# Wireless Sensor Network for Monitoring Applications

A Major Qualifying Project Report

*Submitted to the University of*

WORCESTER POLYTECHNIC INSTITUTE

*In partial fulfillment of the requirements for the*

Degree of Bachelor of Science

By:


| | |
|---|---|
| _____ | _____ |
| Jonathan Isaac Chanin | Andrew R. Halloran |
| _____ | _____ |
| Advisor, Professor Emmanuel Agu, CS | Advisor, Professor Wenjing Lou, ECE |

# Table of Contents

# Table of Figures

## Abstract

The goal of this project was to design and build a wireless sensor network. Following an exploration of personal area networks and mesh networking, a system was implemented to detect physical intrusion. To that end, our network employed sensor nodes equipped with motion sensors and accelerometers. The network communicated with a generic infrastructure, adaptable to future wireless sensor projects, which stored sensor data in a database. Also included was a user interface to monitor the status of the entire system.

# 1. Introduction

With the advent of low-power embedded systems and wireless networking, new possibilities emerged for distributed sensing applications. These technologies led to the implementation of wireless sensor networks, allowing easily configured, adaptable sensors to be placed almost anywhere, and their observations similarly transported over large distances via wireless networks.

## 1.1 What are Wireless Sensor Networks?

Wireless sensor networks consist of distributed, wirelessly enabled embedded devices capable of employing a variety of electronic sensors. Each node in a wireless sensor network is equipped with one or more sensors in addition to a microcontroller, wireless transceiver, and energy source. The microcontroller functions with the electronic sensors as well as the transceiver to form an efficient system for relaying small amounts of important data with minimal power consumption.

The most attractive feature of wireless sensor network is their autonomy. When deployed in the field, the microprocessor automatically initializes communication with every other node in range, creating an ad hoc mesh network for relaying information to and from the gateway node. This negates the need for costly and ungainly wiring between nodes, instead relying on the flexibility of mesh networking algorithms to transport information from node to node. This allows nodes to be deployed in almost any location. Coupled with the almost limitless supply of available sensor modules, the flexibility offered by wireless sensor networks offers much potential for application-specific solutions.

## 1.2 Why Sensor Networks?

Wireless sensor networks have many advantages over traditional sensing technology, due to their embedded construction and distributed nature. The first, and for many the most notable, feature is their cost. Using low-power and relatively inexpensive microcontrollers and transceivers, the sensor nodes used in wireless sensor networks are often less than one hundred dollars in cost. This opens the

doors for many commercial or military applications, as the relatively diminutive cost of nodes allows for not only large numbers of sensors to be deployed, but also for large numbers of sensors to be lost. For example, sensor nodes can be dropped from a plane, allowing widespread coverage of an area with minimal effort involved in positioning the individual nodes. The relatively low cost of the sensors allows for some nodes to be damaged or lost without compromising the system, unlike larger, more centralized sensors [12].

Another advantage wireless sensor networks hold over traditional wireless sensing technology lies in the mesh networking scheme they employ. Due to the nature of RF communication, transmitting data from one point to another using a mesh network takes less energy than transmitting directly between the two points. While embedded systems must respect their power envelope, the overall energy spent in RF communication is lower in a mesh networking scenario than using traditional point-to-point communication [12].

Sensor networks can also offer better coverage than more centralized sensing technology. Utilizing node cost advantage and mesh networking, organizations can deploy more sensors using a wireless sensor network than they could using more traditional technology. This decreases the overall signal-to-noise ratio of the system, increasing the amount of usable data. For all these reasons and more, wireless sensor networks offer many possibilities previously unavailable with traditional sensor technology [12].

## 1.3 Application Examples

Wireless sensor networks are seeing use throughout the world. Just off of the cost of Maine, The University of California Berkeley is using a wireless sensor network to monitor the nesting behavior of ocean birds. Previously, this research would be performed by scientists physically observing the birds, often interfering with their nesting by physically touching their nests. Now, small sensor nodes are placed near the nests, allowing scientists to collect data without affecting the birds [12].

Another Berkeley team developed a wireless sensor network that could be dropped from an Unmanned Aerial Vehicle (UAV), autonomously record traffic patterns, and report the results back to the UAV. This system saw testing in the Mohave Desert, showing promising results for military and civilian applications [12].

For this project, we will be using the Tmote Sky platform for wireless sensor networks. Networks implemented using the Tmote Sky platform have been used for many diverse applications ranging from climate control for farmers to bridge structural stability monitoring. Among the most interesting of the examples is a Tmote-based firefighter monitoring system. Utilizing sensors to monitor smoke, light, and fire, and incorporating electronic maps, the sensor nodes monitor the status of firefighters, displaying relevant information inside a firefighter's mask [1].

## 1.4 Project Goal

We will implement a wireless sensor network using the Tmote sky platform, including all supporting infrastructure, for a monitoring application. In doing so, we will show that wireless sensors networks are indeed a viable emerging technology available to engineers for a wide range of applications. We will document our process from start to finish, enumerating the requirements for building a functional wireless sensor network and all supporting architecture. By establishing this example we will expose the benefits and drawbacks of moving sensing applications onto an embedded platform and make recommendations for the future of both commercial sensing applications and application specific platforms.

# 2. Background

The Tmote Sky is a platform for low-power high-bandwidth sensing applications manufactured by Moteiv Corporation. Built to meet IEEE 802.15.4 low-power networking standards, the Tmote Sky is a versatile platform for scalable wireless sensing applications. Harnessing the power of TinyOS, an event-driven operating system designed specifically for extremely low power sensor networks, the Tmote Sky can be used with most any sensor with either digital or analog outputs. Utilizing fully scalable mesh networking, Tmotes form ad-hoc networks transparent to both the engineer and user. All low-level network functions are handled by TinyOS, allowing the programmer to simply specify a destination for network messages.

Our project builds on previous work which presented the Tmote Sky as an ideal platform for developing a wireless sensor network. This allowed us access to an ample supply of Tmote Sky units for the establishment of our proof-of-concept. Below, we outline what our research revealed about the functionality of Tmote Sky.

## 2.1 Tmote Sky wireless sensor platform

The Tmote Sky wireless sensor platform offers a robust solution for developing a sensor network, with most of the fundamental networking services built in. The Tmote Sky uses a Texas Instruments MSP430 microprocessor running TinyOS, and a Chipcon SmartRF transceiver. TinyOS is an open-source platform specifically designed for the development of wireless sensor networks, and features a component-based architecture enabling rapid implementation despite the memory constraints of embedded systems [2]. The Chipcon SmartRF transceiver utilizes the IEEE 802.15.4 specification, which includes both physical and MAC protocols, and allows packet routing independent of the microprocessor.

### 2.1.1 TinyOS

TinyOS is a software platform engineered specifically with wireless sensor networks in mind. As such, it includes built in multihop routing, which implements mesh networking without any additional development, and a low-bandwidth synchronized power-saving mode. The low-bandwidth mode keeps nodes in a low-power mode most of the time, while still keeping them synchronized for transmitting and receiving. This allows each node to 'sleep,' saving battery life, but synchronously causes the entire network to wake up and send their information at the same time when necessary [3].

While the MSP430 runs TinyOS, the majority of application development is done in the network embedded systems C language, or NesC. NesC is the programming language designed for building applications in the TinyOS environment. NesC provides an event-driven programming model on top of the C language, abstracting low-level C functionality into a higher-level language designed for network embedded systems. As such, it abstractly handles potential issues regarding distributed systems. For example, NesC detects race conditions via static code analysis before the network is even deployed, preventing hazards at compile-time instead of run-time.

### 2.1.2 Chipcon CC2420 Transceiver

The Tmote Sky utilizes an early IEEE 802.15.4 protocol for mesh networking using a Chipcon SmartRF CC2420 ZigBee-ready wireless transceiver. This transceiver is a low-cost solution for providing robust wireless communication in the unlicensed ISM band. The transceiver provides many desirable features, including a physical protocol, a MAC protocol, and AES encryption. These features are used by the Tmote Sky for its TinyOS mesh networking implementation [4].

The Chipcon CC2420 transceiver interacts directly with TinyOS, allowing the Tmote platform to abstract the networking algorithms from the programmer. Simply specifying a destination node identifier in the NesC program allows TinyOS to transparently rout packets to the correct node, regardless of its location on the mesh network. Between the TinyOS code and the Chipcon transceiver,

the packets sent and received by the Tmote units appear similar to packets from any network. This includes the basic format of MAC, preamble, and data followed by an acknowledgement packet. This basic behavior is outlined in a timing diagram depicted in Figure 1.

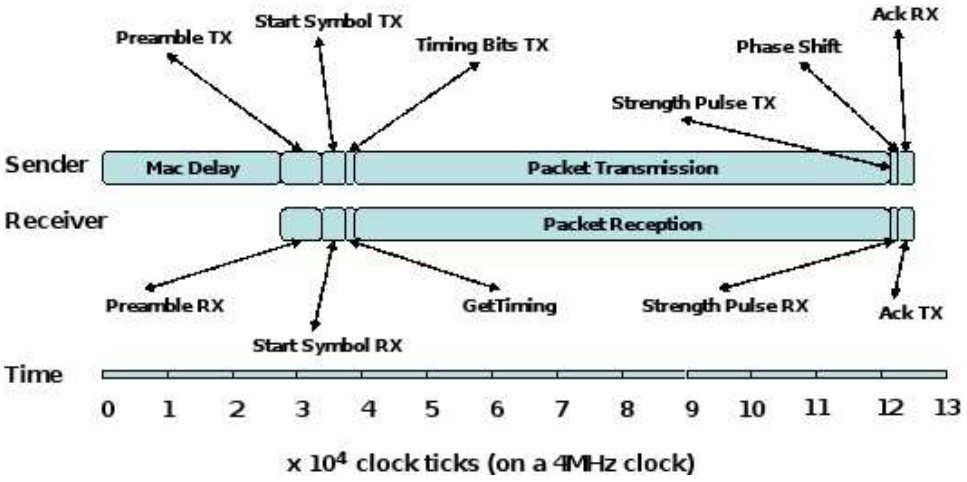


Figure 1: Timing diagram of network transmission and reception

This method of transmission is typical of most networks, wireless or otherwise, and speaks to the reliability of the Tmote Sky's wireless networking capabilities.

## 2.2 Mesh Networking with TinyOS and IEEE 802.15.4

The Tmote Sky's wireless capabilities are based on the mesh networking paradigm implemented by TinyOS. This implementation is described in the following sections.

### 2.2.1 Low-Power Networking Background and Standards

In the wake of the IEEE 802.11 standard, a need evolved for low-power, low-duty-cycle wireless enabled embedded devices. The need for high data rates was often eclipsed by the need for efficient handing of smaller data packets, allowing embedded devices to move information quickly from one point to another and minimize the energy consumed by their transceivers. The IEEE 802.15.4 WPAN standard evolved as a way to implement this functionality on embedded platforms. WPAN units are less expensive than other wireless formats, including Bluetooth and Wi-Fi, and are highly specialized for embedded applications, making WPAN-based networking an ideal choice for cost-effective wireless

sensor networks [5].

Embedded applications are characterized by low-power, high-efficiency electronics, and WPAN units are no exception to this paradigm. As such, any solution implemented with this technology must work around the relatively low transmission power and battery life constraints of embedded systems. In addition, any network formed by these devices must be self-organizing and able to adapt to changing network topologies to be used in a wireless sensing application. These two requirements, in addition to a sensor network's need for distributed intelligence, are met by TinyOS's implementation of mesh networking [6].

