Wireless Sensor Networks for Monitoring HVAC Systems (May 2006)

Ryan Dail, Michael Lewallen, and Armin Moazzami, *Members of IEEE*

***Abstract*- Significant advances in computer technology have made it possible to develop a wireless sensor network (WSN) capable of monitoring and operating a scheduling system for heating, ventilation, and air-conditioning (HVAC). The proposed system includes features such as monitoring, measurement, networking, and control. This system provides an energy efficient solution to the monitoring and controlling of a HVAC system. This design enables users to make cost-effective energy decisions with no interaction with the HVAC system.**

***Index Terms*—**HVAC, motes, sensors, wireless

I. INTRODUCTION

WIRELESS sensor networks (WSN) have rampantly emerged as an important new area in the research community. WSN have numerous applications ranging from indoor deployment scenarios in the home and office to outdoor deployment scenarios in natural and military settings. In some of these scenarios, lives may depend on the timelines and correctness of the sensor data obtained from dispersed sensor nodes. However, our design will focus on an indoor scenario associated with HVAC monitoring, measuring, and control.

The growing energy demands and limited resources have given worldwide attention to energy efficient HVAC technology. As reported in [1], heating and air conditioning account for about 44% of home energy use in the United States. Therefore existing efficient control methods, such as programmable thermostats and zone heating and cooling, should not solely be used to reduce energy expense. Systems such as Direct Digital Control systems (DDC) for building HVAC systems typically perform data acquisition, monitor alarms, automate controls, and produce reports. However, DDC systems require frequent human interaction with software and programming which can cause various problems leading to the system not functioning properly. Our control system will have improved effectiveness, improved operational efficiency, and increased energy efficiency, all with no human interaction.

Manuscript received May 5, 2006. This work was supported in part by the University of North Carolina at Charlotte.

Ryan Dail is a student pursuing a bachelor's degree in Computer Engineering at UNCC (e-mail: rbdail@ uncc.edu).

Michael Lewallen is a student pursing a bachelor's degree in Computer Engineering at UNCC (e-mail: mblewalle@uncc.edu).

Armin Moazzami is a student pursuing a bachelor's degree in Computer Engineering at UNCC (e-mail: amoazzam@uncc.edu).

II. OBJECTIVIES

A list of objectives was created before any work was preformed on the project. These objectives were focused on designing a system that would have an active and inactive state that would decrease the amount of energy the motes use when the HVAC system does not required to be used. The first crucial step in the process was to establish a method for the motes to communicate with each other in order to send data through the network. These motes will need to be programmed to monitor the motion and temperature of the environment. Conditions were established to determine how the sensors should react to the data it collects and programmed into each mote. The last objective was to design a program to act as a control station to send and receive the monitored data from the sensors. The WSN was designed to control the HVAC system in one room or area of a building.

III. BENEFITS OF WSN

A. Improved Effectiveness

WSN provide more effective control on HVAC systems by providing accurately sensed data. Our sensors are more accurate than previous models for measuring the basic HVAC parameters such as temperature, humidity, and pressure. Since the control logic can be readily changed due to the easily reprogrammable motes, the system is extremely flexible in changing the schedules, set conditions and the overall control logic.

B. Improved Operational Efficiency

Using a WSN increases the operational efficiency of the HVAC system by reducing human interaction. This reduces the chance for human error to occur. The motes are more reliable than traditional HVAC controls because they have less mechanical components.

C. INCREASED ENERGY EFFICIENCY

WSN consist of components that are low powered and extremely energy efficient. The motes can operate efficiently for one year without battery replacement and require little adjustment. Less power is consumed while the motes are in an inactive state which increases energy efficiency. The designed system also increases energy efficiency of an HVAC system by operating the system only when an area is occupied by human presence.

IV. HARDWARE

A. MICA2

Wireless sensor networks are a promising technology that will allow people and machines to interact with their environment in a revolutionary way. However, these networks are facing limitations such as energy constraints of the sensor and difficulties in reprogramming the network. The answer to these limitations is the MICA2. The MICA2 is an extremely efficient low power device that can be easily reprogrammed. The MICA2 is widely used in wireless sensor networks, security, surveillance, and environmental monitoring [2].

Figure 1: Crossbow MICA2 programmable base.

Figure 1 shows the MICA2, a wireless sensor mote that the group used to design the WSN. The mote uses TinyOS (TOS) which provides debugging features and support for wireless remote reprogramming. The MICA2 has the capability to interact with a wide range of sensor boards and data acquisition add-on boards.

B. Sensor Boards

The group used the MTS310 sensor board because of the device's innate variety of sensing modalities. These modalities include an accelerometer, magnetometer, light, temperature, and sound. The group focused on the device's light and temperature sensing capabilities. Figure 2 labels the various features on the sensor board [3].

