Internship Progress Report: Summer 2001

Author: Ioan Raicu

Document Created:
6/25/2001

Last Date Modified:
9/17/2001

Table of Contents

1.0 Introduction

…

2.0 Weekly Reports

In the sections to follow, I will be iterating my progress as the project matures. I will be reporting my weekly report to Elisabeth Alben.

2.1 1st WEEK (6/11/01 – 6/15/01)

This first week, I pretty much just got acquainted with the Accenture Technology Labs, formerly know as the CSTAR group. I was introduced to the main topics of the project, and thus just read most of the rest of the week, trying to understand how the motes, sensors in general, the TinyOS, and other related things work. I set up a folder in which I am keeping track of everything I am reading so I can use it for future reference.

2.2 2nd WEEK (6/18/01 – 6/22/01)

I met Owen Richter, who introduced me to the hardware (the motes) and the software (TinyOS). I set up my machine with Cygwin (a UNIX shell needed to program the motes), the TinyOS 4.3 (the OS that the motes run and understand, used to program the motes), and Sun's Java JDK 1.3.1 (used to listen to the serial port for data coming from base station mote). I managed to get the wireless network up and running to a limited degree. I have 3 motes and a programming station (which has 1 parallel port as an input, and 1 serial port as an output). One mote needs to be hooked up to the programming station and be set to be a base station. The other two, I was able to set them up with simple functionality, such as number counting and transmitting the value of the counter over the wireless network. The base station works as data can be received from other motes (in HEX). Both Owen and I have spent a great deal of time trying to resolve a compilation problem. At the moment, we are not able to compile and run the necessary programs to be able to read the sensors and relay the information via RF.

2.3 3rd WEEK (6/25/01 – 6/29/01)

We solved the compilation problems that we were having last week. I managed to get the motes working in terms of reading the light sensor and transmitting it over the RF. Much time was spent investigating the code for the TinyOS and the sample applications in order to be able to write my own applications. Some of my near future goals are to write a simple application that can read the signal strength, and perhaps even the temperature sensor. I have done much reading about the motes themselves and the TinyOS and I am still digging up the answers to some of the questions that came up in last weeks meeting, such as the unique ID of each mote, etc…

2.4 4th WEEK (7/2/01 – 7/6/01)

This was relatively a short week, due to the 4th of July days off, including a personal day off on the 6th of July. I kept working on finding out as much as I could about a way to uniquely identify each mote. I have attempted to contact people like Jason Hill and some others from Berkeley, and Northbrook.

2.5 5th WEEK (7/9/01 – 7/13/01)

I have been working on getting the temperature sensor, however, with no success. The only person who has answered any of my cries for help on some of my questions has been Reena from Northbrook. After talking to Reena, and looking through most of the source code for the TinyOS, here is the breakdown of the known fields in the data that comes in. Each field is 2 bytes long.

- 1st – serial address (TOS_UART_ADDR); this is how the mote knows to send the packet to the serial interface rather than as a broadcast over the radio.
- 2nd – type; I really don't know what this is, but according to Northbrook, it is the handler of the mote hopping, but I cannot verify this yet.
- 3rd – group ID (LOCAL_GROUP); only packets belonging to the group that the base station belongs to will be allowed through; there is no authentication, but rather it is just like switching channels.
- 4th – data; light sensor, temperature sensor, etc…
- 5th – mote ID (TOS_LOCAL_ADDRESS); this will distinguish the mote from other motes, however, uniqueness is not guaranteed since the assigning of mote IDs is done manually at compile time.

2.6 6th WEEK (7/16/01 – 7/20/01)

I finally got the temperature sensor working. I also attempted to get the signal strength to work, and while preliminary results looked positive when I first tried out the signal strength code, I was unable to reproduce the varying signal strength. Since that was not a priority, I abandoned any efforts for now until I have more time. I also tried to get more than 1 sensor working at one time. Theoretically, I believe I understand it, but when it came to coding it, I was running into compiling errors. I also gave up on this in the hopes of pursuing the mote hoping algorithms. I also tried to install Visual Studio .NET with no success because of download errors.

2.7 7th WEEK (7/23/01 – 7/27/01)