### 2.2.2 Networking Protocol Stack

Given that the Chipcon SmartRF chip provides both the physical and MAC layers, it falls to TinyOS to implement the transport and networking layers. It does so using a "packet link" layer, a partial implementation of the transport and networking layers engineered to ensure data is reliably transferred between network nodes in spite of any dropped packets due to range, interference, or various additional factors [7].

### 2.2.3 Mesh Networking with TinyOS

A mesh network topology is the most useful for wireless sensor network applications. Mesh networks are designed such that each node can communicate with each other node in the network without the use of a central control scheme. Figure 2 contains an example of a mesh network before any communication between nodes.

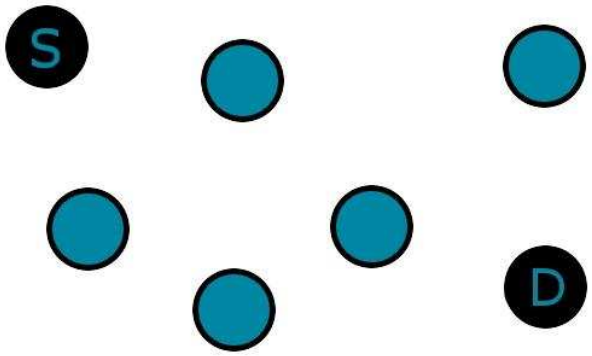


Figure 2: Preliminary mesh network topology

The routing protocol of the mesh network works as follows; the source node shares its routing information with each node surrounding it. Each node performs this action, until a complete routing table is established, and all topology information is shared between the nodes, as shown in Figure 3.

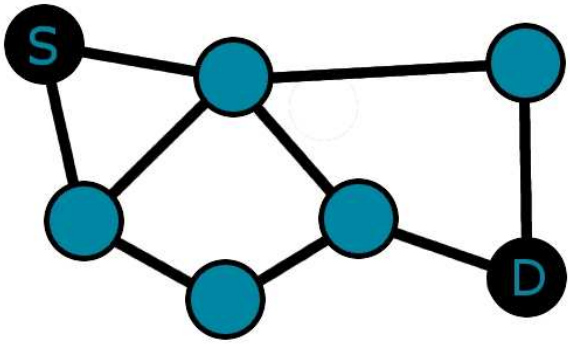


Figure 3: Mesh network topology including routing information

Once the nodes have established their routing table, the source node can send its packets safely to the destination node via the ideal routing path as depicted in Figure 4 [8].

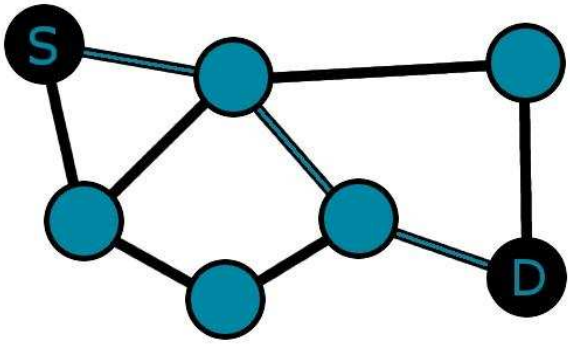


Figure 4: Mesh network topology with ideal routing path highlighted

The main attraction to mesh networking is that it allows for ad-hoc networks with an arbitrarily large number of nodes. Above all, mesh networked embedded systems are built to perform in environments where reliable, scalable, and adaptable low-bandwidth communication is required. This protocol also allows for a particularly resilient network, as data can take more than one path to its destination, meaning the network rarely has a single point of failure. Ad-hoc mesh networking scales exceedingly well, is reliable, and is adaptable to use most any kind of embedded processor [9]. For these reasons, it makes an excellent platform for wireless sensor networks.

# 3. Requirements

Following the completion of our background research, we progressed to defining our application. In order to complete our proof-of-concept wireless sensor network, we needed an application both relevant to modern concerns and appropriate for implementation on an embedded platform. Having arrived at an application, we chose our sensors and laid the groundwork as described below.

## 3.1 Choosing an Application

We began the project with a discussion of the possible applications for our wireless sensor network. After exploring the capabilities of the Tmote Sky platform, we developed a number of different proposals. The most promising implementations we considered involved using the hardware for environmental or security monitoring. The nature of these applications lent themselves well to using large networks of individual nodes. For environmental monitoring, a large wireless sensor network could detect climate changes in environments over a large area. Using a wireless sensor network for security monitoring would allow for quickly deployed large-scale intrusion detection.

After evaluating the feasibility of each of these approaches we chose to pursue security monitoring. Environmental monitoring, though a promising implementation of wireless sensor technology, was discounted due to concerns with testing a large sensor network in an outdoor environment. A small distributed security system provides a solid proof-of-concept without the large-scale outdoor implementation environmental monitoring would require.

## 3.2 Sensors

Having decided that our target application would be security monitoring, we began investigating potential sensors for our mesh network. Among the most promising sensors for a security monitoring application were microphones, motion sensors, and accelerometers.

### 3.2.1 Microphone

Utilizing a microphone as a sensor initially showed much promise. Combined with digital signal processing algorithms, a Tmote Sky module could be programmed to parse microphone data for specific sounds. For example, the Tmote could listen for activity on the frequency ranges that signify a breaking window. This would be accomplished mostly with the use of filters and amplification. Unfortunately, time constraints and inexperience with DSP forced the dismissal of this idea. An alternative involved passing all sounds back to the base station for processing, an unfavorable proposal that would require every mote to transmit far more often than otherwise necessary, reducing battery life and overall network performance.

### 3.2.2 Motion Sensor

Sensing motion with infrared detectors has been a staple feature of security systems for decades. Integrating this functionality into an easily deployed mesh network would allow intrusion detection over a large area. For a security application, we expect that motion would be the most reliable sign of intrusion. Pyroelectric infrared motion sensors are relatively inexpensive and low power, and as such, lend themselves well to an embedded platform.

### 3.2.3 Accelerometer

Accelerometers can be used to determine if a node has changed orientation. For a security application, it is important to know if a node has been rendered insecure. We chose to use accelerometers for this purpose, as it would be difficult for a potential intruder to tamper with a Tmote encased in even a moderately secure box without affecting the output of an accelerometer. Having decided that the integrity of the network was of paramount importance, we included accelerometers into our design. Low-power accelerometers are widely available, and implementing one as a tilt sensor would be a solid addition, making the design more robust.

## 3.3 Network Architecture

Once the Tmote Sky units have been configured with software and external sensors, we will need to build an infrastructure to support them. Without a human interface, there is no way for the collected by the sensors to be used. The purpose of this infrastructure is to provide a method for transporting event data from the sensor network to a human-readable form. To accomplish this, we propose the following design.

### 3.3.1 System Overview

To aid in our explanation of our infrastructure, we have included an overall system diagram, shown in Figure 5.

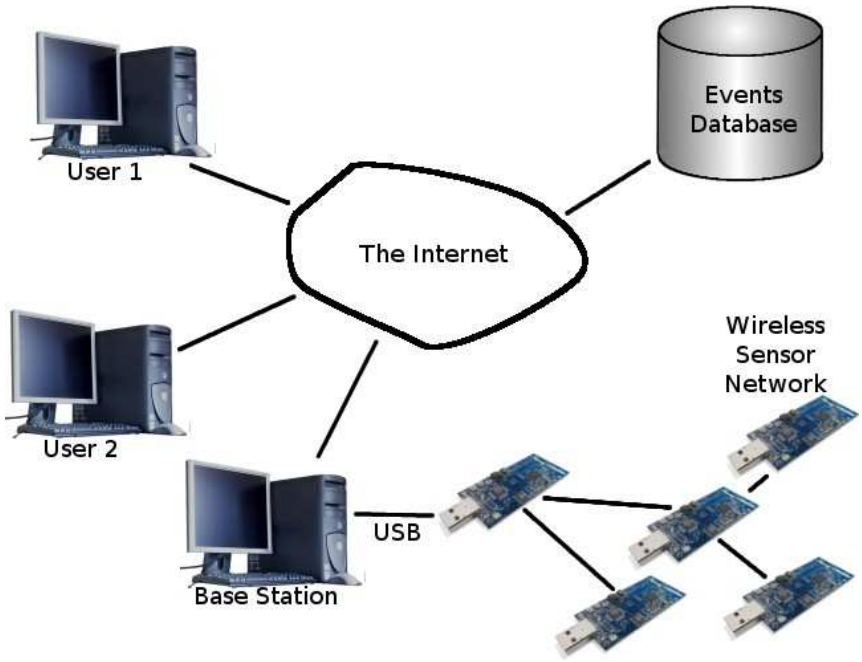


Figure 5: System Overview

This diagram shows the interactions between each major component in the system. Beginning with the network of sensors, information travels to the base station through the sink node. Once the data has been parsed by the base station, it is sent to the database. Finally, users are able to view the

status of the sensor network by using an application that queries event information from the data storage.

### 3.3.2 Base Station

By necessity, our base station will be a computer connected to the mesh network sink node, the node to which all other nodes will be programmed to send their packets. This base node will need to parse sensor data, and format the data for transmission into an appropriate storage medium.

### 3.3.3 Data Storage

For effective implementation of the sensor network, we require a centralized storage mechanism allowing for multiple concurrent clients reading and modifying the system at a given time. This storage mechanism must be able to hold the readings of an arbitrarily large number of sensor nodes, effectively organizing it for efficient read and write access.

### 3.3.4 User Interface

Given that at this point, all of our information will be housed in a central medium, we require an appropriate user interface to parse and display this data in a user-friendly manner. At the minimum, this interface needs to display the current status of each node, along with a description of past node readings. Additionally, users should be able to specify their own names for nodes, forgoing the default behavior of identifying via unique node numbers.

## 3.4 Overall Requirements

Our wireless sensor network must detect physical intrusion. To this end, our nodes have to glean pertinent data from our available sensors, which include motion and acceleration, in order to not only ensure that physical intrusion will be detected, but also to intelligently maintain the trust of the individual sensors. This cohesive application will show that wireless sensor networks are not only a

viable new technology, but also that they can form a good, cohesive solution for today's real-world problems.

# 4. Design

Given the requirements outlined in the section 4, we implemented the sensor network. Beginning with sensor design, we chose specific components for our external sensors and devised methods for connecting them to the Tmote Sky units. Once we had the sensors connected, we moved on to programming, for not only the individual Tmotes, but also for the base station, database, and the user interface.

## 4.1 Sensor Design

For our security application, we decided to use both a motion sensor and an accelerometer. Each sensor is highly useful for a security application. Whenever a sensor node detects motion or is moved from its original location, the node will generate an event, alerting the wireless network that something is wrong. With the sensor types decided upon, we designed our circuits, and chose our specific parts for use with our security system. In the following sections, we outline each circuit and its connections to the Tmote Sky platform.

### 4.1.1 Tmote Expansion Port

In order to connect our external sensors to the Tmote Sky module, we have to interface with the pins of the MSP430 microcontroller. The Tmote Sky module provides two expansion connectors for connecting external sensors. We will be utilizing the larger, ten pin expansion port, a description of which is available in Figure 6.

