TINYOS LABORATORY DEVELOPMENT

A Thesis

Presented to

The Faculty of

Cal Poly, San Luis Obispo

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Rafael David Kaliski

November 2005

Approval Page

Title: TinyOS Laboratory Development

Author: Rafael Kaliski

Date Submitted: November 4, 2005

Dr. James Harris	
Advisor and Committee Chair	Signature
Dr. Albert Liddicoat	
Committee Member	Signature
Dr. John Seng	
Committee Member	Signature

Abstract

TinyOS Laboratory Development
Rafael David Kaliski

Wireless sensor networks offer numerous research opportunities in many areas as well as many novel applications. TinyOS and mica2 motes offer a variety of inexpensive wireless sensor network based research opportunities. One of the research areas is TinyOS laboratory development. When developing laboratories which utilize TinyOS and mica2 motes, the "state of the art" technology is assessed. Custom solutions and workarounds are developed to address limitations in the technology. The TinyOS laboratory offers the students an accelerated introduction to wireless sensor networks without having to discover the limitations of the technology first-hand. Each project builds upon the knowledge from the previous project. By the end of the TinyOS laboratory the student should be capable of writing his or her own TinyOS application. An instructor's manual is provided to aid the instructor with the TinyOS laboratory.

Acknowledgements

I would like to thank Dr. John Seng and Dr. Diana Franklin for sharing the Crossbow hardware acquired from their Cal Poly Central Coast Research Park (C3RP) award, which was necessitated by this thesis.

Table of Contents

List of Figures

Chapter I: Introduction

Wireless sensor networks (WSNs) are a relatively new technology. Previous sensor networks would require wired connections or would require a larger platform, compared to the sensor boards, to gather data and transmit information back to the base station. Wireless sensor networks have provided a relatively small, low-power, and low-cost platform for data collection and transmission.

The current trend in industry is moving towards wireless sensor networks, as they provide both an inexpensive means to gather data and they can operate via self-forming, self-healing, autonomous networks formed by the motes.

Academic institutions find WSNs beneficial as they both allow them to stay current with the technology which industry is currently using and allow them to reinforce concepts from their undergraduate curriculum. Concepts ranging from embedded systems to networks can be covered with WSNs. Also, with the advent of TinyOS, the details of the hardware can be abstracted, so the motes can be programmed in C.

An application of wireless sensor networks, such as collecting data in a structure or an environment, demonstrates the power of wireless sensor networks. Networks setup in a structure, such as a building or a road, provide a cheap and effective means of detecting the presence of certain events of interest, such as temperature, radiation, moisture, and barometric pressure.

By integrating wireless sensor networks into the current curriculum offered by the EE department, students will be exposed to a means of collecting data via remote sensing without the hindrance of setting up a physical network or configuring a network for data

transmission. The use of WSNs in the curriculum will also help to reinforce microcontroller design concepts and introduces the students to an event-driven form of execution. It also allows students to work at a higher level of abstraction than current embedded systems courses. This thesis investigates the possibility of introducing WSNs into the laboratory curriculum for undergraduates. This work presents the development of four laboratory modules, each of which could be performed during one week of activity.

In the lab development process several issues arose which needed to be addressed, such as how and when to introduce students to the capabilities, and the limitations of TinyOS. Many of TinyOS' capabilities are explored in each of the laboratories. When a feature or capability that a laboratory needed was advertised as working, yet it was not, it was identified as a limitation. If the limitation could not be worked around by redesigning the laboratory, a TinyOS workaround or custom solution was created to address the limitation. Crossbow hardware limitations were noted, but unlike software limitations workarounds could not be derived within the allotted time. Unfortunately, addressing the hardware limitations requires massive mote application rewriting in addition to derivation of code specific to the hardware's features. The labs included in this thesis utilize custom solutions to address the limitations of TinyOS, while demonstrating the many features of TinyOS, such as In-Network Programming and Xlisten. Xlisten is a TinyOS data logging application.

The TinyOS laboratory consists of four lab modules. Each lab module is designed to demonstrate subtle points about TinyOS while clearly demonstrating the learning objectives of the laboratory. Each lab module builds off knowledge from the

previous lab. In each of the lab modules more capabilities of TinyOS are exposed to the students. The first laboratory provides an introduction to TinyOS and several of its support programs. The second laboratory provides an introduction to the mica2 mote, mica2 mote communication, programming the mica2 mote, In-Network Programming of the mica2 mote, and logging data from a mote. The third laboratory provides an introduction to ad-hoc networking and another data logging application, Mote Logger, which is more flexible than Xlisten. The second and third laboratories require the students to complete incomplete source code. The fourth and final laboratory is designed to provide the students with the entire program design experience from design requirements thru implementation and testing. The fourth laboratory only provides a proposed set of requirements for the mote application; the students may choose to specify their own requirements if they choose. The students must follow the software development life cycle to meet the requirements.

In chapter II an overview of Wireless Sensor Networks, mica2 motes, TinyOS, and some applications of wireless sensor networks is presented. In chapter III the requirements for the laboratories, the overall goals of the laboratories, and the learning objectives of each of the laboratories is presented. In chapter IV the design of each of the laboratories, each of the experiments, and the problems/limitations encountered is presented. In chapter V the results of the wireless sensor network laboratory design and the content of each laboratory are presented. Chapter VI presents the conclusions and suggestions for future lab development. Appendix A presents the student manual for the four laboratories, Appendix B the instructor's manual, and Appendix C presents a summary of the accompanying CD.

Chapter II: Background

a) Wireless Sensor Networks

Modern research on sensor networks was started by DARPA in the 1980s under the Distributed Sensor Network (DSN) program [10]. Wireless sensor networks usually consist of many inexpensive, low-power, and resource constrained nodes. The nodes consist of a microcontroller, a set of sensors, an energy source, and a radio transceiver. Wireless sensor networks can self-form an ad-hoc network of nodes, thus creating a sensor net. A node usually sends each of its sensor's data. The node's datum is then aggregated by each node in the network, aggregation occurring on the parent node of the transmitting node, until it reaches the destination node, i.e. the gateway or base station node. The aggregated datum is then passed from the gateway/base station node to a computer for storage and processing.

b) TinyOS

TinyOS is an open-source event-driven "real-time" operating system designed by U.C. Berkeley [4].

TinyOS is designed for use in low-power/limited resource applications which utilize wireless embedded sensor networks. TinyOS is a component based operating system which minimizes code size and power consumption. Components which are

not used are not included in the compiled program. Also, the components are initially turned off which assists in reducing power consumption [4].

TinyOS can reside on a multitude of platforms, each of which supports many different sensor boards. The mica2 platform, (MPR400CB) utilizes an Atmel ATMEGA128L microcontroller, a ChipCon CC1000 radio (The CC1000 chip operates in the 915Mhz. range), a USART, and a 51-pin expansion connector (see Figure 1.) The ATMEGA128L also has an 8-channel 10-bit ADC, which is connected to the radio and the 51-pin expansion connect.

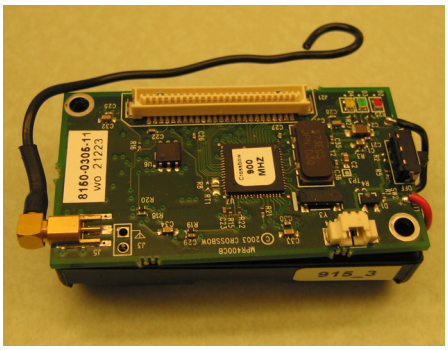


Figure 1: Mica2 mote (MPR400CB)

The 51-pin expansion connector offers an interface to other sensor boards. One of the many sensor boards that the mica2 platform can support is the mica sensor board "micasb" (MTS310CA) (see Figure 2.)

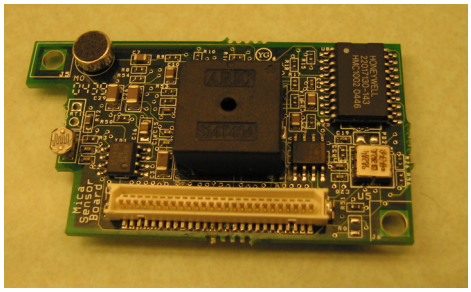


Figure 2: Mica sensor board (MTS310CA)

The mica sensor board provides a tone detection circuit, a microphone, a buzzer, a photometer, a 2-axis accelerometer, and a 2-axis magnetometer.

The mica2 platform, which this thesis uses for the laboratory experiments, was designed by students at U.C. Berkeley and marketed through Crossbow Technology, Inc. [4].

c) NesC

TinyOS is programmed in a C-based language, known as nesC. NesC provides support for tasks, events, and commands, as well as standard C. Tasks are typically posted in response to an event, and cannot preempt one another, but can be preempted by other events. Events are run in response to a hardware interrupt or signaled by a component. Unlike tasks, events can preempt one another. Commands are called via other components, and run in the current execution thread. Two execution threads exist, the task execution thread and the hardware event handler execution thread. NesC checks for potential data races which result from this concurrency model [4].

NesC provides for interfaces and components. Interfaces provide the only means of communication between components. Interfaces consist of commands (commands are called by the user of the interface and are implemented by the provider of the interface) and events (events are implemented by the user and signaled by the provider of the interface). A component can be either a configuration or a configuration and a module. Configurations "wire up" components and, if it exists, the main code module to create a new component or application. Modules consist of code which is executed in response to an event. The main application typically

consists of a configuration (the configuration defines the connections between the main application and its components) and a module (the module defines the main application). A component can provide and use multiple interfaces [4].

d) Basic TinyOS Communication

All mica2 applications which use the standard means of communication utilize the mica2 radio stack (see Figure 3.) The standard means of communication is defined as direct or indirect use of the genericComm, or the genericCommPromiscuous, component.

The UART portion of the TinyOS communications stack is not shown, as UART communication normally occurs between endpoints; UART framing is covered in Figure 6. At the data link layer the UART and Radio communications diverge.

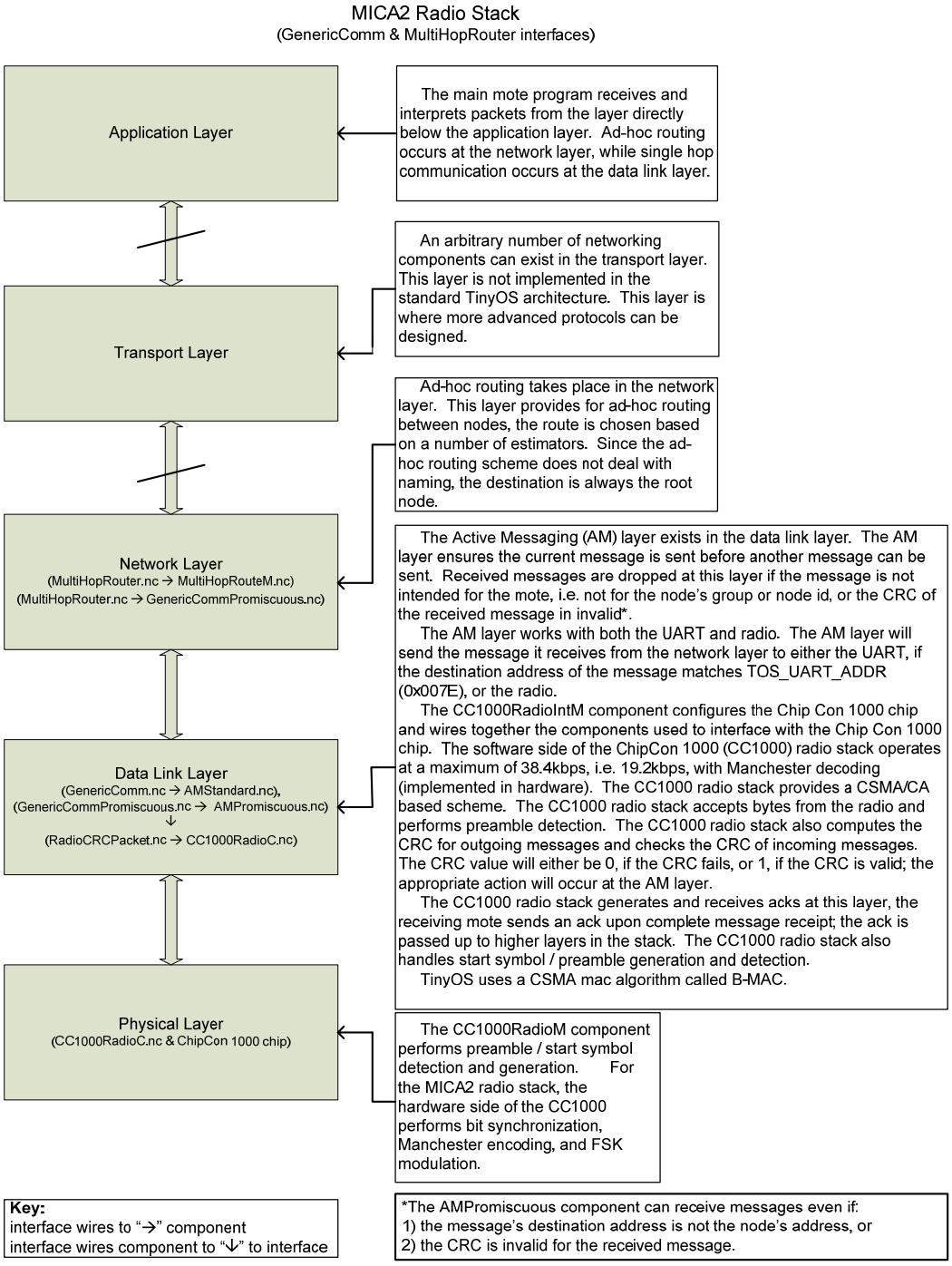


Figure 3: Mica2 radio stack [2]

e) Ad-hoc Routing

Wireless sensor networks show their potential when they exist in an ad-hoc network environment. An ad-hoc wireless sensor network is a self-forming, self-healing, autonomous network of sensors which allows nodes to be beyond direct, single hop, communications distance from the base station. Nodes can easily be moved, removed, or added to the network with minimal impact. With ad-hoc networks, the potential applications for wireless sensor networks grow substantially.

Unfortunately, ad-hoc network operation in a power, and memory, constrained environment creates a limitation on throughput from any given node[1]. The risk of flooding a network becomes higher, and in order to address this problem throughput must be limited by reducing the number of messages sent by any given node.

TinyOS provides an ad-hoc routing component called Multi-hop routing. The Multi-hop routing scheme organizes all the nodes, within communications range, into a routing tree. The root node in the routing tree is the base station node [4]; other nodes are placed in the tree based on their proximity to the root node and the quality of their link with the other nodes.

Multi-hop routing scheme is a collection based routing scheme [4] which has both pros and cons. The Multi-hop routing module can easily be integrated into any application and, as its name implies, provides for a much more expansive range for data collection. Multi-hop's collection-based routing scheme permits data

[1] The terms node and mote are synonymous. A mote is referred to as a node when the discussion pertains to ad-hoc routing.

aggregation and in-network processing of data. Unfortunately, the multi-hop component's collection-based routing scheme precludes messages from being transmitted down the routing tree, i.e. the root node is the implicit and ultimate destination for all messages. Also, due to the multi-hop routing protocol, the throughput of the nodes, compared with standard single-hop communication, is limited; this is in part due to routing update messages.

f) Power Management

There are three methods of using power management, either the user application directly handles power management, the user application enables power management and controls when the mote can enter one of the power saving modes, or the application enables power management and lets the operating system control power management.

The first method would mean the user application would have to determine what sleep level to enter and when to enter it. Directly handling the power management would necessitate that the user application has intimate knowledge of the microcontroller and direct control of the hardware. Consequently, direct handling of power management would defeat one of the purposes of an operating system, i.e. hardware abstraction, in addition to circumventing the scheduling scheme provided by the operating system.

The second method utilizes a component of the operating system to handle power management decisions, but still requires the user application to issue the sleep command. Although the second method does abstract some of the hardware aspects, such as device registers and the devices, it still requires the user application to enter

the sleep command. This method still circumvents one of the components of an operating system, namely scheduling.

The third method both utilizes a component provided by the operating system, and allows the scheduler to enter sleep mode when there are no events or tasks pending.

TinyOS provides a power management component (The TinyOS power management component is defined in the PowerManagement and the HPLPowerManagementM nesC files) and the scheduler already controls when the microcontroller should enter sleep, i.e. when there are no tasks pending or currently executing. The power management component is disabled by default; this does not mean the microcontroller will not sleep. The mote will enter the default sleep level, IDLE, when the sleep command is issued by the scheduler. Some of the TinyOS components that explicitly adjust the power state are the timer, the SPIHPL module, the CC1000 radio stack, the AMStandard module, and the AMPromiscuous module. Any component which uses any of the aforementioned modules, whether implicit or explicit, adjusts the power state. The sleep level is adjusted by calling the power management's adjustpower() command. The sleep level can only be adjusted when power management is enabled.

Under TinyOS 1.1.7, power management is not enabled and power management's interface does not provide either the enable or the disable commands. Consequently the only power state the mote can enter is the IDLE power state. In the IDLE state,

the microcontroller, the internal flash, and the EEPROM are stopped; any external or internal interrupts can wake up the microcontroller [15], i.e. the microcontroller is merely waiting for an internal, or external, event to occur.

g) Applications

There is a plethora of applications for wireless sensor networks. Aside from the labs associated with this thesis, some other applications are the Envisense GlacsWeb project and the University of Washington's CSE "Flock of Birds" project.

In the GlacsWeb project, wireless sensor networks were utilized to monitor the glacial environment. GlacsWeb was designed to assist in sub-glacial bed deformation research. The wireless sensor network nodes / probes were placed in the ice sediment boundary between 50 and 80 meters below the surface of the glacier. Each probe transmitted its sensor (pressure, temperature, and orientation) readings to a base station located on the surface of the glacier for relay to a reference station. The nodes are non-recoverable [8].

In the University of Washington's CSE 466 course, the students are introduced to TinyOS and Mica2dot motes. The students learned concepts from embedded systems, networking, and Real-Time Operating Systems. The student's final project was to create a "flock of birds" where each mote would sing a song and the neighboring motes were notified of what song a particular neighbor was singing. Each mote would sing a song based on its neighbor's song, this could be the same song as before, or a different song [9]. "Originally, the songs were played via a

speaker which was controlled by the mote's Atmel processor; the current version uses a special-purpose sensor board which utilizes a Yamaha synthesizer" [21].

Other research projects utilizing wireless sensor networks can be found at http://www.tinyos.net/related.html.

h) Mote UART communication / Logging data from wireless sensor networks

Data logging applications enable data recording from a wireless sensor network; a conceptual diagram is shown in Figure 4 below. Each mote can communicate with each other and the base station via direct or indirect links. Once the data packet has been sent to the base station mote the data can be recorded on the attached computer via a communications channel, such as a serial, parallel, or ethernet connection. The application on the base station mote must send/forward data via its wired communications channel. For the mica2 mote, the TOSBase application is an example of an application which forwards packets via the UART.