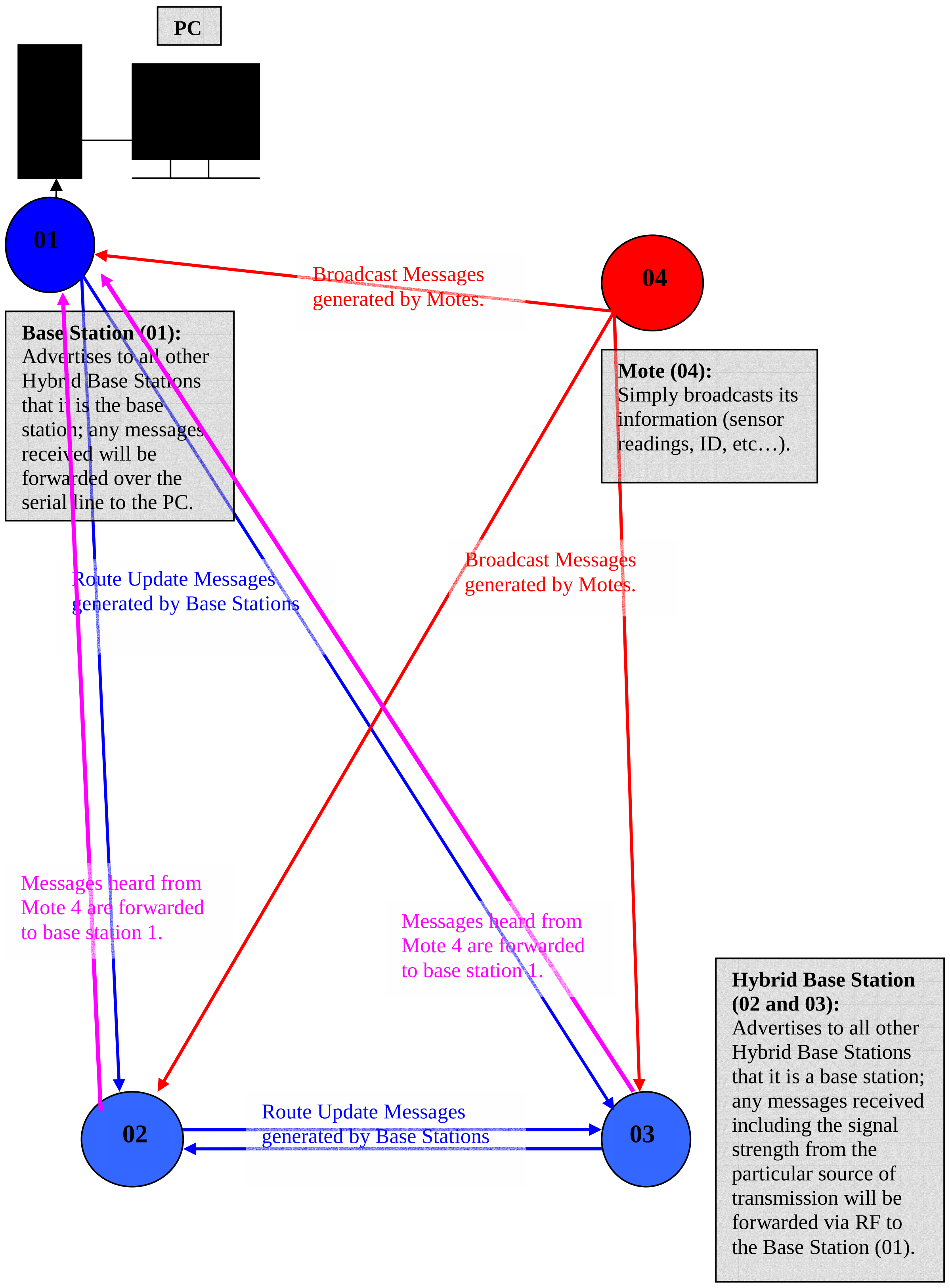
I am attempting to continue the .NET installation in order to compile and run Lisa's C# version of the listen.java. After a day of installs, I managed to finish, and test "listen.cs". It works perfect. I found a new source of great up to date information, at http://sourceforge.net/. With the help of this CVS repository, I was able to update my 5 month old TinyOS, which fixed many of the problems I was having. As time was limited towards the end of the week, I only had time to get the signal strength and CRC working. I incorporated the CRC algorithms into the files that now supported the signal strength, and before I knew it, the CRC check was also working and the base stations are now only outputting error-free packets. Please refer to the technical details in section 3 for a complete break down of the data being transmitted. In the meantime, since I was having such a hard time with batteries, I also attempted to solve the problem by powering the motes by an AC/DC adapter. What I noticed was that the motes are really sensitive to having about 3.2~3.3 volts of power, but only when being programmed. For regular operation, they can work on even 2.7~2.8 volts, which is fine for most batteries. My problem was that most batteries only had about 3.2~3.3 volts when fresh out of the box, and their voltage would quickly drop below the needed voltage and thus I was going through batteries in under an hour. My solution was to use a universal adapter, set on 3 volts, 300 mA, and a 100 OHM resistor. Without the resistor, because of the very small load the motes would have caused the adapter, the voltage would have been above the recommended limits of the hardware. The 100 OHM resistor dropped the voltage to the 3.2 ~3.3 voltage area.

2.8 8th WEEK (7/30/01 – 8/3/01)

My next step was to get both the light sensor and the photo sensor working together, and again, with the help of http://sourceforge.net/, I was able to accomplish it in no time. My last big feat was to get the mote-hopping algorithms to work. By the end of the week, I managed to do it. I now had 4 applications that I wrote, in which they all incorporated sensing both the photo and temperature sensor at the same time, the signal strength, and are doing CRC. They go in pairs, since two of them are meant for the motes as nodes, while the other two are meant as base stations. One pair has simple communication between each mote and the base station, while the other pair supports mote-hopping, in which it relays messages heard from other motes. Please refer to the technical details in section 3 for a complete break down of the data being transmitted. I also spent a little bit of time working on a visual representation of the motes; again, at http://sourceforge.net/, I found a program called “Surge”, which was supposed to graphically represent the network topology in conjunction with the program “connect.desc”. In order to make it compatible with my applications, I had to modify it to suit my needs. Along with the graphical interface, I also found an application that will forward any packets received from the UART and send them over the wired network to any IP address. In essence, base stations could be located throughout a building, and all send their readings back to a server, which collects all the data and comes up with a unified view of all the base stations and all the motes at the same time.