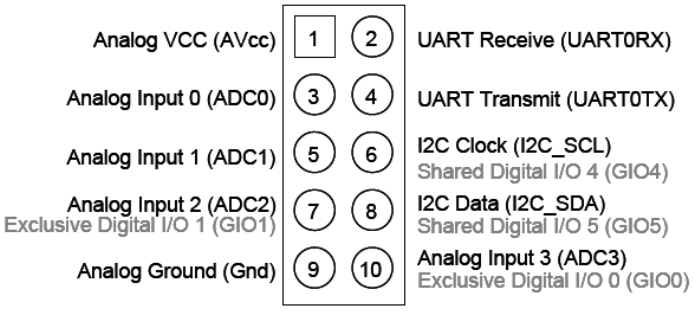


Figure 6: Tmote Sky Expansion Connector

This expansion port provides us with three inputs, power, and ground to connect our sensors. This is sufficient for connecting both the motion sensor and both axes of the accelerometer.

### 4.1.2 Motion Detector

Our sensor node design uses a Panasonic AMN44121 passive infrared motion sensor. This sensor combines passive infrared sensing with the necessary operational amplifiers and comparators to create a low-power digital motion detector that draws less than 46 µA on a three volt source. Incorporated into the sensor is a lens array enabling motion detection at ten meters in a 110° arc.

Connecting the motion sensor proved to be a simple procedure, as the built-in comparator circuitry ensures that the output from the motion sensor is binary. Power, ground, and a signal wire to ADC2/GIO1 (as shown in Figure 6) are the only connections required for a functioning infrared motion detection sensor.

### 4.1.3 Accelerometer

For our design, we will be utilizing an analog devices ADXL322 accelerometer. This accelerometer is a low-power dual-axis accelerometer for use with embedded applications. It accurately measures dynamic and static acceleration at +/- 2 g-forces on each axis.

Connecting the accelerometer outputs to the expansion port is a straightforward task, with each axis having a dedicated output from the IC. However, for proper functionality, the accelerometer requires two 0.1 µF capacitors, one placed between each signal node and ground. These capacitors limit

the bandwidth of the sensor to 50Hz, and function as a low-pass filter for anti-aliasing and noise reduction purposes. Conveniently, we were able to obtain samples of this IC directly from Analog Devices, preassembled with the requisite filtering capacitors. Our completed accelerometer schematic can be seen in Figure 7.

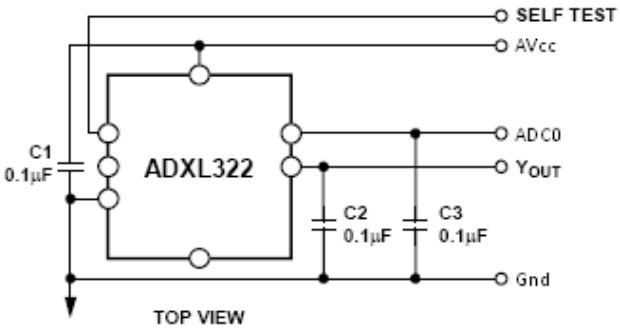


Figure 7: Accelerometer Sensor Schematic

This circuit allows us to monitor the magnitude of the acceleration affecting our sensor node in two distinct directions. Our methods for processing this information are documented in a later section.

## 4.2 Sensor Node Software Design

With the motion and acceleration sensors attached to the Tmote Sky units, we proceeded to write software for the sensor nodes.

### 4.2.1 A/D Resolution

In order to accurately obtain values from an analog to digital converter, we must first find the resolution of the converter. The resolution of an A/D converter is the voltage potential per bit of the converted value. To find this resolution, we can apply the Equation 1.

$$R = \frac{Vref\ hi - Vref\ low}{2^n}$$

Equation 1: Resolution of an A/D converter

In this equation, n is the number of bits in the A/D converter, $Vref_{hi}$ is the supply voltage for the converter, and $Vref_{low}$ is ground. The Tmote sky platform functions on a supply voltage of three volts

and uses a twelve-bit A/D converter. Applying these values to Equation 1 yields the results in Equation 2.

$$R = \frac{3v - 0v}{2^{12}} \cong 0.732\ mV/bit$$

Equation 2: Resolution of Tmote Sky's A/D Converter

In order to obtain mathematically correct values for arithmetic operations on the Tmote Sky, we multiply the values returned by the A/D converter by the calculated resolution. In contrast to the 12-bit A/D converter employed by our sensor node, the ADXL322 accelerometer outputs an analog voltage corresponding to 420 mV/g. Given this value, our resolution of 0.732 mV/bit is acceptable for processing the output of the accelerometer.

### 4.2.2 Accelerometer Algorithm

In order to retrieve useful data from the accelerometer, it is necessary to process the output of the analog to digital converter. The first step in this algorithm is applying the resolution of the A/D converter to its output. Multiplying the V/bits resolution with the output of the A/D converter yields the voltage level of the A/D input. The ADXL322 datasheet indicates that an accelerometer output of 1.50 Volts corresponds to 0g, and that the accelerometer outputs 420 mV/g. Combining these figures yields an equation used to calculate the acceleration affecting an axis of the accelerometer. This equation is shown in Equation 3.

$$a = \frac{(ADC\ Output)\left(0.000732\ \frac{V}{B}\right) - 1.5V}{.42\ \frac{V}{g}}$$

Equation 3: Converted A/D Converter Value in g's

This acceleration value is mathematically relevant and useful, but again referencing the ADXL322 datasheet, there is an additional mathematical operation to perform.

Since the ADXL322 accelerometer measures static acceleration, it is possible to use it as a tilt sensor. Assuming there are no external forces, and both axes of the accelerometer are oriented parallel

to the earth's surface; the arcsin of the accelerometer output will be equal to the angle between the earth's surface and the accelerometer axis, as shown in Equation 4. This is the end result of the algorithm we implemented, as changes in this angle reveal that the sensor node is moving.

$$\angle\Theta = \arcsin(a)$$

Equation 4: A/D Converter Value as an angle

Utilizing this simple algorithm, we can accurately measure the orientation of our Tmote sky wireless sensor.

### 4.2.3 Node Program Design

The software for each individual sensor node is a straightforward loop. Each involves monitoring the analog-to-digital converter port on the MSP430, and sending packets to the base station if an event is detected. The program flow of the accelerometer and motion sensor nodes is depicted in Figure 8.

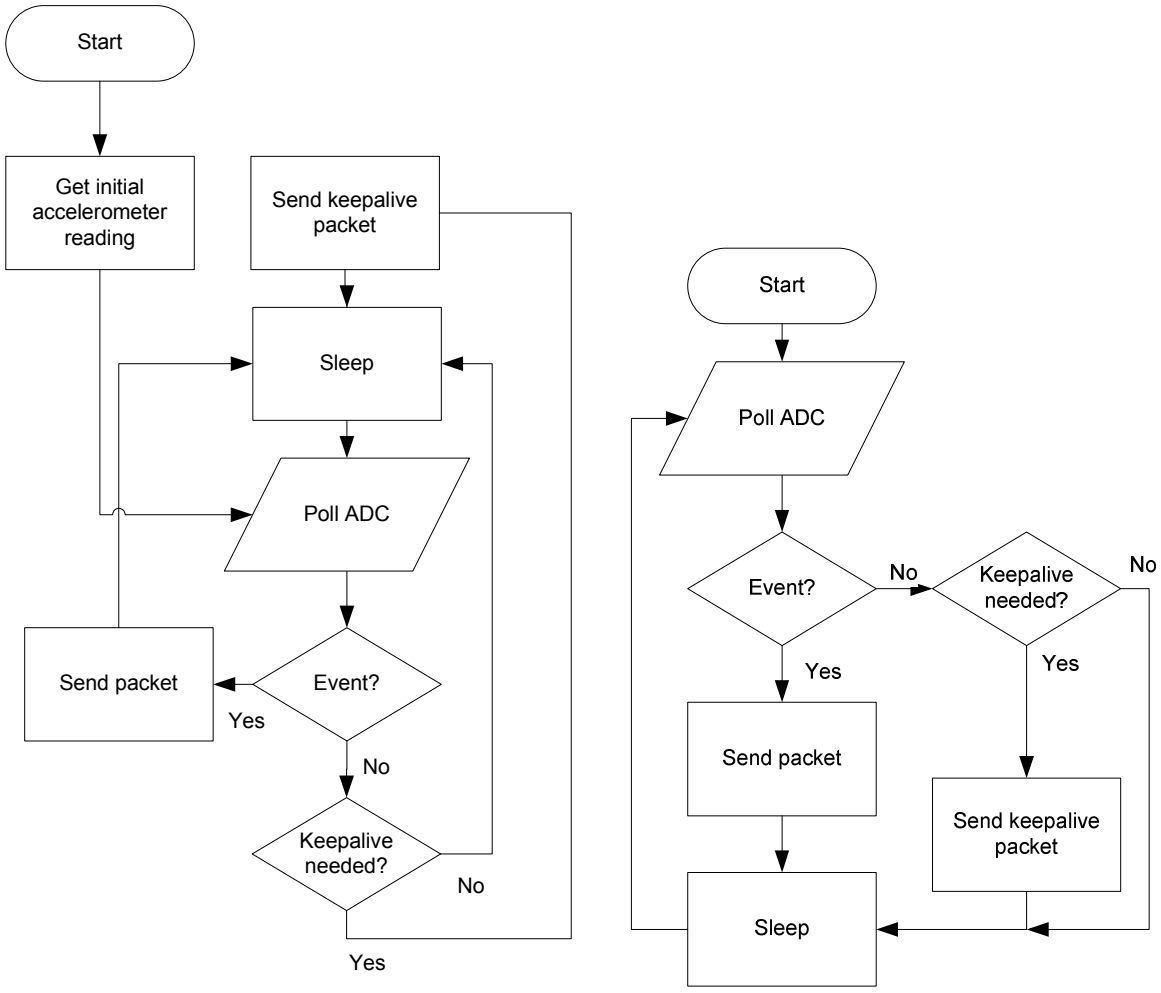


Figure 8: Accelerometer (left) and motion sensor (right) mote program flows

Each program begins with a startup sequence, initializing needed variables, the analog-to-digital converter, and user button functionality. The startup sequence for the accelerometer also incorporates an initial accelerometer reading to determine the starting orientation of the Tmote. Following these startup procedures, each Tmote begins monitoring their respective sensors; deciding if an event needs to be generated based upon the sensor readings it acquires. If no event is detected, the Tmotes provide user feedback in the form of a blinking LED for easily verifiable functionality.

Provided no event occurs for 60 seconds, the node will send a keep-alive packet to the base station, informing it that the mote is, in fact, still functional. A user can also send this packet from the Tmote manually by depressing the user button on the back of the Tmote Sky. If an event occurs, the

Tmote Sky will reset the keep-alive timer, ensuring that the mesh network is not cluttered with spurious data. Additionally, it will send a packet to the base station containing the requisite identifying information before returning to the beginning of the program. For the entire NesC source code of these programs, see Appendix B.

## 4.3 Infrastructure Programming

With the sensor node programming complete, the network could transmit its collective data as far as the sink node of the mesh network. At this point, we implemented the functionality necessary for the sink node to transfer its information to our storage medium, and eventually to the user.

### 4.3.1 Base Station Software

The base station software is marginally more complicated than the software present on the individual Tmote nodes. A high level depiction of the program flow can be seen in Figure 9.