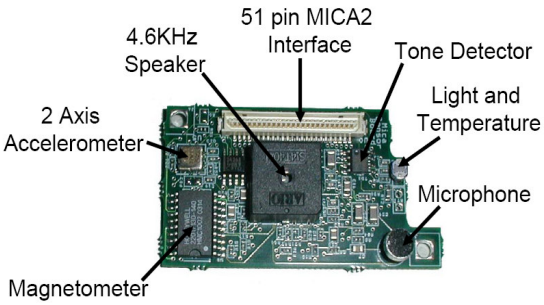

Figure 2: A MTS310 sensor board that attaches to a MICA2 base.

C. Base Stations

A base station allows the aggregation of sensor network data onto a PC or any other computer platform. This device acts as a bridge between motes and computer platform. It provides a programming interface for the MICA2 and allows for data to be collected or sent from the WSN to the computer. The mote interface board uses an RS-232 serial interface for both programming and data communications. Figure 3 below shows an MIB510 [4].

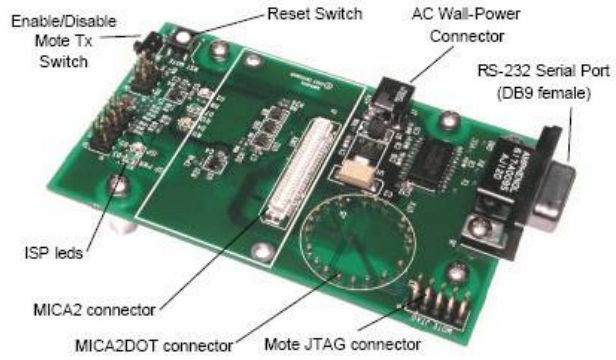

Figure 3: Crossbow MIB510 base station.

IV. TINYOS (TOS)

TinyOS is an open-source operating system designed for wireless embedded sensor networks that have very limited resources. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. TinyOS is small and energy efficient, capable of supporting large scale, self-configuring wireless sensor networks [5].

TinyOS has been ported to various platforms and numerous sensor boards. A wide community uses it in simulation to develop and test various algorithms and protocols. Over 500 research groups and companies are using TinyOS on the Berkeley/Crossbow Motes [5]. Numerous groups are working together, actively contributing to the code making TinyOS increasingly popular and easy to use.

V. NESC

The programming language nesC is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS. The basic concepts of nesC are:

- Separation of construction and composition: programs are built out of components, which are assembled to form whole programs. Components have internal concurrency in the form of tasks. Thread of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt [6].

- nesC gives specification of component behavior with a set of interfaces. Interfaces may be provided or used by components. The provided interfaces are intended to represent the functionality that the component provides to

its user. The used interfaces represent the functionality the component needs to perform its job [6].

- Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider and a set to be implemented by the interface's user. This allows a single interface to represent a complex interaction between components. This is critical because all lengthy commands in TinyOS are non-blocking; their completion is signaled through an event. By specifying interfaces, a component cannot call the *send* command unless it provides an implementation of the *sendDone* event [6].

- Components are statically linked to each other via their interfaces. This increases runtime efficiency and encourages robust design [6].

- nesC is designed under the expectation that code will be generated by whole-program compilers. This allows better code generation and analysis [6].

VI. JAVA APPLICATIONS

For the computer to receive and send data to and from the WSN, a variety of Java based applications are used. The main application that makes this possible is the *Serial Forwarder* program. This program is capable of receiving packets from the WSN and writing packets to the WSN [7]. The received data can be sent to other Java based applications such as *Oscilloscope*. The *Oscilloscope* program graphs data such has a change in the temperature, light, or sound that is recorded in each sent packet. From this, the user can understand at what levels the motes are measuring the acquired data from each programmed device in the mote.

One of the unique features of this WSN is the ability to reprogram or change the behavior of the motes without directly connecting the mote to the base station. The Java application *BcastInject.java* gives the user the ability to change some of the motes actions almost automatically. By default, the program is design to inform the mote to turn off or on an LED or adjust the radio level [8]. This program was later modified so the user could specify the ideal temperature of an area.

Other Java applications exist that adds more functionality to the WSN. However, these three were outline since they are used in the control of the HVAC system. Details about how these applications are used are described in the System Setup section.

VII. SYSTEM OPERATION

Our wireless HVAC control system requires three motes: one that uses a light sensor to detect motion, a temperature sensor, and a mote to be use has a base station to send and receive packets from the WSN. Bundled with the TinyOS package are several applications that could be used by the motes or on the computer. To establish the HVAC control system, the code from the *OscilloscopeM.nc, Oscilloscope.nc,* and *BcastInject.java* files were modified slightly. This section points out some of the changes that were made to the original code. Details about how the original code works can be found at [9].

Before any code was written, a flow chart for both the light sensor and the temperature sensor was constructed. Also, a directory in the computer was created for each sensor. This section gives an in-depth explanation of the light sensor and the temperature sensor followed by a brief comment about the changes made to the *Oscilloscope.nc* and *BcastInject.java* files. This paper does not provide details about functionally or the code that is used for the base mote, *TOSBase*. However, such details can be found [10].