2.9 9th WEEK (8/6/01 – 8/10/01)

I spent a little bit of time updating this report, and cleaning up my source code. I am now working on a proximity detector with Owen, specifically to be able to transmit the signal strength from one mote to another as the mote hopping does its thing. I am also trying to see if motes could have some dynamic reprogrammable parameters, such as the frequency at which it updates the sensor readings. As for the proximity detector, I have made great progress in making another program which is a hybrid between a base station and a sensor mote. I had to do this since when I tried to read the signal strength from a mote to another mote, it was 0, and thus was not supported in the architecture of the system. This hybrid mote was pretty much exactly like a base station, however instead of forwarding its received data to the serial port, it is broadcasting it over the radio. When I tried using motes that had mote hopping routing enabled, the system would occasionally hear echoes of its own messages, and thus I decided to further simplify the problem. In terms of a proximity detector, such as locating a person wearing a badge, that badge should only have capabilities for simple communication between a base station and itself; it should not forward other messages it hears since this will create a whole lot of traffic and even echoes. On the next page, you will find a diagram in which you will find the description of the current setup in terms of the proximity detection setup. The next few steps would include involving some way to average out the signal strengths received and apply it to some math functions to get a rough proximity estimator.


PC
01
04
02
03
Broadcast Messages generated by Motes.
Broadcast Messages generated by Motes.
Route Update Messages generated by Base Stations
Route Update Messages generated by Base Stations
Messages heard from Mote 4 are forwarded to base station 1.
Messages heard from Mote 4 are forwarded to base station 1.

Base Station (01): Advertises to all other Hybrid Base Stations that it is the base station; any messages received will be forwarded over the serial line to the PC.

Mote (04): Simply broadcasts its information (sensor readings, ID, etc…).

Hybrid Base Station (02 and 03): Advertises to all other Hybrid Base Stations that it is a base station; any messages received including the signal strength from the particular source of transmission will be forwarded via RF to the Base Station (01).

Figure 2.1

2.10 10th WEEK (8/13/01 – 8/17/01)

I finally finished with Owen's proximity detector in a state that works rather well. The program was written in Java and pretty much analyzes the incoming data, keeps some history data of everything, and comes up with decisions in which when a mote is closer to some other mote instead of the one it was closest to. Please refer to Figure 1 for details on the setup of the motes for such an example. Everything is being logged to a file which can be viewed at a later time for closer examination. Below can be viewed a sample log file:

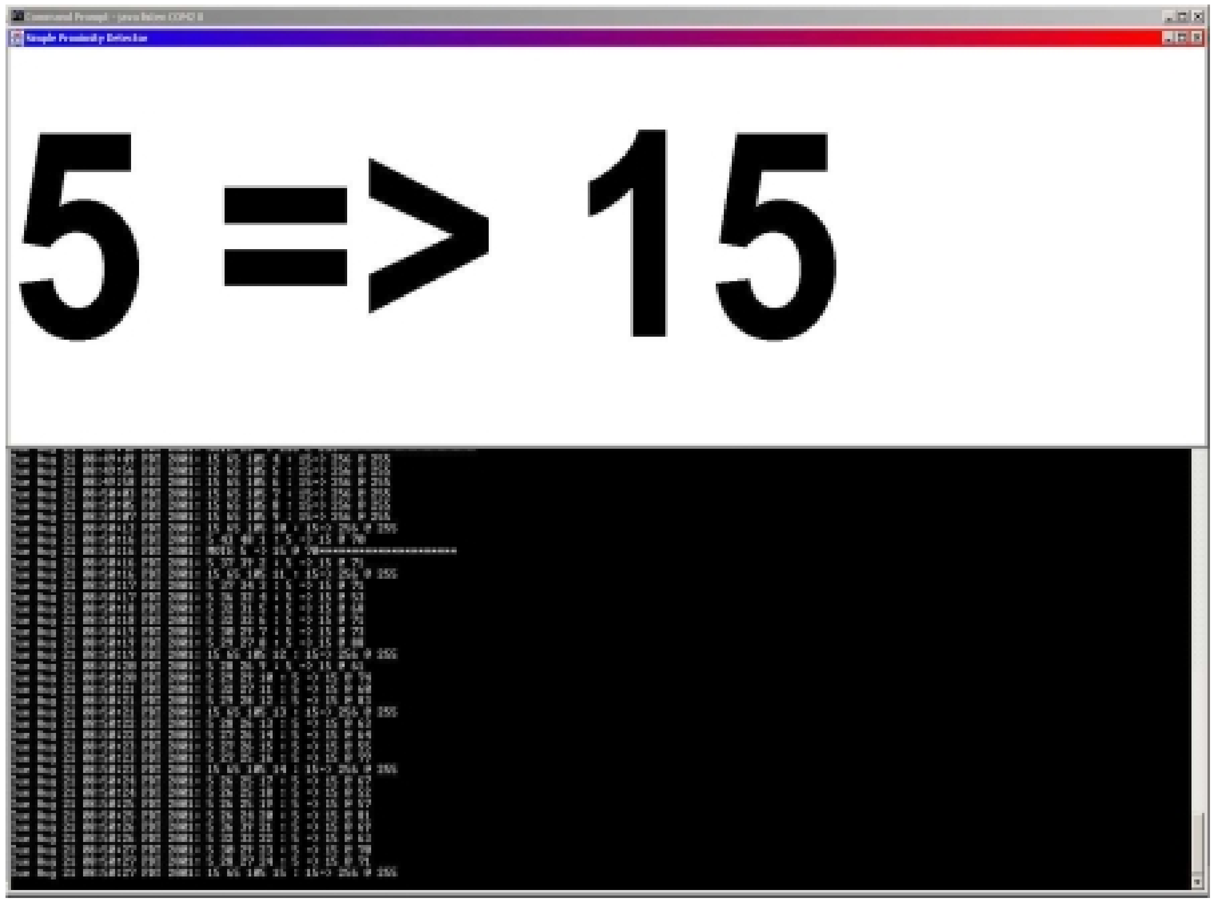
```
Tue Aug 21 08:49:48 PDT 2001: NEW SESSION
Tue Aug 21 08:49:48 PDT 2001: ID TEMP LIGHT COUNTER
Tue Aug 21 08:49:48 PDT 2001: 15 65 105 3 : 15=> 256 @ 255
Tue Aug 21 08:49:48 PDT 2001: MOTE 15 => 256 @ 255*********************
Tue Aug 21 08:49:48 PDT 2001: 126 8 34 0 15 65 105 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Tue Aug 21 08:49:49 PDT 2001: 15 65 105 4 : 15=> 256 @ 255
Tue Aug 21 08:49:49 PDT 2001: 126 8 34 0 15 65 105 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Tue Aug 21 08:49:56 PDT 2001: 15 65 105 5 : 15=> 256 @ 255
Tue Aug 21 08:49:56 PDT 2001: 126 8 34 0 15 65 105 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Tue Aug 21 08:50:07 PDT 2001: 15 65 105 9 : 15=> 256 @ 255
Tue Aug 21 08:50:07 PDT 2001: 126 8 34 0 15 65 105 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Tue Aug 21 08:50:13 PDT 2001: 15 65 105 10 : 15=> 256 @ 255
Tue Aug 21 08:50:13 PDT 2001: 126 8 34 0 15 65 105 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Tue Aug 21 08:50:16 PDT 2001: 5 43 48 1 : 5 => 15 @ 70
Tue Aug 21 08:50:16 PDT 2001: MOTE 5 => 15 @ 70*********************
Tue Aug 21 08:50:16 PDT 2001: 126 8 34 1 5 43 48 1 70 15 65 105 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255
48 70 0
```

