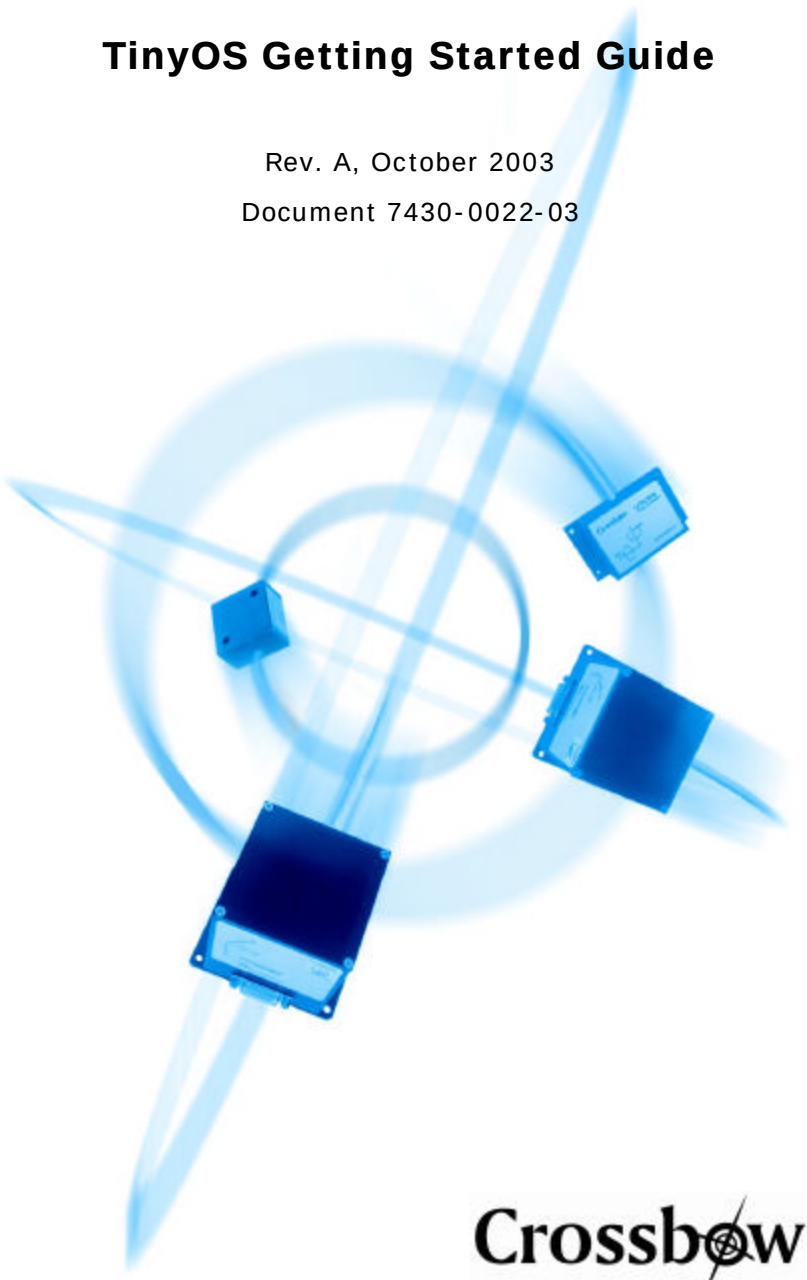
TinyOS Getting Started Guide

Rev. A, October 2003

Document 7430-0022-03



Crossbow

Crossbow Technology, Inc., 41 Daggett Dr., San Jose, CA 95134
Tel: 408-965-3300, Fax: 408-324-4840
email: info@xbow.com, website: www.xbow.com

©2002-2003 Crossbow Technology, Inc. All rights reserved.
Information in this document is subject to change without notice.

Crossbow and SoftSensor are registered trademarks and DMU is a trademark of Crossbow Technology, Inc. Other product and trade names are trademarks or registered trademarks of their respective holders.

1 Introduction

This guide walks you through the installation, verification, compilation and running the TinyOS 1.x application on Windows based PC. There are several key steps to getting up and running with TinyOS.

1.1 Installing TinyOS development tools

> NOTE The installation instructions for TinyOS 1.1.0 are found in "Readme.htm" found on the CD. If you have a previous version of TinyOS on your system, you must uninstall it. Instructions for this are found in the "Uninstalling TinyOS.htm" on the CD. **This must be completed before proceeding any further**.
>
> You must install with Administrator privileges. If you don't, the setup will eventually abort but it could leave unwanted files and program registries behind.

The TinyOS 1.1.0 InstallShield Wizard setup offers the following software packages:

- TinyOS
- TinyOS Tools
- NesC
- Cygwin
- Support Tools
- Java 1.4 JDK & Java COMM 2.0
- Graphviz
- AVR Tools
  - avr-binutils
  - avr-libc
  - avr-gcc
  - avarice
  - avr-insight

Choose between "COMPLETE" and "CUSTOM" install. A custom install allows the user to choose the features that are installed. A "COMPLETE" install includes all of the above features (recommended). The user chooses the destination directory. Any chosen features are installed under that destination directory. **Hereafter the destination directory is referred to as <install dir>.**

JDK. If the user choose to install the JDK module, a dialog asking if the user has read Sun's terms and conditions appears. If the user selects 'No', then setup is ended. If the user selects 'Yes,' then the setup continues. Verify the

install directory and setup type; click on “Next” proceed or “Back” to make changes.

Cygwin and Necessary RPMs. If the user chooses to install cygwin and any necessary RPMs by clicking on the “Continue” button, the 1.1.0 cygwin package tree is copied over to a directory called `<install dir>/cygwin-installation` files. `setup.exe` is called to perform automatic installation of those files in `<install dir>/cygwin`. This is the most lengthy step. On some PCs it takes at least 30 minutes. You are done after this step completes!

TinyOS. All the TinyOS `apps`, `contrib`, `doc`, `tools`, and `tos` folders are located under `<install dir>/cygwin/opt/tinyos-1.x`. In addition the “`Makefile`” is in this folder. The environment variables for `TOSROOT` is set to `<install dir>/tinyos-1.x`. The TinyOS Tutorial is located under `<install dir>/cygwin/opt/tinyos-1.x/doc/tutorial`

Setting Aliases

Once you have successfully installed TinyOS, it is recommended that you setup aliases to commonly used commands and accessed directories. Aliases are to be edited in the `profile` file which is located in `<install dir>/tinyos/cygwin/etc`.

- These aliases are useful for quickly changing the TinyOS-1.x and java tools directories.

```
alias cdtinyos=“cd c:/<install dir>/tinyos/cygwin/opt/tinyos-1.x”

alias cdjava=“cd c:/<install dir>/tinyos/cygwin/opt/tinyos-1.x/tools/java”
```

◀ NOTE If the `<install dir>` is the folder “Program Files,” then you must enter in the text “`Program\ Files`” to correctly handle the space between the two words.

- This is a useful alias for users who program with the MIB510.

```
alias mib510=“MIB510=/dev/ttyS<n>”
```

where <*n*> is the serial port number (0 for COM1, 1 for COM2, etc.) where the MIB510 is attached.