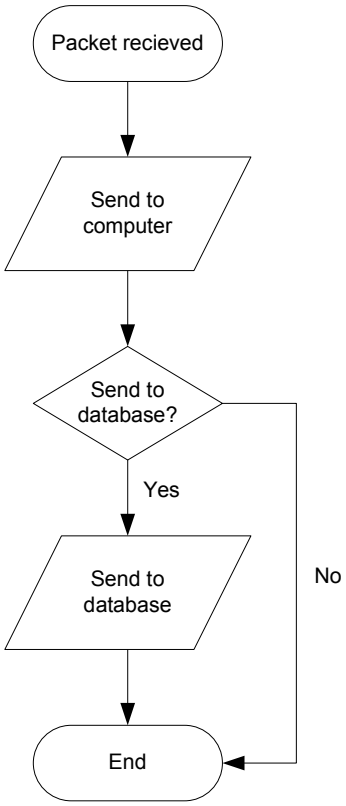


Figure 9: Base station program flow

Once the packet information is received by the sink node attached to a computer via the USB interface, the node forwards the information to a Java program running on the Cygwin command line. Next, the data is passed from the Java program to an AWK script to remove spurious data from the packet before passing on the data to a Bash script. This script then determines whether or not the data should be sent to the database, consulting the latest event from the given mote. If the mote has recently submitted information to the database, the packet is discarded so as not to flood the database with repeated data. Otherwise, the information is passed from Bash to the MySQL client. In either case, the Bash script presents some debug information to the user via Cygwin command line.

### 4.3.2 SQL Database

After the base station software has formatted the data appropriately, it is sent to the MySQL database. This database can be located on any remote computer and accessed via the Internet, or on

the base station itself. The database consists of a single table, 'alerts', which contains the information for every event generated by the wireless sensor network. Recorded data includes a timestamp of the event, event type, and any corresponding sensor data.

### 4.3.3 User Interface

With the data successfully sent to the MySQL database, any user with appropriate credentials can access the sensor network data through our user interface. This does not require Internet access if both the user interface and the MySQL server are running on the same computer. The user interface first retrieves all database rows, placing the data in tree data structures sorted by timestamp and Tmote id number. This storage scheme allows quick access to all information. Following the initial data retrieval, the user interface requests *only* new data from the database on a regular basis. This keeps the interface current while not requesting superfluous data. A screen capture of the user interface is shown in Figure 10.


Figure 10: User Interface

The user interface is divided into two main sections. The top section displays the status of each Tmote, and the timestamp of the last known contact with the node. In addition, users may input their own custom aliases for each Tmote, allowing for easy recognition of events. Upon clicking on a node in the status table, an additional mote information section is made available. This information is located in the lower area of the UI, and consists of every event in the database associated with the given node. Both sections of the interface update in real time, constantly providing the user with the current status of the network. Finally, the 'Reset selected mote' button enables the user to reset the status of the currently selected mote, allowing them to correct for false alarms or accidental triggering of the security system.

# 5. Further Research

During our project, we also spend time researching the functionality of the Tmote Sky device and looking for potential solutions to the problems inherent in the Tmote platform. Foremost among these problems is that notable lack of localization features present in the Tmote Sky device. Our research centered primarily on finding potential solutions to this problem.

## 5.1 Sensor Localization

The ease of deployment offered by wireless sensor networks often comes with a lack of precise localization. Short of using an absolute positioning system, such as GPS, to position and monitor each node, knowing a definite location for a node is a challenge. We explored two distinct methods of sensor node localization, both of which took advantage of the Tmote Sky's network capabilities to generate localization data. We first conducted an analysis of transmission latency as a prediction tool for distance between sensor nodes. Subsequently, we conducted an investigation of ultrasonic localization and its feasibility of implementation.

### 5.1.1 RF Localization Pilot Studies

In order to test the effectiveness of RF localization, we used an example program bundled with the Tmote Sky units, called Tmote Trawler. A feature of this program allowed us to display the current network topology as well as a sliding average of the latency between the sensor nodes. We set three sensor nodes at increasing distances apart in a flat parking lot, and recorded the latency between the nodes as observed by the Trawler program. A table of preliminary results is included below.

| Distance (meters) | Latency (ms) | Latency/Distance |
|---|---|---|
| **14** | 46 | 3.25 |
| **18** | 64 | 3.56 |
| **20** | 76 | 3.80 |

Table 1: Flat land RF Ranging Effectiveness

Preliminary statistical analysis of these values was very promising, as the correlation coefficient of distance to latency was greater than 0.99. A least squares regression yielded a line with a y-intercept close to zero and a slope of about 4.92. These results indicated a clear trend which could likely be used to determine the relative locations of sensor nodes, given that each node had direct line of sight with at least two other nodes. With these promising results, we connected two more nodes to the network, one with slightly obstructed line of sight to another node, and one fully obstructed from the view of any nodes. A table of results is included below.

| Distance (meters) | Latency (ms) | Latency/Distance | Notes |
|---|---|---|---|
| **5** | 64 | 12.8 | Complete Obstruction |
| **12** | 179 | 14.9 | Slight Obstruction |

**Table 2: Obstructed RF Ranging Effectiveness**

Unfortunately, this test showed wildly differing results from our initial testing. The obstructions did not interfere with the overall functionality of the network, but did vastly increase latencies between affected nodes. In our security application, it is certain that some sensor units will be placed without direct line of sight to another unit, and this result is unacceptable for node localization given the inconsistencies between the latency of nodes with line of sight and nodes with an obstructed view from one another.

### 5.1.2 Ultrasonic Localization

Ultrasonic localization utilizes high frequency sound waves combined with transmission latency and power calculations to accurately triangulate sensor node locations. This is ideally accomplished by fixing several “beacon” nodes, or nodes equipped with high-frequency sound emitters, in known positions using GPS or other absolute localization technology. In addition to the standard RF transceivers, the standard sensor nodes are equipped with microphones and programmed to detect the ultrasonic pulse. Using the ultrasonic pulse propagation time, RF signal propagation time, and RF signal strength it is possible to calculate the position of each node in one of several different ways.

To determine the locations of each sensor node, each beacon node in turn transmits an RF packet at the same time as a high-frequency sound pulse, and the sensor nodes compare the time of arrival of the ultrasonic pulse and the RF packet. At this point, it is mathematically possible to calculate node position using several methods, principally hyperbolic tri-lateration, triangulation, or maximum likelihood multilateration [10].

Hyperbolic tri-lateration locates a node by intersecting circles drawn around the three nodes nearest to the target of the search. Triangulation utilizes the relative locations of three nodes and the laws of sines and cosines to find the relative *direction* of each node, as opposed to the relative *distance*. Maximum likelihood multilateration locates a node by minimizing the differences between actual distance measurements and estimated distances of all surrounding nodes [10].

The localization systems derived from these technologies are robust and very accurate. Several graduate papers have been written on the subject, and practical implementations of this system have been developed. We do not have the temporal or fiscal resources to implement a system of this caliber, but it is certainly a technology worth pursuing in future research.

# 6. Recommendations and Conclusion

In the end, we designed and implemented an entire wireless sensor network in approximately six months using the Tmote Sky platform. Given the research inherent in this process, we feel this is a reasonable timeframe for this developing technology. Our proof-of-concept demonstrates the viability of WPAN-based wireless sensor networks despite the immaturity of the Tmote Sky platform. Given our experiences with wireless sensor networks, we recommend the following course of action for future work.

## 6.1 TMote Sky Platform and Alternatives

At about the halfway point in our project, the Tmote Sky sensor module reached end-of-life, and ceased to be supported by Moteiv. Even before this point, the provided development software was clearly not mature, lacking an API for the supported TinyOS version and often failing to compile provided example programs. This made developing software for the Tmote Sky frustrating, as necessary features were undocumented, and the only resources with these features included, the example programs, failed to compile with the provided code. Additionally, the Tmote Sky utilizes an outdated version of TinyOS. According to the TinyOS developers and our own research, TinyOS version 2 offers much improved functionality, stability, and documentation compared to the version included with the Tmote Sky modules.

For these reasons, any future wireless sensor network development absolutely *requires* adopting a new sensor platform. After implementing our proof-of-concept using the Tmote Sky, we believe that utilizing more modern wireless sensor technology would tremendously improve ease of application development, allowing for more effective sensor networks. Below are some examples of other wireless sensor network platforms that, we believe, offer the potential for marked improvement over Tmote Sky.

### 6.1.1 Sun Microsystems – Project Sun SPOT

Sun Microsystems' wireless sensor network platform is called the Sun SPOT, or Sun Small Programmable Object Technology. The Sun SPOT is an embedded platform, powered by rechargeable batteries and utilizing a 32-bit ARM9-based microcontroller in conjunction with a newer version of the same CC2420 transceiver, now rebadged the TI2420 and manufactured by Texas Instruments, found on the Tmote Sky. There are several features of the Sun SPOT that we feel makes it superior, at least from a development perspective, to the Tmote Sky platform.

The Sun SPOT units run a version of the Java Virtual machine optimized for the ARM9 microcontroller. This negates the need for an operating system, and allows developers to write node programs in Java, a language with which most developers will already have experience. Code written for the Sun SPOT platform can be debugged using Sun's development environment, which is based on the industry standard Eclipse IDE. This environment includes emulation of the sensor's hardware, allowing developers to rapidly implement node programs and test them on their local machines, with plenty of pertinent debugging information, instead of sensor node hardware.

Sun has also provided several resources for Sun SPOT developers. The Sun SPOT documentation offers APIs for all Java libraries in standard JavaDoc format, as well as full user's and developer's guides, both features sorely lacking from the Tmote Sky documentation. Additionally, Sun has established a community for users implementing Sun SPOT-based sensor networks which is moderated and contributed to by Sun employees and Sun SPOT engineers. Given these features, the Sun SPOT seems like a much improved platform for the construction of a wireless sensor network, with a much shorter implementation time.

### 6.1.2 Sentilla – Formerly Moteiv

In October of 2007, Moteiv announced their reorganization, reforming as the company Sentilla. With this corporate change came a shift in company initiatives. Working with existing Tmote hardware,

Sentilla soon announced the Sentilla Software Suite, which allows users to develop Java-based software for MSP430-based embedded platforms. This suite is aimed at users wishing to implement what Sentilla calls "pervasive computing," technology easily adapted to wireless sensor networks. Sentilla has yet to allow its development environment past beta testing, but in the future it will certainly be a technology worth investigating.

### 6.1.3 Implementing a ZigBee Network

Another direction a future wireless sensor network could take is the abandonment of TinyOS and the Java Virtual Machine altogether, choosing instead a ZigBee mesh network implementation. Currently, both TinyOS and the Sun SPOTs implement their own mesh networking protocols. The networking standards provided by the ZigBee Alliance are a third option for wireless sensor networks, but require more time from involved engineers.

As stated in Section 2.2.1, IEEE standard 802.15.4 outlines the standard for low power Wireless Personal Area Networks (WPAN), which provides the physical and MAC layers of a ZigBee network. To implement an entire ZigBee-based wireless sensor network, an engineer needs to implement the Network, Security, and Application layers described in the ZigBee Specification. This involves several tasks, including PCB-level design of the proposed node, and would take more time than implementing a sensor network on an existing platform. The advantage of this approach lies in its flexibility. Custom designing a sensor node allows the engineer to implement all necessary features from scratch instead of relying on a pre-built platform, ensuring that the resulting sensor would meet the required specifications.