A. Light Sensor

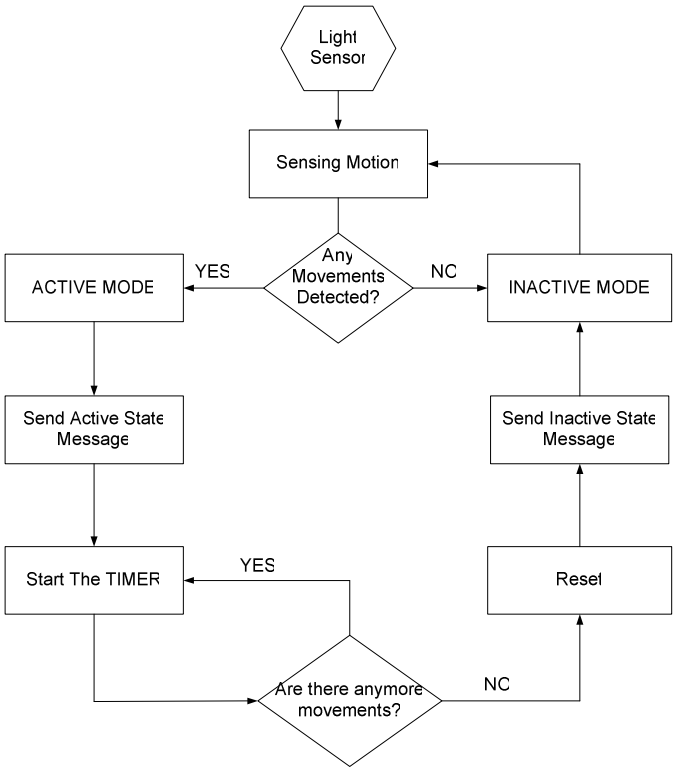

Figure 4: Flow chart for the light sensor.

The first step was to modify the *OscilloscopeM.nc* to give the light sensor the required logic to perform various tasks. To make the WSN conserve power when no human presence is detected, the rate at which packets were sent to the receiver was decreased. Once the WSN detected human presence, the system goes in an active mode and sends packets at a higher rate. In order to do this, a variable was established to allow the timing of the system to be easily changed as shown in figure 5.

```
command result_t StdControl.start() {
    call SensorControl.start();
    call Timer.start(TIMER_REPEAT, TimeIntv);
//Variable used to eaSlly adjust the time
```

```
    call CommControl.start();
    return SUCCESS;
```

Figure 5: Added variable (in bold) to adjust the system timing.

Once motion was detected, the light sensor needs to send a message to the temperature sensor to go into an active state. Each packet sent out on the network is received by the temperature sensor. This sensor looks for a field in the packet that tells the sensor what state it should be in. Figure 6 shows code that was added to add the extra field to the packets sent by the light sensor. The packet structure in the *OscopeMsg.h* file was also changed to include the new field.

```
task void dataTask() {
    struct OscopeMsg2 *pack;
    atomic {
      pack=(structOscopeMsg2 *)msg[currentMsg].data;
      packetReadingNumber = 0;
      pack->lastSampleNumber = readingNumber;
    pack->STATE = StateNumber;  //Added field to be
sent in packet.
    }
```

Figure 6: Bolded line shows the new field added to the packet structure.

Now the senor must be programmed to respond to the conditions sensed by the light sensor. Every possible scenario had to be taken into consideration. Once the light sensor is turned on, it measures the level of the light that is currently in the room and stores a value in the mote's memory. If no motion is sensed after 10 packets have been sent, the light sensor records a new value for the level of light in the room and stores that value. This is does so that as the amount of sunlight changes during the day will not trick the sensor into thinking human presence was detected. The code below shows a counter that will increment only during the inactive state.

```
if (call DataMsg.send(TOS_BCAST_ADDR, sizeof(struct
OscopeMsg2), &msg[currentMsg]))
//Execute this if the packet has been sent.
      {
  atomic {
    currentMsg ^= 0x1;
        }
  call Leds.yellowToggle();       //Flash YELLOW LED
  if (StateNumber==0)
 //Everytime a packet is sent in INACTIVE state,
  {atomic{YellowFlashCount++;}}  //increment by 1.
      }
```

Figure 7: Counter used to refresh base light level.

Originally, we assume that no motion is present. The lines in bold in figure 8 shows the sensor records the value of the light level in memory after 0.5 seconds. The sensor continues to sense the motion in the room and sends a packet to the base station every 4 seconds. If this value is about the same as the recorded value, the mote remains in the inactive state. This process is repeated until the reset condition has been reached. The reset condition will be described later.

```
//ENTER TEST CONDITIONS TO STORE BASE LIGHT HERE
state = inactive
  if (StateNumber==0 &&  TakeBase==1)//Grab 1st
packet in inactive state since it is incremented
    {atomic{
      BaseLight = data;//Data is the light ADC value
      TakeBase = 0;
        }
    }
```

Figure 8: Record the base light level of the room.