These and other alias can be setup to make changing directories and other commands easier. To make your own use the format as shown in the examples above.

1.2 Platforms

The new release of TinyOS supports the MICA2 and MICA2DOT platforms using a new frequency tunable FM radio with improved range. Application code is built for these platforms by invoking (in a cygwin window):

```
make mica2
make mica2dot
make mica
```

1.3 Device Addressing

The programming tools also include a method of programming unique node addresses without having to edit the TinyOS source code directly. To set the node address during program load, use the following install syntax:

```
make (re)install.<addr> <platform>
```

where `<addr>` is the desired device address and `<platform>` is the target platform. Do not use the reserved values `TOS_BCAST_ADDR (0xFFFF)` or `TOS_UART_ADDR (0x007E)`.

`install` - compiles the application for the target platform, set the address and programs the device (mote).

`Reinstall` - sets the address and programs the (mote) ONLY and does not recompile. This option is significantly faster.

1.4 Programming Boards

The TinyOS development environment supports a variety of programming tools. The supported programmers include:

- The MIB500 (Crossbow's) parallel port programmer board with serial output.
- The MIB510 (Crossbow's) serial port programming board.
- The Atmel AVRISP
- The Ethernet PRogramming Board (EPRB)

The standard programming software used in TinyOS is the µ In-System Programmer or `uisp`. This program, which comes as a part of the TinyOS release, takes various arguments according to the programmer hardware and the particular programming action desired (erase, verify, program, etc.). To simplify using this tool, the TinyOS environment invokes uisp for you with

the correct arguments whenever you issue an "`install`" or "`reinstall`". You only need specify the type of device you are using and how to communicate with it. This is done using environment variables.

MIB500/Parallel Port Programmers

This is the default programmer device. No additional command line parameters need to be specified when using this programmer.

MIB510/Serial Port Programmers

Define: `MIB510=<dev>` where `<dev>` is the name of the serial port where the device is attached (e.g. /dev/ttyS0).

example:

```
bash% MIB510=/dev/ttyS0 make install mica
```

where S0 is for COM1, S1 for COM2, etc.

◀ NOTE If your computer does not have a DB9 serial port and are using a USB to DB9 serial port converter, you must know what port (COM) number your computer has assigned to the USB port. Use that COM port number when doing the above command. However, there are cases where your computer will issue a COM port number but is not what cygwin will communicate through. That is, by trial and error you will have to try different numbers for `ttyS#`.

AVRISP

Define: `AVRISP=<dev>` where `<dev>` is the name of the serial port where the device is attached (i.e. /dev/ttyS0).

example:

```
bash% AVRISP=/dev/ttyS0 make install mica
```

EPRB

Define: `EPRB=<host>` where `<host>` is the DNS name or IP address of the EPRB device.

example:

```
bash% EPRB=123.45.67.89 make install mica
```

1.5 Radio Frequency

The new FM radios support multiple frequencies. Units are delivered either at base frequencies of 315 MHz, 433 MHz, or 916 MHz. Within each frequency band multiple channels can be programmed. All of the

coefficients for radio tuning are contained in the TinyOS file `/tos/platform/mica2/CC1000Const.h`

Users must compile in the correct base radio frequency otherwise radio communication will fail. The correct frequency is selected by modifying the line 213 in the CC1000Const.h file:

```
[212]  #ifndef CC1K_DEF_PRESET
[213]  #define CC1K_DEF_PRESET (CC1K_<freq>_MHZ)
[214]  #endif
```

Change the <*freq*> to one of the values listed in the table below that matches your motes frequency

For motes running at…	…use these values for <*freq*>
315 MHz	315_778
433 MHz (choose one)	433_002
	434_845
915 MHz (choose one)	914_077
	915_998

These numbers correspond to the first six digits of the radio's frequency. For example, 433_002 is for a mote with radio set at 433,002,000 Hz.

1.6 Sensor Boards

Multiple sensor boards are supported in this release. These sensor boards are invoked by modifying the "SENSORBOARD=" statement in the local `Makefile` (in the application directory). Remember to set the `SENSORBOARD` option to either `micasb`, `micawb`, or `basicsb` depending on the type of sensor board you have. For example, to use the MICA2DOT weatherboard sensor, the `Makefile` in the

`/apps/MicaWBVerify/TestHumidity` directory contains:

```
COMPONENT=TestHumidity

SENSORBOARD=micawb
include ../../Makerules
```

1.7 Setting the Group ID and Node ID for the Mote Network

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 environment, 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` (hex). You can set the group ID by

defining the preprocessor symbol `DEFAULT_LOCAL_GROUP` in `Makerules` file found under `/apps`.

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

In addition, the message header carries the destination node number, which is a 16-bit value.

Setting the node ID will become more clear in Section 5 and is included here for reference. The node ID or local address of your mote is done when you download the application into the mote by using the command

```
make mica install.<n>
```

where <*n*> is the local node ID in decimal that you wish to program into the mote. For example,

```
make mica install.38
```

programs the mote with ID 38.

◀ NOTE The latest TinyOS code is available on Sourceforge (http://sourceforge.net/projects/tinyos/). Source code should be updated as new releases and bug fixes are checked into Sourceforge.

2 System and Hardware Verification

When working with embedded devices, it is very difficult to debug applications. Because of this, you want to make sure that the tools you are using are working properly and that the hardware is functioning correctly. This will save you countless hours of searching for bugs in your application when the real problem is in the tools. This section will show you how to check your system and the hardware.

2.1 TinyOS PC Tools Verification:

A TinyOS development environment requires the use of the avr gcc compiler, perl, flex, cygwin (if you use windows operation system), and the JDK 1.4.x or above. First, we will check that the tools have been installed correctly and that the environment variables are set. The "`toscheck`" is a script that will perform these functions.

- Run the cygwin application by double-clicking the icon that can be found on your desktop.
- Change into the `/tools/scripts` directory and type "`toscheck`".

The last line of the output should be "`toscheck completed without error`". If any errors are reported, make sure to fix the problem.

2.2 Mote Hardware Verification:

The mote hardware functionality can be tested in two different ways:

- `MicaHWVerify` application contained in the TinyOS distribution.
- `Mote-Test` GUI provided by Crossbow.

NOTE Users are strongly advised to use a Crossbow MIB500/MIB510 interface board with an external wall mounted power supply (5-7 VDC). If using batteries, check the battery voltage. If the battery voltage is less than 3.0 V the flash memory may not reprogrammed correctly. This can also cause the ATmega128 fuses to be set incorrectly which will defeat any further reprogramming. There have been numerous reported difficulties with programming motes. These include program failure, flash verification errors, and dead motes.

If you still get flash verification errors, please refer to the suggestions provided in the application note: http://www.xbow.com/Support/Support_pdf_files/UISPHELP.pdf. Also, rebooting your PC, power cycling the MIB500/MIB510, and hitting the RESET switch on the MIB may also work.