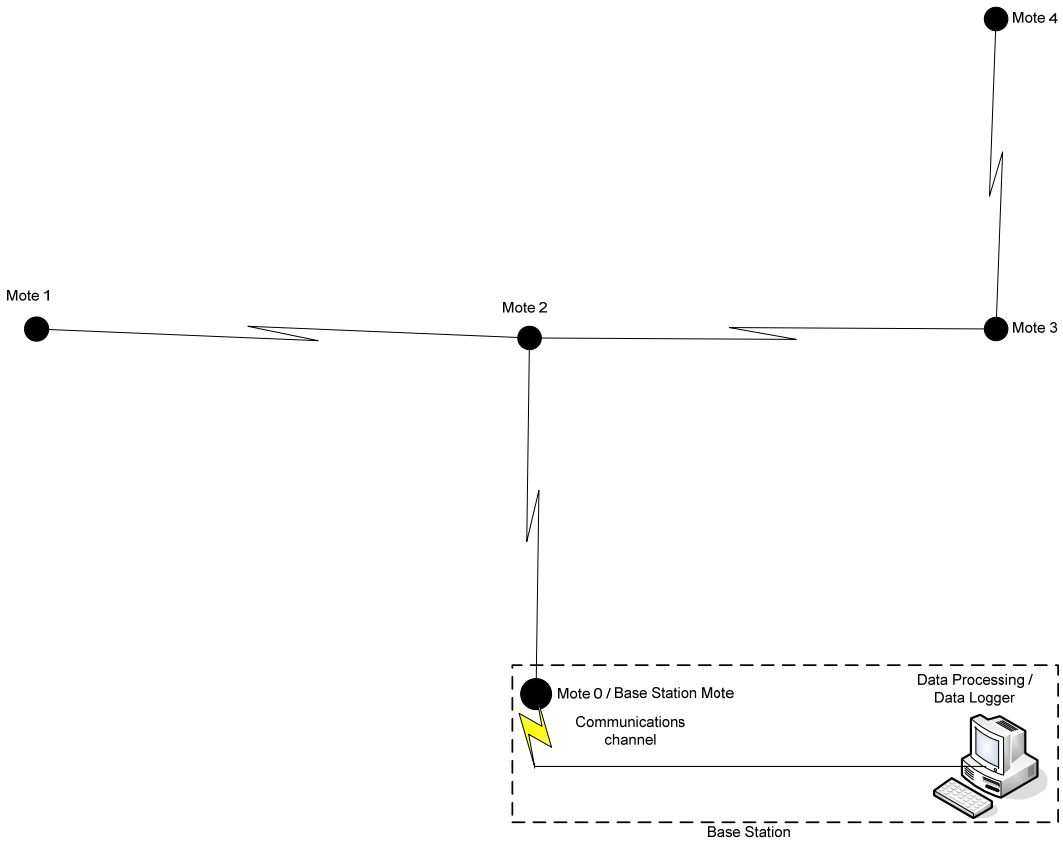


Figure 4: Wireless Sensor Network, ad-hoc setup

The mica2 mote is programmed via the MIB510 serial programmer. The MIB510 serial programmer, shown in Figure 5, uses a UART for serial communication with the attached computer and enables a mote to act as the base station mote when the mote is properly programmed. The TinyOS serial line communication packet structure is given in Figure 6.

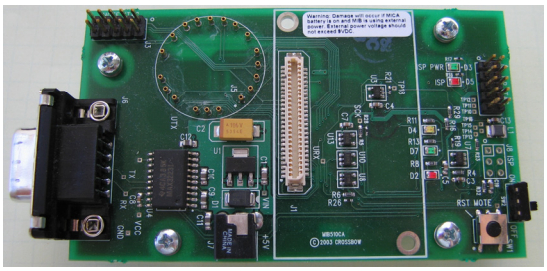


Figure 5: MIB510 serial programmer (MIB510CA)

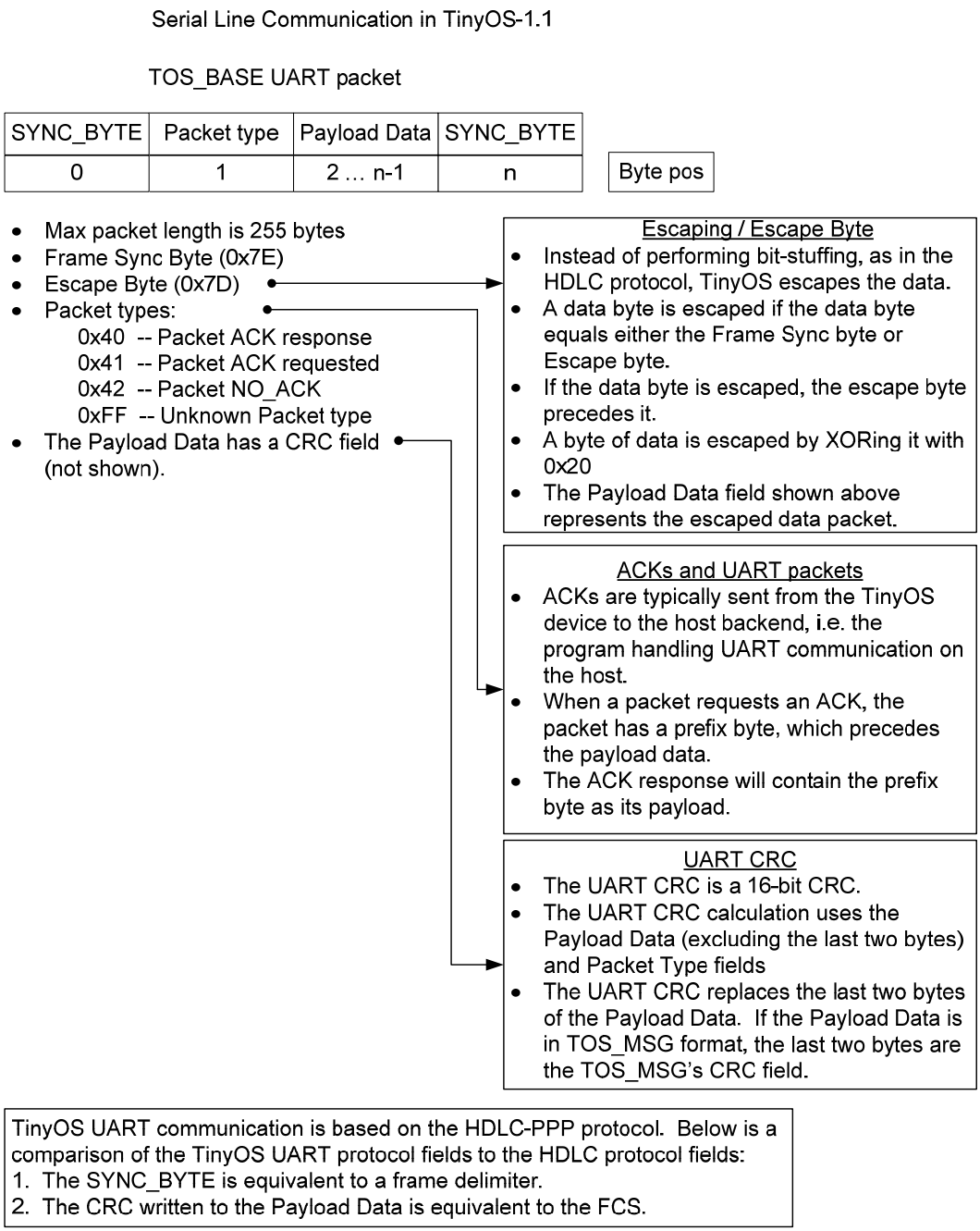
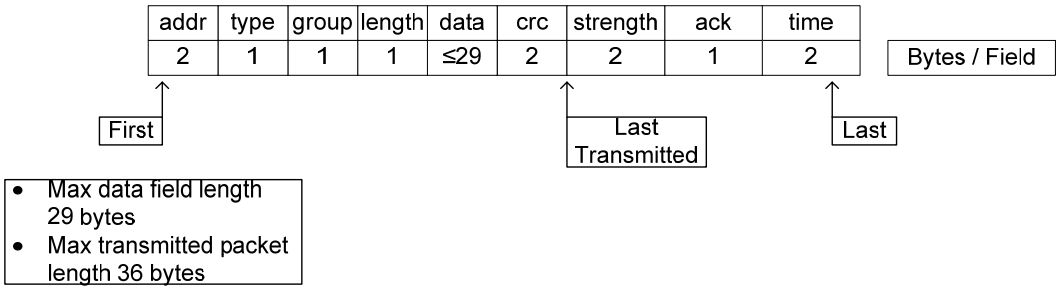


Figure 6: Serial line communication [10]

Once the data packet has been extracted from the UART packet, and un-escaped, the resulting data packet can be parsed according to the data logger's filters. The data packet is typically in a TOS_Msg packet format (currently the only officially supported packet format is the TOS_Msg packet format). The TOS_Msg packet format is described in Figure 7.

TOS_MSG packet format



addr – Address field, destination of the packet.

- The address must be the destination node id, the reserved broadcast address (0xFFFF) or the UART address (0x007E).
- If the address field is the UART address, the data will be sent directly to the UART versus the Radio.
- If the received packet's address is not the receiving node's ID or the broadcast address, the packet will be dropped at the AM level.

type – Type field, specifies which packet handler will be called at the AM level, upon receipt.

group – Group field, specifies a channel for the motes on the network.

- If the group field does not match the receiving node's group id, the packet will be dropped at the AM level.

length – Length field, specifies the length of the data section of the packet.

data – Data field, this field contains the data to be transmitted with the packet.

crc – CRC field, incrementally calculated as each byte of the packet is transmitted.

strength – set when a message is received via the ChipCon 1000 mica2 radio stack.

ack – used for reliability in the communication stack.

time – stores atomic capture of clock.

The Data field, of the TOS_MSG packet, can have a format imposed on it by the mote application. The data field can be interpreted by the packet handler. The appropriate packet handler to interpret the data with is determined by the type field.

Figure 7: TOS_Msg packet format [4]

Chapter III: Requirements for Lab

a) Prerequisites

The Crossbow motes, used in the wireless sensor network experiments, run the TinyOS operating system. The motes are embedded systems running on an Atmel chip. The motes are programmed using a C based language known as nesC. In order to understand how wireless sensor networks work and how to program them a basic understanding of C and embedded systems is required.

Under the 2005 CPE / EE curriculum, CPE 329 should provide the basic skill set required by the wireless sensor network experiments. The C programming language book, "The C Programming Language (2nd Edition)" by Kernigan and Ritchie, is recommended as a refresher to the C programming language.

The development environment for the laboratory experiments consist of TinyOS 1.1.7, TinyOS 1.1.7 support programs, the mote data logging application "Mote Logger", the X2c conversion script, and the mote hardware. The mote hardware consists of: 1 MIB510CA serial programmer board, 3 MPR400CB mica2 motes, and 3 MTS310CA mica sensor boards.

TinyOS 1.1.7 requires a minimum of approximately 600MB for a complete installation. The TinyOS system requirements are: 1) an available com port 2) a system capable of running java 1.4.1 and CygWin. Instructions for installing TinyOS 1.1.0, the requisite release for TinyOS 1.1.7, and TinyOS 1.1.7 can be found under

the TinyOS website [4] via following URL http://www.tinyos.net/tinyos-1.x/doc/install.html. The rpm of the CVS snapshot for TinyOS 1.1.7 can be found at http://www.tinyos.net/dist-1.1.0/tinyos/windows/ . There is an install shield for TinyOS 1.1.7, if preferred; it is located at http://www.tinyos.net/dist-1.1.0/tinyos/windows/.

In order to use the TinyOS support programs and the Mote Logger application, JDK 1.5 and JavaComm 2.0 should be installed. Ensure that the java 1.4.x is the first version of java that CygWin finds; some TinyOS java programs will not compile with JDK/JRE 1.5+. The TinyOS support programs are included with the TinyOS installation. The Mote Logger can be downloaded via the link http://www.netprl.calpoly.edu/files/phatfile/Mote_Logger.zip.

The X2c script requires Perl, which is included in the default TinyOS installation. The X2c script can be downloaded via the link http://www.netprl.calpoly.edu/files/phatfile/X2c.pl.

b) Overall Goals

An overall goal of the experiments is to provide students with an accelerated introduction to wireless sensor networks. This goal is realized by introducing the students to TinyOS, the mica2 motes, the mica sensor board, various useful TinyOS applications, and by utilizing the components introduced in each laboratory in each laboratory's experiment. Since there is a time restriction of four weeks for this laboratory, the experiments are fairly simple.

Another goal is to provide the students with the necessary knowledge base for designing, implementing, and testing an application in less than four weeks. By the end of the laboratory, the student should be familiar with most of the major TinyOS components and each of the component's major functions.

c) Learning objectives

When designing the laboratory experiments, the learning objectives were always the first thing written in the experiment. By writing down the learning objectives first, the requirements for each laboratory were specified before the laboratory was developed.

The learning objectives for the 1st laboratory are:

1. To learn how to acquire and install the development environment.
2. To become familiar with TinyOS and its components.
3. To become familiar with event-driven execution.
4. To learn how to develop applications in TinyOS with nesC.
5. To learn how to simulate applications in TinyOS with TOSSIM.

The learning objectives for the 2nd laboratory are:

1. To become familiar with Crossbow mica2 motes and the mica sensor board.
2. To learn how to program mica2 motes.
3. To learn how to read data from the communications port provided by the serial programmer.

The learning objectives for the 3rd laboratory are:

1. To learn how to harness the power of ad-hoc mote networks.
2. To learn how to utilize the Mote Logger application to collect data from the base station.
3. To provide a system knowledge base so the user can design, implement, and test their custom application.

The learning objective of the 4th laboratory is:

1. To apply previous knowledge to design, implement, and test a system from a set of requirements.

Chapter IV: Designing the Labs

a) Designing the experiments

The laboratories were developed around TinyOS 1.1.7, Xlisten v 1.16, X2c v 0.1, and Mote Logger v 0.4, i.e. the development environment. As such, the laboratories are limited to the capabilities of TinyOS and Xlisten; the Mote Logger application and X2c script are used only for data deciphering purposes.

The first laboratory was designed to introduce the students to the development environment. In this laboratory the students are introduced to TinyOS, CygWin, the surge application, the TinyOS support programs TOSSIM/Tython, and the serial forwarder. The students are given instructions on how to configure CygWin for TinyOS. The students are also given the requisite knowledge to develop the necessary tools to be able to analyze nesC programs, as well as simulate nesC programs with TOSSIM. This laboratory experiment offers the students an opportunity to observe the power of ad-hoc networks via TOSSIM, TinyViz, and the surge application.

The second laboratory builds upon the students' knowledge gained from the first laboratory. In this laboratory the students are introduced to mica2 radio and UART communication. They are also introduced to In-Network Programming, i.e. reprogramming motes over the network. Finally, the students are introduced to Xlisten and its capabilities and the X2c conversion script which is required to convert

data form Xlisten into a usable CSV-compatible format. In the lab experiment the students are given source code with some critical lines missing, they must apply their knowledge from the previous laboratory to complete the source code. Although the students are given instructions on how to use Xnp, this laboratory still requires them to familiarize themselves with the location and use of TinyOS libraries as well as the sensor interfaces. The laboratory consists of a mote application which can be configured to listen to any sensor on the board, with the exception of the tone detector. They must wire the main code module to two different sensors. In addition to the output from Xlisten, the leds provide visual feedback for each sensor they are connected. This laboratory experiment offers them the opportunity to see the power of reconfigurable code and motes.

The third laboratory builds upon the students' knowledge from the previous laboratories. In this laboratory the students are introduced to Ad-Hoc networking, the Mote Logger application, and radio power control. The students do not use the In-Network Programming feature of TinyOS as it does not function across ad-hoc networks. Also the students no longer use Xlisten as the Mote Logger application provides more functionality than Xlisten, based on the features the students are utilizing in the laboratories. In the laboratory experiment the students are given source code with critical lines missing; they must make use of the information from TinyOS and in the laboratory to build a successfully working mote. The leds provide useful debugging information which will assist them in determining if their application has successfully established an ad-hoc network. The students utilize the Mote Logger application to gather, parse, and separate data from each of the motes in

the network. The temperature conversion formula is given to assist in the interpretation of data. Since both the temperature and light sensors are utilized the temperature conversion will assist in debugging their application as the data will be meaningless if the sensors are read incorrectly, i.e. the data is invalid if both sensors are on at the time of the reading. This laboratory offers them the opportunity to create an ad-hoc network and see the behavior of the ad-hoc network, i.e. the packet drop rate. If the packet transmission rate is too high their packet drop rate will increase, this requires the students to ensure their application does not flood the throughput constrained network.

The fourth laboratory builds upon the students' knowledge from all of the previous laboratories. In this laboratory the students must design, implement, and test their solution to a set of defined requirements. The students are only provided with a description of a proposed mote application, i.e. the Sounder System application. They may choose to either build an application which meets the requirements, or they may propose a set of requirements and build their own application. The students may use any component of TinyOS in order to meet the requirements. The students must fully document their solution to the set of requirements and the approach they chose.

The Sounder System application requires the students to devise unique sequences of tones for each mote in the sounder network. This application also requires the students to synchronize the receiving mote to the transmitting mote and determine if the tone pattern being received is in fact the trigger for the mote. Each unique sequence of tones triggers a mote. After a mote is triggered, it will transmit another unique sequence of tones, thereby triggering yet another mote. The first mote which

started the chain reaction also acts as the last mote in the chain. The tone detector sensor and the microphone sensor interfaces need to be examined by the students in order to determine the features the tone detector offers. Also the sounder interface needs to be examined by the students in order to determine what features the sounder offers.

The appendix is split based upon what the students needed to know to begin work on the experiment and what a potential solution to the laboratory consists of. The laboratory has two different versions, a student version where the experiment is not completed, and an instructor version where a potential solution to laboratory is included.

In the student's version of the laboratory, labs 2 and 3 present the students with source code which is missing critical lines of code; the critical lines of code are removed based upon what is covered in the laboratory and the previous laboratories. The graphs and calculations, necessitated by the procedures, are also removed. The student must create these graphs and calculations in order to complete the experiment. Finally, the commands required to run the various applications are removed, as the information is presented in the laboratory through the reading material associated with each laboratory.

In the instructor's version of the laboratory, potential solutions to the experiments are given. Lab solutions consist of graphs, results from calculations, and working source code for each of the experiments, depending on the requirements of the experiment. The source code solution for experiment 4 is a solution to the set of requirements provided to the students. The instructor's version of the laboratory also

consists of the necessary commands required to run the various TinyOS support applications.

b) Designing the X2c conversion script

Xlisten gathers data over a communications port of the computer. The Xlisten application displays the data from the serial communications port either as raw (raw data is in hex or ASCII format), parsed, or converted data. An optional timestamp displays the time of packet reception. Unfortunately, data from custom packet formats, such as in laboratories 2 to 4, cannot be converted or parsed as Xlisten is incapable of parsing data from applications other than X applications[2], Crossbow developed, and the surge application. Only raw data and the optional timestamp can be displayed.