If a significant difference between the base light and the current measurement is observed, the system assumes human motion was detected. The sensor goes into the active state and lights up the red LED to signal the sensor is in the active state. Now the system will send data to the base station every second and send a message to the temperature mote signaling it to go into the active state. Figure 9 shows the code used to perform these tasks.

```
//ENTER CONDITIONS TO GO INTO ACTIVE MODE WHEN
MOTION IS DETECTED (StateNumber=1), state = inactive
//NOTE:  A condition of tolerance has been
established to ignore very slight changes in light.
//Also the light sensor may vary +/- 1 ADC reading
each pass.
  if (StateNumber==0 && BaseLight > 0 && (data >
(BaseLight+0x002) || data < (BaseLight-0x002)))
//Motion Detected
      {atomic{
        TimeIntv=103;  //Change timer 1 second
        StateNumber=1;
        post dataTask();
 //Send packet with state change
        call Leds.redOn();
        call Timer.start(TIMER_REPEAT,
TimeIntv);//Reset timer to send packets every second
        }
    }
```

Figure 9: Actions taken when the light sensor enters the active state.

Once motion is detected, the system remains in an active mode until motion is no longer sensed. If the system receives a measurement that is very close to the previous measurement, a counter starts and the green LED turns on. The counter is incremented by 1 every time a packet is sent with the same measurement. After 4 packets in a row are sent with a close measurement to the previous one, the system begins the process to go back into the inactive state. However, if a rapid change in the level of light is suddenly detected, it is interpreted has movement and the counter is reset to zero and the green LED turns off. The two "if" statements that perform these actions are shown below.

```
//ENTER CONDITIONS TO START RESET DELAY COUNTER FOR
GO INACTIVE HERE WHEN NO MOVEMENT IS PRESENT,
state = active
  if (StateNumber==1 && (lastMeasure-0x001<= data)&&
(lastMeasure+0x001 >= data))
    {atomic{
      delayCnt++;//No movement?
      call Leds.greenOn();
        }
    }

//ENTER CONDITIONS TO RESET INACTIVE DELAY COUNTER
SINCE MOTION WAS DETECTED AGAIN, state = active
  if (StateNumber==1 && delayCnt > 0 &&
((lastMeasure-0x001 > data)|| (lastMeasure+0x001 <
data)))
```

```
    {atomic{
      delayCnt =0; //reset counter
      call Leds.greenOff(); //show the delayCnt is
now 0 but system is still active
        }
    }
```

Figure 10: Delay counter functional in active state.

The last phase of the program is the reset condition. If the sensor is in the inactive mode, the reset condition forces the light sensor to record a new base level of light to compare against future values. This is done after the yellow LED flashes 10 times. If the system is in the active mode, a message is sent to the temperature sensor to go into the inactive state. The packet timer is increased to 4 seconds and conditions are established so the light sensor will record the current level of light in the room as the base level on the next pass. Finally, the red and green LEDs are turned off. The code for the reset condition is given below in figure 11.

```
//ESTABILISH RESET CONDITIONS HERE!
//inactive, YellowFllash count=10, grab new light
level on next pass
//active,  1.  Send packet to temp sensor to set it
inactive
//     2.  Turn OFF RED and GREEN LEDs
//     3.  Reset timer to slow system down
//     4.  Make light sensor grab a new light level
on the next pass

  if ((delayCnt==80 && StateNumber==1)||
YellowFlashCount==10)
    {atomic{
      StateNumber=0;     //Return everything to
inactive
        post dataTask();   //Send message
      call Leds.greenOff();
      call Leds.redOff();
        delayCnt=0;        //Reset delay counter for
active state
      BaseLight = 0;     //Reset base light data
      lastMeasure=0;     //Reset last measure for
active state
      TimeIntv=413;      //Reset timer to 4 seconds
      YellowFlashCount=0;    //Used for inactive
state to grab new base light as needed
      TakeBase=1;        //Force a new base light
level to be grabbed next pass
      call Timer.start(TIMER_REPEAT,TimeIntv);
//Using the new TimeIntv, reset packet timer
        }
    }
```

Figure 11: Reset conditions for both states and the conversion from active state to inactive state.

The light sensor will continue to go through the process described above until the batteries in the mote fail or the power switch on the mote is turned off.