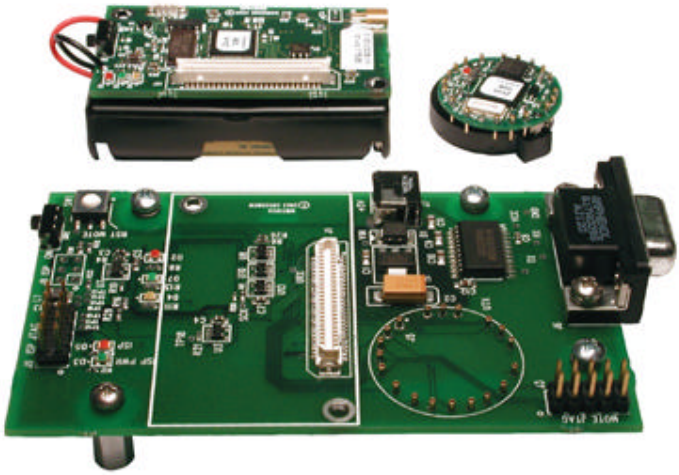


Fig. 2-1. MIB510 interface board pictured with a MICA2 and a MICA2DOT

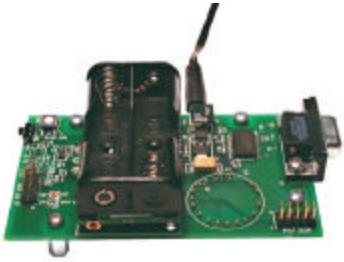


Fig. 2-2. MICA/MICA2 plugged into top-side of an MIB510

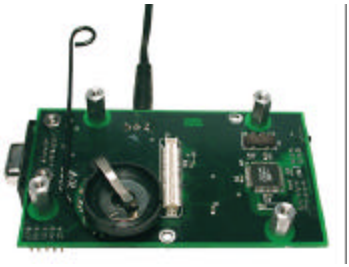


Fig. 2-3. MICA2DOT plugged into bottom-side of an MIB510

WARNING When programming a MICA2 with the MIB500/MIB510, turn of the battery switch. For a MICA2DOT, remove the battery before inserting into the MIB500/MIB510. The MICA2s and MICA2DOTs **do not** have switching diodes to switch between external and battery power.

2.3 Hardware Verification Using MicaHWVerify

To test the hardware, we have provided the `MicaHWVerify` application. It is designed for the purpose of verifying MICA/MICA2/MICA2DOT mote hardware only. If you have a different hardware platform, this application is not suitable.

- If necessary set the radio frequency for the MICA2/MICA2DOT as described in Section 1.5.
- Change to the /apps/MicaHWVerify directory
- For a MICA platform type "make mica"
- For a MICA2 or MICA2DOT platform type "`PFLAGS=-DCC1K_MANUAL_FREQ=<freq> make <mica2|mica2dot>`", respectively.

The compilation process should complete without any errors.

- Place a mote into a programming board and power it with either batteries or an AC wall power adaptor. (The red LED on the programming board should light.)
- Connect the programming board to the parallel port of your computer if you have an MIB500. Or if you have an MIB510, connect it to the serial port of your computer.
- Load the application on to the device. If programming the mote with an MIB500 (parallel port programmer), type

  ```
  make reinstall <mica|mica2|mica2dot>
  ```

  Or if programming the mote with an MIB510 (serial port programmer), type

  ```
  MIB510=/dev/ttyS# make reinstall
  <mica|mica2|mica2dot>
  ```

  Where # is 0, 1, 2, etc. for COM1, COM2, COM3 **assuming** that the numbering begins with zero. See Section 1.4 for installation instructions for other programmers.

A typical output when programming with an MIB510 looks like

```
$ mib510 make reinstall mica2
   installing mica2 binary
uisp -dprog=mib510 -dserial=/dev/ttyS0 -dpart=ATmega128 --
wr_fuse_e=ff  --erase --upload  if=build/mica2/main.srec
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash
Fuse Extended Byte set to 0xff
```

Now you know that the programming tools and the computer's parallel port are working.

The next step is to verify the mote hardware. First, confirm that the LEDs are blinking like a binary counter. Next, connect the programming board to the serial port of the computer. The `MicaHWVerify` application will send data over the UART that contains it status. To read from the serial port, we provide a java tool called `hardware_check.java`. It is located in the same directory. Build and run this tool. The commands are shown below assuming you are using `COM1` at 57600 baud to connect to the programming board.

NOTE If you are using the MIB510, please be sure to turn the SW2 switch to OFF position before using it to read the data from the Serial Port. If SW2 is set to ON position, this will disable mote's T_x line making it not transmit any data.

- Type "`make -f jmakefile`". The output from this command should look like

```
$ make -f jmakefile
mig java -java-classname=DiagMsg MicaHWVerify.nc DiagMsg
-o DiagMsg.java
mig java -java-classname=RxTestMsg MicaHWVerify.nc
RxTestMsg -o RxTestMsg.java
javac -sourcepath . hardware_check.java
```

- Then type "MOTECOM=serial@COM1:57600 java hardware_check" The output on the PC should be something like

```
hardware_check started
Hardware verification successful.
Node Serial ID: 1 60 48 fb 6 0 0 1d
```

This program checks the serial ID of the mote (except on the MICA2DOT), the flash connectivity, the UART functionality and the external clock. If all status checks are positive, the hardware verification successful message will be printed on your PC screen.

NOTE Since MICA2DOTs don't have a Serial ID, when you build the MicaHWVerify application, a warning message appears saying that "Serial ID not supported on mica2dot platform". However the application still builds and installs. If you run the hardware_check on MICA2DOT, it performs hardware verification, but the serial ID displayed is simply all 0xFF.

2.3.1 Mote Radio Verification:

To verify radio, you need two nodes. Use the second node (that has passed the hardware check up to this point) to act as a radio gateway to the first node. Install it with the application `TOSBase`.

- Change directory to the `/apps/TOSBase` directory
- Compile the TOSBase application by typing "PFLAGS=-DCC1K_MANUAL_FREQ=<freq> make <mica2|mica2dot>"
- Install the program into the mote via MIB500 or MIB510. Leave this mote in the programming board and place the other node next to it.
- Run the `hardware_check` java application by typing "`MOTECOM=serial@COM1:57600 java hardware_check`". The output should be the same as shown in the previous section (but will display the serial ID of the remote mote). The indication of a working radio system is, again, something like:

```
hardware_check started
Hardware verification successful.
Node Serial ID: 1 60 48 fb 6 0 0 1e
```

If the remote mote is turned off or not functioning, it will return a message "`Node transmission failure`".

If your system and hardware pass all the above tests, you are all set for having some fun with TinyOS. Congratulations.

2.4 Hardware Verification Using Mote-Test

To test the hardware, we have provided the application `Mote-Test`. Its purpose is to provide a quick way to check you're your MICA2 and MICA 2DOT motes are working.

- Install Mote-Test from the CD by running setup.exe found under `/Crossbow Software/Mote-Test` folder. This would also install LabVIEW run time engine.
- Copy the folders MICA2_TEST_315, MICA2_TEST_433, MICA2_TEST_916, MICA2DOT_TEST_315, MICA2DOT_TEST_433, and MICA2DOT_TEST_916 from the CD folder `/Crossbow Software/Mote Firmware` to your `/apps` directory.
- Power the MIB500/MIB510 programming board with the AC wall-power adaptor.
- Connect the MIB500/MIB510 to a PCs serial port with a DB9 (female)/DB9 (male) serial cable.
- **For MIB500 users only:** Connect the MIB500 to a PCs parallel port with a DB25 (female)/DB25 (male) parallel port printer cable.