To address the Xlisten parsing limitation the X2c script was written as a supplement to Xlisten. The X2c Perl script converts a file containing the output of the Xlisten application to a comma separated variable format (CSV) file. The X2c script only works with the RecSensNet application, used for the reconfigurable sensor network experiment in laboratory 2.

Unlike the Mote Logger application, this script does not take into account packet types; consequently the packet is hard-coded to the RecSensNet application. The format of the CSV file is given in Table 1.

Table 1: X2c output format

[2] X applications were not tested as they are not part of the TinyOS distribution. X application support was advertised by Crossbow's MOTE-VIEW documentation. MOTE-VIEW uses an application similar to Xlisten, known as xserve. Custom packet formats cannot be defined for MOTE-VIEW or xserve.

Time	Photo / Temp	Microphone	Mote ID	Packet ID

The student must denote which CSV file corresponds to which RecSensNet packet type.

X2c design details

The X2c script examines each line of the specified file; the file must contain the I/O redirected output from the Xlisten application. If the line contains either a timestamp or raw, hexadecimal only, serial communications data the line is parsed, otherwise the line is ignored. In the data parsing step, extraneous data, such as the length of the received message and formatting associated with Xlisten is removed. The result of the data parsing step is serial communication data and the timestamp for each line of serial communication data.

Serial communication data are then parsed into TOS_Msg fields and serial communication headers and trailers, the serial communication headers and trailers are discarded. After the TOS_Msg packet has been extracted, the packet is then parsed into payload data and TOS_Msg headers and trailers, the TOS_Msg headers, and trailers are discarded. Finally the data payload is parsed into single/multi-byte application fields, the endian for each of the multi-byte fields is adjusted, i.e. multi-byte fields are converted from the little-endian to big endian format for integer conversion and storage, see Figure 8.

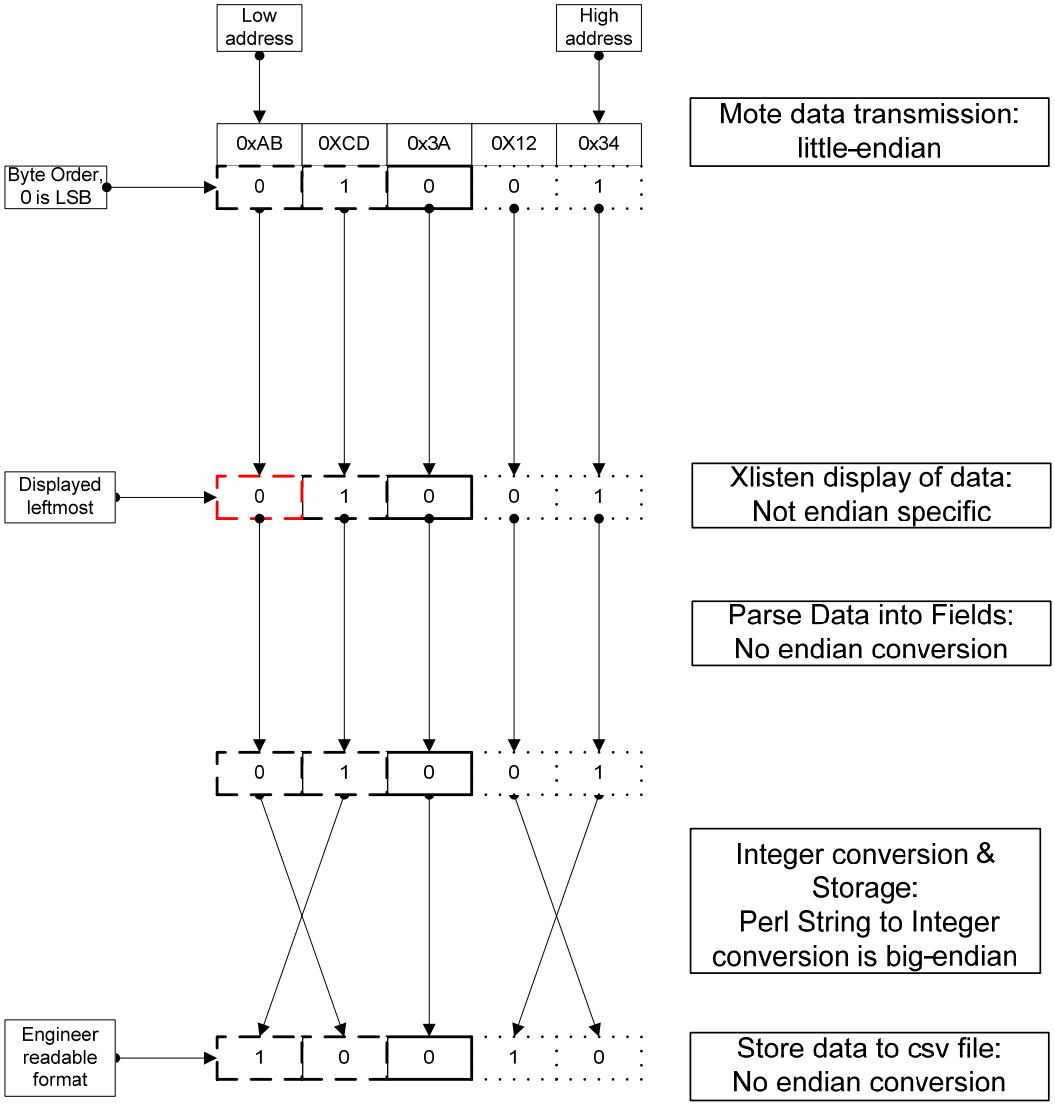


Figure 8: Endian of Mote and X2c script

After the serial communication data has been parsed into application fields, the data is then stored in comma separated variable format, with the timestamp of each received packet in the first column.

X2c enables the students to utilize Xlisten and become familiar with its capabilities and limitations. Unlike the Mote Logger application, which is described

in the next section, Xlisten will continue to be developed and supported. Also Xlisten is included in the distribution, while the Logger application is not.

c) Designing a mote data logging application

TinyOS is packaged with a data logging utility known as Xlisten. This utility enables display and interpretation of data from a serial communications port. Data can also be decoded based on the packet type that is received and interpreted based on known sensor boards. There are several limitations to Xlisten, they are:

1. Xlisten cannot interpret arbitrary packet types, i.e. the data payload can only be displayed in its raw form. Arbitrary packet types cannot easily be added to Xlisten.
2. Xlisten cannot separate data from different motes or different packet types. Xlisten data can only be stored to a file via I/O redirection, ergo all data recording will reside in a single file regardless of packet type and mote id.

Crossbow produced a similar application, based on Xlisten, known as MOTE-VIEW [1]; this application provides a GUI front-end and database recording capabilities. Still one of the main limitations of Xlisten continues to be prevalent in MOTE-VIEW. MOTE-VIEW cannot interpret arbitrary packet types; arbitrary packet types cannot be added to MOTE-VIEW. To address this main limitation, the Mote Logger application was created.

Unlike the X2c Perl script, the Mote Logger application is designed as a replacement for the Xlisten application. The Mote Logger application provides the user with the ability to record packets from user defined packets into multiple files. The determination of which received packet is recorded to a file is determined by a

user designated field in the packet format given by the user. The data can optionally be parsed into its respective fields.

The Mote Logger application is only designed to address limitations in Xlisten and MOTE-VIEW; it does not have all of Xlisten's or MOTE-VIEW's features. Mote Logger was designed to be used in conjunction with any comma separated variable compatible spreadsheet application. The Mote Logger application only updates packet received statistics, decodes data, and performs radix conversion on data before storing it to a CSV file. The Mote Logger application can only receive data over a serial communications port.

Mote Logger design details:

The Mote Logger application extracts TOS_MSG packets from the data received over the serial communications port connected to the PC; the data is extracted based on the TinyOS serial line communication protocol (see Figure 6 for the format of TinyOS serial packets.) The TOS_MSG packet is parsed and the data payload extracted (see Figure 7 for the TOS_MSG packet format.) The data payload is then parsed / interpreted based on the user defined packet formats. The field in the user defined packet format delimited as the mote identifier field is used to update packet received statistics; the mote id field determines which packet received statistic is updated. The statistics are displayed under the "Motes Detected" section of the Mote Logger GUI. If the received TOS_MSG datum is from a mote not listed under the Mote ID column, the mote is added to the Mote ID column.

When interpreting packets the data of each multi-byte field must have its byte ordering, i.e. endian, adjusted to be properly decoded, i.e. converted from bytes to integers, and displayed in engineering format.

Figure 9 illustrates what the byte ordering is, and why the byte ordering changes, at each data storage/parsing stage of the Mote Logger application. The data of each multi-byte field only has its endian adjusted if it can be decoded, i.e. the packet type is defined, and the user has selected TOS_MSG as the "packet type" in the "packet type / data format" section of the GUI.

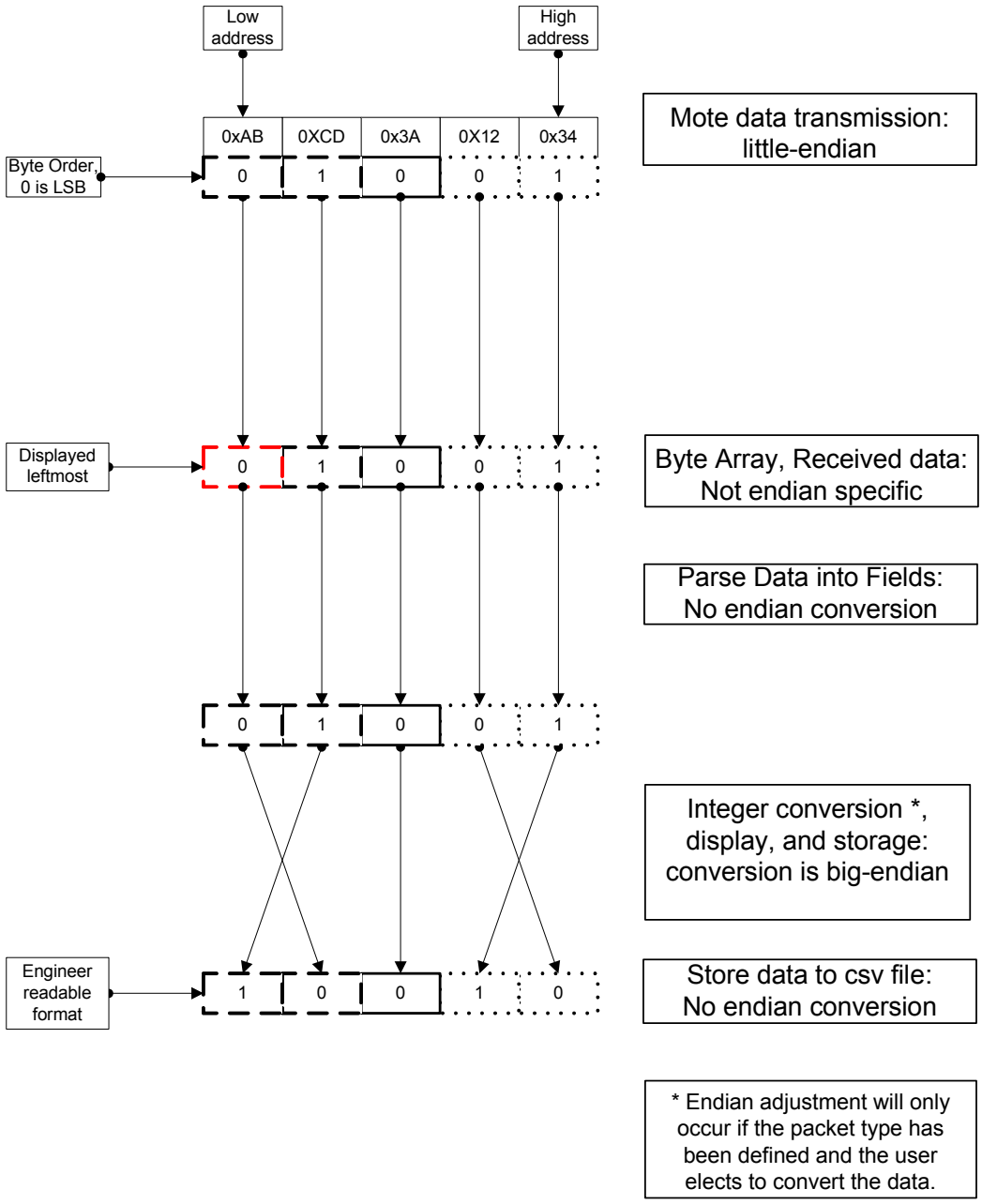


Figure 9: Endian of Mote and Mote Logger application

If the data payload cannot successfully be decoded, i.e. the packet type does not have a defined format, the RAW data is logged in the same order that they were received.

The Mote Logger application only provides parsing, logging, and integer conversion of the received packet to a CSV compatible format. The data logging converts known fields to integers.

*Please see the Mote Logger read me file for information on how to use the Mote Logger application. The Mote Logger read me file is on the accompanying CD.

d) Problems encountered

One of the problems is the mica2 compilation process; for unknown reasons a user mote application may cause the compiler to display the following error:

> "nesC: Internal error. Please send a bug report to the nesC bug mailing list at nescc-bugs@lists.sourceforge.net"

This error was generated when the first attempt at lab 4, the Morse code application, was being developed. The error was caused by the toupper library function; the workaround entailed creating a custom toupper function. The toupper function used by the Morse code encoder converts the ASCII message string to all uppercase; as Morse code only has a single case associated with the letters, so does the lookup table.

A related problem is TOSSIM; it would not simulate the Morse code application or the Sounder System application, i.e. the timer would never trigger. The problem is, the code works correctly on the mote, so TOSSIM's behavior does not necessarily match that of the mote's behavior.

The Morse code project was abandoned because the tone detector interrupt occurred multiple times when enabled, at least on the base station node. The working

lab 4 project is the Sounder System application. The Sounder System application required tone detector invalid interrupt removal code in order to prevent premature triggering.

Other encountered problems stem from the fact that TinyOS is not clearly documented. Discovered limitations of TinyOS include the following: As of TinyOS 1.1.7, the ad-hoc network is unidirectional, which precludes one of the major functions which must inherently exist in a wireless sensor network. Ad-hoc networks must be bidirectional in order to allow motes to be remotely configured, via In-Network Programming. Also, without bidirectional ad-hoc networking, it becomes difficult to poll motes beyond the single hop distance. Another TinyOS 1.1.7 problem is the lack of support for power management enabled applications; the interface does not include the required commands to enable/disable power management. This precludes the use of the mica2 motes in any long-term observation, as the battery life will be substantially less than that of a power managed mote.

Problems which have a direct impact on the laboratories are addressed by workarounds, which include supplemental programs and scripts, redesigning of the laboratory and experiments, or are clearly noted. Compilation and programming problems have scripts to workaround or address the problem. The .bashrc, makelocal, makefile, and mib510.extra scripts permit each laboratory to compile and program the connected mote correctly. To address data logging issues that are related to Xlisten, the Mote Logger application and the X2c script were created as supplemental

programs. Issues related to mote application development are noted in each laboratory.

Chapter V: Results

The laboratories were developed by first performing a study of TinyOS. The listed features were studied in more depth to gain a greater understanding of the features and how to use them. Features which did not work, as suggested by TinyOS documentation, were documented. Even though the version of TinyOS used was changed multiple times during thesis development, many of the features which were previously advertised as working still did not work.

The second step in the laboratory development entailed documenting the prerequisites necessary for the laboratories and the overall goals of the laboratories. Once these were documented, the laboratories were outlined based on how the overall goal could be meant. This provided an outline for the learning objectives for each laboratory.

The third step in the laboratory development was to document which features of TinyOS were necessary to facilitate the learning objectives for each laboratory. Features which were deemed necessary, and did not work, were researched in more depth. Each necessary non-working feature, or limitation, of TinyOS necessitated a workaround or the creation of a custom program. For the configuration of TinyOS the .bashrc file was created. For the programming of motes in TinyOS the makelocal file was created. For individual mote programming, the makefile for each experiment was created; this file was modeled after other applications' makefiles. To augment the TinyOS serial line application, Xlisten, the X2c conversion script was created. The X2c script allows the students to continue using Xlisten while permitting them to interpret and analyze the data

in a CSV-compatible spreadsheet application. Finally, a more functional Xlisten replacement program, known as the Mote Logger application, was created. The Mote Logger application is a stand-alone application which listens to the serial communications port, parses the received data based on a user defined packet type, and separates the data into different CSV files based upon a user defined mote identification field.

The fourth step in laboratory development was to design the experiments. This step was discussed with Dr. Harris. It was determined that the first experiment should be an introduction to TinyOS laboratory; the second experiment should involve an In-Network Programming laboratory; the third laboratory should involve ad-hoc networking, and the last laboratory should be a culmination of the previous laboratories, i.e. a design laboratory, with one or two sets of proposed requirements.

Once the experiments were specified, the next step meant programming a solution to the laboratory and documenting the procedures. This step involved some thought as the solution, with key lines missing, would be provided to the students. In the first laboratory no code writing is required, so there is no source code solution. In the second and third laboratories the students are given the incomplete source code, for each of the experiments and they must complete it. In the last laboratory, the students are not given any code; the proposed set of requirements does have a source code solution. Both the solution and incomplete source code is included in appendix C.

The fifth step in laboratory development was to design the contents of each of the laboratories. Each laboratory contains information which is necessary to complete the experiment. This includes component usage information, extracted from TinyOS source

code and documentation, useful information pertinent to TinyOS and programming such as scripts and the Mote Logger application, and pitfalls. Pitfalls related to a given laboratory are included in each laboratory. By providing the students with a list of pitfalls, they should be able to avoid making a mistake which is already known as a potential problem. The pitfalls are extracted from component source code and problems encountered during laboratory development.

The sixth and final step is to split the laboratories. The student version of the laboratory includes the necessary information required to complete the experiment, all the necessary commands are located within the required laboratory readings. Each student laboratory includes incomplete source code, if it is necessitated by the laboratory. The instructor's manual uses the student laboratory as reference, but it includes the commands necessary to run the tools for each laboratory. The instructor's manual also includes the graphs, calculations, and working source code which are required to complete each of the experiments, with the exception of experiment 1.

The laboratories are included in the appendix. Appendix A contains the student version of the laboratory. Appendix B contains the instructor's version of the laboratory. Appendix C contains Mote Logger usage information.

The accompanying CD contains:

1. This thesis
2. The X2c Perl script
3. The Mote Logger source code
4. The Mote Logger jar file
5. The Mote Logger README file

6. The student version of each experiment's source code
7. The instructor version of each experiment's source code
8. The CSV and excel files used for the lab solutions

Chapter VI: Conclusions

TinyOS, the Crossbow mica2 mote, and the Crossbow mica sensor board offer many research projects and course development opportunities. Unfortunately, TinyOS 1.1.7 has significant limitations, such as bidirectional ad-hoc networking, In-Network Programming over an ad-hoc network, and power management. These shortcomings preclude many research and course development endeavors. Any laboratory development must take into account the limitations and either design around them or create their own custom solution. Hopefully many of TinyOS' limitations will be addressed with the next major release of TinyOS, i.e. TinyOS 2.0. Also, the future Crossbow 802.15.4/Zigbee compatible platform promises to offer many more IEEE standard related research opportunities than the mica2 platform offers.