Detailing the construction of such a network would be a complicated task, and is beyond the scope of this project. However, an MQP group at WPI has recently completed a project incorporating a ZigBee-based mesh network, which can provide additional information [11].

## 6.2 Additional Technologies

There are several more technologies relevant to the construction of wireless sensor networks that we did not have the means to pursue. These technologies could be useful to sensor networks, and should be considered in any future implementation. Our experiences with the Tmote Sky platform lent themselves to supporting two technologies in particular: alternative energy sources and node localization.

### 6.2.1 Alternative Energy Sources

The Tmote Sky units we used are powered using 1.5V AA size batteries. While they do provide a reasonable lifetime for the sensor nodes, disposable batteries are not the most environmentally friendly energy sources, and are not an ideal solution for a technology that has many uses in environmental monitoring. Future implementations of wireless sensor networks should consider the use of a more power-aware platform utilizing alternative energy sources such as solar energy, or even rechargeable batteries. The low-power nature of sensor nodes could lend itself well to a more environmentally friendly power source.

### 6.2.2 Node Localization

As discussed in Section 5.1, sensor localization is of prime importance to ad-hoc wireless sensor networks. Future work with ad-hoc mesh networks should include robust methods for node localization, as this will markedly increase the practicality of an ad-hoc network.

# 7. References


[1] TMote Sky Blog, Moteiv Corporation, Accessed 2/2008 <http://blog.moteiv.com/archives/tmote_sky/>

[2] TinyOS Mission Statement. TinyOS Community, Accessed 9/2007 – 11/2007 <http://www.tinyos.net/special/mission>

[3] Tmote Sky Quickstart Guide, Moteiv Corporation, Accessed 9/2007 – 11/2007 <http://www.moteiv.com/products/docs/tmote-sky-quickstart.pdf>

[4] SmartRF CC2420 Datasheet, Chipcon AS, Accessed 9/2007 – 11/2007 <http://inst.eecs.berkeley.edu/~cs150/Documents/CC2420.pdf>

[5] Adams, Jon and Heile, Bob, "Busy as a ZigBee," *IEEE Spectrum*, October 2006 <http://www.spectrum.ieee.org/oct06/4666/3>

[6] ZigBee Specification, ZigBee Alliance, Accessed 12/2007 <http://www.zigbee.org/en/spec_download/download_request.asp>

[7] Packet Link Layer, TinyOS Community, Accessed 2/2008 <http://www.tinyos.net/tinyos-2.x/doc/html/tep127.html

[8] Akyildiz, Ian F. and Wang, Xudong, "A Survey of Wireless Mesh Networks," *IEEE Radio Communications*, September 2005

[9] Wheeler, Andy, "ZigBee Wireless Networks for Industrial Systems," Accessed 9/2007 <http://www.microcontroller.com/Embedded.asp?did=149>

[10] Han, Chih-Chieh, Savvides, Andreas, and Strivastava, Mani B., "Dynamic Fine-Graned Localization in Ad-Hoc Networks of Sensors," Presented at the seventh annual international conference on Mobile Computing and Networking

[11] Bosman, Joseph A., Olivieri, Steven, Ozil, Ipek, and Steacy, Brandon C., Worcester Polytechnic Institute “Design of a Completely Wireless Security Camera System,” <http://www.wpi.edu/Pubs/E-project/Available/E-project-101107-142534/>

[12] Guibas, Leonidas and Zhao, Feng, Wireless Sensor Networks: An Information Processing Approach, Morgan Kaufman Publishers, San Francisco, CA, 2004

# 8. Appendix A – User Guide

**General Information**

- Terms
  - **Cygwin** – The Linux-in-Windows environment used to run the wireless sensor network side of the application (base station.)
  - **Base Station** – The computer that is connected to the sink node and moves information from the wireless sensor network to a database.
  - **MySQL** – The database software used to store event information.
  - **Sink Node** – The node (mote) that is to act as a relay between the wireless sensor network and a computer.
- Mote Specifics

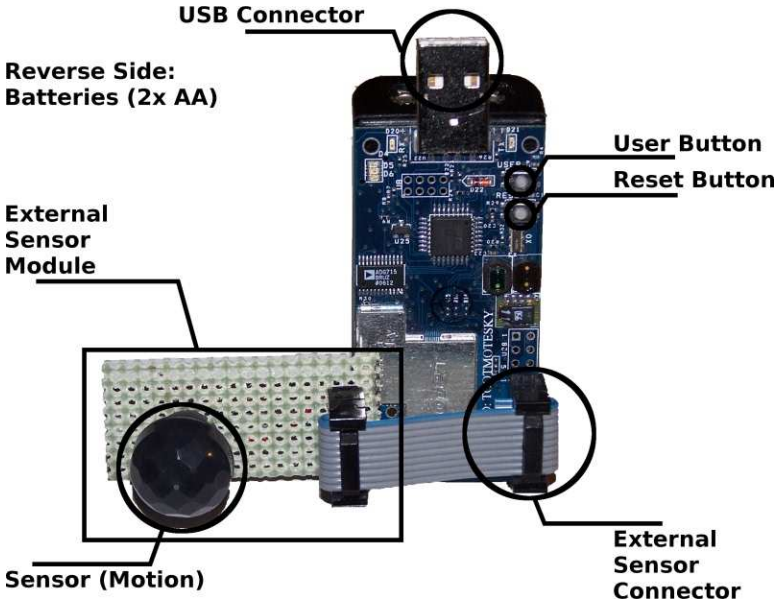


Figure 11: Motion Sensing Node

**General Instructions**

- Base Station Computer Prerequisites
  - Insert WJLMQP CD-ROM.
  - Extract the cygwin directory to the hard drive from *cygwindir.tar.gz*
  - Start a cygwin command prompt.

**Database**

- Initial Setup
  - The implementation housed on the CD-ROM is hardcoded to access a specific database for demonstration purposes. If you want to have the base station write to a database of your choosing, proceed with the following
    - Acquire a MySQL account on a machine well suited to hold sensed event information.

(Such as the accounts available on mysql.wpi.edu.)
    - Run the *mysql -hMySQL.Machine.Address -uUSERNAME -pPASSWORD wjlmqp < /cygwin/wjlmqp/extras/database.sql* command from the base station.
    - Update the */wjlmqp/extras/script.sh* file with your database information
- Running
    - Nothing, the database does not need to be directly interacted with by the user once the system is functional.

**Base station**

- Initial Setup
    - Change to the */cygwin/wjlmqp/base* directory.
    - Insert the mote to be the base station to any USB port on the computer (no batteries necessary.)
    - Run the *make tmote,1* command, this compiles the base station mote code and installs it giving the mote the address of 1 (Note: the base station mote should always have address 1.)
- Running
    - Insert the programmed base station mote to any USB port on the computer (no batteries necessary.)
    - Change to the */cygwin/wjlmqp/extras* directory.
    - Run *./run.sh* script to listen for data on all virtual COM ports.

**Motion Sensing Node**

- Initial Setup
    - Change to the */cygwin/wjlmqp/sensor-motion* directory.
    - Insert the mote to be a motion sensing node to any USB port on the computer (no batteries necessary.)
    - Run *make tmote,X* where *X* is a unique number used to identify the mote. *X* should not be less than 2, nor greater than 50, nor shared with any other mote on the network.
    - Repeat these steps for each motion sensing node to be on the network.
- Running
    - Place the motion sensing node in the location from which it should monitor.
    - Insert batteries into the motion sensing node.
    - (Optional) Press the user button or trigger the sensor with motion to ensure that the node is functional. Doing either should generate a LED change and an event visible from the graphical user interface.

**Acceleration Sensing Node**

- Initial Setup
    - Change to the */cygwin/wjlmqp/sensor-accel* directory.
    - Insert the mote to be a motion sensing node to any USB port on the computer (no batteries

necessary.)
    - Run *make tmote,X* where *X* is a unique number used to identify the mote. *X* should not be less than 2, nor greater than 50, nor shared with any other mote on the network.
    - Repeat these steps for each acceleration sensing node to be on the network.
- Running
    - Place the acceleration sensing node in the location from which it should monitor.
    - Insert batteries into the acceleration sensing node.
    - (Optional) Press the user button or trigger the sensor with acceleration ot ensure that the node is functional. Doing either should generate A LED change and an event visible from the graphical user interface.

**Graphical User Interface**

- Initial Setup
    - Ensure the system has a JRE or JDK properly installed and functioning.
    - Ensure the system has a network connection allowing it to access the database server.
    - For demonstration purposes, the UI is hardcoded to connect to a specific database on WPI's mySQL server. If you changed the mySQL server above, you'll need to modify the source code file ui/src/ui/Ui.java (included on the CD-ROM) accordingly.
- Running
    - Double-click on the ui/Sensor Network UI.jar file.
    - Monitor the interface for important events and clear false alarms as needed.

**Testing Operation**

In order to ensure proper functioning of a simple wireless sensor network take the following steps:

- Set up, per above, a database, base station, one sensing node and the graphical user interface.
- Press the user button on one of the sensing nodes and ensure when the user button is pressed
    - The LED lighting changes on the sensing node
    - The LED lighting changes on the sink node
    - The base station console output reports an event
    - The user interface reports an event
- Should the first step fail, ensure the sensing mote has the sensing software properly installed, is turned on and has sufficient battery power.
- Should the second step fail, ensure the sink mote has the sink mote software installed and is turned on.
- Should the third step fail, ensure that the sink mote is properly detected as attached to the base station by the base station's operating system.
- Should the forth step fail, ensure the base station has the ability and privileges necessary to access and modify the database. Additionally, ensure that the GUI has the ability to access and read from the database.

# 9. Appendix B – Mote Code

## 9.1 Base Station (Sink Node) Mote Code

### 9.1.1 baseC.nc

```
includes Message;

/**
 * Interface wiring for base/sink node
 */

configuration baseC
{
}
implementation
{
      components Main
            , baseM
            , GenericComm
            , LedsC
            , UserButtonC;

      Main.StdControl -> baseM;
      Main.StdControl -> GenericComm;

      baseM.SendMsg -> GenericComm.SendMsg[MSG_MAX];
      baseM.ReceiveMsg -> GenericComm.ReceiveMsg[MSG_MAX];
      baseM.Leds -> LedsC.Leds;
      baseM.Button ->UserButtonC;
}
```