B. Temperature Sensor

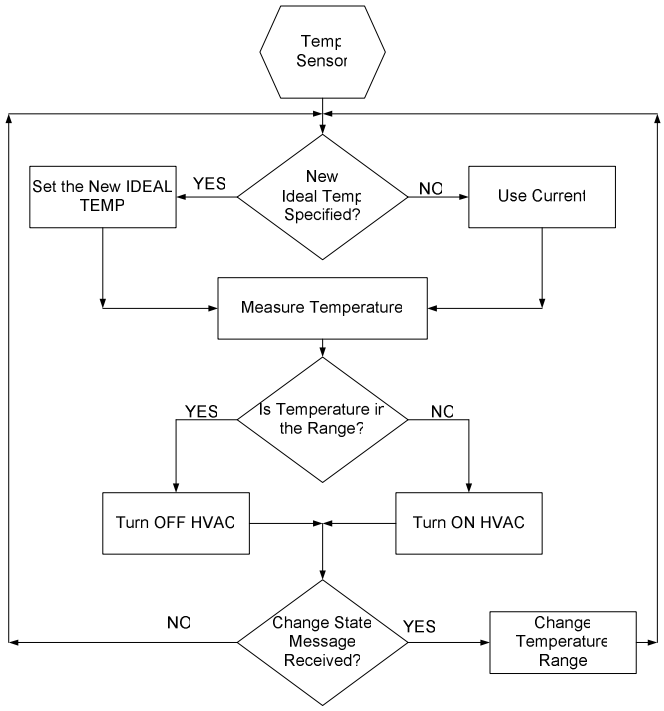

Figure 12: Flow chart for the temperature sensor.

When the temperature sensor is turned on, it assumes it is in the inactive state and has a default ideal temperature setting of 75°F. In the inactive state, the mote measures the temperature of the room and compares it to that of the ideal temperature every 8 seconds. If the temperature goes above 85°F, it will turn on the HVAC system to cool the room until it is 83°F. If the temperature drops below 65°F the HVAC system will turn on the heat until the room reaches 67°F. This rage of temperatures was used to only turn on the HVAC system when the temperature reaches extreme conditions. It is important to ensure the area is kept from freezing and kept from getting too warm at all times. The 2°F change reduces the wear and tare on the HVAC system by not requiring it to constantly turn on and off for slight changes in the temperature.

Once the temperature mote receives a packet from the light mote, it views the state field of the received packet. If the value of the state has is different from the mote's current state, the temperature sensor will change states. If a 1 is sent in the field, the mote goes into the active state, if a 0 is sent, the mote goes into the inactive state. The code used for this process is shown in the figure below.

```
event TOS_MsgPtr StateChangeMsg.receive(TOS_MsgPtr
m) {
    struct OscopeMsg2 *data= (struct OscopeMsg2 *)m-
>data;
  atomic { TempState=data->STATE;}//received from
packet
    if (TempState==1 && PrevState==0)
      {atomic{ TimeIntv=105;
          call Timer.start(TIMER_REPEAT, TimeIntv);
        }
      }
    if (TempState==0 && PrevState==1)
    {atomic{ TimeIntv=840;
```

```
         call Timer.start(TIMER_REPEAT, TimeIntv);
        }
  }
    PrevState=TempState;  //MUST  be  used  after  if
statements so if statements will work on change of
state signal
    return m; }
```

Figure 13: Code used to receive a change state message.

In the active state, the range of acceptable temperatures decreases to +/- 2°F that of the idea temperature. The mote will also measure the room temperature every second. If the temperature is 2°F or higher than the ideal temperature, the green LED will turn on signaling the HVAC should cool the room. Once the temperature reaches about 1°F higher than the ideal temperature, the HVAC will shut off and the green LED will turn off. If the temperature is 2°F or lower than the ideal temperature, the red LED will turn on signaling the HVAC should cool the room. Once the temperature reaches about 1°F lower than the ideal temperature, the HVAC will shut off and the red LED will turn off. The logic that was used to code the difference between the two states is shown in figure 14.

```
  if ((TempState==0 && data <= 449)|| (TempState==1
&& data <= IdealTemp-14))
//Too COLD, need heat, Ideal - 40
    {call Leds.greenOff();
     call Leds.redOn();}

    if((TempState==0 && data >= 591)|| (TempState==1
&& data >= IdealTemp+14))
//Too HIGH, need AC, Ideal + 40
    {call Leds.redOff();
     call Leds.greenOn();}

  if ((TempState==0 && data > 464 && data < 576) ||
      (TempState==1 && data > IdealTemp-7 && data <
IdealTemp+7))//Ideal temperature Keep air off!
     { call Leds.greenOff();
       call Leds.redOff(); }
```

Figure 14: Code used to decide the status of the HVAC system.