Figure 2.2

- Tue Aug 21 08:49:48 PDT 2001: NEW SESSION
    - Time stamp: announcing that a new session started.
- Tue Aug 21 08:49:48 PDT 2001: ID TEMP LIGHT COUNTER
    - Time stamp: specifying that the 1st field is the mote ID, the 2nd is the temperature, the 3rd is the light, and the 4th is the packet counter.
- Tue Aug 21 08:49:48 PDT 2001: 15 65 105 3 : 15=> 256 @ 255
    - Time stamp: ID TEMP LIGHT COUNTER : source mote ID is closest to destination mote ID @ the particular signal strength.
- Tue Aug 21 08:49:48 PDT 2001: MOTE 15 => 256 @ 255*********************
    - Time stamp: source mote ID is closest to destination mote ID @ the particular signal strength; the "*********************" mean that this is a new location, and therefore will only be displayed once until the system sees another change; this is also the information that is being displayed on the graphical GUI.
- Tue Aug 21 08:49:48 PDT 2001: 126 8 34 0 15 65 105 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    - Time stamp: packet data as it came over the serial port; please see figure 3.2 for further details on the arrangement of the data.

2.11 11th WEEK (8/20/01 – 8/24/01)

Since my presentation is on Wed, I need to prepare some slides and my entire talk. I also wanted to present the Proximity Detector, and thus I spent a few hours developing a simple application that graphically represents who the mote is closest to, in full screen mode. Below you can see a snap shot of the interface. The above white part means that mote with ID 5 was last closest to mote with ID 15. This will only change if the system has some reason to believe otherwise. Below that, you can see the background information that is coming in and being examined. Each line starts out with a time stamp, after which there is the packet info (mote ID, temperature reading, photo sensor reading, and packet number). After the colon, there is information regarding where did this information come from, and at what signal strength it came at. For example, for the first line, it states that mote 15 data came from mote 256 with a signal strength of 255. Mote 256 is denoted as the computer itself, and therefore the signal strength of 255 is justified because the signal travels through the serial port rather than the radio. Notice that there is no way to exit the program unless Ctrl-c is used; the process will continue forever if not terminated.



```
5 => 15
```

```
[illegible]
```

Figure 2.3

2.12 12th WEEK (8/27/01 – 8/31/01)

As this was the last week, I was frantically trying to wrap everything up, especially the documentation. As for the proximity detector, I came up with a better looking interface rather than the one depicted in Figure 2.3. The new interface is a graphic user interface designed in JBuilder4. A screen shot of it can be seen in Figure 2.4.