The lack of working features in TinyOS 1.1.7 is not necessarily a flaw in its design; it is inevitable because of the immaturity of the state of the art technology. The problem with TinyOS exists in its marketing; there are many features which TinyOS claims to offer, yet in reality the features have not been fully implemented. TinyOS development seems to not focus on a given specific area but instead on many at once, this is a result of individual projects which do not have a group focused on them. The result is a number of partially implemented features which may or may not ever be fully implemented or bug free.

To address some of TinyOS 1.1.7's limitations, workarounds and custom solutions were developed. These workarounds include scripts for TinyOS and CygWin configuration, a script for data extraction from Xlisten, and a program to log data from

the base station mote. The scripts and custom solutions are designed to workaround the limitations of TinyOS while permitting the student to utilize various features of TinyOS.

The purpose of the TinyOS laboratories is to enable a student the ability to successfully write mote applications, by utilizing TinyOS, by the end of the four, one week-long laboratories. This is achieved by creating laboratories which require the student to examine TinyOS source code, and TinyOS documentation, in order to derive a solution to each of the experiments. Each laboratory builds upon the core knowledge of the pervious laboratory. Unfortunately, due to TinyOS' limitations certain features are restricted to only one or two laboratories. But the fact the features are not used in other laboratories does not detract from the learning experience. With any state of the art technology the researcher and the developer must learn its features along with its limitations.

The first laboratory builds the core knowledge about TinyOS, nesC, and the TinyOS simulator, TOSSIM. The second laboratory continues building upon the core knowledge of TinyOS by introducing the students to the mica2 platform, mica sensor board, and program reconfigurability. The third laboratory builds upon the core knowledge base from the previous laboratories by introducing the students to intricacies of the mica2 hardware and ad-hoc networking. The final laboratory builds upon the knowledge base from all of the previous laboratories and requires the students to build their own application based on either a proposed set of requirements or a set of requirements which they propose.

a) Future Lab development

Currently the next version of TinyOS, known as TinyOS 2.0, is in its prerelease stage. This next version of TinyOS was not developed using pre-existing TinyOS code. TinyOS 2.0 was created from a clean slate by the TinyOS working group, which consists of companies and institutions from three continents. TinyOS 2.0 is not backwards compatible with the previous version of TinyOS. TinyOS 2.0 promises to offer better platform portability via its hardware abstraction architecture, i.e. "the 2.0 Hardware Abstraction architecture" [4].

The next generation of motes supports the IEEE 802.15.4 standard. The IEEE 802.15.4/Zigbee standard is designed for low-power, low-cost, wireless sensor network applications. Utilizing a mote which supports this standard ensures that the mote can interact with other 802.15.4/Zigbee compliant devices.

Due to the next version of TinyOS, and the standardization of wireless sensor network communication, lab development should wait until the release of TinyOS 2.0 and until IEEE 802.15.4/Zigbee compliant motes become available. Any laboratory developed before these technology releases will most likely require redevelopment.

Unfortunately, the support programs included in this thesis most likely will not operate correctly, if at all, with the next version of TinyOS. However, the structure of the four lab modules provides a viable model for any future lab development. The learning objectives of each of the laboratories should remain unchanged. But the content of the laboratories must be updated in order to provide a similar learning experience. The source code will require changes as necessitated by TinyOS 2.0. The algorithm used in the 4th laboratory may require changes in order to properly

handle tone detection; the hardware/software bug addressed in lab 4 may not exist in TinyOS 2.0 or on the 802.15.4/Zigbee compliant platform.

Other non-TinyOS approaches offer an insight into the required development time and cost associated with TinyOS and Crossbow technology. Cal Poly's Computer Science department is currently developing a short range wireless sensor network which utilizes custom motes and a custom device driver to coordinate the movement of a robot's legs [22]. The device driver offers both hardware abstraction and functions, but does not provide scheduling. Development with TinyOS would be more involved due to TinyOS' constraints. Also, utilization of Crossbow technology would require more financial investment which is unnecessary when an in-house solution will suffice.

References

1. Crossbow Technology; accessed 28 July 2005, available from http://www.xbow.com/.
2. "MOTE-VIEW 1.0 User's Manual". Crossbow Technology, Doc. #7430-0008-02 (Rev. A), Mar. 2005 [software manual online]; accessed Sep. 1st 2005; available from http://www.xbow.com.
3. Turon, Martin. "MOTE-VIEW 1.0 Quick Start Guide." Crossbow Technology, Application Note 7410-0008-02 Rev. A, 1 Mar. 2005 [Application Note online]; accessed Sep. 1st 2005; available from http://www.xbow.com.
4. TinyOS. University of California, Berkeley (25 Jul. 2005); accessed 29 July 2005; available at http://www.tinyos.net/.
5. Sun Developer Network; accessed 15 Apr. 2005, available from http://java.sun.com/.
6. Sintes, Tony. "Events and Listeners: How do you create a custom event?" JavaWorld Aug. 2000 [magazine online]; accessed 4 Apr. 2005; available from http://www.javaworld.com/.
7. Culler, David; Estrin, Deborah; Srivastava, Mani. "Overview of Sensor Networks." IEEE Computer Society, Vol. 91, No. 8 (August 2004); 41-49.
8. Martinez, Kirk; Hart, Jane; Ong, Royan. "Sensor Network Applications." IEEE Computer Society, Vol. 91, No. 8 (August 2004); 50-56.
9. Hemingway, Bruce; Brunette, Waylon; Anderl, Tom; Borriello, Gaetano. "The Flock: Mote Sensors Sing in Undergraduate Curriculum." IEEE Computer Society, Vol. 91, No. 8 (August 2004); 72-78.
10. Thorn, Jeff. "Deciphering TinyOS Serial Packets." Octave Technology, Octave Tech Brief #5-01, 10 Mar. 2005 [tech brief online]; accessed Apr. 29th 2005; available from http://www.octavetech.com/solutions/pubs.html.
11. Wikipedia; accessed 2 July 2005; available from http://en.wikipedia.org/.
12. Nico, Kim. "TinyOS MultiHop Routing," 25 July 2005, professional email (25 July 2005).
13. "SmartRF CC1000 Datasheet". Chipcon, Chipcon AS SmartRF CC1000 Datasheet (Rev. 2.2), 22 Apr. 2004 [datasheet online]; accessed Jun. 10th 2005; available from http://www.chipcon.com.
14. Sudha, Krish. "XNP/INP help," 13 August 2005, professional email (14 May 2004).
15. "ATmega128(L) Complete Datasheet". ATmel, 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash, ATmega128 / ATmega128L (Rev. 2467M–AVR–11/04), Nov. 2004 [datasheet online]; accessed Mar. 13th 2005; available from http://www.atmel.com.
16. Larry Wall, Tom Christiansen, and Randal L. Schwartz. Programming Perl, Second Edition. Sebastopol: O'Reilly & Associates, Inc, 1996.
17. Randal L. Schwartz and Tom Christiansen. Learning Perl, Second Edition. Sebastopol: O'Reilly & Associates, Inc, 1997.

18. About Perl and PHP; accessed 7 Sep. 2005, available form http://perl.about.com/.
19. Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Second Edition. Upper Saddle River: Prentice-Hall, Inc, 1988.
20. Tuan Le, Khanh. "Designing a ZigBee-ready IEEE 802.15.4-compliant radio transceiver." RF Design Magazine, Nov. 2004 [article online]; accessed Sep. 17th 2005; available from http://www.rfdesign.com.
21. Borriello, Gaetano. "TinyOS Laboratory development (Cal Poly)," 24 September 2005, email correspondence (24 Sep 2005).
22. Seng, John. Interview by author. San Luis Obispo, CA. 18 October 2005.

Appendix A: Lab Modules (Student Manual)

Lab 1: TinyOS, nesC, and TOSSIM

Estimated Lab time: approximately 4 hours 30 minutes.

Learning Objectives

- To learn how to acquire and install the development environment.
- To become familiar with TinyOS and its constituents.
- To become familiar with event-driven execution.
- To learn how to develop applications in TinyOS with nesC.
- To learn how to simulate applications in TinyOS with TOSSIM.

TinyOS and mica2 mote overview

For more information about the background of TinyOS and the Crossbow motes, read chapter 2 of the "TinyOS Laboratory Development" thesis by Rafael Kaliski.

Development Environment

The development environment used for each of the laboratories consists of the TinyOS 1.1.7 distribution, the X2c Perl script, and the Mote Logger application. The TinyOS 1.1.7 distribution will be referred to as TinyOS 1.1.7.

TinyOS 1.1.7 is used for developing mote applications, programming the motes, and listening to the mote network.

The X2c script is used to parse data output from Xlisten into mote application specific data fields, in addition to a timestamp field which corresponds to the time the data packet was received. The fields which the data are parsed into are hard coded to the RecSensNet application, the RecSensNet application is used in the second laboratory. The parsed fields are stored into a comma separated variable (CSV) file whose name is determined by the user.

The Mote Logger application reads TinyOS UART packets from the serial port and parses the data based on the packet handler. The formatted data are stored to a CSV file whose name is determined by the user, the packet type, and the mote identifier field that is delimited by the user.

TinyOS, X2c, and Mote Logger Installation

To install TinyOS 1.1.7, go to the TinyOS download website http://www.tinyos.net/download.html, download TinyOS 1.1.0 and the CVS snapshot of TinyOS 1.1.7. Install TinyOS 1.1.0 first, and then proceed to install 1.1.7. Follow the directions on the TinyOS download website.

After installation and updating of TinyOS has completed, run the toscheck script in order to ensure that TinyOS is installed correctly. Instructions on how to run the toscheck script can be found at http://www.tinyos.net/begin.html.

The X2c Perl script should have execute permission within CygWin. It is recommended that the X2c Perl script should be located in your Lab 2 directory, this will simplify path issues. The X2c Perl script may be downloaded from http://www.netprl.calpoly.edu/files/phatfile/X2c.pl.

Read the Mote Logger read me file for installation instructions and for more information about the Mote Logger application. The Mote Logger application may be downloaded from http://www.netprl.calpoly.edu/files/phatfile/Mote_Logger.zip.

As a side note, some TinyOS java programs may not compile with JDK/JRE 1.5 or later. Also, using any other version of CygWin, other than the one included with TinyOS, may prevent installation/update of TinyOS. The windows install shield for TinyOS includes a compatible version of java and a compatible version of CygWin.

TinyOS Java applications compilation

To compile all of the TinyOS java applications, for the net.tinyos java package, navigate to the TinyOS "tools/java/net/tinyos" directory and run **make**. This will compile all of the java applications located in the "tools/java/net/tinyos" directory and all of the sub-directories of the aforementioned directory.

Hardware requirement for this laboratory

Each laboratory which programs a mote utilizes the Crossbow MIB510 serial programmer, the mica2 motes, and the mica sensor boards. The MIB510 serial programmer is used to download the code to the connected mote. The MIB510 serial programmer also enables the mote connected to it to act as the base station mote for the network of motes. This hardware is specified in the .bashrc and makelocal files; see Custom file A- 1 and Custom file A- 2. Additional programmer support code is specified in the mib510.extra file shown in Custom file A- 4.

There is no hardware required for this laboratory.

Introduction to TinyOS 1.1.7 and CygWin

Customizing CygWin for TinyOS Programming

There are a number of environmental variables that need to be setup in order to use TinyOS. Environmental variables which will most likely remain the same for most of the invocations within TinyOS can be set in a .bashrc file (The .bashrc file gets run each time the bash shell is opened). A working .bashrc file for TinyOS is listed in Custom file A- 1.

```
#user .bashrc file
#$Id: .bashrc 2004/06/24 rkaliski

#Create the tinyos root environmental variable.
#Simplifies reference to the TinyOS root directory.
export TOSROOT=`ncc -print-tosdir`/..

#Use new makerules instead of the old makerules.
export MAKERULES=$TOSROOT/tools/make/Makerules

#Define the location of the Makelocal file.
export TINYOS_MAKELOCAL=$TOSROOT/apps/Makelocal

#Make MIB510 the default programmer,
#Makelocal should set the com port.
export DEFAULT_PROGRAM=MIB510
```

Custom file A- 1: user .bashrc file, user environmental variables setup

The programmer used in these laboratories is the MIB510, as specified by the DEFAULT_PROGRAM environmental variable, set in .bashrc file as shown in Custom file A- 1. If the MIB510 serial programmer is connected to a comm. port, other than com 6, the comm. port can be specified by either altering the makelocal file or by specifying the extra option, **mib510,COM**<#>, when using the make command. Where <#> is the number of the comm. port that the MIB510 serial programmer is connected to.

In addition to environmental variables, there are several command line options, pertaining to compilation and programming, that can be stored in a makelocal file. If there are command line options for ncc (ncc is the nesC compiler for TinyOS) or other programs which are launched during the make process and these command line options are common for most of the applications, then these command line options can be specified in a makelocal file. A working makelocal file is listed in Custom file A- 2. The makelocal file is located in the directory $TOSROOT/apps, as specified by the TINYOS_MAKELOCAL environmental variable.

```
# Makelocal file
#$Id: Makelocal,v 1.1 2005/09/02 rkaliski

#Add a directory to the nesC compiler's search path.
#Use %T in lieu of the TinyOS ROOT directory specified by
#the TOSROOT environmental variable.
#For example, the line PFLAGS += -I%T/lib/Route will include
#the route library in the nesC compiler's search path.

PFLAGS += -I~/Projects

#Workaround for bug in new makerules,
#adds XNP to the compiler's search path.

#Only includes XNP in the compiler's search path if
#the XNP variable is equal to yes.
ifeq ($(XNP),yes)
 XNP_DIR = $(TOSDIR)/lib/Xnp
 PFLAGS += -I$(XNP_DIR)
endif

#Define the group the node should belong to.
DEFAULT_LOCAL_GROUP = 0x7D

#Set the frequency then mote radio should operate at.
CFLAGS += -DCC1K_DEF_FREQ=915998000

#If not defined,
#identify the com port the MIB510 programmer is connected to.
ifndef MIB510
MIB510=COM6
endif
```

Custom file A- 2: Makelocal, setup TinyOS apps compiler and programmer flags / variables.

The Makelocal file, shown in Custom file A- 2, also has some code necessary for compilation of the In-Network Programming application in laboratory 2.

In order to use the TinyOS make facility, the local makefile must include the TinyOS makerules file, as shown in Custom file A- 3.

```
#Local Makefile for RecSensNetPhoto application
#$Id: Makefile,v 1.0 2005/08/17 rkaliski

#Specifies the mote's component name.
#This should match the mote's configuration
#name.
COMPONENT=RecSensNetPhoto

#Specifies which sensor board the mote will use
SENSORBOARD=micasb

#Includes a directory containing files specific
#to this application.
PFLAGS = -I../RecSensNet #PFLAGS must precede include

#Includes TinyOS' Makerules file.
include /opt/tinyos-1.x/apps/Makerules
```

Custom file A- 3: Local makefile, set up application-specific compiler flags/variables

The local makefile is the makefile located in the mote application's directory.

Both the .bashrc, and the makelocal, files can be copied to your installation of TinyOS. The local makefile is specific to your application, the example local makefile, listed above, can be used as a template for your own local makefile. Read the comments in each of the files to determine what to change for your specific installation and application. If you are not including a directory, i.e. the PFLAGS assignment is empty, comment out the PFLAGS assignment.

Programming motes

Setting the Node ID

The Node ID of a mote is specified at install time. Each node id must be unique and cannot be equal to the TinyOS broadcast address 0xFFFF or the TinyOS UART address 0x7E. The Node ID field is 2 bytes in size, up to 65,534 different motes can exist in the same Group.

Setting the Group ID

In order to set the group id of all motes which are programmed, set the DEFAULT_LOCAL_GROUP environmental variable in the makelocal file, as shown in Custom file A- 3. The Group ID field is 1-byte in size; up to 256 different groups may exist on a single communications channel, which corresponds to a set radio frequency.

The Group ID field can be used to separate data from motes running on the same communications channel. Other motes operating in the same channel cannot receive data from motes which belong to a different group.

Tuning the MICA2 series (MICA2 / MICA2DOT) radio

In order to tune the mica2 radio you must know the operating frequency range of the mica2 series mote. If you are unsure what the operating frequency of the mica2 series mote is see the "How to determine the operating frequency range of a MICA2 or MICA2DOT mote" document located in the TinyOS doc folder under the name mica2freq.html.

The MICA2 radio can be tuned at compile time or at runtime. To determine what the radio frequency will be, given a desired radio frequency, use the *channelgen* program. To run the *channelgen* program, type "channelgen" in a CygWin shell with the desired frequency as an argument, the actual frequency, and offset from the desired frequency, will be displayed by the *channelgen* program.

To tune the radio at compile time, change the –DCC1K_DEF_FREQ parameter, in the makelocal file, to the desired frequency. To tune the radio at runtime, wire the CC1000Control module to your application, and call the CC1000Control.TuneManual(uint32_t) command with the desired frequency in Hz; the actual frequency of the channel is returned. To avoid inter-channel interference, a minimum frequency spacing of 150 KHz is suggested.

Compiling / Installing the mote application

Once TinyOS and CygWin have been setup and customized, the mote application can be compiled for the pc platform via the following command; this command must be executed in the mote application's directory:

make pc

The mote application can be compiled for the mica2 platform via the following command; this command must be executed in the mote application's directory:

make mica2

The mote application can be optionally compiled, and downloaded, to a mote connected to the MIB510 serial programmer via the following command, executed in the mote application's directory:

make mica2 (**re**)**install.<**moteid#**>**
The moteid# is the node id you want the mote to have. If install is specified, then the application will be (re)compiled before it is uploaded. If reinstall is specified, then the application will be uploaded. Note: If the application has not been compiled, then reinstall will fail.

TinyOS libraries, sensor interfaces, and mote/ generic interfaces

The TinyOS core source code can be divided into four different sections; libraries, sensor interfaces, mote interfaces, and generic interfaces. The TinyOS

libraries provide additional features not necessary to the operation of a basic mote application. The sensor interfaces provide the necessary interfaces required to interact with each of the sensors. The mote interfaces provide the access to components specific to the mote, for example the mica2 radio stack. Finally, the generic interfaces provide access to features which apply to many motes, for example the leds. The generic interfaces operate at a higher level of abstraction then the mote specific interfaces.

The micasb sensor interfaces are located in the TinyOS "tos/sensorboards/micasb" directory. The TinyOS libraries are located in the TinyOS "tos/lib" directory. The generic TinyOS interfaces are located in the TinyOS "tos/interfaces" directory. The mica2 radio, ADC interfaces, and the mica2 specific components are located in the TinyOS "tos/platform/mica2" directory. The generic TinyOS components are located in the TinyOS "tos/system" directory.

These directories, and their subdirectories, provide the resources necessary to build the mote applications in each of these laboratories. If a component exists in both the system and mica2 directories, refer to the mica2 version.