Your MICA units are shipped from the factory with the hardware test firmware already installed. In the event you require confirmation that the hardware is still fully functional after altering the firmware, you may compile and reinstall the test firmware with the procedure described in Section 2.4.1. Of course, you need to have TinyOS installed into your PC. Otherwise, proceed to Section 2.4.2.

2.4.1 Loading in MICA2/MICA2DOT Test Firmware

- From a cygwin window, go to the folder for `MICA2_TEST_XXX` or `MICA2DOT_TEST_XXX` (`where XXX` is the frequency of your unit)
- Type "`make mica2 reinstall`" for each mote you wish to test. This will install the test firmware in your MICA2 or MICA2DOT mote(s).

> **NOTE** It is important to note that the command used for this chapter to install the mote test firmware into the motes is "`make mica2 reinstall`" and **not** "`make mica2 install.`"

If successfully installed, upon pressing the hardware RESET button on the programming board (marked as SW1), you should see the LEDs blink for a MICA2. This verifies that the programming tools and computer's parallel port are working correctly.

2.4.2 Setting Up Mote-Test to Verify the Motes

- If not already done, connect the MIB500/MIB510 programming board to the serial port of the PC.
- Next, run the `Mote-Test.exe` application from the Start Menu>Programs>Mote-Test.

The opening screen (Fig 2-4) will have button options to select various functions.

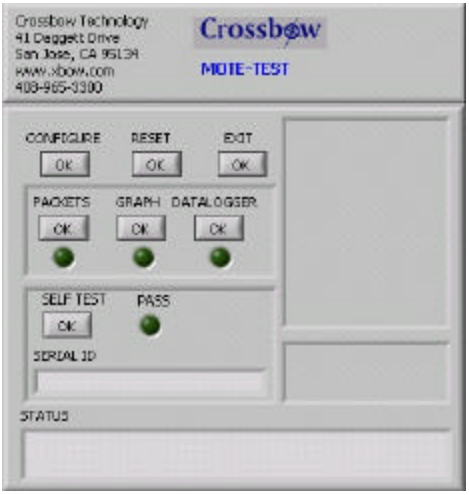


Fig 2-4. Mote-Test main window. Note: the PACKETS, GRAPH, and DATALOGGER buttons do not perform any function

Select the CONFIGURE button, and one of the two pop-up screens will appear (Fig. 2-5):

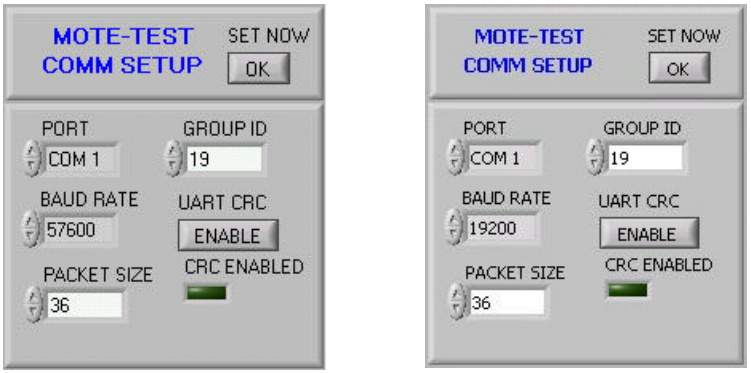


Fig 2-5. COMM Setup windows for the MICA2 (left) and MICA2DOT (right)

Make sure that the correct port is selected for your computer. The MICA2 firmware is configured to run at baud rate of 57600 and the MICA2DOT at 19200. The default packet size should be 36, **and UART CRC should not**

be enabled. Once the configuration is properly entered, push the SET NOW button. The main screen will now reappear.

2.4.3 Confirming that the Mote Hardware is Working

`Mote-Test` enables you to use either a MICA2 or a MICA2DOT in the MIB500/MIB510 to create a base station or serial gateway to a PC. This feature is unlike most TinyOS applications. This was done to test and demonstrate that two-way radio communications between two motes. These tests also verify that the mote hardware and the computer's serial port are working correctly.

- Supply a mote (MICA2 or MICA2DOT) (pre)programmed with `MICA2_TEST_XXX` or `MICA2DOT_TEST_XXX` firmware with batteries. This mote becomes the *remote* unit to test the radio functions.
- Connect either a MICA2 or MICA2DOT (pre)programmed with the `MICA2_TEST_XXX` or `MICA2DOT_TEST_XXX` firmware on the MIB programming board in which "XXX" is the same frequency as the remote unit.
- If not already done, connect the MIB500/MIB510 programming board to the serial port of the PC.
- Press the SELF TEST button. A pop-up screen (see Fig. 2-6) will ask "Which unit to test?" Choose either MICA2 or MICA2DOT depending on what you have plugged into the MIB board.

Fig 2-6. Unit selection pop-up screen

- A second pop-up (Fig. 2-7) will appear with instructions for configuring the test, click "OK." Place the remote unit 2 feet or more from the base.

Fig 2-7. An example instruction screen for placing motes

One of two Mote-Test Hardware Verify (Fig. 2-8) screens will appear:

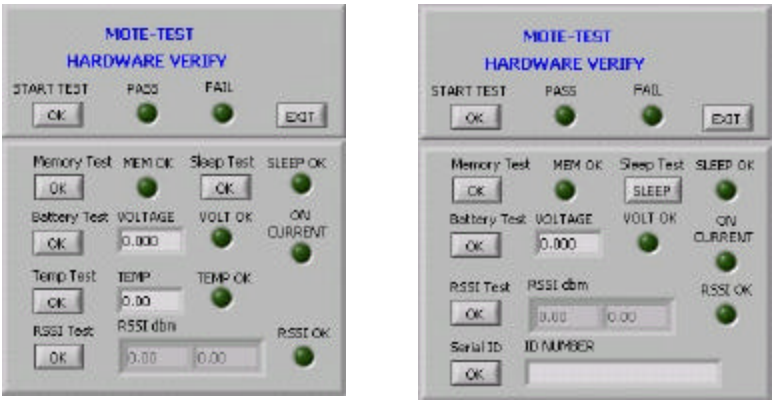


Fig 2-8. Mote-Test Hardware Verify screens for the MICA2 DOT (left) and MICA2 (right).

The MICA2DOT does not have a ID number and so this test is not done. Instead it reports back the on-board temperature reading (in °C)

- To run either the MICA2 or MICA2DOT hardware verify test, press the START TEST button.

For the MICA2 observe the status lights for the Memory, Battery, RSSI Test's, and the ID NUMBER in the Serial ID test. For the MICA2DOT observe the Memory, Battery, Temp, and RSSI Tests status lights. If all hardware components are functional, the PASS indicator will be lit. All these hardware tests are performed for the Base Station Mote except the RSSI.

The RSSI dbm has two boxes: the left one shows what the remote unit measured from the base stations unit. If the RSSI value is less than -80.00, you will get a FAIL indicator. This does not necessarily mean that your mote's radio has failed. If you have placed the remote MICA2 or MICA2DOT far away, the radio signals are too attenuated. The solution is to place your mote closer to the base station. If the remote unit is a MICA2DOT, be sure to use fresh 3 V coin cells.