The user can also change the ideal temperature of the room at any time during either state. This process is described in the system setup section. Once the sensor receives a message from the computer, it takes the ideal temperature in the packet, converts it to its equivalent ADC value. This will be the new temperature the senor uses to compare the room temperature with. Through multiple trails, it was determined that an ADC value of 450 was about 65°F and an ADC value of 590 was about 85°F. In this range, a change in temperature by 1°F is approximately equal to a change of +/- 7 of the ADC value. For example, 67°F has an ADC value of 464 while 68°F has an ADC value of 471. It is important to note that the ADC does not measure temperatures in a linear fashion. As the temperature drops, the ADC is more sensitive, thus the ADC value between temperatures is higher. The next figure shows the code used to receive the number and convert it to its matching ADC value.

```
 event TOS_MsgPtr SimpleCmdMsg.receive(TOS_MsgPtr m)
{struct SimpleCmdMsg *cmdi = (struct SimpleCmdMsg
*)m->data;
  atomic{ NewTemp= cmdi->action;
      IdealTemp= 450 + ((NewTemp-65)*7); }
//Byte to ADC temperature (Fahrenheit)
    return m; }
```

Figure 15: Method used to receive the ideal temperature from the computer.

The light sensor will continue to go through the process described above until the batteries in the mote fail or the power switch on the mote is turned off.

C. Oscilloscope.nc

The *Oscilloscope.nc* file was changed for the temperature sensor has shown in figure 16. The file was modified to include both types of packets, the temperature sensor interface, and methods to handle the incoming packets.

```
OscilloscopeM.StateChangeMsg-> Comm.ReceiveMsg
[AM_OSCOPEMSG2];  //Receive State Change Message
OscilloscopeM.SimpleCmdMsg -> Comm.ReceiveMsg
[AM_SIMPLECMDMSG]; //Receive new ideal temperature
```

Figure 16: The *Oscilloscope.nc* file for the temperature sensor.

D. BcastInject.java

BcastInject.java is part of the simple command suite for the motes. This file was modified so a two digit number could be entered for a temperature. The figure below shows the modifications made to this file. This file was later renamed to *BcastInjectNew.java* so the original file would not be saved with the modifications.

```
byte cmdi; //Temperature from command line
  cmdi = Byte.parseByte(cmd);

  if ( cmdi > 64 && cmdi < 86)
  {
    packet.set_action(cmdi);
  }
  else
    {
    System.out.println("The number is outside the of
range, Please Retry (65-85). ");
    System.exit(-1);

    }
```

Figure 17: Changes made to the *BcastInject.java* file. This file was renamed to *BcastInjectNew.java*.

VIII. SYSTEM SETUP

The first step to setting up the wireless control system is do load the motes with the appropriate program. There are three programs that must be loaded on to the motes. To load these programs, the user must install 2 AA batteries in to the back of the MICA2 base and connect white strip of the MICA2 to the white strip of the MIB510 board. Then, the MIB510 must be connected to the computer through a serial port and the mote must be turned on. Figure 18 shows a MICA2 connected to a MIB510 base station.

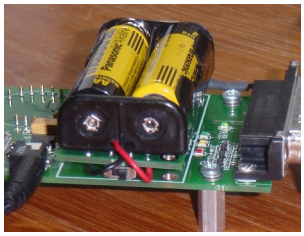

Figure 18: A MICA2 mote connected to a MIB510 base station receiving code from the computer.

The motes are designed to be programmed with the UNIX operating system. To simulate this environment, a software application called Cygwin is used and is bundled in the TinyOS package. In the proceeding paragraphs, a list of commands and steps are given to load the program on to each mote. After the desired code has been loaded, the mote must be turned off and disconnected from the MIB510 base station. The Cygwin window should be closed and a new Cygwin window should be opened to program the next mote. This procedure is required each time a program is loaded to the motes. The MTS310 sensor board should be attached to the motes designated to be the temperature sensor and the light sensor after the mote has been programmed and turned off.

One mote needs to be programmed to detect the motion. This program is stored in the *OscilloscopeRFLight* directory. The following commands must be entered with Cygwin to get into the correct directory. The commands are case sensitive!

cd c:
cd tinyos/cygwin/opt/tinyos-1.x/apps/OscilloscopeRFLight

Once the user is in the right directory, the following command is used to load the program in to the memory of the MICA2*:*

MIB510=COM1 make mica2 install.2

Note the *install.2* part of the command. This ensures that the mote uses a different channel to transmit packets on the network to avoid collisions. If no channel is specified, the mote is set to communicate on the network in channel 1.

The second mote needs to be programmed to monitor the temperature. This program is located in the *OscilloscopeRFTemp* directory. The following commands are used to get into the correct directory:

cd c:
cd tinyos/cygwin/opt/tinyos-1.x/apps/OscilloscopeRFTemp2

Once in the user is in the correct directory, the following command is used to load the program into the mote's memory.

MIB510=COM1 make mica2 install

The third mote needs to be programmed to function as base station. This program is called *TOSBase* and is stored in the *TOSBase* directory. To load this program, the following commands are used to get into the right directory:

cd c:
cd tinyos/cygwin/opt/tinyos-1.x/apps/TOSBase

Once in the user is in the correct directory, the following command is used to load the program in to the mote's memory:

MIB510=COM1 make mica2 install

After this mote has been programmed, it is not to be removed from the base since it is used to send data to the WSN and allow the computer to receive data from the WSN. Once the programming part is done, a virtual wireless channel must be established between the motes. This can be achieved by running the *Serial Forwarder* program as shown in figure 19.