Introduction to NesC

Go thorough lessons 1 to 4 in the TinyOS tutorial, http://www.tinyos.net/tinyos-1.x/doc/tutorial/, for a gentle introduction to the nesC programming language. Lesson 1 provides an introduction to TinyOS and nesC. This lesson covers the differences between tasks, events, and commands. This lesson also discusses how components are connected together. Lesson 2 demonstrates a simple event-driven sensor application. This lesson discusses a basic sensor application's code. This lesson also covers the timer component and the concept of parameterized interfaces and their use. Lesson 3 introduces tasks and their use in "background" data processing. Lesson 4 discusses the hierarchical decomposition of components and how to use radio communication.

Introduction to TOSSIM, TinyViz, serial forwarder, and Tython

Read Lesson 5 of the TinyOS tutorial to become familiar with TOSSIM and TinyViz.

Read the Getting Started section of the Tython manual, located in the TinyOS doc directory under the Tython subdirectory, named manual.html to become familiar with the basic Tython commands. Read nido.pdf, located in the TinyOS doc directory, to become familiar with TOSSIM and the serial forwarder.

For a quickstart guide for TOSSIM and Tython, read the README file located under the TinyOS directory "/tools/java/net/tinyos/sim/".

Surge application description

The surge application takes light sensor readings and transmits them via the ad-hoc network. The surge application also supports putting individual nodes

into sleep mode, focusing on individual nodes, and setting the sampling rate of all nodes which are within the broadcast range of the root node, i.e. the base station node. When a node is in sleep mode it does not transmit any sensor reading packets and its green and yellow leds will turn off, if they are on. When a node is woken up, its sampling rate is set to 2.048 seconds / sample. When a node is in focus, its sampling rate decreases to 1 second per sample and it begins to chirp, other nodes sampling rate will also decrease to 1 second per sample. When the focus is canceled, i.e. no node is focused on, the sampling rate will revert to the normal 2.048 seconds per sample, and the chirping node will stop chirping.

The accompanying surge java application, shown in Figure A- 2, enables visualization of the ad-hoc network topology. The surge java application also allows the user to send focus/unfocus, sleep/wakeup, and timer update messages as well.

Procedure: Surge Application and Tools

Notes: Using a USB serial comm. port cable may cause your computer to blue screen if ran for a prolonged period of time.

1) Compile the **surge** application for the pc platform.
2) Compile the **surge** java application, the serial forwarder application, and Tython, if not already compiled.
3) For this experiment ensure the environment variable DBG is set to none, otherwise the simulation may run very slow.
4) In a CygWin window, run **Tython** with no console and 5+ motes in the simulation; use the options nosf, noconsole, and gui. The TinyViz simulator will open with the simulation stopped.
5) Load the following TinyViz plug-ins:

    a. Radio links – this plug-in will show if a mote is broadcasting a message or sending a message to a specific mote.
    b. Radio model – this plug-in will enable you to specify the radio model. For ad-hoc applications, like **surge**, the location of the mote does matter when using the empirical radio model.

6) Organize the motes in a structure which uses ad-hoc communication. To assist in mote organization a grid can be displayed by clicking on the button with a grid image on it. In ad-hoc communications, the destination node[*] is not necessarily within direct communications range of a given node, yet it is within indirect communications range of a given node, i.e. an intermediate node will be used for communication with the destination node. Note: The ad-hoc routing component provided in TinyOS 1.1.7

[*] The terms node and mote are synonymous. A mote is referred to as a node when the discussion pertains to ad-hoc routing.

only support unidirectional communication with the base station as the implicit destination node.

7) The serial forwarder application enables data forwarding from a serial port to a network port. In another CygWin window, run the serial forwarder. If not already set, set the server port field is set to **9001**. Set the Mote Communications field to **TOSSIM-serial**.

8) In the serial forwarder click **Start Server**, this will open the serial port and the network port.

9) In another window, start the **surge java application**. Information on how to start the surge java application can be found in the README file located in the TinyOS surge java application directory "tools/java/net/tinyos/surge". Be sure to set the group id field to the group id of the motes being simulated.

10) Click the "Play" button in TinyViz to start the surge mote application.

11) In the surge java application, click the "Start root beacon" button, and then click the "Send wakeup" button.

12) After the motes have established their network, you should see a **TinyViz** screen similar to Figure A- 1. You should also see **Sensor Network Topology** screen similar to Figure A- 2.

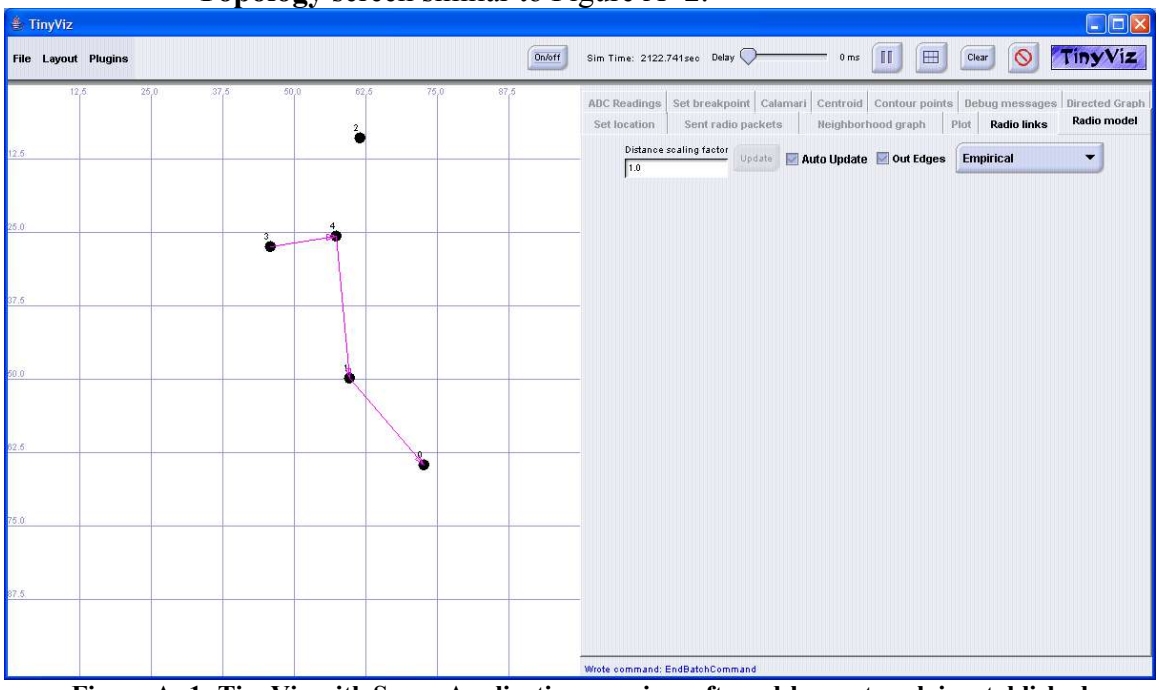


Figure A- 1: TinyViz with Surge Application running, after ad-hoc network is established

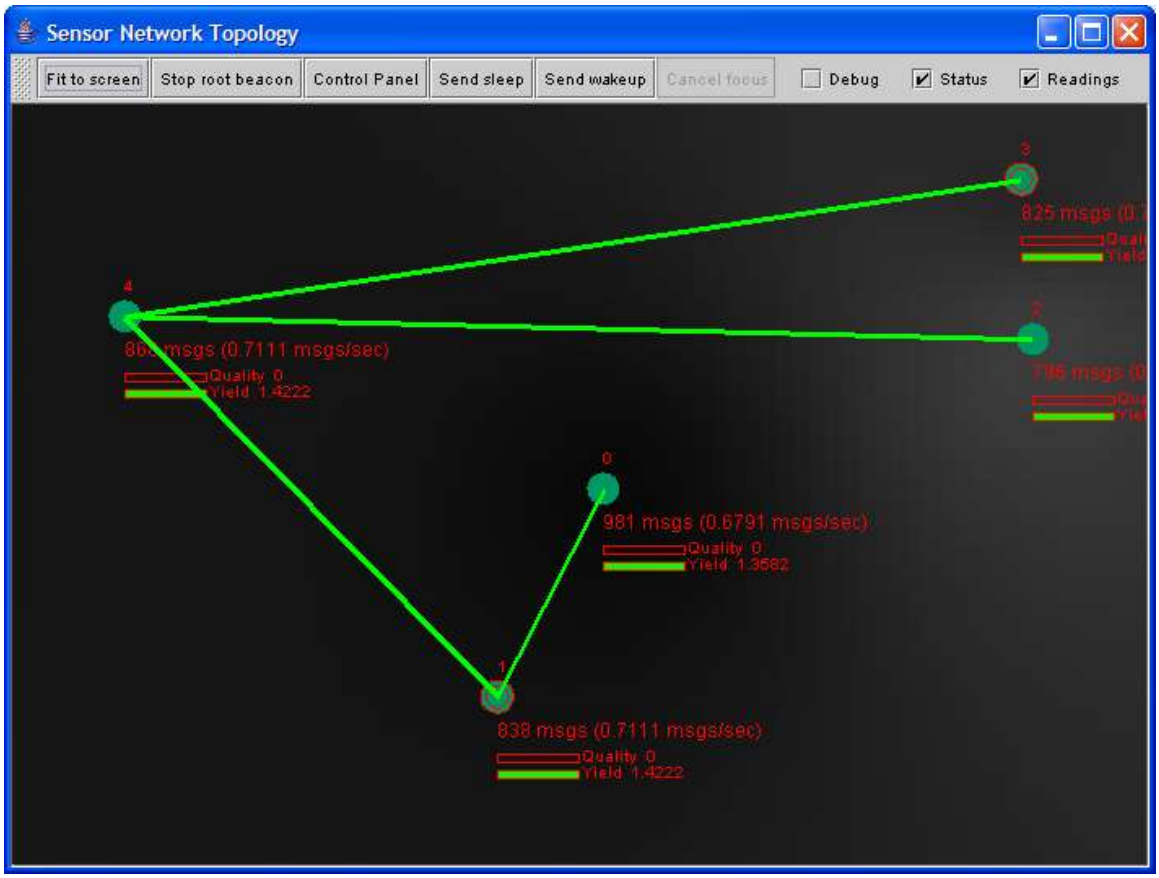


Figure A- 2: Surge java application, after ad-hoc network is established

Note: Although the surge mote application supports chirping, when the mote is in focus, the simulation does not support audio output to a sound device.

13) Test the various commands and note the behavior resulting from activating them.

14) Examine the surge source code, i.e. Surge.h, Surge.nc, SurgeM.nc, and SurgeCmd.h, and determine how the TinyOS components are interconnected and interact.

Conclusion

Discuss what you learned from this lab and how it applies to real-world applications. The discussion should focus on TOSSIM and sensor networks. The discussion should not just summarize the procedure or what you did.

Lab 2: Crossbow Motes, Programming, In-Network Programming, and Listening

Estimated Lab time: approximately 4 hours.

Learning Objectives

- To become familiar with Crossbow mica2 motes and the mica sensor board.
- To learn how to program mica2 motes.
- To learn how to read data from the communications port provided by the serial programmer.

Hardware requirements for this laboratory

This laboratory utilizes the MIB510 serial programmer, 3 MPR400CB mica2 motes, and 2 MTS310CA "micasb" mica sensor boards. The sensor boards should be connected to the motes not directly connected to the serial programmer. The mote connected to the serial programmer is considered part of the base station.

The micasb sensor board is specified in the application makefile; see Custom file A- 3.

Mica2 communication

The GenericComm component provides a means of sending packets from one node to one of its direct neighbors. The GenericComm component also provides for bidirectional communication between neighbors. If a received message's destination id does not match the node's id, then the message is dropped by the Active Message (AM) Layer of the Mica2 radio stack. If the CRC for the received message does not match the CRC in the message, the message is dropped at the AM Layer.

The GenericCommPromiscuous component provides a CommControl interface which enables a mote to configure whether or not a received message is dropped at the AM layer due to failed CRC or different destination id.

The MultiHopRouter component provides a means of collecting data over an ad-hoc network. Since the MultiHopRouter component does not support naming, only unidirectional communication is supported and the destination of all sent packets is the root node. The MultiHopRouter uses a data aggregation routing scheme, i.e. many nodes to a single "root" node.

Other components for radio communication exist, yet the GenericComm, GenericCommPromiscuous, and MultiHopRouter components will suffice for most of the laboratory experiments.

The above components do not permit inter-group communication between nodes. If a received message's group id does not match the group id of the node, the message is dropped at the AM layer. Motes running compatible, if not the same, applications should reside in the same group. Motes running incompatible applications should reside in different groups. By utilizing the group id field motes running different, incompatible, applications can co-exist on the same communications channel.

Every mote is configured at run-time, or compile-time, to communicate on a specific communications channel. A communications channel is a specific frequency used for communication between nodes. Usable communications channels have a large enough frequency separation from adjacent channels to prevent interference. The "Tuning the MICA2 series (MICA2 / MICA2DOT) radio" section of the previous lab describes how to determine what frequency a mote can operate at and how to configure the mote to operate at a given frequency.

The UART can be used for communications, with a computer or gateway, by utilizing any of the above components. If a mote wants to send a message over the UART, the destination address (node id) of the message needs to be set to 0x007E. When any of the aforementioned components is used, with the SendMsg interface, the message will be sent over the UART versus the radio. Incoming messages, sent to the mote over the UART, will utilize the component connected to the ReceiveMsg interface.

The SendMsg and ReceiveMsg interfaces are examples of parameterized interfaces. In order to connect a component to any of these interfaces, or any parameterized interface, the component must be connected to a given instance of the interface.

For more information on the aforementioned components/interfaces, see:

1) The SendMsg, ReceiveMsg, Intercept, and CommControl interfaces located in the TinyOS interfaces directory. The SendMsg and ReceiveMsg interfaces are provided by the GenericComm, GenericCommPromiscuous, and MultiHopRouter components. The CommControl interface is provided by GenericCommPromiscuous. The Intercept interface is provided by the MultiHopRouter component.
2) The MultiHopRouter configuration located in the TinyOS lib/route directory. This component provides access to the ad-hoc network. In order to use this component, this component's directory must be in the search path of the compiler, see the "Customizing CygWin for TinyOS programming" section of lab 1.
3) The GenericComm / GenericCommPromiscuous configurations located in the TinyOS system directory. These files define what interfaces are provided for the communications component that the main mote application can use.

In-Network Programming

In-Network Programming (INP) enables motes, including the mote's group id and node id, to be reprogrammable over the network. In order to use INP:

1) The motes must be "wired" to the Xnp module in TinyOS
2) The motes must have the INP in-system programmer (ISP) installed.
3) An Xnp compliant application to send the mote application to the base station for program transmission.

The relevant **Xnp** interface commands, from Xnp.nc under the TinyOS lib/Xnp directory, and events prototypes are listed below, along with the reason for each of these commands and events:

Commands:

```
command result_t NPX_DOWNLOAD_ACK(uint8_t cAck);
```

- Acknowledgement from Mote application that indicates whether the network programming download operation should proceed. The EEPROM must be released before a cAck of SUCCESS should be sent.
- cAck = SUCCESS - The program code download can begin (The program code is downloaded to the EEPROM). The mote's main application should discontinue its current program until the download is completed.
- cAck = FAIL - The mote application cannot proceed with the downloading of the application.
- This command should be called within the NPX_DOWNLOAD_REQ event.

```
command result_t NPX_SET_IDS();
```

- Sets the mote's group and node ids. When the mote is programmed directly from the programmer, the mote's group and node ids are downloaded to a reserved section of the EEPROM.
- Network programming requires that the Mote's Group and Node ID be restored in code space.
- This command should be called during initialization of the mote.

Events:

```
event result_t NPX_DOWNLOAD_REQ(uint16_t wProgramID, uint16_t
wEEStartP, uint16_t wEENofP);
```

- This event is signaled when an in-network program download request message has been received.
- The Program ID, the planned EEPROM start page, and planned number of EEPROM pages are arguments for this event.
- If the mote's application chooses to grant the download request, the mote's main application should discontinue its operation until the download completes (see the NPX_DOWNLOAD_ACK command for more information regarding acknowledgements).

```
event result_t NPX_DOWNLOAD_DONE(uint16_t wProgramID, uint8_t
bRet, uint16_t wEENofP);
```

- This event is signaled when the download operation has completed.
- The Program ID, the success of the download, and the actual number of EEPROM pages used are passed in as arguments for this event.
- If the download succeeds, bRet will be true, otherwise bRet will be false.
- The user program must ensure the integrity of the downloaded program, i.e. the user program must ensure that it does not overwrite the program in the EEPROM by not writing to any of the memory locations where the program is stored. The EEPROM pages used for storing the program is given by the Start Page (wEEStartP) and the Number of Pages (wEENofP) values.

Due to a bug in the new makerules, the default In-Network Programming option does not install the In-Network Programming bootloader, i.e. the mica2 INP bootloader. A quick fix is to change the mib510.extra file located under the TinyOS directory "tools\make\avr". The mib510.extra file should contain the inpisp upload code given in the mib510.extra file listed below.

```
#-*-Makefile-*- vim:syntax=make
#$Id: mib510.extra,v 1.2 2004/04/24 09:33:37 cssharp Exp $
#Added inpisp upload code.  2005/08/14 rkaliski

ifeq ($(MIB510),)
$(error MIB510 must be defined, try "make $(TARGETS) help")
endif

PROGRAM = mib510
PROGRAMMER_FLAGS = -dprog=mib510 -dserial=$(MIB510)
$(PROGRAMMER_PART) $(PROGRAMMER_EXTRA_FLAGS_MIB)

program: FORCE
      @echo "    installing $(PLATFORM) binary using mib510"
      $(PROGRAMMER) $(PROGRAMMER_FLAGS) --erase --upload
if=$(INSTALL_SREC)

#inpisp upload code
ifeq ($(XNP),yes)
      @echo "    Uploading Bootloader"
      $(PROGRAMMER) $(PROGRAMMER_FLAGS) --upload if=$(BOOTLOADER)
endif
```

Custom file A- 4: mib510.extra, enable TinyOS inpisp program download

The changes to the makelocal file, see Custom file A- 2, allows any Xnp enabled application to successfully compile. To compile an Xnp enabled application the **XNP, yes** option must be specified as an argument to make.

If the **XNP, yes** and the **install/reinstall** option is specified as argument to the make command, then the above mib510.extra fix will upload the mica2 / mica2dot inpisp to the mote after the main code has been uploaded.

Even if the Xnp module is "wired up" to the mote application, a boot loader must be installed to write the program to the flash memory. The INP bootloader will load the code from the EEPROM into the flash when a reprogram request is received. If the INP bootloader is not installed, when the reprogram request attempts to activate the boot loader, the mote will crash/reboot. To install the INP bootloader type the following command:

> **make mica2 (re)install.<**moteid#**> XNP,yes**
> The moteid# is the node id you want the mote to have. If install is specified, then the application will be (re)compiled before it is uploaded. If reinstall is specified, then the application will be uploaded. Note: If the application has not been compiled, then reinstall will fail.