### 9.1.2 baseM.nc

```
includes Message;

module baseM
{
      provides interface StdControl;
      uses interface SendMsg;
      uses interface ReceiveMsg;
      uses interface Leds;
      uses interface Button;
}

implementation
{
      /* The message to send over UART */
      TOS_Msg m_msg;

      /* Whether or not we're already transmitted over UART */
      bool m_sending;
```

```
/**
 * Initialization, called on boot
 */
command result_t StdControl.init()
{
      /* We start not sending anything */
      m_sending = FALSE;

      /* Make LEDs usable */
      call Leds.init();

      return SUCCESS;
}

/**
 * Start, also called on boot
 */
command result_t StdControl.start()
{
      /* Enable user-button */
      call Button.enable();

      return SUCCESS;
}

/**
 * Required for interface, noop
 */
command result_t StdControl.stop()
{
      return SUCCESS;
}

/**
 * Function called when we receive a packet
 */
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msg)
{
      /* Blink yellow LED on every packet */
      call Leds.yellowToggle();

      /* If we're not sending and we're the sink node */
      if(!m_sending && TOS_LOCAL_ADDRESS == 1) {
            /* Extract data and send it over uart */
            Message* body = (Message*)msg->data;
            Message* toc = (Message*)m_msg.data;
            memset(toc, 0, sizeof(Message));
            toc->num = body->num;
            toc->type = body->type;
            toc->src = body->src;
            toc->body = body->body;

            /* Blink red LED on too large packet */
            if(msg->length >= MAX_LENGTH) {
                  call Leds.redOn();
            }
```

```
                  /* Try to send over uart */
                  if(call SendMsg.send(TOS_UART_ADDR, sizeof(Message),
&m_msg) == SUCCESS) {
                        /* Blink green if we sent properly */
                        call Leds.greenToggle();
                        call Leds.redOff();
                        m_sending = TRUE;
                  } else {
                        call Leds.redToggle();
                        call Leds.greenOff();
                  }
            }
            return msg;
      }

      /**
       * Called when we're done sending over the uart
       */
      event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
      {
            /* We're no longer busy sending */
            m_sending = FALSE;

            return SUCCESS;
      }

      /**
       * Required for interface
       */
      async event void Button.pressed(uint32_t time)
      {
      }

      /**
       * Blink a light on user-button press -- why not?!
       */
      async event void Button.released(uint32_t time)
      {
            call Leds.redToggle();
      }
}
```

## 9.2 Accelerometer Mote Code

### 9.2.1 accelC.nc

```
includes Message;
includes adcConfig;

/**
 * Interface wiring for an accelerometer mote
 */

configuration accelC
```

```
{
}

implementation
{
	components Main
		, accelM
		, TimerC
		, GenericComm
		, LedsC
		, ADCC
		, UserButtonC;

	Main.StdControl -> GenericComm;
	Main.StdControl -> ADCC;
	Main.StdControl -> accelM;
	Main.StdControl -> TimerC;

	accelM.Button -> UserButtonC;

	accelM.Timer -> TimerC.Timer[unique("Timer")];
	accelM.SendMsg -> GenericComm.SendMsg[MSG_MAX];
	accelM.Leds -> LedsC.Leds;

	accelM.ADC -> ADCC.ADC[TOS_ADC_GIO0_PORT];
	accelM.ADCControl -> ADCC;
}
```

### 9.2.2 accelM.nc

```
includes Message;

module accelM
{
	provides interface StdControl;
	uses interface Timer;
	uses interface SendMsg;
	uses interface Leds;
	uses interface ADC;
	uses interface ADCControl;
	uses interface Button;
}

implementation
{
	/* Holding variable for any message to send */
	TOS_Msg m_msg;

	/* How many 1/5ths of a second it has been since any packet was sent */
	unsigned int keepalive;

	/* Data read over the ADC */
	int m_int;

	/* Boolean to indicate if we are currently busy sending */
	bool m_sending;
```

```
	/* The number of packets we have sent since bootup */
	uint32_t packets;

	/* Holds the count down to when we pull for an initial accelerometer
reading */
	int countdown;

	/* Holds the 'normal' (initial) accelerometer reading */
	int normal;

	/**
	 * Initialization, called on boot
	 */
	command result_t StdControl.init()
	{
		/* Initialize variables to sensible starts */
		atomic m_int = 0;
		packets = 0;
		countdown = 15 * 5;
		keepalive = 1;
		m_sending = FALSE;

		/* Make LEDs usable */
		call Leds.init();

		/* Initialize ADC */
		call ADCControl.init();
		call ADCControl.bindPort(TOS_ADC_GIO0_PORT,
TOSH_ACTUAL_ADC_GIO0_PORT);

		return SUCCESS;
	}

	/**
	 * Start, also called on boot
	 */
	command result_t StdControl.start()
	{
		/* Start the periodic timer at every 200 ms */
		call Timer.start(TIMER_REPEAT, 200);

		/* Pull some initial data from the ADC */
		call ADC.getData();

		/* Enable the user-button */
		call Button.enable();

		return SUCCESS;
	}

	/**
	 * Required for interface, noop
	 */
	command result_t StdControl.stop()
	{
		return SUCCESS;
	}
```

```
	/**
	 * Called when the timer fires, once every 200 ms
	 * This is also how often the ADC is polled, quicker events will be
	 * missed.
	 */
	event result_t Timer.fired()
	{
		/* Atomic query const for m_int */
		int n;

		/* Atomic query const for m_sending */
		bool s;

		atomic n = m_int;
		atomic s = m_sending;

		/* If we have not finished the countdown, and thus don't have
		 * an initial sensor reading to go from */
		if(countdown > 0) {
			/* Blink the yellow LED */
			call Leds.yellowToggle();

			/* Countdown and possibly get an initial reading */
			--countdown;
			if(countdown == 3) {
				call ADC.getData();
			} else if(countdown == 1) {
				normal = n;
			}
			return SUCCESS;
		}

		/* If the data most recently pulled from the ADC does not
		 * constitute an event */
		if(abs(normal - n) < 400) {
			/* If it's been 60 seconds with no event, reset the
keepalive clock */
			if(++keepalive >= 60 * 5) {
				keepalive = 0;
			}

			/* Send a keepalive packet, if the keepalive clock has been
reset */
			if(keepalive == 0 && s == FALSE) {
				Message* send = (Message*)m_msg.data;
				send->body = 0;
				send->src = TOS_LOCAL_ADDRESS;
				send->type = KEEPALIVE;
				send->num = packets++;

				/* Send the packet and note that we're now busy
sending */
				if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
				{
					atomic {
```

```
                                    m_sending = TRUE;
                              }
                        }
                  }

                  /* Blink the red LED */
                  call Leds.redToggle();
                  call Leds.greenOff();
                  call Leds.yellowOff();

                  /* Refresh our reading from the ADC */
                  call ADC.getData();

                  return SUCCESS;
            }

            /* If we are not busy sending and we have an event */
            if(s == FALSE)
            {
                  Message* send = (Message*)m_msg.data;
                  send->body = abs(normal - n);
                  send->src = TOS_LOCAL_ADDRESS;
                  send->type = ACCELR;
                  send->num = packets++;

                  /* Send the event data to the sink node */
                  if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
                  {
                        /* Send high-order bits to the LEDs */
                        call Leds.set( n >> 9 );
                        atomic {
                              m_sending = TRUE;
                        }
                  }
            }

            /* Get new data from the ADC */
            call ADC.getData();

            return SUCCESS;
      }

      /**
       * Called when data is ready from the ADC
       */
      async event result_t ADC.dataReady(uint16_t data)
      {
            /* Put the new data into our common ADC storage */
            atomic m_int = data;

            return SUCCESS;
      }

      /**
       * Called when we are done sending
       */
```

```
	event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
	{
		/* Mark the fact that sending is complete and we are free to
		 * do so again */
		atomic {
			m_sending = FALSE;
		}

		return SUCCESS;
	}

	/**
	 * Called when the user-button is pressed
	 */
	async event void Button.pressed(uint32_t time)
	{
		/* Atomic const to query for sending */
		bool s;
		atomic s = m_sending;

		/* Send a keep alive packet */
		call Leds.redOn();
		call Leds.yellowOn();
		call Leds.greenOn();
		if(s == FALSE) {
			Message* send = (Message*)m_msg.data;
			send->body = 0;
			send->src = TOS_LOCAL_ADDRESS;
			send->type = KEEPALIVE;

			/* Send the packet and note that we are now busy sending */
			if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
			{
				atomic {
					m_sending = TRUE;
				}
			}
		}
	}

	/**
	 * Called when the user button is released
	 * This is merely used to indicate, on the mote, that the keepalive
	 * packet has been sent
	 */
	async event void Button.released(uint32_t time)
	{
		/* Blink some LEDs, probably */
		call Leds.redOn();
		call Leds.yellowOn();
		call Leds.greenOn();
	}
}
```

## 9.3 Motion Sensor Mote Code

### 9.3.1 motionC.nc

```
includes Message;
includes adcConfig;

/**
 * Interface wiring for a motion sensor mote
 */

configuration motionC
{
}

implementation
{
      components Main
            , motionM
            , TimerC
            , GenericComm
            , LedsC
            , ADCC
            , UserButtonC;

      Main.StdControl -> GenericComm;
      Main.StdControl -> ADCC;
      Main.StdControl -> motionM;
      Main.StdControl -> TimerC;

      motionM.Button -> UserButtonC;

      motionM.Timer -> TimerC.Timer[unique("Timer")];
      motionM.SendMsg -> GenericComm.SendMsg[MSG_MAX];
      motionM.Leds -> LedsC.Leds;

      motionM.ADC -> ADCC.ADC[TOS_ADC_GIO0_PORT];
      motionM.ADCControl -> ADCC;
}
```

### 9.3.2 motionM.nc

```
includes Message;

module motionM
{
      provides interface StdControl;
      uses interface Timer;
      uses interface SendMsg;
      uses interface Leds;
      uses interface ADC;
      uses interface ADCControl;
      uses interface Button;
}
```

```
implementation
{
      /* Holding variable for any message to send */
      TOS_Msg m_msg;

      /* How many 1/5ths of a second it has been since any packet was sent */
      unsigned int keepalive;

      /* Data read over the ADC */
      int m_int;

      /* Boolean to indicate if we are currently busy sending */
      bool m_sending;

      /* The number of packets we have sent since bootup */
      uint32_t packets;

      /**
       * Initialization, called on boot
       */
      command result_t StdControl.init()
      {
            /* Initialize variables to sensible starts */
            atomic m_int = 0;
            packets = 0;
            keepalive = 1;
            m_sending = FALSE;

            /* Make LEDs usable */
            call Leds.init();

            /* Initialize ADC */
            call ADCControl.init();
            call ADCControl.bindPort(TOS_ADC_GIO0_PORT,
TOSH_ACTUAL_ADC_GIO0_PORT);

            return SUCCESS;
      }

      /**
       * Start, also called on boot
       */
      command result_t StdControl.start()
      {
            /* Start the periodic timer at every 200 ms */
            call Timer.start(TIMER_REPEAT, 200);

            /* Pull some initial data from the ADC */
            call ADC.getData();

            /* Enable the user-button */
            call Button.enable();

            return SUCCESS;
      }

      /**
```

```
	 * Required for interface, noop
	 */
	command result_t StdControl.stop()
	{
		return SUCCESS;
	}

	/**
	 * Called when the timer fires, once every 200 ms
	 * This is also how often the ADC is polled, quicker events will be
	 * missed.
	 */
	event result_t Timer.fired()
	{
		/* Atomic query const for m_int */
		int n;

		/* Atomic query const for m_sending */
		bool s;

		atomic n = m_int;
		atomic s = m_sending;

		/* If the data does not constitute an event */
		if(n < 2000) {
			/* If it's been 60 seconds with no event, reset the
keepalive clock */
			if(++keepalive >= 60 * 5) {
				keepalive = 0;
			}

			/* Send a keepalive packet, if the keepalive clock has been
reset */
			if(keepalive == 0 && s == FALSE) {
				Message* send = (Message*)m_msg.data;
				send->body = 0;
				send->src = TOS_LOCAL_ADDRESS;
				send->type = KEEPALIVE;
				send->num = packets++;

				/* Send the packet and note that we're now busy
sending */
				if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
				{
					atomic {
						m_sending = TRUE;
					}
				}
			}

			/* Blink the red LED */
			call Leds.redToggle();
			call Leds.greenOff();
			call Leds.yellowOff();

			/* Refresh our reading from the ADC */
```

```
                call ADC.getData();

                return SUCCESS;
        }

        /* If we are not busy sending and we have an event */
        if(s == FALSE)
        {
                Message* send = (Message*)m_msg.data;
                send->body = n;
                send->src = TOS_LOCAL_ADDRESS;
                send->type = MOTION;
                send->num = packets++;

                /* Send the event data to the sink node */
                if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
                {
                        /* Send high-order bits to the LEDs */
                        call Leds.set( n >> 9 );
                        atomic {
                                m_sending = TRUE;
                        }
                }
        }

        /* Get new data from the ADC */
        call ADC.getData();

        return SUCCESS;
    }

    /**
     * Called when data is ready from the ADC
     */
    async event result_t ADC.dataReady(uint16_t data)
    {
        /* Put the new data into our common ADC storage */
        atomic m_int = data;

        return SUCCESS;
    }

    /**
     * Called when we are done sending
     */
    event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
    {
        /* Mark the fact that sending is complete and we are free to
         * do so again */
        atomic {
                m_sending = FALSE;
        }

        return SUCCESS;
    }
```

```
	/**
	 * Called when the user-button is pressed
	 */
	async event void Button.pressed(uint32_t time)
	{
		/* Atomic const to query for sending */
		bool s;
		atomic s = m_sending;

		/* Send a keep alive packet */
		call Leds.redOn();
		call Leds.yellowOn();
		call Leds.greenOn();
		if(s == FALSE) {
			Message* send = (Message*)m_msg.data;
			send->body = 0;
			send->src = TOS_LOCAL_ADDRESS;
			send->type = KEEPALIVE;

			/* Send the packet and note that we are now busy sending */
			if(call SendMsg.send(1, sizeof(Message), &m_msg) ==
SUCCESS)
			{
				atomic {
					m_sending = TRUE;
				}
			}
		}
	}

	/**
	 * Called when the user button is released
	 * This is merely used to indicate, on the mote, that the keepalive
	 * packet has been sent
	 */
	async event void Button.released(uint32_t time)
	{
		/* Blink some LEDs, probably */
		call Leds.redOn();
		call Leds.yellowOn();
		call Leds.greenOn();
	}
}
```