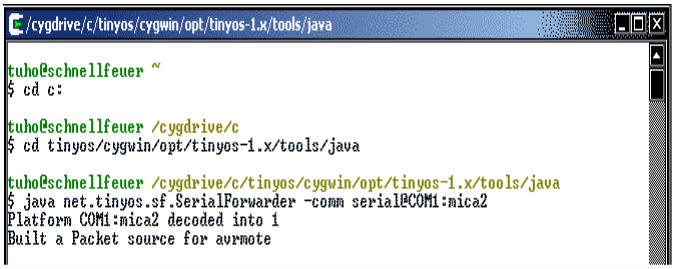

Figure 19: Serial Forwarder command window.

Once the program is running, the Serial Forwarder window appears as shown in the figure 20. This program shows the number of packets sent and received. It will also give the status of the WSN and inform the user of packets written that were not received by the WSN.

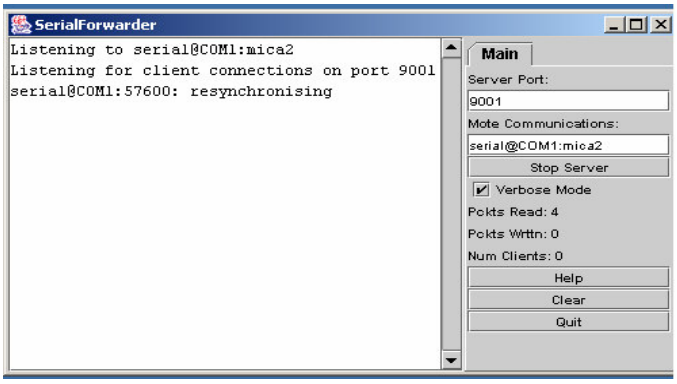

Figure 20: Serial Forwarder window.

The next step is to find a way to monitor the data collected by the motes. This can be done by using the *Oscilloscope* program, and running the following commands as shown in figure21.

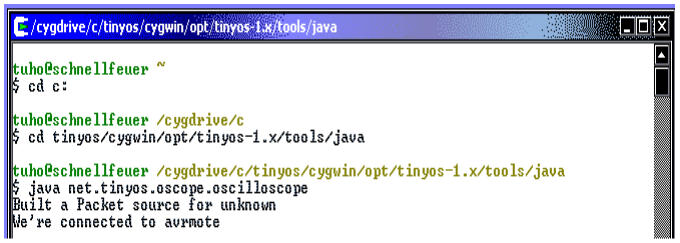

Figure 21: *Oscilloscope* command window.

After executing the program, the window shown in figure 22 will be open, and the data collected by wireless sensors is shown. The green line shows the level of light measured by the light sensor and the red line shows the temperature measured by the temperature sensor. The program allows the user to scroll through the collected data and zoom in and out on each axis.

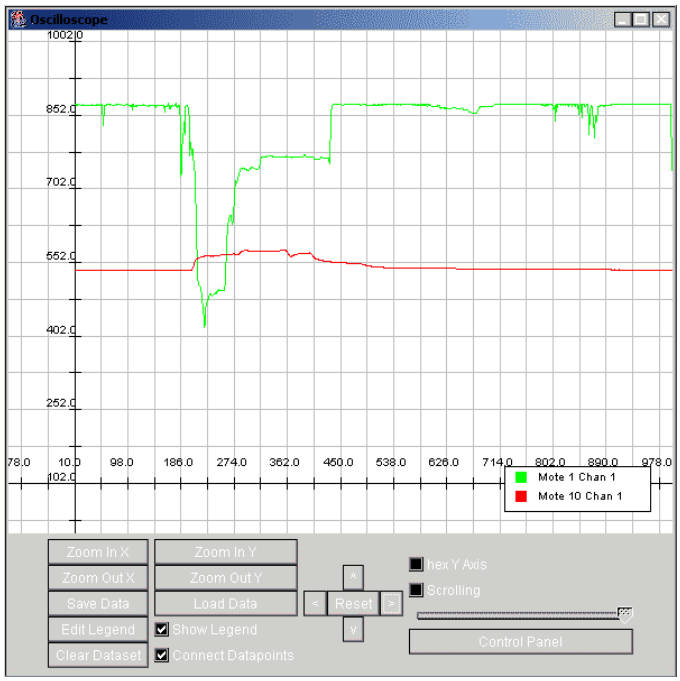

Figure 22: *Oscilloscope* Window

BcastInjectNew program can be used to send a new ideal temperature to the temperature sensor. For an example, figure 23 demonstrates how the temperature mote can be reprogrammed to the new ideal temperature of 80°F using the java command.