Individual tests on each interface can be performed separately by selecting the appropriate test button. This verifies that the mote hardware is working correctly. When the test is completed, press the EXIT button. The main screen will reappear. You can continue checking other motes or end the program and continue on with other applications.

3 Introduction to TinyOS and NesC

The TinyOS operating system, libraries, and applications are all written in nesC, a new structured component-based language. The nesC language is primarily intended for embedded systems such as sensor networks. 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 principal goal is to allow application designers to build components that can be easily composed into complete, concurrent systems, and yet perform extensive checking at compile time.

TinyOS also defines a number of important concepts that are expressed in nesC. 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.

Components

Specification

A nesC application consists of one or more components linked together to form an executable. 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** that the interface provider must implement and another set of functions called **events** that the interface user must implement. 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.

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.

When looking at the files in an application directory, you can identify the nesC files because it uses the extension ".nc" for all source files—interfaces, modules, and configurations.

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 runs 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. The norace keyword should be used with extreme caution.

3.1 An Example Application: Blink

So far this is all fairly abstract—let's look at a concrete example: the simple test program "`Blink`" found in `/apps/Blink` in the TinyOS tree. This application simply causes the red LED on the mote to turn on and off at 1 Hz.

Blink consists of two components: a module, called "`BlinkM.nc`", and a configuration, called "`Blink.nc`". Remember that all applications require a single top-level configuration, 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 the executable for the mote. `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 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, 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.

Sometimes (as is the case with `Blink` and `BlinkM`) you will have a configuration and a module that go together. When this is the case, the convention used in the TinyOS tree is that `Foo.nc` represents a configuration and `FooM.nc` represents the corresponding module. While you could name an application's implementation module and associated top-level configuration anything (ncc uses the 'COMPONENT' definition in the application's `Makefile` to find the top-level configuration), to keep things simple we suggest that you adopt this convention in your own code. There are several other naming conventions used in TinyOS code.

3.1.1 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 first at the module `Blink.nc`:

Blink.nc

```
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 `/tos/interfaces/StdControl.nc`:

StdControl.nc

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

We see that `StdControl.nc` defines three commands: `init()`, `start()`, and `stop()`. `init()` is called when a component is first initia lized, 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`.nc are `init* (start | stop)*`. All three of these commands have “deep” semantics; calling `init()` on a component will make it call `init()` on all of its subcomponents.

The following two 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` .`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 exp licitly 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.

The line

```
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`), whereas `SingleTimer.Timer` on the right side of the arrow is refering 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.

3.1.2 The BlinkM.nc Module

Now let's look at the module `BlinkM.nc`.

BlinkM.nc

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

```
    interface Leds;
  }
}
// Continued below...
```

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:

Timer.nc

```
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:

BlinkM.nc, continued

```
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.

3.2 Compiling the Blink Application

TinyOS supports multiple platforms. Each platform has its own directory in the `/tos/platform` directory. In this section, we will use the MICA2 platform as an example.

Run the cygwin application by double-clicking the icon that can be found on your desktop.

- Enter the `/apps/Blink` directory using your shell (cygwin under Windows); it is a good application to make sure that the most basic hardware is working.
- Type "`make mica2`" in a cygwin window. This should complete successfully and create a binary image of your program for the motes.
- All objects, generated includes and executables are place in the bin directory for the specific platform, e.g., `/build/mica2`

You should, of course, observe errors and warnings that arise in building your application. This example should not have any. At the very end, the Make shows you a piece of the load map that tells you whether your application fits.

3.3 Programming a Mote and Running Blink

To download an application into the MICA2 mote, connect the 51-pin male connector of the MICA2 into the 51-pin female connector of on the MIB programming board. To download into a MICA2DOT, connect the female connectors of the MICA2DOT to the female connectors of the MIB's MICA2DOT programming bay located on the "underside" of the MIB programming board.

- You can either supply a 3 V supply to the connector on the programming board or power the node directly. The red LED labeled D2 on the programming board will be on when power is supplied.
- If you have MIB500CA, plug the 25-pin connector into the parallel port of a laptop configured with the TOS tools, or connect use a standard DB-25 parallel port cable.
- Type "`make mica2 install`". If you are using windows and the install doesn't work, you make need to fiddle with the port specified

to uisp; depending on the hardware, cygwin can map parallel ports to widely different names (use the `-dlpt=#` option, where # may be 1, 2, or 3).

- If you are using an IBM ThinkPad, it may be necessary to tell the tools to use a different parallel port. You can do this by adding the line

  `HOST = THINKPAD` before the `include` statement in `/apps/Blink/Makefile`

You should see the upload take place (this may take several seconds) and the red LED should light up every second.

3.4 Generating the Component Structure Documentation

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.

To generate the documentation, go to the `/tinyos-1.x/apps/Blink` directory and type "`make <platform> docs`". The html documentation will have the filename be generated in the `/docs/nesdoc/<platform>.docs/nesdoc/<platform>/index.html` is the main index to all documented applications.

NOTE If the install is located on the drives other than C: you must check the expansion of the `$TOSROOT` variable. If you get any errors during "`make mica2 docs`", then do the following (assuming the TinyOS was installed on D: drive)

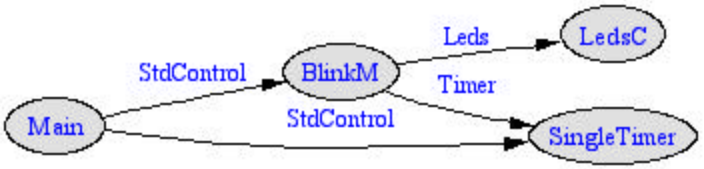
Open the file `Makerules` from `tinyos-1.x/apps` and under "Rules for documentation generation" section, replace the line,
"`NCC := $(NCC) -docdir=$(DOCDIR)/$(PLATFORM) -fnesc-is-app`"

with
"`NCC := $(NCC) -docdir=$(DOCDIR)/$(PLATFORM) -topdir=/cygdrive/d/tinyos-1.x -fnesc-is-app`"

The directory index takes you to an html file that looks like the figure below.

App: Blink

Component Graph (text version, help)



Browsing through the graphical representation of the component wiring using your mouse is really helpful to understand the overall structure of TinyOS.

For more details on different component modules please refer to Lesson 1 in the Tutorial.

4 Component Composition and Radio 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!

If you're using MICA2 or MICA2DOT motes, you will need to ensure that you've selected a radio frequency compatible with your motes. Refer back to section 1.5 on how to set the radio frequency.

4.1 Sending Messages with CntToLedsAndRfm

Assuming you are using a MICA2 mote, after it is installed you should see a 3-bit binary counter on the mote's LEDs. And while it is not apparent, it is, of course, transmitting the value over the radio.

- Build and install the application by typing "`MIB510=/dev/ttyS0 make mica2 install`", assuming you are programming with the MIB510 serial port programming interface board on COM1.

4.2 Receiving Messages with RfmToLeds

- In a similar manner program another mote with `RfmToLeds`.

When you turn on `CntToLedsAndRfm`, you should see the count displayed on the `RfmToLeds` device. If you turn the transmitter mote off, you will see that the LED counting stops on both motes.