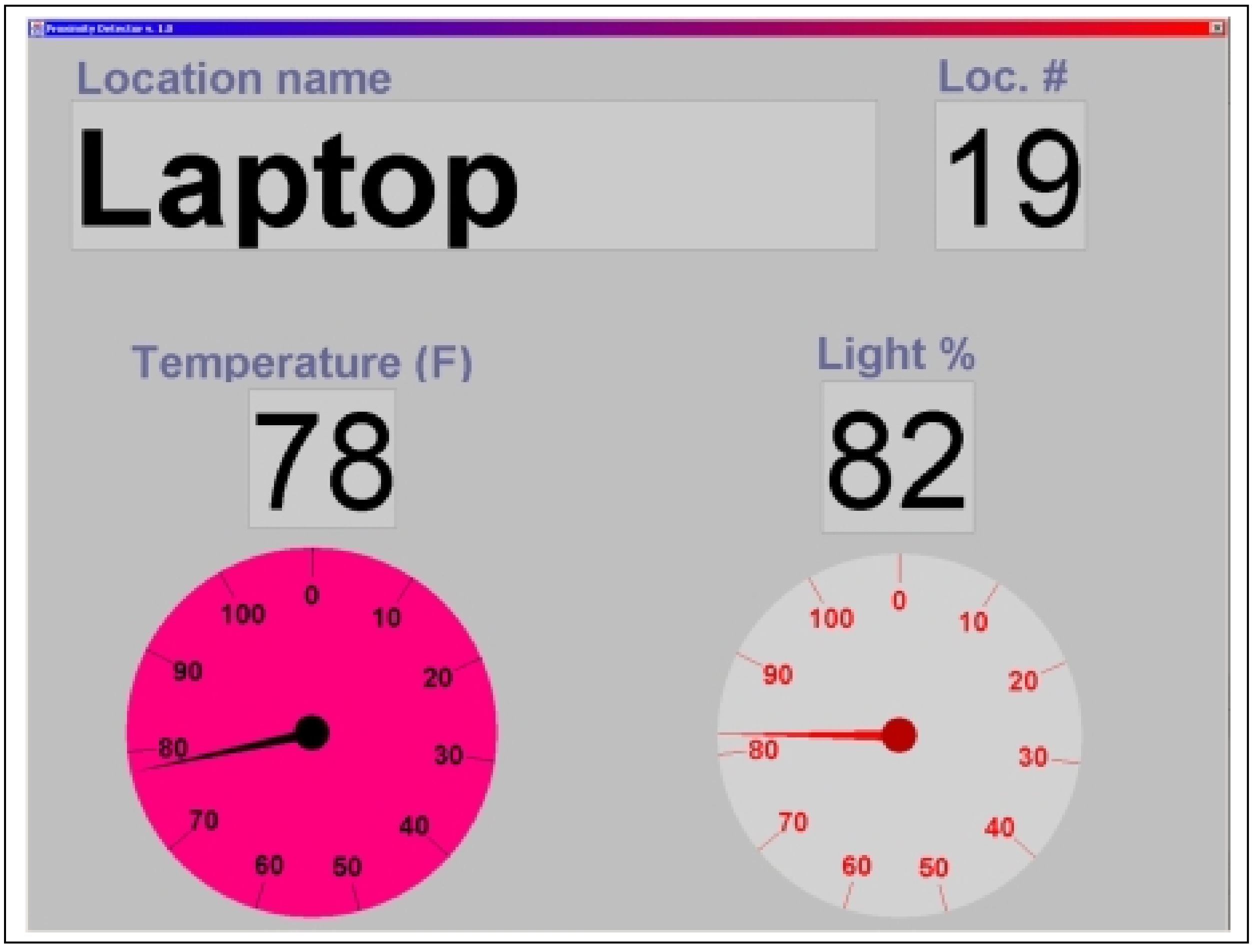


Figure 2.4

In figure 2.4, the name Laptop indicates that I (tagged with a mote as well) am closest to the location Laptop with an ID of 19. The temperature and light readings are indicating the conditions at the displayed location. The temperature was calibrated for Fahrenheit degrees and the light was calibrated to show percentages of light. The gauges will update along with the digital numbers, and also change colors as the values change. For the temperature gauge, the colors change from blue to red as the temperature rises from cold to hot. The light gauge also changes colors from black to white as the light levels increase.

3.0 Technical Details

The processor within the MCU (ATMEL 90LS8535), is an 8-bit Harvard architecture with 16-bit addresses. It provides 32 8-bit general registers and runs at 4 MHz and 3.0 V. The system is very memory constrained: it has 8 KB of flash as the program memory, and 512 bytes of SRAM as the data memory. Additionally, the processor integrates a set of timers and counters which can be configured to generate interrupts at regular time intervals.

The coprocessor represents a synchronous bit-level device with byte-level support. In this case, it is a very limited MCU (AT90LS2343, with 2 KB flash instruction memory, 128 bytes of SRAM and EEPROM) that uses I/O pins connected to an SPI controller. SPI is a synchronous serial data link, providing high speed full-duplex connections (up to 1 Mbit) between various peripherals. The coprocessor is connected in a way that allows it to reprogram the main microcontroller. The sensor can be reprogrammed by transferring data from the network into the coprocessor's 256 KB EEPROM (24LC256). Alternatively the main processor can use the coprocessor as a gateway to extra storage.

The radio is the most important component. It represents an asynchronous input/output device with hard real time constraints. It consists of an RF Monolithics 916.50 MHz transceiver (TR1000), antenna, and collection of discrete components to configure the physical layer characteristics such as signal strength and sensitivity. It operates in an ON-OFF key mode at speeds up to 19.2 Kbps, 115 Kbps using amplitude shift keying, but usually about 10 Kbps as raw data; the speeds are as low as they are because of the bit-level processing, and the conservation of the most valuable resource – battery life. Control signals configure the radio to operate in either transmit, receive, or power-off mode. The radio contains no buffering so each bit must be serviced by the controller on time. Additionally, the transmitted value is not latched by the radio, so jitter at the radio input is propagated into the transmission signal. The task scheduler is just a simple FIFO scheduler.

Commercial systems work in the 915 MHz to 2.8 GHz, thus having the motes transmit at higher frequencies (such as 5.8 GHz), would make them much less susceptible to interference, and also yield a higher throughput. However, the lower frequency is much less prone to problems because of line of sight, which seems to be the main concern for the motes. The range of the motes communication is 30 ft ~ 100 ft. The signal strength can be controlled through a digital potentiometer (DS 1804) from 0 ~ 50 kOhms, however, there is no observable effects on power usage; optimal setting seems to be around 10 kOhms. All the motes contend for a single channel RF radio. They use carrier sense multiple access (CSMA) and have no collision detection mechanism. Channel capacity is about 25 packets per second, so the wireless network can easily get congested if too many sensors are in the vicinity.