## 9.4 Shared Mote Code

### 9.4.1 adcConfig.h

```
enum
{
	TOS_ADC_GIO0_PORT = unique("ADCPort"),

	TOSH_ACTUAL_ADC_GIO0_PORT = ASSOCIATE_ADC_CHANNEL(
		INPUT_CHANNEL_A0,
```

```
            REFERENCE_VREFplus_AVss,
            REFVOLT_LEVEL_1_5
      ),
};
```

### 9.4.2 Message.h

```
/* Maximum body length */
enum {
      MSG_MAX = 4,
      MAX_LENGTH = 3
};

/* Sensor types */
enum {
      KEEPALIVE = 0,
      MOTION = 1,
      ACCELR = 2,
      AUDIBL = 3
};


/* All fields in a message */
typedef struct message {
      uint32_t num;
      uint8_t type;
      uint16_t src;
      int32_t     body;
} Message;
```

# 10. Appendix C – Base Station (Computer) Code

## 10.1 run.sh

```
#!/bin/bash

# Runs the alert processing script on all COM ports

for i in `seq 1 10` ; do
      C="COM$i"
      echo "Listening for data on $C..."
      MOTECOM=serial@$C:tmote java net.tinyos.tools.ListenRaw $C 2>/dev/null
| awk '{ print $15 $16 $17 $18 " " $19 " " $20 $21 " " $23 $24 } fflush()' |
./script.sh
Done
```

## 10.2 script.sh

```
#!/bin/bash

# Sets last event for
sle ()
{
      le[$1]=$2
}

# Checks if another event should be added
cle ()
{
      # Make sure the source is part of our network
      if [ $1 -gt 256 ] ; then
            echo
            echo "Spurious garbage from supposed, ID $1."
            return 0
      fi

      cur="`date +%s`"
      thn="${le[$1]}"
      if [ $((cur-thn)) -ge 60 ] ; then
            sle $1 $cur
            return 1
      fi
      return 0
}


# Last events array
declare -a le

# Make sure we process all new events
for i in `seq 256` ; do
      sle $i 0
done
```

```
# Options to bc to convert from hex to decimal
bcp="obase=10;ibase=16;"

# Read in data passed over stdin from run.sh
while read -r num typ src bdy
do
      source="`echo $bcp$src | bc`"
      if [[ "$source" -gt "50" ]] ; then
            continue
      fi

      if [ -n "$num" ] ; then
            body_b1="`echo $bdy | cut -b3,4`"
            body_b2="`echo $bdy | cut -b1,2`"

            body="`echo $bcp$body_b1$body_b2 | bc`"
            type="`echo $bcp$typ | bc`"

            if [ "$type" -eq "0" ] ; then
                  echo
                  echo "DB: adding keepalive for $source"
                  echo "INSERT INTO alerts (type,mote,data) VALUES
('$type','$source','0');" | /mysql -hmysql.wpi.edu -uchanin -pMN4oHN wjlmqp
            fi

            if [ "$type" -eq "1" -a $body -gt 2000 ] ; then

                  cle $source

                  if [ "$?" -eq "1" ] ; then
                        num_b1="`echo $num | cut -b7,8`"
                        num_b2="`echo $num | cut -b5,6`"
                        num_b3="`echo $num | cut -b3,4`"
                        num_b4="`echo $num | cut -b1,2`"

                        number="`echo $bcp$num_b1$num_b2$num_b3$num_b4 | bc`"
                        echo
                        echo "DB: number: $number, type: $type (motion),
source: $source, body: $body"
                        echo "INSERT INTO alerts (type,mote,data) VALUES
('$type','$source','$body');" | /mysql -hmysql.wpi.edu -uchanin -pMN4oHN
wjlmqp
                  else
                        echo -n "$source "
                  fi
            fi


            if [ "$type" -eq "2" -a $body -gt 20 ] ; then

                  cle $source

                  if [ "$?" -eq "1" ] ; then
                        num_b1="`echo $num | cut -b7,8`"
                        num_b2="`echo $num | cut -b5,6`"
                        num_b3="`echo $num | cut -b3,4`"
                        num_b4="`echo $num | cut -b1,2`"
```

```
                        number="`echo $bcp$num_b1$num_b2$num_b3$num_b4 | bc`"
                        echo
                        echo "DB: number: $number, type: $type (accel),
source: $source, body: $body"
                        echo "INSERT INTO alerts (type,mote,data) VALUES
('$type','$source','$body');" | /mysql -hmysql.wpi.edu -uchanin -pMN4oHN
wjlmqp
                else
                        echo -n "$source "
                fi
            fi
      fi
done

exit 0
```

# 11. Appendix D –Database Table SQL Description (database.sql)

```
-- This file describes the structure of the alerts table, for MySQL


-- Create the table

CREATE TABLE alerts (`type` varchar(255) NOT NULL, `mote` varchar(4) NOT
NULL, data TEXT, `date` timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL, `id`
BIGINT AUTO_INCREMENT PRIMARY KEY);
```