Congratulations! You are doing wireless networking.

For more details on different component modules please refer to Lesson 4 in the Tutorial.

5 Displaying data on the PC

The goal of this section 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 will demonstrate a Java application that displays sensor readings graphically. Finally, we will close the communication loop by showing how to send data back to the motes.

5.1 The Oscilloscope application

The Oscilloscope application 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, but it can be easily extended to have it send the data over the radio instead.

- Set the `SENSORBOARD` option in `/apps/Oscilloscope/Makefile` to either `micasb` or `basicsb` depending on the type of sensor board you have.
- Compile and install the `Oscilloscope` application on a mote.
- Connect a sensor board (51-pin female) to the underside of the programming interface board (51-pin male) to get the light readings. See Fig 2a for proper connection of the sensor board to the programming interface board.
- Connect the programming board with mote and sensor board to the **serial port** of your computer. See Fig. 2b for the proper stack of the mote, interface board, and sensor board (top to bottom).

When the `Oscilloscope` application is running, the red LED lights when the sensor reading is over some threshold, set to `0x0300` (hex), 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.

> NOTE The MICA2 UART runs at 57600 baud and MICA and MICA2DOTs run at 19200 baud.

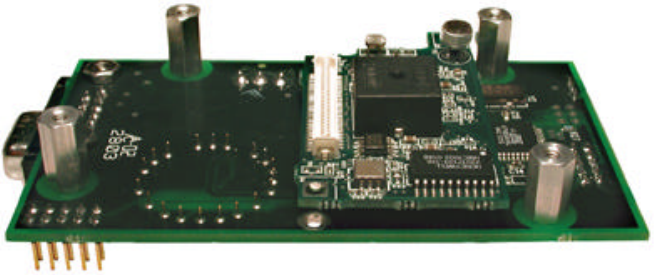


Fig 5-1. Sensor board plugged into the bottom side of the MIB510

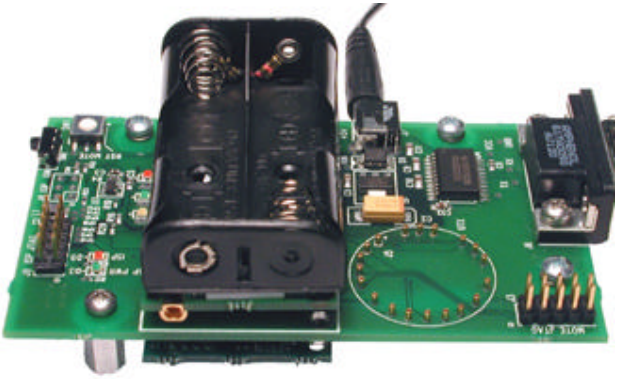


Fig 5-2. MICA2 plugged into the top side and the MTS310 sensor board plugged into the bottom side of the MIB510. Also shown are the serial cable and AC power wall adaptor connected to the MIB510.

5.2 The 'listen' Tool: Displaying Raw Packet Data

The first step to establishing communication between the PC and the mote is to connect up your serial port cable to the programming board, and to make sure that you have Java and the `javax.comm` package installed. After programming your mote with the `Oscilloscope` code, `cd` to the `/tools/java` directory, and type

```
make
export MOTECOM=serial@serialport:baudrate
```

The environment variable `MOTECOM` tells the java Listen tool (and most other tools too) which packets it should listen to. Here `serial@serialport:baudrate` 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 baud rate of the mote. For the MICA and MICA2DOT motes, the baud rate is 19200, for the MICA2 it is 57600 baud. You can also use a mote name as the baud rate (in which case that motes baud rate is selected). So you could do any of:

```
export MOTECOM=serial@COM1:19200 # MICA baud rate
export MOTECOM=serial@COM1:mica  # MICA baud rate, again
export MOTECOM=serial@COM2:mica2 # the MICA2 baud rate,
on a different serial port
export MOTECOM=serial@COM3:57600 # explicit MICA2 baud
rate
```

- Set MOTECOM appropriately
- Run `java net.tinyos.tools.Listen`

You should see some output resembling the following:

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

```
serial@COM1:19200: resynchronising
7e 00 0a 7d 1a 01 00 0a 00 01 00 46 03 8e 03 96 03 96 03 96 03
97 03 97 03 97 03 97 03 97 03
7e 00 0a 7d 1a 01 00 14 00 01 00 96 03 97 03 97 03 98 03 97 03
96 03 97 03 96 03 96 03 96 03
7e 00 0a 7d 1a 01 00 1e 00 01 00 98 03 98 03 96 03 97 03 97 03
98 03 96 03 97 03 97 03 97 03
```

The program is simply printing the raw data of each packet received from the serial port. 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/system/AM.h`. The data payload of the message, which is defined by the application, is defined in `/tos/lib/OscopeMsg.h`. The overall message format for the `Oscilloscope` application is as follows:

- Destination address (2 bytes)
- Active Message handler ID (1 byte)
- Group ID (1 byte)
- Message length (1 byte)
- Payload (up to 29 bytes):
    - source mote ID (2 bytes)
    - sample counter (2 bytes)
    - ADC channel (2 bytes)
    - ADC data readings (10 readings of 2 bytes each)

Before continuing, execute `unset MOTECOM` to avoid forcing all java applications to use the serial port to get packets

5.3 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, go to `tools/java` and run the program by typing

```
  java net.tinyos.sf.SerialForwarder -comm
serial@COM1:<baud rate>
```

where *<baud rate>* is the baud rate of your serial port (it will typically be either 19200 or 57600) The `-comm` argument tells `SerialForwarder` to communicate over serial port `COM1`. 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 which in turn should talk to, e.g., a serial port. You wouldn't want the `SerialForwarder` to talk to itself...).

The `<baud rate>` argument tells `SerialForwarder` to communicate at specified baud rate.

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

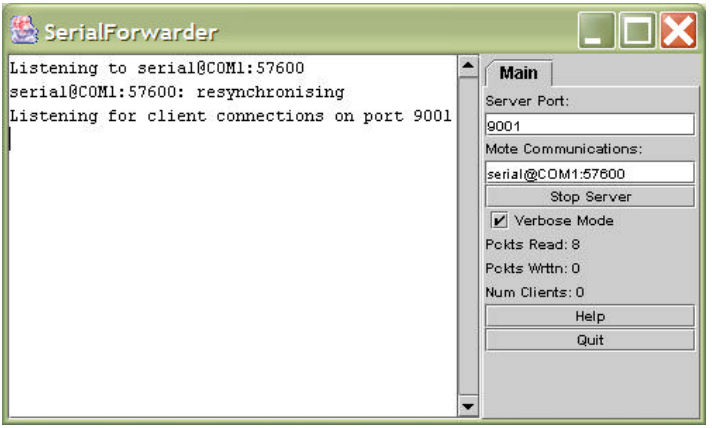


Fig. 5-3. Screen shot of java application SerialForwarder when it is properly running

`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.

As packets arrive from the mote connected to the serial port, you will see the "*Pckts Read*:" field in the lower right corner begin to increment.