Three LEDs represent outputs connected through general I/O ports; they may be used to display digital values or status. The photo-sensor represents an analog input device. The input signal can be directed to an internal ADC in continuous or sampled modes. The temperature sensor (Analog Devices AD7418) represents a large class of digital sensors which have internal A/D converters and interface over a standard chip-to-chip protocol. In this case, the synchronous, two-wire I2C protocol is used with software on the microcontroller synthesizing the I2C master over general I/O pins. In general, up to eight different I2C devices can be attached to this serial bus, each with a unique ID. The protocol is rather different from conventional bus protocols, as there is no explicit arbiter. Bus negotiations must be carried out by software on the microcontroller. The serial port represents an important asynchronous bit-level device with byte-level controller support. It uses I/O pins that are connected to an internal UART controller. In transmit mode, the UART takes a byte of data and shifts it out serially at a specified interval. In receive mode, it samples the input pin for a transition and shifts in bits at a specified interval from the edge. Interrupts are triggered in the processor to signal completion events.

The type of battery that is feasible to use in motes is a watch size battery (Energizer CR2450 lithium battery @ 575 mAh). More noteworthy are the three sleep modes: idle, which just shuts off the processor, power down, which shuts off everything but the watchdog and asynchronous interrupt logic necessary for wake up, and power save, which is similar to the power down mode, but leaves an asynchronous timer running. There are three modes of operation, which last accordingly to the numbers below:

- Active – 30 hours
- Idle – 200 hours
- Inactive (Power down) – >1 year

Some of the voltage requirements that I have been able to find are:

- Atmel MCU – 6.6 volts max
- TR1000 radio transmitter and receiver – 4 volts max
- In order to program the motes, at least about 3.2 volts is needed
- Other components might have different voltage ratings, I still need to look into them.

In terms of security, there is some but not much. There is trusted communication with the bases; it uses RC5 cryptography to secure data transmissions as it shares secret keys between bases and each device. The overhead is minimal, as it only incurs less than 5 ms delay all the way from key setup, authentication, and encryption. The data transmitted and received to and from the motes adheres to the following standards:

- At the start of communication, <Base ID, 0, **>
- After parent has been established, each mote transmits <identity, depth, data>. The identity is the parent or routing path; the depth is the depth in the tree that the mote is situated. If a new parent comes along that has a lower depth that the current one, the routing will change to the new better path.

3.1 Interpreting the data

As for the actual data that comes across the wireless network, here is a sample output for the light sensor using the application "sens_to_rfm.desc".

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***	***21***	***22***	***23***	***24***	***25***	***26***	***27***	***28***	***29***	***30***
7E	04	13	6C	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7E	04	13	6F	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7E	04	13	75	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 3.1

- 1st (7E) – serial address (TOS_UART_ADDR); this is how the mote knows to send the packet to the serial interface rather than as a broadcast over the radio.
- 2nd (04) – type; I really don't know what this is, but according to Northbrook, it is the handler of the mote hopping, but I cannot verify this yet.
- 3rd (13) – group ID (LOCAL_GROUP); only packets belonging to the group that the base station belongs to will be allowed through; there is no authentication, but rather it is just like switching channels.
- 4th (6C) – data; light sensor, temperature sensor, etc…
- 5th (0F) – mote ID (TOS_LOCAL_ADDRESS); this will distinguish the mote from other motes, however, uniqueness is not guaranteed since the assigning of mote IDs is done manually at compile time.

For the applications I wrote, here is a summary:

- **envmon_crc_signal.desc** – program to be loaded on a mote that is to be used as a network node for simple communication between each node and the base station; it's supported features are: photo sensor readings, temp sensor readings, signal strength readings, and CRC checking implementation.

- **envmon_crc_signal_base.desc** – program to be loaded on the base station, in which it simply waits for incoming packets, and as long as they adhere to the right group, they will be forwarded to the UART; be aware that this application had to be given the support of both the signal strength and CRC in order for it to work properly with the "envmon_crc_signal.desc" application.

- **envmon_crc_signal_router.desc** – program to be loaded on a mote that is to be used as a network node for communication between each node and the base station; it implements mote hopping in which it relays any messages it hears from other nodes and includes them in the same packet with its own data, and forwards it on to the base station. Besides the mote hopping feature, it implements the same features as "envmon_crc_signal.desc".

- **envmon_crc_signal_router_base.desc** – program to be loaded on the base station; besides just waiting for incoming packets, and as long as they adhere to the right group, they will be forwarded to the UART, it will also periodically send messages containing routing update topologies; be aware that this application had to be given the support of both the signal strength and CRC in order for it to work properly with the "envmon_crc_signal_router.desc" application.