The INP bootloader must be reinstalled each time the mote is programmed directly by a programmer. The INP bootloader will persist in the motes memory when the mote is reprogrammed over the network.

The TinyOS application, Xnp, is an In-Network Programming application which enables the user to download code to, as well as reprogram, the listening motes.

The Xnp application can be used with, or without, the serial forwarder. In order to use the Xnp application with the serial forwarder, ensure that the server port field is set to 9001 and the Mote Communications field is set to serial@<COM #>:mica2, where <COM#> is the communications port number that the MIB510 serial programmer is connected to. The serial forwarder is the default means of communication for the Xnp application. In order to use the Xnp application without the serial forwarder, ensure that the **MOTECOM** environmental variable is set to the correct serial port and baud rate. For example, if the mica2 mote were the mote you wanted to communicate with via the serial port, COM 6, the below command would set the **MOTECOM** environmental variable:

> **export MOTECOM=serial@COM6:mica2**
> *mica2* is equivalent to the mica2 baud rate *57600*, either can be specified.

To run the Xnp application type the below command:

> **java net.tinyos.xnp.xnp &**

This will launch the Xnp application in background mode, see Figure A- 3. The Xnp application is covered in more detail in the "Mote In-Network Programming User Reference" located in the TinyOS doc directory under the name Xnp.pdf.

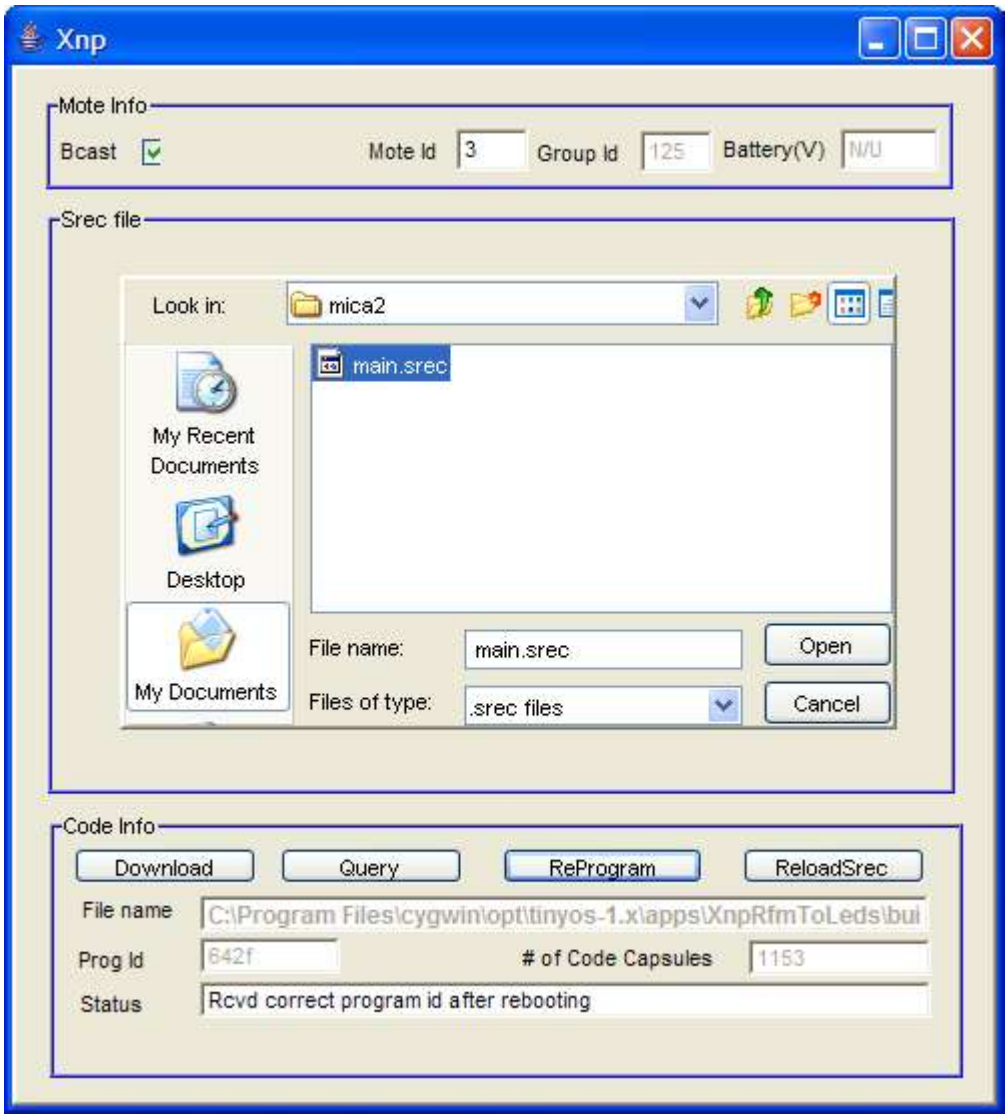


Figure A- 3: Xnp java application

Xnp Pitfalls:

Some of the pitfalls listed below are extracted from the XnpM.nc, Xnp.nc, Xnp.h, and XnpC.nc files located in the TinyOS lib/Xnp directory.

1. In-Network Programming does not work across ad-hoc networks; only the nodes within range of the base station will be reprogrammed.
2. The mote's main application may lose data if it utilizes the EEPROM for data persistence / storage. Although the starting EEPROM page to store the new program is always the same, the number of pages required to store the new program can change.

3. The Group ID of the mote should not be 255, nor should the Node ID of the mote be 65,535. Note: both of these are the maximum values for both the Group ID and the Node ID. The Group ID and Node ID values mentioned will cause Xnp to assume that the EEPROM was erased at the location considered reserved for both the group and node ids. As a result, both the group and node IDs will default to those stored in the code space.
4. The Active Message (AM) ID 47 is reserved for Xnp messages; if any of the nodes are configured for Xnp, the user should not use this ID for their own custom packet format.
5. Because Xnp uses the EEPROM, there is a potential problem if data logging is used. Xnp uses a static EEPROM ID of 47 for the parameterized EEPROM write interface, the Logger component uses a unique number based on the string "EEPROMWrite" for the EEPROM write interface. The potential problem exists due to the fact that the unique function only guarantees a unique number for a given unique string.

   In order to avoid this potential problem, ensure that the main mote application releases the EEPROM before yielding to the Xnp download.

   If the EEPROM is not released before replying SUCCESS to the NPX_DOWNLOAD_REQ, the download will most likely fail.

Reading data from the Base Station with Xlisten

Xlisten listens to the data being transmitted over the UART to the host computer. The data can be displayed as in raw, ASCII, or converted format. The data can also be recorded to a file, forwarded to another computer, or read from another computer. The Xlisten options and usage information is reproduced below for convenience.

```
Xlisten Ver:$Id: Xlisten.c,v 1.16 2004/09/30 21:23:56 mturon Exp $
Using params: [help]

Usage: Xlisten <-?|r|p|c|x|l|d|v|q> <-l=table>
               <-s=device> <-b=baud> <-i=server:port>
   -? = display help [help]
   -r = raw display of tos packets [raw]
   -a = ascii display of tos packets [ascii]
   -p = parse packet into raw sensor readings [parsed]
   -x = export readings in CSV spreadsheet format [export]
   -c = convert data to engineering units [cooked]
   -l = log data to database or file [logged]
   -d = debug serial port by dumping bytes [debug]
   -b = set the baudrate [baud=#|mica2|mica2dot]
   -s = set serial port device [device=com1]
   -i = use serial forwarder input [inet=host:port]
   -o = output (forward serial) to port [onet=port] -!TBA!-
   -h = specify header size [header=offset]
   -t = display time packet was received [timed]
   -q = quiet mode (suppress headers)
   -v = show version of all modules
```

Xlisten pitfalls

1. The parse packet into raw sensor readings, the convert data to engineering units, the export readings, and log data options only support the surge application packet handler. If any other packet handler, other than the surge packet handler, is used the following error will be displayed (The export readings option will not display any error, but nor will it display useful data.):

   ```
   error: no packet handler for tos type 0x??
   ```

   ?? is the hex value of the packet handler of the received packet.

2. The MOTECOM environmental variable is not used by this application. The connection to the base station, whether over a tcp or serial connection, must be explicitly specified.

Xlisten to CSV (X2c)

Even though Xlisten is incapable of logging or decoding unknown packet types, Xlisten can still be used for this laboratory. The X2c.pl Perl script enables us to utilize Xlisten to gather data from a single mote packet type and separate the data into separate fields, based on the RecSensNet application used in this laboratory. The format of the fields is given in Table A- 1.

Table A- 1: X2c output field format

Time	Photo / Temp	Microphone	Mote ID	Packet ID

The X2c script requires a file containing the output of Xlisten. This can be generated by using the I/O redirection operator > with Xlisten, when called on the command line.

If the X2c script has its executable property set, you can run it from the same directory you are located by typing:

./X2c.pl
./ refers to the current directory, this can be replaced with the path to the X2c script.

If the X2c script does not have its executable property set, you can run it from the same directory by typing:

perl X2c.pl
The path to the X2c script can be specified after the perl command on the command line.

Lab Pitfalls

1. On the MTS310CA sensor board, i.e. the micasb, the temperature, and the light sensor share the same ADC channel. Ensure that only one sensor is on at a time, otherwise the ADC data will be invalid.
2. A read cannot take place on the microphone immediately after the microphone sensor is activated. After microphone activation there is a certain time which the mote application must wait before it can retrieve valid data from the microphone sensor. Since the microphone sensor has a dedicated ADC channel, the microphone sensor's can either be started when the main mote application starts or it can be wired directly to the main component's stdcontrol interface.
3. Xnp does not recognize multiple motes downloading at once, i.e. multiple motes could be downloading the program and all but one could fail, if the one that succeeded sends the correct program ID and the other motes fail to respond to the query performed after download then Xnp will go unaware of the failed download. Ensure a manual query is performed on all motes after download has finished and the correct program id has been received. The mote application program code can be written such that this query is not always necessary.
4. Due to the limited range of the motes; ensure that each mote is within sending and receiving distance of the base station mote. As the motes get further away the packet drop rate will increase. Ensure the motes have different sensing environment so the data received will be information rich, versus near identical.
5. The time format for Xlisten provides accuracy in order of seconds; ensure this is reflected in your results.

Reconfigurable sensor network application description

The RecSensNet application is an Xnp enabled application which utilizes different sensors based on the configuration of the application.

Each mote is connected to the Xnp component and can be reprogrammed over-the-air.

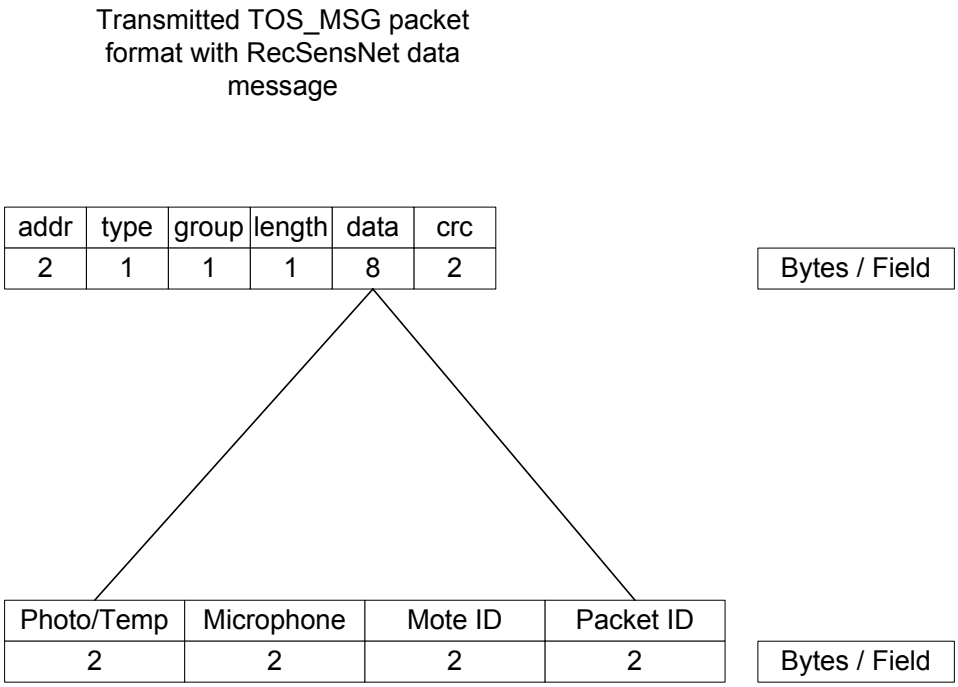
This laboratory utilizes both the microphone and either the light or the temperature sensor, depending on the configuration. The interval between initiating sensor ADC conversion is determined by the constant `TIMER_INTERVAL`, which is in terms of milliseconds. The default sampling interval is 1 second.

The leds are used to display status information pertinent to the mote. The pertinent led behavior is shown in Table A- 2. Each of the leds' behavior is defined by Xnp once a download, query, or reprogram request has been received. When Xnp has finished, the leds behavior is dictated by the main mote program.

Table A- 2: RecSensNet Led description

Led	Display
Green	• On – On duration is equal to the upper 8 bits of light/temperature sensor reading in terms of milliseconds. Each ADC channel, and sensor reading, has 10 bits of resolution.
Red	• Toggling – The mote has received a message from the base station.
Yellow	• Toggling – The mote has sent sensor data to the base station or over the serial communications line.

The structure for the transmitted RecSensNet packet is shown in Figure A- 4.



Type – Type field, indicates which packet handler should be used to decode the packet. The RecSensNet packet types are:

- AM_RSNMSG_PHOTO, used for node to base transmission of photo sensor packets, has the value 0x0E.
- AM_RSNMSG_TEMP, used for node to base transmission of temperature sensor packets, has the value 0x0F.
- AM_RSNBASEMSG, used for base to node transmission of base message packets, has the value 0x10.

Photo/Temp – Photometer or Temperature reading, based on configuration of RecSensNet application.
Microphone – Raw microphone reading.
Mote ID – ID of the sending mote.
PacketID – Packet ID of the sending mote.

Figure A- 4: RecSensNet transmitted packet structure, with pertinent field descriptions

Procedure: Reconfigurable sensor network

1. The RecSensNet module, *RecSensNetM*, uses the microphone and another sensor, connected via its configuration, to collect data from the environment. Several sections of the *RecSensNetM* code are missing, fill in the missing sections of code to complete the module.
2. The *RecSensNetPhoto* configuration utilizes the *RecSensNetM* module to create an application which utilizes the microphone and light sensors. Fill in the missing sections of code in the *RecSensNetPhoto* configuration.

3. The *RecSensNetTemp* configuration utilizes the *RecSensNetM* module to create an application which utilizes the microphone and temperature sensors. Fill in the missing sections of code in the *RecSensNetTemp* configuration.
4. Program a mote with the *TOSBase* application. The *TOSBase* programmed mote should be connected to base station via the serial programmer board. Ensure that SW2 is in the off position, otherwise the mote's transmit line will be disabled, see Figure A- 5.

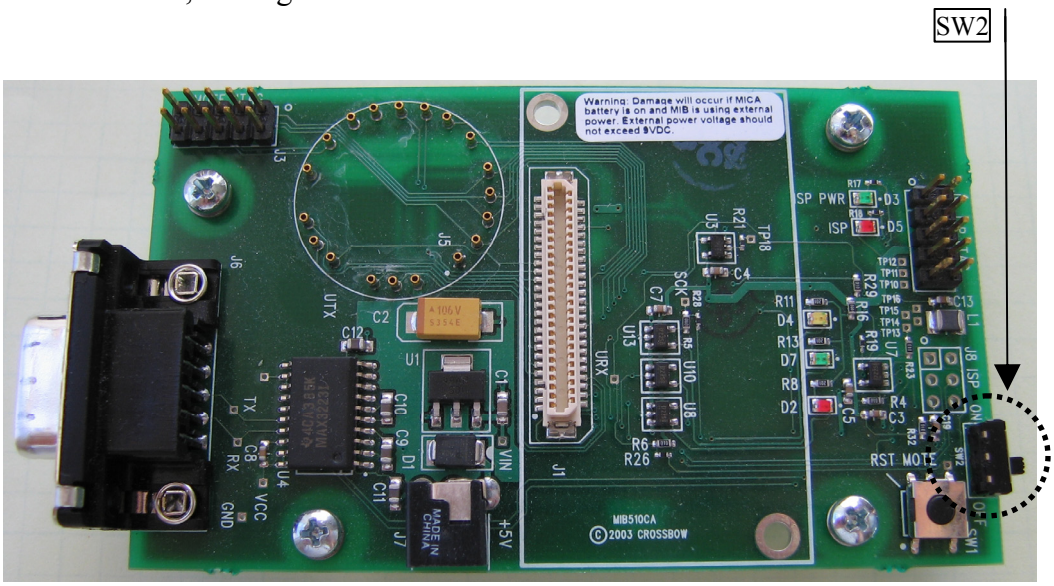


Figure A- 5: MIB510 serial programmer (MIB510CA) with SW2 circled

5. Program the other two motes with the *RecSensNetTemp* application, via the serial programmer. The INP bootloader will be uploaded during this programming process and will persist in memory until the mote is reprogrammed by the serial programmer.
6. Connect a mica sensor board to each of the *RecSensNetTemp* programmed motes.
7. Place each of the *RecSensNetTemp* programmed motes in different locations. Ensure that they can still transmit their sensor readings to the base station. Test a mote at a time, at its new location, Xlisten can be used to indicate if packets are being received.
8. Turn all motes on and run the Xlisten application configured with the raw and timed options. Redirect the output of Xlisten to a file.
9. Run the X2c.pl script on the file containing the output from Xlisten.
10. Open the resulting comma separated variable file. Graph the microphone readings for each mote on a graph, each series should correspond to the mote which transmitted the data, use the MoteID field to separate data. Also graph the sensor data in a separate graph from the microphone readings graph. The MoteID field should be included to show which mote the data is originating from.
11. Denote each graph with the application which transmitted the data in the graph.

12. The Packet ID field can be used to compute what the packet error rate is. The time stamp field is used to indicate when the packet was received; this can be used for calculating the packet transmission rate. Compute the approximate packet reception duration, the number of packets transmitted, the number of packets received. Also compute the approximate packet transmission rate **per second** and the error rate, i.e. the percent of lost packets and the percent of packets in error.
13. Reprogram the two *RecSensNetTemp* programmed motes, via In-Network Programming, with the *RecSensNetPhoto* application. The motes are reprogrammable via the Xnp java application. The Xnp application utilizes the *TOSBase* programmed mote and the base station to relay commands to the Xnp enabled motes. Note: The code downloading may take a couple of minutes. If the downloading fails, you will have to download the code to the motes again. To download the code to a failed download mote(s), cycle the affected mote(s) power and download the code via In-Network Programming again. Ensure that each mote has successfully received the new program code; this may require that a query be performed on each mote.
14. Once the code has been successfully downloaded to both of the motes, reprogram them.
15. Repeat steps 7 through 12 with the reprogrammed motes.