# 12. Appendix E – User Interface Code

## 12.1 UI.java

```
package ui;

/* Import necessary swing and awt libraries */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

/* Import java SQL library */
import java.sql.*;

/* Import java util libraries */
import java.util.*;

/**
 * A user interface for a security system implemented using TMote Sky and
mySQL
 * @author Andrew Halloran, Jonathan Chanin
 * @version 1.0
 */
public class Ui extends JPanel implements TableModelListener,ActionListener {

	static final long serialVersionUID = 1;

	protected static JTextArea console;
	protected static JTable moteTable;
	protected static JScrollPane tablePane, consolePane;
	protected static JLabel statusLabel, reportLabel;
	protected static JButton resetButton;
	protected static DefaultTableModel tableModel;
	protected static TreeMap<Integer,Mote> motes;

	/**
	 * Default constuctor method
	 */
	Ui() {

		super(new GridBagLayout());
		GridBagConstraints c = new GridBagConstraints();

		this.setSize(640, 480);

		/* Mote table status label (tm) */
		statusLabel = new JLabel(" Mote Status:");
		c.fill = GridBagConstraints.HORIZONTAL;
		c.weightx = 1.0;
		c.gridx = 0;
		c.gridy = 0;
		this.add(statusLabel, c);
```

```
            /* Mote table definition */
            tableModel = new DefaultTableModel();
            moteTable = new JTable(tableModel);
            tableModel.addColumn("ID");
            tableModel.addColumn("Alias");
            tableModel.addColumn("Status");
            tableModel.addColumn("Last Heard From");

            tablePane = new JScrollPane(moteTable);
            c.fill = GridBagConstraints.HORIZONTAL;
            c.weightx = 1.0;
            c.gridx = 0;
            c.gridy = 1;
            this.add(tablePane, c);

            /* Mote report text area label */
            reportLabel = new JLabel(" Mote Information:");
            c.fill = GridBagConstraints.HORIZONTAL;
            c.weightx = 1.0;
            c.gridx = 0;
            c.gridy = 3;
            this.add(reportLabel, c);

            /* Mote report text area */
            console = new JTextArea(12,1);
            consolePane = new JScrollPane(console);
            c.fill = GridBagConstraints.HORIZONTAL;
            c.weightx = 1.0;
            c.gridx = 0;
            c.gridy = 4;
            this.add(consolePane, c);

            /* Reset button definition */
            resetButton = new JButton("Reset selected mote");
            resetButton.addActionListener(this);
            c.fill = GridBagConstraints.CENTER;
            c.weightx = 1.0;
            c.gridx = 0;
            c.gridy = 5;
            this.add(resetButton, c);

            tableModel.addTableModelListener(this);
      }

      /**
       * @param args Command line arguments, not used.
       */
      public static void main(String[] args) {
            mainProgram();
      }

      /**
       * The main loop of the program.  Makes the GUI visible, establishes a
connection to the mySQL
       * database, polls the database, and updates the contained information
periodically
```

```
	 */
	private static void mainProgram() {
		/* mainWindow is the main JFrame */
		JFrame mainWindow = new JFrame("Sensor Network UI");
		mainWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		/* Adds the Ui to the mainWindow JFrame */
		mainWindow.setResizable(false);
		mainWindow.add(new Ui());

		/* Packs the UI and makes it visible */
		mainWindow.pack();
		mainWindow.setVisible(true);

		/* String to hold the last updated date */
		String lastUpdated = "\"2008-01-01 00:00:00.0\"";

		/* Create SQL statement, results set, and connection variables */
		Statement stmt;
		ResultSet res;
		Connection DBConn = null;

		/* Create ArrayList of available motes */
		motes = new TreeMap<Integer,Mote>();

		try {
			/* Load the SQL Driver */
			Class.forName("com.mysql.jdbc.Driver").newInstance();

			/* Obtain a connection from DriverManager */
			DBConn =
DriverManager.getConnection("jdbc:mysql://mysql.wpi.edu/wjlmqp?" +

"user=chanin&password=MN4oHN");
		}
		catch(Exception e) {
			console.append("Epic fail detected.\n");
			console.append(e.getMessage() + "\n");
		}

		/* Database polling loop */
		while(true) {
			try {

				/* Execute a select query */
				stmt = DBConn.createStatement();
				res = stmt.executeQuery("SELECT * FROM alerts WHERE
date > " + lastUpdated +
                                                                " order by
date;");

				/* Move through Mote list and update */
				if(res.last()) {
					/* Save the last updated date */
					lastUpdated = '\"' + res.getString("date") +
'\"';
					/* Parse the new events */
```

```
                        motes = parseLine(motes, res);
                        while(res.previous()) {
                              motes = parseLine(motes, res);
                        }
                  }

                  /* Begin closing connections */
                  if(res != null) {
                        try {
                              res.close();
                        }
                        catch(SQLException sqlex) {
                              /* Ignore the exception */
                        }
                  }

                  if(stmt != null) {
                        try {
                              stmt.close();
                        }
                        catch(SQLException sqlex) {
                              /* Ignore the exception */
                        }
                  }
            }
            catch(Exception e) {
                  console.append("Epic fail detected.\n");
                  console.append(e.getMessage() + "\n");
            }

            try {
                  parseMotes(motes, tableModel, moteTable, console);
                  Thread.sleep(250);
            }
            catch(Exception e) {
                  System.out.println("Sleep exception.");
                  /* Do nothing */
            }
      }
}

/**
 * Parses a line in an SQL ResultSet, adding the appropriate Event to
the appropriate Mote
 * in the TreeMap<Integer,Mote> passed to the function.
 * @param motes A TreeMap of type Integer,Mote
 * @param res A ResultSet from a mySQL database query
 * @return The TreeMap motes, with the Event from the ResultSet added
to the appropriate Mote
 * @throws SQLException
 */
private static TreeMap<Integer,Mote> parseLine(TreeMap<Integer,Mote>
motes, ResultSet res) throws SQLException {

      int tempMoteID;
      Mote tempMote;
```

```
            /* Retrieve Mote ID from the database row */
            tempMoteID = res.getInt("mote");

            /* If a Mote exists with the same MoteID in the motes TreeMap */
            if(motes.containsKey(tempMoteID)) {
                  /* Find the Mote, and pull it from the list */
                  tempMote = motes.get(tempMoteID);
                  motes.remove(tempMoteID);
            }
            /* Else create a new Mote for insertion */
            else {
                  tempMote = new Mote(tempMoteID);
            }

            /* Add the event from the database into the Mote's event tree */
            Event tempEvent = new Event(res.getInt("type"),
res.getTimestamp("date"));
            tempMote.addEvent(tempEvent);

            /* Add the Mote into the motes TreeMap */
            motes.put(tempMoteID, tempMote);

            /* Return the Mote ArrayList */
            return motes;
      }

      /**
       * Parses a TreeMap of Motes and a DefaultTableModel to ensure that the
information is correct
       * @param motes A TreeMap of type Integer,Mote
       * @param tableModel a DefaultTableModel to be checked
       * @param moteTable a JTable with mote information
       * @param console a JTextArea to be updated
       */
      private static void parseMotes(TreeMap<Integer,Mote> motes,
DefaultTableModel tableModel, JTable moteTable, JTextArea console) {

            /* Retrieve an iterator for the keys of the motes TreeMap */
            Set<Integer> iSet = motes.keySet();
            Iterator<Integer> iterator = iSet.iterator();

            /* If there are more motes than table rows, remake the table */
            if(tableModel.getRowCount() != iSet.size()) {
                  while(tableModel.getRowCount() > 0) {
                        tableModel.removeRow(0);
                  }
                  while(iterator.hasNext()) {

      tableModel.addRow(motes.get(iterator.next()).getTableRow());
                  }
            }
            /* Else check each row for changes and update as necessary */
            else {
                  /* Retrieve an array of keys for updating */
                  Object[] iArray = iSet.toArray();
                  /* Check for selected row, and update the mote status text
area */
```

```
                  int selectedRow = moteTable.getSelectedRow();
                  if(selectedRow != -1) {
                        String tempText =
motes.get(iArray[selectedRow]).toString();
                        if(!tempText.equals((String)console.getText())) {
                              console.setText(tempText);
                        }
                  }
                  else {
                        console.setText("");
                  }
                  /* For each row in the table */
                  for(int i=0; i<tableModel.getRowCount(); i++) {
                        /* Retrieve the table row from the selected mote */
                        Object[] temp =
motes.get((Integer)iArray[i]).getTableRow();
                        /* For each column in the table */
                        for(int j=0; j<tableModel.getColumnCount(); j++) {
                              /* If the mote pulled from the list isn't the
same as the UI table */
                              if(!temp[j].equals(tableModel.getValueAt(i,
j))) {
                                    /* Update the table */
                                    tableModel.setValueAt(temp[j], i, j);
                              }
                        }
                  }
            }
      }

      /**
       * TableModelEvent listener for the JTable, updates the alias field if
necessary
       * (Static variables read: motes, tableModel)
       */
      public void tableChanged(TableModelEvent evt) {

            /* If a table entry was updated, not inserted or removed */
            if(evt.getType() == TableModelEvent.UPDATE) {

                  /* Retrieve the row and column of the update, and the table
model */
                  int row = evt.getFirstRow();
                  int column = evt.getColumn();
                  TableModel tableModel = (TableModel)evt.getSource();

                  /* If the Alias field was changed */
                  if(column == 1) {

                        /* Retrieve an array of keys from the TreeMap */
                        Set<Integer> iSet = motes.keySet();
                        Object[] iArray = iSet.toArray();

                        /* Remove the mote from the list, update the alias,
and put it back */
                        Mote temp = motes.remove(iArray[row]);
```

```
                        temp.setAlias(tableModel.getValueAt(row,
column).toString());
                        motes.put((Integer)iArray[row], temp);
                  }
            }
      }

      /**
       * ActionEvent listener for the JButton, sets the selected Mote status
to secure
       * (Static variables read: motes, moteTable)
       */
      public void actionPerformed(ActionEvent evt) {

            /* Retrieve an array of keys from the TreeMap */
            Set<Integer> iSet = motes.keySet();
            Object[] iArray = iSet.toArray();

            /* Retrieve the selected database row */
            int selectedRow = moteTable.getSelectedRow();

            /* Set mote to secure */
            motes.get(iArray[selectedRow]).setSecure();
      }
}
```

## 12.2 Mote.java

```
package ui;

import java.util.*;

/**
 * A class for holding information about a Mote as described in an SQL
database
 * @author Andrew Halloran, Jonathan Chanin, 2008
 * @version 1.0
 */
public class Mote {

      private int id;
      private String alias;
      private TreeSet<Event> events;
      private Timer timer;
      private boolean isConnected, isSecure;

      /**
       * Default constructor method
       * @param id The id number of the Mote
       */
      Mote(int id) {
            this.id = id;
            this.alias = "Mote #" + Integer.toString(id);
            this.events = new TreeSet<Event>();
            this.isConnected = false;
            this.isSecure = true;
```

```
            this.timer = new Timer();
      }

      /**
       * Retrieves the id number of the Mote
       * @return The id number of the Mote
       */
      public int getId() {
            return this.id;
      }

      /**
       * Retrieves the most recent Event stored in the Mote
       * @return The most recent recorded Event
       */
      public Event getLastEvent() {
            return events.last();
      }

      /**
       * Adds a new event to the Mote
       * @param event The Event to be added
       */
      public void addEvent(Event event) {
            events.add(event);

            this.isConnected = true;
            this.timer.cancel();
            this.timer = new Timer();
            this.timer.schedule(new KeepAliveCheck(this), new
Date(this.getLastEvent().eventTimestamp.getTime() + 305000));
            if(event.eventType != 0) this.isSecure = false;
      }

      /**
       * Retrieves the alias of the Mote
       * @return The alias of the Mote
       */
      public String getAlias() {
            return this.alias;
      }

      /**
       * Sets the alias of the Mote
       * @param alias The alias of the Mote
       */
      public void setAlias(String alias) {
            this.alias = alias;
      }

      /**
       * Asserts the secure boolean on the Mote
       */
      public void setSecure() {
            this.isSecure = true;
      }
```

```
	/**
	 * Retrieves the status of the Mote {Connected, Not connected} and
{Secure, event detected!}
	 * @return A string containing the Mote status
	 */
	public String getStatus() {
		String temp = "";
		if(isConnected) temp += "Connected, ";
		else temp += "Not connected, ";
		if(!isSecure) temp += "event detected!";
		else temp += "secure.";
		return temp;
	}

	/**
	 * Retrieves the Mote's information, formatted for use as a row in the
main GUI JTable
	 * @return An Object[4] containing the Mote's information
	 */
	public Object[] getTableRow() {
		Object[] tempArray = {Integer.toString(this.id),
					this.alias,
					this.getStatus(),

this.getLastEvent().eventTimestamp.toString()};
		return tempArray;
	}

	/**
	 * Converts the Mote's information, including id and a list of events,
into String format
	 */
	public String toString() {
		String temp = this.alias + " has the following events:\n";
		Iterator<Event> iterator = this.events.iterator();
		Stack<String> tempStack = new Stack<String>();
		while(iterator.hasNext()) {
			tempStack.push("    " + iterator.next().toString() + '\n');
		}
		while(!tempStack.isEmpty()) {
			temp += tempStack.pop();
		}
		return temp;
	}

	/**
	 * A task for monitoring the connection status of the Motes
	 * @author Andrew Halloran, Jonathan Chanin
	 * @version 1.0
	 */
	class KeepAliveCheck extends TimerTask {
		Mote mote;
		KeepAliveCheck(Mote mote) {
			this.mote = mote;
		}
		public void run() {
			mote.isConnected = false;
```

```
			}
		}
}
```

## 12.3 Event.java

```
package ui;

import java.sql.Timestamp;;

/**
 * A class for holding information about Mote events
 * @author Andrew Halloran, Jonathan Chanin
 * @version 1.0
 */
public class Event implements Comparable<Event> {

	int eventType;
	Timestamp eventTimestamp;

	/**
	 * Default constructor method
	 * @param type The type of the Event
	 * @param date The date and time of the Event
	 */
	Event(int type, Timestamp date) {
		this.eventType = type;
		this.eventTimestamp = date;
	}

	/**
	 * Retrieves the type of the Event
	 * @return the type of the Event
	 */
	public int getEventType() {
		return this.eventType;
	}

	/**
	 * Retrieves the date and time of the Event
	 * @return date and time of the Event
	 */
	public Timestamp getEventDate() {
		return this.eventTimestamp;
	}

	/**
	 * Chronologically compares another Event to this Event.
	 * @param event the Event to be compared to this Event
	 * @return If the parameter event occurred before this event, returns
1, if it occurs later
	 * than this Event, returns -1.  If they occur at the same time,
returns 0.
	 */
	public int compareTo(Event event) {
		if(event.eventTimestamp.before(this.eventTimestamp)) {
```

```
                return 1;
            }
            else if(event.eventTimestamp.after(this.eventTimestamp)) {
                return -1;
            }
            return 0;
      }

      /**
       * Retrieves a string with a description of this Event, incuding a time
and description of the
       * event type.
       */
      public String toString() {
            String temp = "[" + this.eventTimestamp + "] Alert type: ";
            switch(this.eventType) {
                case 0: temp += "Keep Alive";
                        break;
                case 1: temp += "Motion Sensor Active";
                        break;
                case 2: temp += "Accelerometer Active";
                        break;
                default: temp += "Unknown event type detected";
            }
            return temp;
      }
}
```