A sample output for these 4 programs is, using a 4 byte header, 30 bytes of data, and 4 byte trailer:

```
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
7E 08 22 00 19 02 DB 81 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7E 08 22 00 19 02 E0 82 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7E 08 22 01 0B 08 A0 CC 19 02 E0 82 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 45 A7 66 00
7E 08 22 00 19 03 E0 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7E 08 22 01 0B 08 AA CD 19 03 E0 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 D8 8A 55 00
7E 08 22 01 0B 08 9B CE 19 03 E0 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF B1 5B 00
7E 08 22 00 19 03 E1 84 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00
7E 08 22 02 13 05 9B 01 0B 08 9C CF 19 03 E0 84 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 23 51 2A 00
7E 08 22 02 13 05 9B 02 0B 08 9B D0 19 03 E1 85 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FA DB 2B 00
```

Figure 3.2

- 1st – serial address (TOS_UART_ADDR); this is how the mote knows to send the packet to the serial interface rather than as a broadcast over the radio.
- 2nd – type; the handler of the mote hopping.
- 3rd – group ID (LOCAL_GROUP); only packets belonging to this group will be heard by the base station; there is no authentication, but rather it is just like switching channels.
- 4th – the depth of the tree that the data came from; a depth of 0 means that the message came straight from a mote; a depth of 1 means that there is a message from a lower depth that is attached to the message as well; a depth of 2 means that there are 2 messages attached from lower levels, and so on…
- 5th ~ 9th – these 5 fields are the data of the mote; the next 6 sets of data, 10~14, 15~19, 20~24, and 25~29 are all identical in terms of the composition:
    - 1st – mote ID, which will distinguish each mote from one another
    - 2nd – the temperature data
    - 3rd – the light sensor data
    - 4th – packet counter, which could help in determining if a certain piece of data has already been received or not.
    - 5th – signal strength from received node

  If the depth is 0, then these 5 fields will hold the data of the sending mote; if the depth is 1, then these 5 fields will hold the data of the forwarded packet, since the original data from this packet was shifted down 5 fields; the original data from the sending mote will always be found in the last 5 fields that are being used. This can easily be extracted since we know the depth, we can just go to position "5+depth*5", and we have found the original data; with each level less, we get the next depth data, all the way till we reach the depth of 0, in which it was the last set of data from the furthest level.
- 10th ~ 14th – please refer to 5th ~ 9th.
- 15th ~ 19th – please refer to 5th ~ 9th.
- 20th ~ 24th – please refer to 5th ~ 9th.
- 25th ~ 29th – please refer to 5th ~ 9th.
- 30th ~ 33rd – undecided.
- 34th ~ 35th – CRC values; used by lower layers to do error detection.
- 36th ~ 37th – signal strength reading

4.0 Installation Instructions

All necessary files can be found in \\paw\motes. The instructions bellow assume that you have Windows 2000 Professional, however it should also work with Windows NT 4.0 with a late (3 or above) service pack.

4.1 Reading (listening) motes

To be able to listen to base stations, all you need is a properly installed JAVA system. Doing the following few steps can accomplish that.

4.1.1 JDK 1.3.1

Execute "\motes\install\jdk 1.3.1\j2sdk-1_3_1-win.exe". Agree to all default options and settings. Once installation is completed, open "\motes\install\jdk 1.3.1\path.txt" and added to the CLASSPATH under System Properties, Environmental Variables, System Variables; in the same place, also add "C:\jdk1.3.1\bin" to the PATH variable. To double check everything is good, open up a new command prompt window and type "java", which it should recognize it as a valid program, and display a list of possible options. Do the same thing with "javac", in which the result should be similar. If it comes back with a message that the program is not recognized as an internal or external command, operable program or batch file, then you do not have the path set properly and it cannot find the compiler.

4.1.2 COMMAPI

This package is an add-on to the core classes for the JDK which will allow the reading of the serial / parallel port with ease. The examples in this document assume that your JDK installation is in "C:\jdk1.3.1" and that the COMMAPI is in "C:\commapi".

- Copy win32com.dll to your <JDK>\bin directory.
  - C:\>copy c:\commapi\win32com.dll to c:\jdk1.3.1\bin
- Copy comm.jar to your <JDK>\lib directory.
  - C:\>copy c:\commapi\comm.jar c:\ jdk1.3.1\lib
- Copy javax.comm.properties to your <JDK>\lib directory.
  - C:\>copy c:\commapi\javax.comm.properties c:\ jdk1.3.1\lib
  - The javax.comm.properties file must be installed. If it is not, no ports will be found by the system.
- Add comm.jar to your classpath:
  - If you don't have a classpath defined:
    - C:\>set CLASSPATH=c:\ jdk1.3.1\lib\comm.jar
  - If you already have a classpath defined:
    - C:\>set CLASSPATH=c:\ jdk1.3.1\lib\comm.jar;%classpath%

4.1.3 JBuilder 4

Execute "\motes\install\JBuilder4\install_windows.exe". Select Full Installation, and agree to all default options and settings. Once the installation is complete, start up JBuilder4 and do the following actions:

- If prompted for serial number and authentication key, they are:
  - Serial number: XA22-?XYXU-ZZT8X
  - Authentication Key: HX6-TZY
- Go to the Project menu and select default project properties
  - Screen "Default Project Properties": Click on the … on the right side of JDK, which will allow you to select a different JDK
  - Screen "Select a JDK": click on NEW at the bottom left corner
  - Screen "New JDK Wizard": click on the … on the right side of Existing JDK home path
  - Screen "Select Directory": find the path of the SUN JDK 1.3.1, which should be C:\jdk1.3.1 and select it, and confirm all the windows until you get back to the screen "Default Project Properties".
  - Screen "Default Project Properties": make sure that under User Home, you have selected "java 1.3.1-B24", and not "java 1.3.0-C".
- You are now ready to make a new project or open an existing one.

4.2 Writing (programming) motes

To be able to program the motes, you need a UNIX shell, a C compiler, the Tiny OS, and device drivers to write to the parallel port. Follow the following steps to accomplish that.