Conclusion

Describe what you learned from this lab. Also, describe the various aspects of the lab, their value, and how they relate to real-world engineering problems. Do not just summarize what you did.

Lab 3: *Ad-hoc Networking, Logging data from multiple motes*

Estimated Lab time: approximately 4 hours.

Learning Objectives

- To learn how to harness the power of ad-hoc mote networks.
- To learn how to utilize the Mote Logger application to collect data from the base station.
- To provide a system knowledge base so the user can design, implement, and test their custom application.

Hardware requirements for this laboratory

This laboratory utilizes the MIB510 serial programmer, 3 MPR400CB mica2 motes, and 3 MTS310CA "micasb" mica sensor boards. A sensor board should be connected to each of the motes. Sensor boards connect to the serial programmer by using the header underneath the serial programmer.

The micasb sensor board is specified in the application makefile; see Custom file A- 3.

Introduction to ad-hoc networking with motes

Adjusting the power of the Chipcon® 1000 Radio

The output power level of the mica2 is initialized to 0 dBm, i.e. 0x80, upon mote application startup. The output power level can be adjusted at runtime by wiring up the CC1000 module to the mote application and by calling the SetRFpower(uint8_t) command in the CC1000Control module. The RF output power levels, their corresponding (hex) values, the amount of current consumed is listed in Table A- 3 below. The Chipcon datasheet says, to minimize current leakage the PA_POW register should be set to 0x00 (PA_POW is the register on the CC1000 radio which controls the RF output power, the SetRFPower function is considered a wrapper function to allow a nesC program access to the register).

Table A- 3: Chipcon® CC1000 Output power settings and typical current consumption. Copied from the SmartRF CC1000 Datasheet (Rev 2.2), p. 32 of 50.

Output power [dBm]	RF frequency 433 MHz		RF frequency 868 MHz	
	PA_POW [hex]	Current consumption, typ. [mA]	PA_POW [hex]	Current consumption, typ. [mA]
-20	01	5.3	02	8.6
-19	01	6.9	02	8.8
-18	02	7.1	03	9.0
-17	02	7.1	03	9.0
-16	02	7.1	04	9.1
-15	03	7.4	05	9.3
-14	03	7.4	05	9.3
-13	03	7.4	06	9.5
-12	04	7.6	07	9.7
-11	04	7.6	08	9.9
-10	05	7.9	09	10.1
-9	05	7.9	0B	10.4
-8	06	8.2	0C	10.6
-7	07	8.4	0D	10.8
-6	08	8.7	0F	11.1
-5	09	8.9	40	13.8
-4	0A	9.6	50	14.5
-3	0B	9.4	50	14.5
-2	0C	9.7	60	15.1
-1	0E	10.2	70	15.8
0	0F	10.4	80	16.8
1	40	11.8	90	17.2
2	50	12.8	B0	18.5
3	50	12.8	C0	19.2
4	60	13.8	F0	21.3
5	70	14.8	FF	25.4
6	80	15.8		
7	90	16.8		
8	C0	20.0		
9	E0	22.1		
10	FF	26.7		

To retrieve the RF output power setting call the command GetRFPower(). The command will return the contents of the PA_POW register.

Aside from the potential power savings associated with adjusting the transmission power of the radio, there is an additional benefit for laboratory purposes. By adjusting the power of the radio, ad-hoc networks can be tested within a more constrained space.

Ad-hoc Routing (MultiHop Routing)

TinyOS provides an ad-hoc routing component, named MultiHop. From the user prospective, the ad-hoc routing component offers an extended network but constrains the throughput of data.

Extra features offered to the user, not offered through the standard means of communication, are packet interception and packet snooping interfaces. When the intercept interface is used the mote application is notified when a packet is

received and can choose to forward, or not forward, via the value it returns from the intercept event. When the snoop interface is used the mote application can passively monitor network traffic.

The basic functionality of the multi-hop routing module can be integrated into a mote with only a couple of changes to the mote application. Most of the changes will reside in the configuration file for the application.

The multi-hop routing module is designed around a data aggregation based routing scheme. As of TinyOS 1.1.7, the MultiHop routing module only supports unidirectional communication. The destination of all packets is the base node, i.e. the mote with node ID 0x0000.

Although the multi-hop routing module extends the effective range of the mote network, the multi-hop module doesn't replace GenericComm component. The GenericComm component enables bidirectional communication between nodes within a single hop.

Read the multihop_routing file and the ad-hoc.pdf file under the TinyOS doc directory for MultiHop routing component usage specifics. As a side note, the reserved MultiHop routing Active Message (AM) number is 250.

Pitfalls / Important notes regarding the CC1000 and MultiHop components:

1. If any of the CC1000 components are used in the main mote application, the application must only be compiled with them. This is a result of the fact that TOSSIM does not simulate the ChipCon CC1000 radio communication stack.

   The CC1000 chip and the CC1000 radio stack only exist on the MICA2 and the MICA2DOT motes. In order to only have the CC1000 radio stack specific components, use the code below around the blocks of code that you wish to make CC1000 specific.

   ```
   #if defined(PLATFORM_MICA2) || defined(PLATFORM_MICA2DOT)
   //CC1000 component related code
   #endif
   ```

2. In order to receive any application level messages with other motes in the MultiHop network, the MultiHop receivemsg interface should be wired to GenericCommPromiscuous's receivemsg interface. This interface is parameterized. Use the application's active message ID, i.e. message type, for receivemsg's parameter interface.

   GenericCommPromiscuous allows the motes to receive messages not destined for them, within their group.
3. In order to avoid flooding the network with packets, the main mote application must ensure that it sends packets in an interval greater than 2 seconds. Since the main mote application must utilize a timer, ensure the string given to the unique function is "Timer". Since the multi hop router uses a timer to send its route messages, using the aforementioned string to create a timer instance will avoid the potential conflict between MultiHop's timer instance and the main mote application's timer instance(s).
4. Do not use the Active Message ID 250, this is used for MultiHop routing updates.

Logging data from motes (Mote Logger)

Xlisten has many abilities, yet it does not support custom message formats or multiple files to record to. Since there is not support for custom message formats, the end user cannot create an application and expect to have Xlisten to print fully parsed data to the screen; only raw or ASCII data will be printed to the screen. Since the only data recording ability is through I/O redirection, Xlisten is also incapable of recording messages to different files based on the message's contents. For these reasons the Mote Logger application was created. Read the Mote Logger read me file for installation instructions and for more information about the Mote Logger application.

The data, after interpretation, are stored in a CSV or a set of CSV files. The next section covers the formula(s) which may be used to convert data from an ADC reading to engineering units.

Conversion to Engineering Units

In order to read useful data from a sensor on the mote, the data should be converted to the correct units. The temperature sensor is the only sensor on the MTS310CA which has a conversion formula valid for all applications which utilize the temperature sensor.

The temperature sensor's ADC result can be converted to degrees Fahrenheit via the below equations.

$$1/T(K) = a + b*\ln(R_{thr}) + c*[\ln(R_{thr})]^3$$

$$where:$$
$$a = 1.30705*10^{-3}$$
$$b = 2.14381*10^{-4}$$
$$c = 9.3*10^{-8}$$
$$R_{thr} = R1*(ADC\_FS - ADC)/ADC$$
$$R1 = 10K\Omega$$
$$ADC\_FS = 1023$$
$$ADC = adc\ reading$$

(1)

(Thermistor ADC reading to kelvin conversion)

$$T(F) = (T(K) - 273.15)*(9/5) + 32 \quad (2)$$

(Kelvin to Fahrenheit conversion)

Ad-Hoc network application description

The Multi_Sens_Net application is an ad-hoc network enabled application which collects data from the light, temperature and microphone sensors. Each sensor is read successively. The interval between initiating a sensor ADC

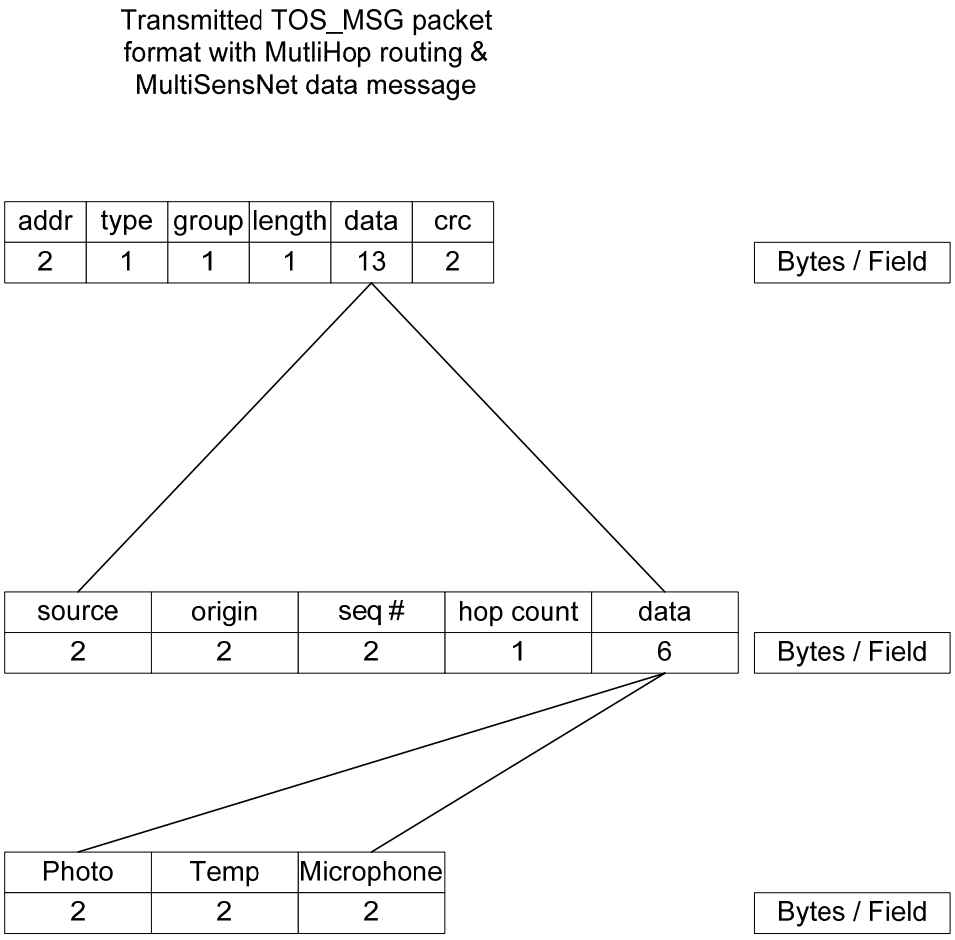
conversion is determined by the constant `DEF_SAMPLE_TIMER_RATE`. The default timing sampling interval is 1 second. The collected data is transmitted every 4 seconds.

The leds are used to display status information pertinent to the mote. The pertinent led behavior is shown in Table A- 4.

Table A- 4: Multi_Sens_Net Led description

Led	Display
Green	• On – The parent node of the mote is not the base station. The mote has an indirect, or no, link to the base station. • Off – The parent node of the mote is the base station. The mote has a direct link to the base station. • Toggling – The node is the base station, i.e. the mote id is 0. Led toggles every second.
Red	• Toggling – The mote is sending sensor data, the mote is the origin. Led toggles every 4 seconds. • On – The mote is not in the ad-hoc network.
Yellow	• Toggling – The mote is forwarding a Multi_Sens_Net packet from another node.

The structure for the transmitted Multi_Sens_Net packet is shown in Figure A- 6.



Type – Type field, indicates which packet handler should be used to decode the packet. The Multi_Sens_Net, node to base, packet type is AM_MSENSNET_MSG which has a value of 0x1E.

source – Address field, address of the node forwarding the packet. If the node is the origin this field will be the same as the origin field.
origin – Address field, address of the packet's originating node.
seq # – Sequence number of the packet, used to calculate the number of missing packets from a neighbor. As the number of missing packets increases, the link quality of the sending node decreases.
hop count – used for creating / updating the routing tree. The neighbor closest to the base station & with the best link quality will become the parent of the sending node.
Photo – Photometer reading.
Temp – Temperature reading.
Microphone – Raw microphone reading.

Figure A- 6: Multi_Sens_Net transmitted packet structure, with pertinent field descriptions

Procedure: Ad-hoc network

1. The Multi_Sens_Net application uses the microphone, temperature, and light sensors to gather data from the environment. Several sections of code are missing from the Multi_Sens_NetM module and the Multi_Sens_Net configuration files; fill in the missing sections of code to complete the module and configuration files.
2. Program the motes, giving each a different node id. The serial programmer, and the mote connected to it, constitutes the base station. All data sent using the ad-hoc routing component propagates under the data aggregation scheme towards the mote with node id 0. Due to the data aggregation scheme, the base station mote must be given the node id of 0.
3. Turn off all of the motes and attach a sensor board to each mote. The sensor board for the base station is attached on the bottom of the serial programmer.
4. Turn all of the motes on and arrange them in a configuration such that ad-hoc routing is occurring, i.e. at least one mote has an indirect link to the base station. See Table A- 4 for more information regarding the leds and their meaning in regards to the ad-hoc routing component.
5. Open the Mote Logger application and provide it with the format for the Multi_Sens_Net packet, see Figure A- 6 for Multi_Sens_Net configuration information. Configure the Mote Logger application to use the TOS_Msg packet type and to log data; the data log is in comma separated variable, CSV, format. Delimit the origin field as the mote id; this will ensure that data from each mote will be stored in a separate log file.
6. Connect to the base station, using the Mote Logger. Record at least 100 packets from each mote. The number of packets received from each mote is displayed under the "Motes Detected" portion of the display. If no field was delimited as the mote identifier field, the packets received will appear next to the "Mote ID" field with the value -1.
7. Open each of the data log files and graph the temperature, light, and microphone sensor readings. All sensor readings, for a given sensor, should reside in the same graph. Be aware the motes which are further away from the base may have a lower packet count than motes closer to the base; this is in terms of the routing tree, not necessarily related to the physical distance to the base station.
8. Convert the temperature readings to Fahrenheit. Graph the converted temperature readings in a separate graph. This graph should look similar to the ADC temperature graph.

Conclusion

Describe what you learned from this lab. Also, describe the various aspects of the lab, their value, and how they relate to real-world engineering problems. Do not just summarize what you did.

Lab 4: Group Project

Estimated Lab time: approximately 5 hours.

Learning Objectives

- To apply previous knowledge to design, implement, and test a system from a set of requirements.

Data Logging with on-board data logger

The mica mote has a 512KB EEPROM which can be used for persistent storage. The EEPROM has a total of 32,768 addressable lines (each line of the EEPROM is 16-bytes in size). Every 16 lines of the EEPROM are considered a page. The first page of the EEPROM is considered reserved and is therefore inaccessible through the logger interface.

To utilize the logger module, wire the Logger component to your main mote application. The Logger write interface will enforce exclusive read/write access, i.e. no other logger instance can read or write data to the EEPROM until the last logger instance has received an readDone or writeDone event.

When the append command or the readNext command is used to store or read data to/from the EEPROM, the current line pointer is incremented*. If the current line pointer, after being incremented, happens to point beyond addressable memory, the current line pointer will reset to the beginning of non-reserved memory.

If Xnp is also present in your design, ensure that the main mote application releases the EEPROM before replying SUCCESS to the NPX_DOWNLOAD_REQ. Also ensure the main mote application does not overwrite the downloaded program code to the EEPROM, by Xnp. A simple calculation using the downloaded program's start page number, the download program's number of pages used, and the number of lines per EEPROM page can easily avoid the risk of overwriting the downloaded program code.

Read Lesson 8 of the TinyOS tutorial for an introduction to the data logging component in TinyOS.

Requirements for the laboratory

This laboratory is a design laboratory which represents a culmination of what you have learned in the previous laboratories. You are required to design, implement, and test a system from a set of requirements. Two options exist for

*For correctness sake, any valid logger read or logger write operation will increment the current line pointer upon completion. The next read/write only cares about the pointer position if the next read/write operation is a readNext or append operation.

this set of requirements. In the first option you propose a set of requirements, i.e. identify a problem and list the requirements, and build a system that addresses the problem. The second option you have is to build a system that addresses the problem, and requirements, presented below. Be sure to demonstrate your solution to the instructor.

Sounder System

Create a system of 3 motes, where each mote plays a unique sequence of tones which triggers the next mote to play a unique sequence of tones. The base node should trigger the next node, and the next node should trigger the last node, and the base node should receive tones from the last node. Each node should only respond to its given triggering tone sequence. Each node should provide a visual means of valid tone sequence receipt verification.

The triggering tone and sending tone sequences should utilize the node id. Under this tone scheme the node id would be used to determine how many tones to send and how many to trigger on, the base node would require special handling. The solution should consist of 1 mote application which can adapt its triggering and tone sending sequences based upon its node id.

Hint: Review lesson 4 of the TinyOS tutorial, http://www.tinyos.net/tinyos-1.x/doc/tutorial/.

Note: These requirements are more difficult than they first appear[**].

Conclusion

Write up what procedure you followed to design, implement, and verify your program. Discuss what you learned from this lab.

[**] Pitfall: The Tone Detector circuit may generate a false interrupt when its interrupt is enabled. The Tone Detector circuit should be used to indicate the presence of a tone, and to synchronize the listening mote to the tone, not to count the tone.

Important Notes about TinyOS and mica2 motes

1. TinyOS uses the Unique phrase “Timer” when creating TimerC instances. To avoid conflict with TinyOS components use the phrase “Timer” with the “unique” function when creating any TimerC instances.
2. TinyViz, the TOSSIM GUI front-end, may become unresponsive if used for an extended period of time or excessive debug messages are displayed, i.e. too many messages are being displayed in TinyViz. Limit the number of messages by either not enabling the Debug messages plug-in, or by setting the DBG environmental variable to only the desired debug message types.
3. TinyViz, and TOSSIM, may run slow if too many debug messages are enabled. Ensure the DBG environmental variable is set to only the desired debug message types, the default message type is “all” if the environmental variable is not set.
4. Java applications which utilize a communications port, such as a serial port, use the javacomm driver. The javacomm driver, version 2.0, can cause Windows to crash if used for a prolonged period of time or if a java program is frozen, such as in debugging, for an extended period of time. Ensure that all work is saved before use of any communications port capable java application.
5. On the micasb sensor board, the light and temperature sensors are connected to the same ADC channel. Only turn on one of the sensors at a time, otherwise the data read from the ADC, regardless of photo or temp interface, will be invalid.
6. TOSSIM does not simulate several TinyOS components. These include, but are not limited to:
    1. The CC1000 mica2/mica2dot radio stack
    2. The Xnp module

    Programs using any of the aforementioned modules will not compile for the pc platform, but they will compile for the mica2 platform.

Appendix B: Lab Modules (Instructor's Manual)

The instructor's manual provides the results for each of the experiments. The source code, both the incomplete and solution, is included on the accompanying CD. The content of the instructor's manual is merely the commands necessary to run each of the experiments' programs, in addition to the graphs, and the data required to complete each of the experiments.