5.4 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. 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:

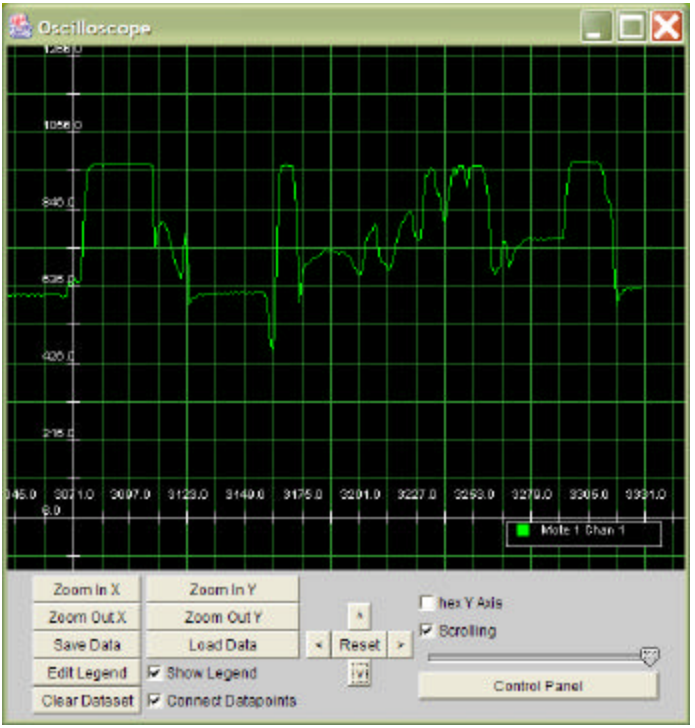


Fig. 5-4. Example screen shot from java application Oscilloscope

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, be sure that you have not zoomed in on the display.

5.5 Transmitting Sensor Data Over the Radio to Serial Port

The `Oscilloscope` mote application is written to use the serial port and the light sensor. Instead, look at `/apps/OscillosopeRF`, which transmits the sensor readings over the radio. In order to use this application, you need to provide a bridge that receives data packets over the radio and transmits them over the serial port. The `/apps/TOSBase` is an application that does this; it simply forwards packets between the radio and the UART (in both directions).

NOTE The `Oscilloscope` GUI is already capable of displaying sensor readings from multiple motes. You have to ensure that those readings are correctly transmitted and received over the network. This setup would look like the following diagram.

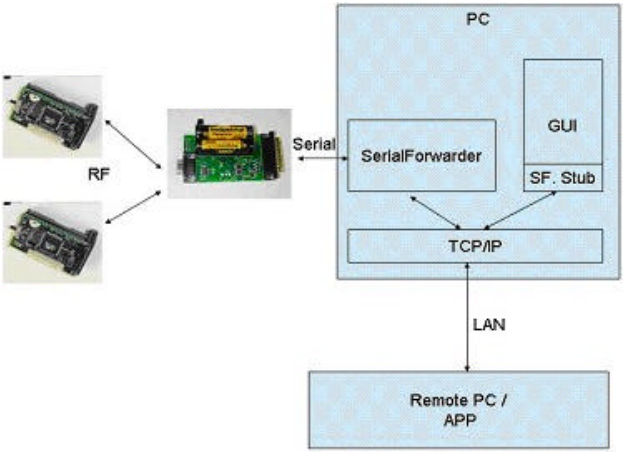


Fig. 5-5. Illustration of how OscilloscopeRF works

For this exercise you will need at least 3 motes.

- Program the two motes with the `OscilloscopeRF` application, setting their nodeid's to 1, 2, etc. using command

  ```
  make mica2 install.<nodeid>
  ```

  where, `<nodeid>` is the ID you wish to program into the mote.

- Program the other mote with the `TOSBase` application.

- With the `TOSBase` mote plugged into the programming board and connected to the serial port, spread around the rest of motes (with the `OscilloscopeRF` application with sensor boards connected (See Fig. 3).

- Repeat Steps 5.3 and 5.4 to invoke the `SerialForwarder` and `Oscilloscope` GUI. You should see the following GUI with specific Mote IDs displaying the light sensor data. If you cover the light sensor with the hand you should see the data from that particular mote change.

Fig. 5-6 A battery-powered MICA2 mote with an MTS310 sensor board used in the OscilloscopeRF application

> **NOTE** For MICA2DOT users, it is highly recommended that they use MICA2 as a base station, because the MICA2DOT UART is not stable as a base-station.

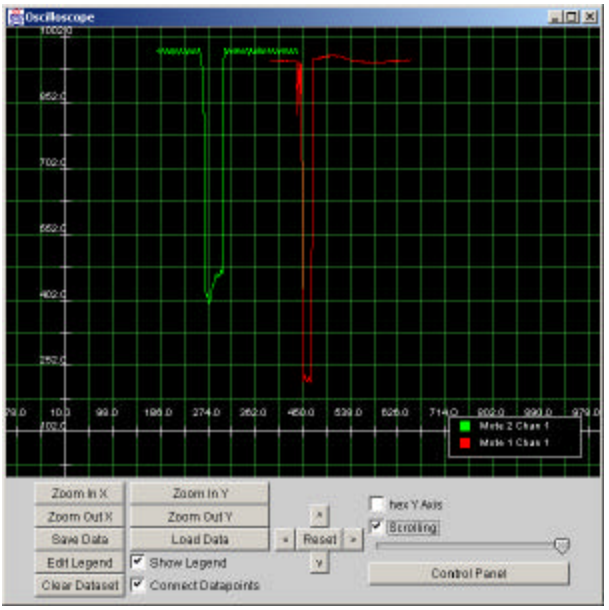


Fig. 5-7. Screen shot of the java GUI for OscilloscopeRF. Two motes' sensor data displayed in two different colors is visible in this example

For more details on different component modules please refer to Lesson 6 in the tutorial.

6 Multihop Routing

The TinyOS-1.1 release and later include library components that provide ad-hoc multi-hop routing for sensor network applications. The implementation uses a shortest-path-first algorithm with a single destination node (the root) and active two-way link estimation. The data movement and route decision engines are split into separate components with a single interface between them to permit other route-decision schemes to be easily integrated in the future. Use the multi-hop router is essentially transparent to applications (provided they correctly use the interface).

Use of the multi-hop library component is mostly transparent to the application. Any application that uses the Send interface can be connected to this component to achieve multi-hop functionality. One limitation of multi-hop, however, is the aggregate data rate. Applications should maintain average message frequency at or below one message every two seconds. Higher rates can lead to congestion and or overflow of the communication queue.

6.1 Surge Demo

The Surge application, in the `/apps/Surge`, is a simple example of a mutlihop application. Surge takes light sensor readings and sends them over the mesh to the base node (nodeid 0). Accompanying this application is a Java program that can be used to visualize the logical network topology and the sensor readings. Users are encouraged to review the application, `SurgeM.nc`, and it's configuration, `Surge.nc`, to better understand how to use the multi-hop tools.

- Build the application `Surge`: `cd` to the `/apps/Surge` directory and type a "`make mica2`".
- Build the java tools: `cd` to the `/tools/java/net/tinyos/surge` directory and type "`make`".
- Install the application onto the target nodes, giving each node a unique node ID by typing "`MIB510=/dev/ttyS0 make reinstall.<nodeid> mica2`" assuming you are programming through the MIB510 connect to serial port COM1. Remember, the **base station** mote must be installed with node ID of 0. This node should be connected to a PC via a serial or network link.