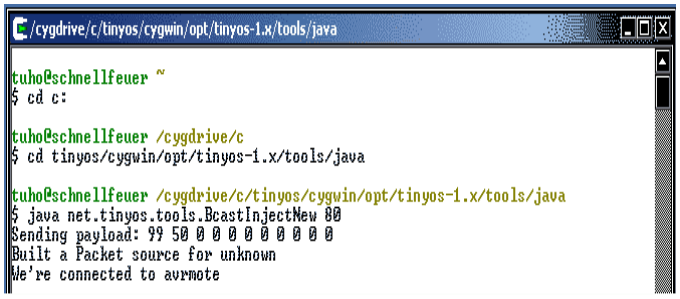

Figure 23: *BcastInjectnew.java* command for 80°F.

IX. SYSTEM LIMITATIONS

The control system was design to help control a HVAC system, but it does not have the capabilities to actually activate the HVAC system. This project is only illustrates part of the control elements needed to establish such a system. Additional components will need to be designed and integrated into the HVAC system to receive a signal from the temperature sensors to turn the system on or off.

If this system is used in an area that has very little light or is very bright, the system may not be able to detect motion causing it to fail. In these areas, it might be possible do develop methods to detect human presence by setting up the mote to monitor the sound in the room or movement of the mote.

This system does not feature a very robust method for handling transmission errors that may be caused by interference or power failure to one of the motes. Also, if building loses power, the motes have no way of knowing a power failure has occurred. However, when the power is restored, the motes should function without needing to setup the WSN again since the network uses battery power. In theory, when the power is restored, the HVAC will activate immediately as needed. The system may not function properly if the building is damaged.

The control system was only designed to accept a range of temperatures between 65°F and 85°F. Should the user attempt to enter a number out of this range, the packet will not be sent and an error message will appear on the computer screen informing them the value they have entered was out of range.

This paper describes an HVAC system that is designed to control only one room or area inside of a building. Simple modifications can be made to the code to design zones or controls for many rooms in a building and these areas can be controlled by one computer. However, the motes have a broadcast range of up to 50 feet away from a base or another mote. The system will have to be very robust and sources of interference such as too many packets on one channel should be limited as much as possible. Another feature that could be added to the system is a log of the HVAC system's activity. This log system should have the ability to update itself every time the system is turned on and off. This can give the user a great deal of helpful information and help determine the energy efficiency of the system.

X. CONCLUSION

Our research shows that it is possible to create an efficient wireless HVAC control system that uses less energy by establishing a WSN. This system combined with an integrated and programmable HVAC structure will likely be used in the construction of future buildings and in the remodeling of existing buildings. The decreasing cost of monitoring devices makes it become a more attractive offer for the average user in addition to its benefits. This system can be easily used in large

buildings as well as average homes. The exact amount of energy that this system will save is difficult to calculate. However, it can be speculated that wide use of this system could cause a significance reduction in worldwide energy consumption. The flexibility of the system makes it easy to customize and adjust as needed. Moreover, this idea could be used to reduce the energy consumed for other applications that use a large amount of energy.

ACKNOWLEDGMENT

The group thanks Dr. Nasipuri for giving us access and ample time with the motes. We would also like to thank him for his guidance through our difficulties and supporting our research.

REFERENCES

[1] Azzi, D., Virk, G.S. "Wireless Temperature Sensing for Building Management Systems," *Wireless Technology (Digest No. 1996/199), IEE Colloquium*, London, 1996, p. 2/1-2/4.

[2] Anonymous. "MICA2 Datasheet." Crossbow Technologies Inc., San Jose, p. 1-2. <http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf>.

[3] Anonymous. "MTS MDA Datasheet." Crossbow Technologies Inc., San Jose, p. 1. <http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MTS_MDA_Datasheet.pdf>.

[4] Roeder, Rughoonundon, Deming, and Kenny Chung. "Hardware Wireless Project Report." *Rochester Institute of Technology*. <http://www.ce.rit.edu/~fxheec/cisco_urp/docs/wireless%20networks%20-%20hardware%20group%20-%20project%20report.htm>.

[5] Anonymous. "TinyOS" *TinyOS*, 27 Apr. 2006. <http://www.tinyos.net>.

[6] Brewer, Culler, et al. "nesC: A Programming Language for Deeply Networked Systems." *UC Berkeley WEBS Project*, Berkeley, 14 Dec. 2004. <http://nesc.sourcefourge.net>.

[7] Hull, Brent. "SerialForwarder v1.1." *SourceForge.net,* 10 Oct. 2001. <http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-0.6.x/doc/serialforwarder.pdf?rev=1.2>.

[8] Anonymous. "Lesson 7: Injecting and Broadcasting Packets." *TinyOS*, 23 August 2003. <http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson7.html>.

[9] Anonymous. "Lesson 6: Displaying Data on a PC." *TinyOS*, 23 August 2003. <http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson6.html>.

[10] Marrón, Lachenmann, Minder, et al. "Management and Configuration Issues for Sensor Networks." *International Journal of Network Management*, *Wiley InterScience*, 2005, p. 235-253.