The instructor's manual is designed to be used in conjunction with the student manual.

Source Code and X2c zip structure

The folder names denote the laboratory, source code version i.e. incomplete/solution, and the module/configuration/application name. NesC filenames ending in an upper-case M are modules, while filenames not ending in an upper-case M are configurations. The graphs/tables reside in excel files, files with a .xls extension, parsed data resides in CSV files, files with a .CSV extension, and re-directed Xlisten output resides in files with no extension.

The X2c Perl script is located in the Lab 2 directory.

Lab 1: TinyOS, Nesc, and TOSSIM

1. To run Tython with 5 motes in the simulation, no console, no serial forwarder and, the TinyViz GUI, type the below command in a CygWin window:

   java net.tinyos.sim.SimDriver -noconsole -nosf -gui -run ./build/pc/main.exe 5

2. To run the serial forwarder application, type the below command into another CygWin window.

   java net.tinyos.sf.SerialForwarder

3. To run the surge java application, type the below command into another CygWin window Be sure to set the group id field to the group id of the motes being simulated, the group id field is shown as 125 below, this is the group id the mote application was compiled with.

   java net.tinyos.surge.MainClass 125

Lab 2: Crossbow Motes, Programming, In-Network Programming, and Listening

1. To run the Xlisten application configured with the raw and timed options, type the below command. The greater than sign redirects the output of Xlisten to a file, in this case xrt.

   Xlisten -s=COM6 -b=57600 –r –t > xrt

2. The graphs generated from the comma separated variable files are presented in the graphs below. The MoteID field is used to separate data from each of the transmitting motes and each graph is denoted with the application which transmitted it and the sensor which the data was collected from.
3. The approximate packet reception duration, the number of packets transmitted, the number of packets received, the approximate packet transmission rate **per second**, and the error rate, i.e. the percent of packets with errors or lost, is presented in the table below.

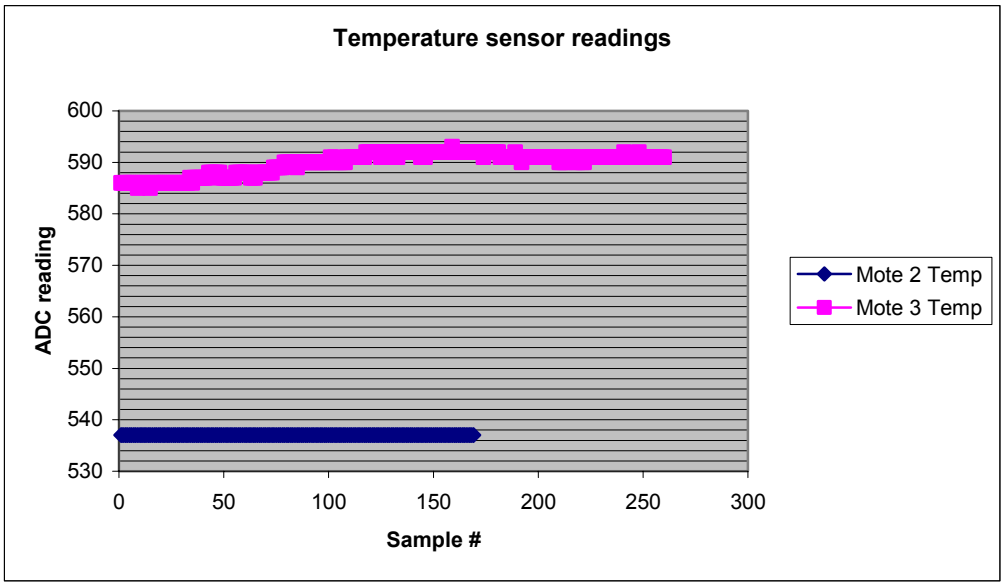


Figure B- 1: RecSensNetTemp temperature sensor readings

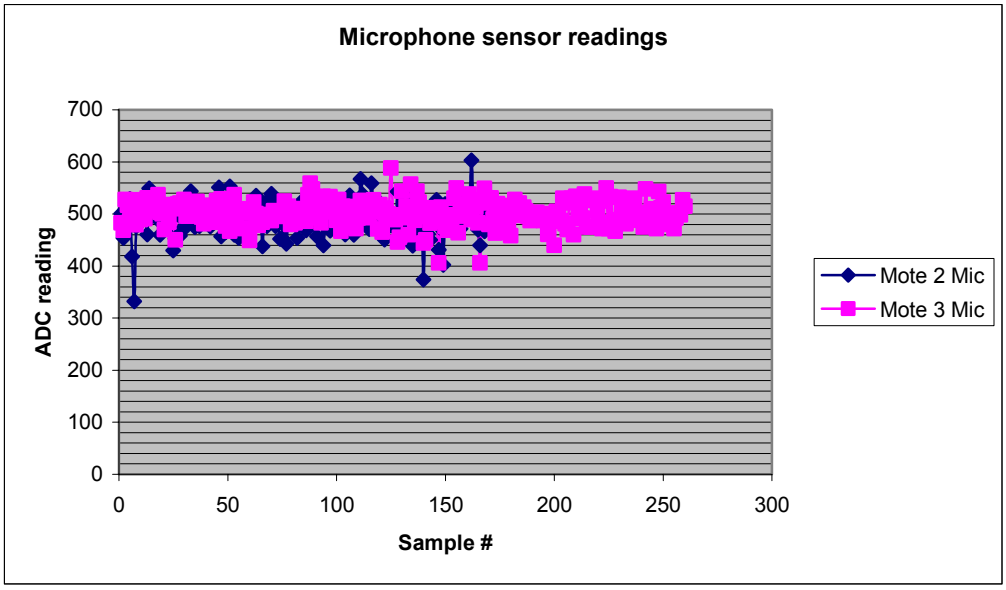


Figure B- 2: RecSensNetTemp microphone sensor readings

RecSensNetTemp	Duration (h:mm:ss)	TX Pkt Count	RX Pkt Count	Duration (s)	TX Pkts/sec	Error Rate
Mote 2	0:04:46	293	189	286	1.0245	35.49%
Mote 3	0:04:47	294	276	287	1.0244	6.12%

Table B- 1: RecSensNetTemp calculations tabulation

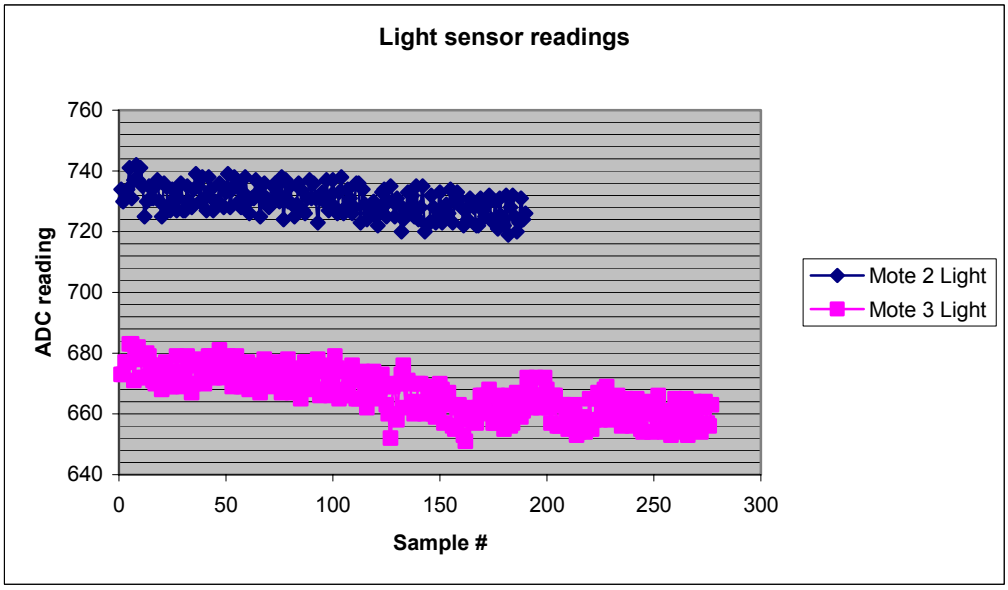


Figure B- 3: RecSensNetPhoto light sensor readings

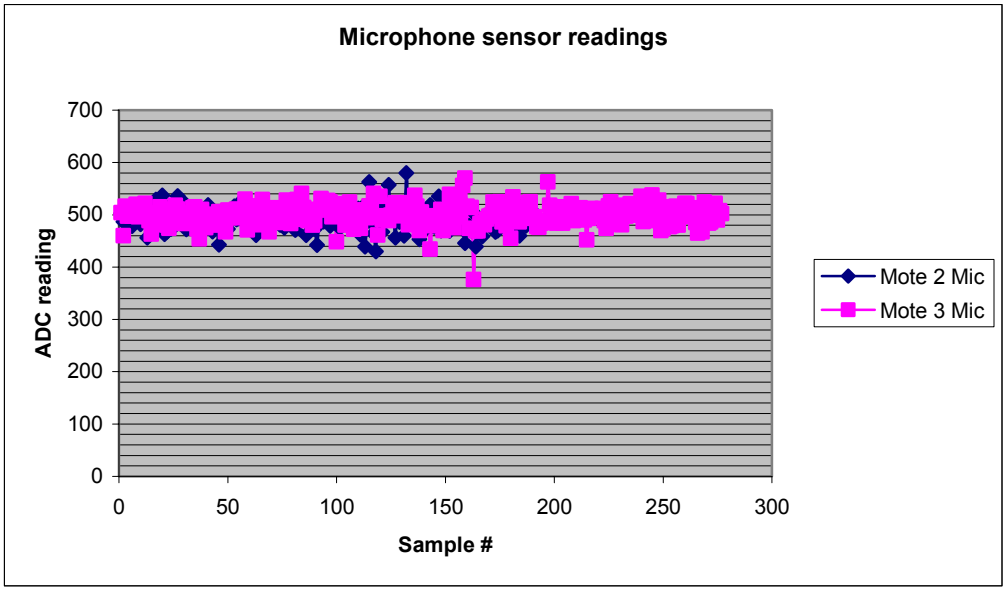


Figure B- 4: RecSensNetPhoto microphone sensor readings

RecSensNetPhoto	Duration (h:mm:ss)	TX Pkt Count	RX Pkt Count	Duration (s)	TX Pkts/sec	Error Rate
Mote 2	0:04:54	302	189	294	1.0272	37.42%
Mote 3	0:04:56	304	276	296	1.0270	9.21%

Table B- 2: RecSensNetPhoto calculations tabulation

Lab 3: Ad-hoc Networking, Logging data from multiple motes

1. Each sensor reading, for a given sensor, is graphed below.

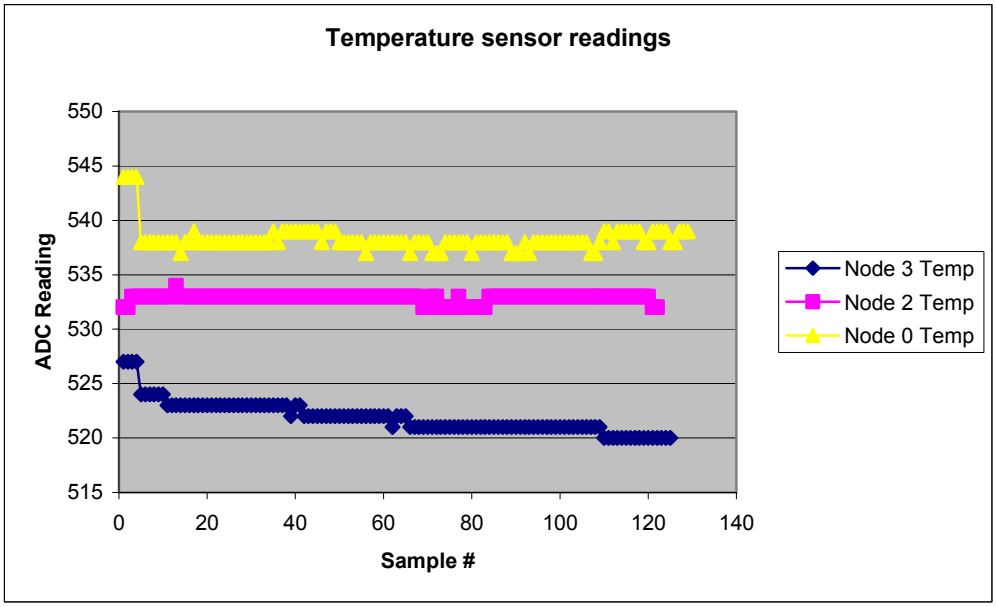


Figure B- 5: Multi_Sens_Net temperature sensor readings

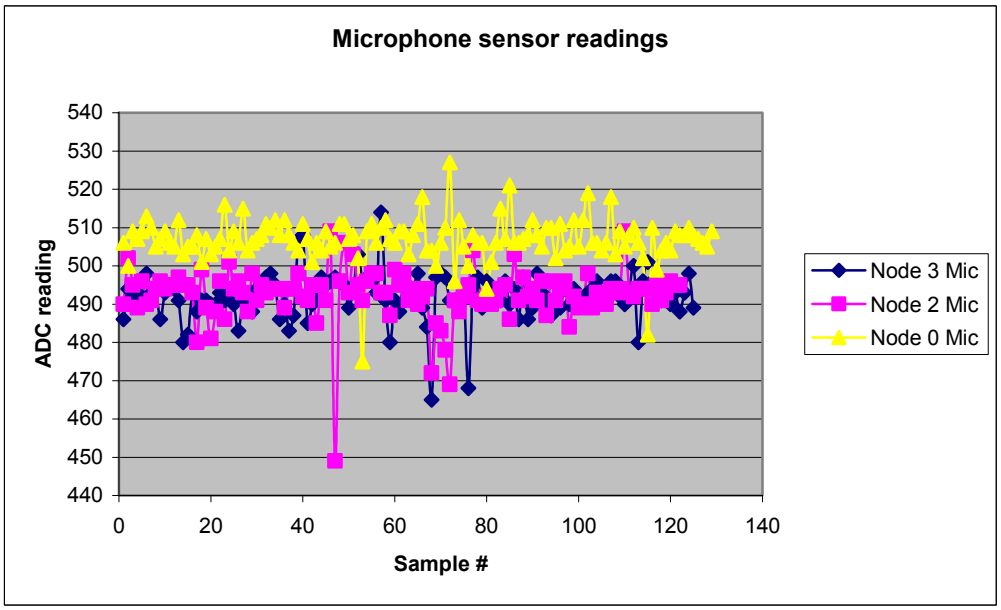


Figure B- 6: Multi_Sens_Net microphone sensor readings

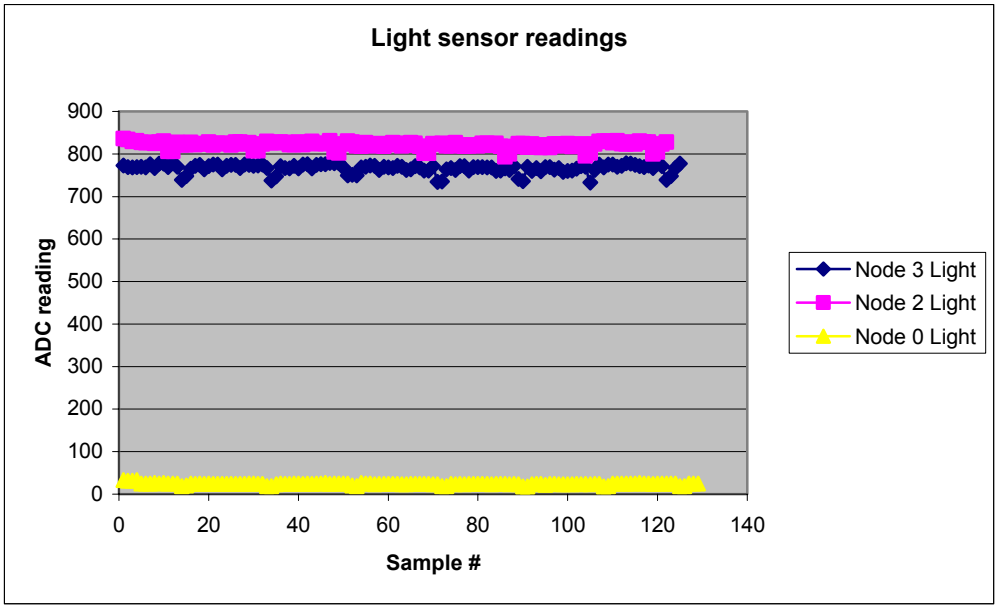


Figure B- 7: Multi_Sens_Net light sensor readings

2. The converted temperatures are given in the graph below.

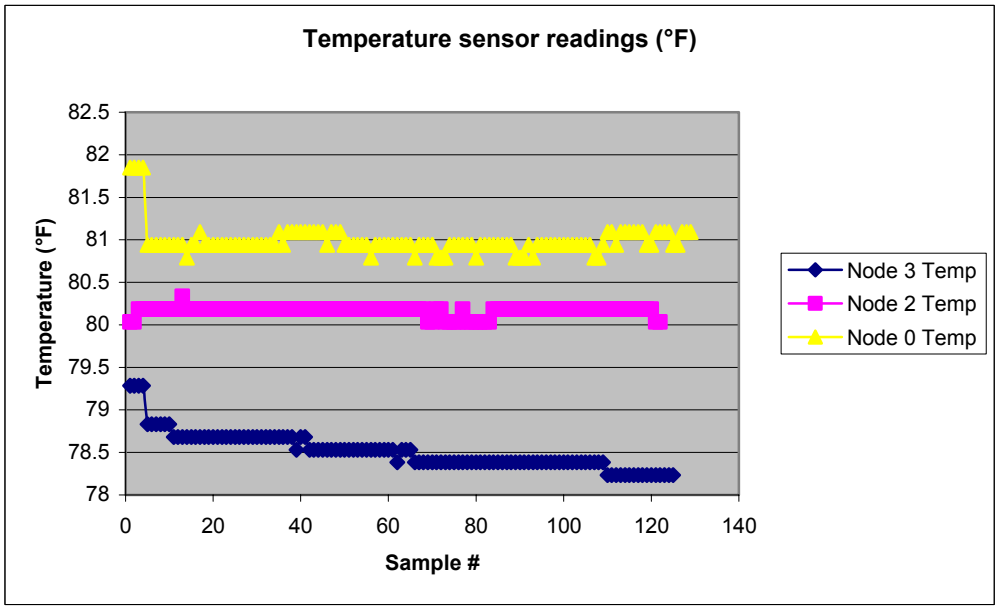


Figure B- 8: Multi_Sens_Net temperature sensor readings converted to Fahrenheit

Lab 4: Group Project

There are no results for this experiment other than a demonstration of the student's solution to the instructor.

Appendix C: Source Files and Mote Logger README

The source files, both incomplete and solution versions, for the lab experiments, the X2c script, and the Mote Logger application are included on the accompanying compact disc. The Mote Logger README file is also included on the accompanying compact disc.