Next run the java applications.

- First start `SerialForwarder` to link the base node and the PC (e.g., typing "`java net.tinyos.sf.SerialForwarder –comm serial@COM1:mica2`" in the java directory).

- Next start the GUI for the `Sensor Network Topology` from the `/tools/java` directory by typing "`java net.tinyos.surge.MainClass <Groupid>`", where <*Groupid*> is the AM group ID number (in decimal) used when compiling the mote application (e.g., the default group ID of hex 0x7d is decimal 125).

When the application starts, you should immediately see the base node reporting sensor values. After about 1 minute, other nodes should appear as the network topology builds.

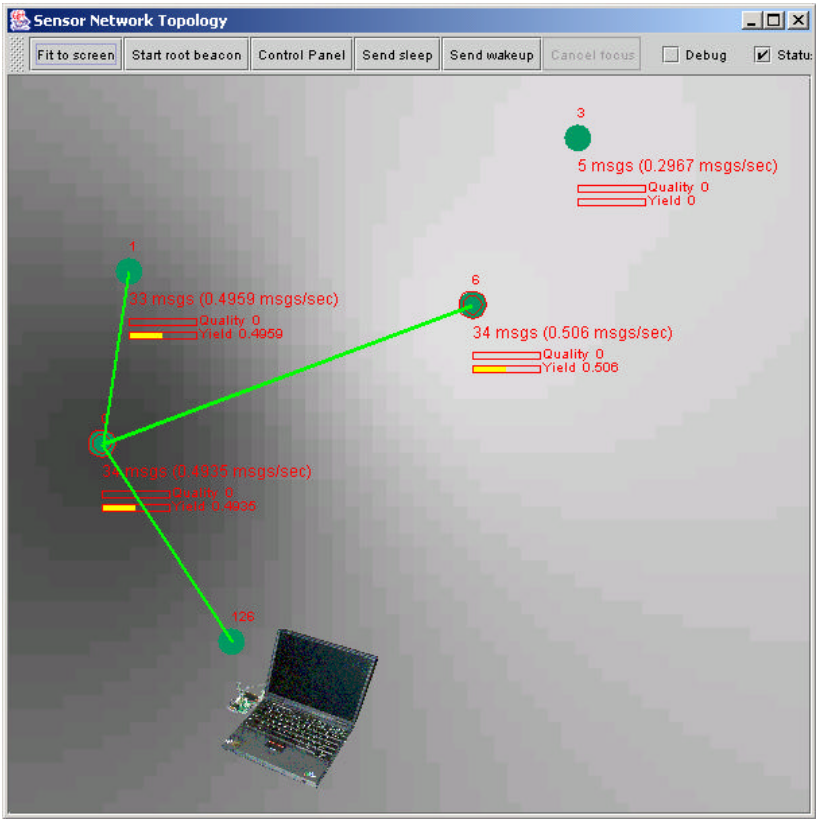


Fig. 6-1. Network topology GUI in Surge

If you cover the light sensor with the hand, that particular mote should get dark on the screen. You can make a particular mote disappear from the screen by powering it off or move it farther away. The solid green lines indicate the active data transmission link and the red line represents previously active communication links, which has now become inactive, because the node found a better transmission path.

NOTE The location of the node IDs on the screen does not represent the physical location of the nodes in the neighborhood of the PC.

6.2 Learning More About Multi-hop Protocols

Multi-hop protocols for ad-hoc networks is an active area of research. The nature ad-hoc networks makes them very different than fixed-powered and even cellular networks. Ad-hoc networks are characteristically not reliable over the long term. Nodes may suddenly stop working or may physically move out of range. Multi-hop allows for these dynamic changes in network topology.

Ad-hoc networks also are designed to minimize the use of energy to so it can work off of batteries for at least many months to a few years. Powered networks by contrast can afford to expend a lot more energy to manage links. One of the challenges to ad-hoc networks is that broadcasting is energy and time inefficient. So a multi-hop protocol must be able to dynamically determine which nodes (motes) would be the better or the best parent to a transmitting mote.

There have been several multi-hop protocols designed specially for TinyOS. The application in this section, `Surge`, is one of the oldest and was developed at the University of California, Berkeley. It is a useful demonstration of multi-hopping but does not include power management. Another protocol is `DSDV` (located in `contrib/hsn/tos/lib`), developed as Intel-Berkeley Labs. It has power management features. One of the latest and perhaps most promising to date is `Blast` by Alec Woo of UC, Berkeley. `Blast` is reported to benefit from the extensive use of estimators and uses advanced power management. These files are located in `contrib/hsn/tos/lib/route`

6.3 Learning More About TinyOS

Now that you have TinyOS working, you can start learning about it and writing your own programs. In the `tinyos-1.x/doc/tutorial` there is a comprehensive tutorial for learning TinyOS. Read and follow the instructions in different Lessons.

7 Warranty and Support Information

7.1 Customer Service

As a Crossbow Technology customer you have access to product support services, which include:

- Single-point return service
- Web-based support service
- Same day troubleshooting assistance
- Worldwide Crossbow representation
- Onsite and factory training available
- Preventative maintenance and repair programs
- Installation assistance available

7.2 Contact Directory

United States:	Phone:	1-408-965-3300 (7 AM to 7 PM PST)
	Fax:	1-408-324-4840 (24 hours)
	Email:	techsupport@xbow.com
Non-U.S.:	refer to website	www.xbow.com

7.3 Return Procedure

7.3.1 Authorization

Before returning any equipment, please contact Crossbow to obtain a Returned Material Authorization number (RMA).

Be ready to provide the following information when requesting a RMA:

- Name
- Address
- Telephone, Fax, Email
- Equipment Model Number
- Equipment Serial Number
- Installation Date

- Failure Date
- Fault Description

7.3.2 Identification and Protection

If the equipment is to be shipped to Crossbow for service or repair, please attach a tag TO THE EQUIPMENT, as well as the shipping container(s), identifying the owner. Also indicate the service or repair required, the problems encountered, and other information considered valuable to the service facility such as the list of information provided to request the RMA number.

Place the equipment in the original shipping container(s), making sure there is adequate packing around all sides of the equipment. If the original shipping containers were discarded, use heavy boxes with adequate padding and protection.

7.3.3 Sealing the Container

Seal the shipping container(s) with heavy tape or metal bands strong enough to handle the weight of the equipment and the container.

7.3.4 Marking

Please write the words, "***FRAGILE, DELICATE INSTRUMENT***" in several places on the outside of the shipping container(s). In all correspondence, please refer to the equipment by the model number, the serial number, and the RMA number.

7.3.5 Return Shipping Address

Use the following address for all returned products:

Crossbow Technology, Inc.
41 Daggett Drive
San Jose, CA 95134
Attn: RMA Number (XXXXXX)

7.4 Warranty

The Crossbow product warranty is one year from date of shipment.

Crossbow Technology, Inc.
41 Daggett Drive
San Jose, CA 95134
Phone: 408.965.3300
Fax: 408.324.4840
Email: info@xbow.com
Website: www.xbow.com