4.2.1 Cygwin (UNIX shell)

Execute "\motes\install\Cygwin\setup.exe". Agree to all default options and settings unless otherwise specified. The following are the exceptions to the default options:

- Screen "Cygwin setup": select "Install from Local Directory"
- Screen "Cygwin setup": under "Select Packages to Install", select all packages under SRC?

Once the installation is complete, restart the "\motes\install\Cygwin\setup.exe", following the same exact steps as the first time, however this time when you get to the part of selecting packages to install, choose "Exp" instead of "Cur" at the top of the screen. Follow the same steps in choosing all those packages as well and continue the install just like before.

In the event that things do not work as expected in terms of the functionality of the shell, please copy the "\motes\c_drive\cygwin" directory over the fresh installed CYGWIN which should be in "C:\cygwin", overwriting all files.

4.2.2 AVRGCC (C compiler)

Execute “\motes\install\avrgcc\avrgcc20010211.exe”. Agree to all default options and settings.

In the event that things do not work as expected in terms of the functionality of the compiler, please copy the “\motes\c_drive\avr-gcc” directory over the fresh installed AVRGCC which should be in “C:\avr-gcc”, overwriting all files.

4.2.4 TVICPORT (device driver for writing to parallel port)

Execute “\motes\install\tvicport\install.exe”. Agree to all default options and settings.

4.2.3 TinyOS

Copy “\motes\c_drive\cygwin\tos4.3” and all its contents to your local Cygwin directory.

5.0 Documentation Details

Below you will find documentation on important files dealing with the TinyOS and the Proximity Detector.

5.1 Important Files and Short Description

- <u>c:\cygwin\tos4.3\Makefile</u>
  - File used to compile and upload the programs to the motes
  - Procedure of compiling and uploading are as follows:
    - make clean – cleans up any ".o" files; should always do this before compiling in order to do fresh recompiles of all new code.
    - make – compiles the specified program in the Makefile
    - make install_windows.15 – uploads the compiled program up to the mote; if the ".15" is omitted, then the mote will be assigned the default ID specified in the "c:\cygwin\tos4.3\main.c"; if ".15" is specified, then the corresponding decimal value will be assigned to the mode as its ID.
  - To compile a simple mote, please comment all programs except for (note that everything past a "#" on that line is considered commented out):
    - DESC=apps/envmon_crc_signal.desc
    - CFLAGS += -DMOTE
  - To compile a simple base station, please comment all programs except for:
    - DESC=apps/envmon_crc_signal_base.desc
    - CFLAGS += -DBASE_STATION
  - To compile a mote that will handle routing, please comment all programs except for:
    - DESC=apps/envmon_crc_signal_router.desc
    - CFLAGS += -DMOTE
  - To compile a base station that will handle routing, please comment all programs except for:
    - DESC=apps/envmon_crc_signal_router_base.desc
    - CFLAGS += -DBASE_STATION
  - In order to achieve the proximity detection properly, I also devised another program which combines the functionality of a routing mote and a routing base station; these are going to be the mote base stations, and they are going to be fixed throughout the environment within range of the base station that is hooked up to the main computer. At the moment, in order to avoid message echoes from bouncing back and forth, I made the route static towards the base station, however this should be dynamic and learned by the motes themselves. To compile a base station that will handle routing, please comment all programs except for:
    - DESC=apps/envmon_crc_signal_router_base.desc
    - CFLAGS += -DBASE_STATION_MOTE

- c:\cygwin\tos4.3\apps\envmon_crc_signal.desc
    - Simple mote program that implements CRC error detection, signal strength reading, photo sensor, and temperature sensor, all at once; it periodically sends packets out over the radio as a broadcast with all of the above data. For details on the organization of the data contained in the packet, please refer to Figure 3.2.
- c:\cygwin\tos4.3\apps\envmon_crc_signal_base.desc
    - Simple base mote program that simply listens and any packets that come through with the proper Group ID are forwarded along the UART. It implements the CRC error checking, in which it throws out any bad packets; and also implements the signal strength in which it calculates the strength based on the information received from the mote.
- c:\cygwin\tos4.3\apps\envmon_crc_signal_router.desc
    - The program is very similar to “c:\cygwin\tos4.3\apps\envmon_crc_signal.desc”, however it also implements mote hoping. Besides doing the usual work load of reading its sensors and sending them out, it also listens for messages coming from other motes; if it does receive anything, it concatenates that data to its own data, places it all in a packet, and forwards it along. It implements all the usual stuff, such as CRC, signal strength, photo sensor, and temp sensor.
- c:\cygwin\tos4.3\apps\envmon_crc_signal_router_base.desc
    - This program is similar to “c:\cygwin\tos4.3\apps\envmon_crc_signal_base.desc” in functionality, however it also periodically advertises that it is the base station so other motes can establish a routing path back to it. It also implements all the usual stuff, such as CRC, signal strength, photo sensor, and temp sensor.
- c:\cygwin\tos4.3\env.c
    - Take periodic samples (light and temp) at the same time, perform simple local processing, and periodically emit data message
- c:\cygwin\tos4.3\env.comp
    - Component file for “c:\cygwin\tos4.3\env.c”.
- c:\cygwin\tos4.3\env_router.c
    - This module sends out the value of sensor readings to the radio. It periodically samples the sensor values and then sends out a single radio packet. It also listens to the radio and anything it hears, it places it in its own packet and forwards everything along. Essentially, the received packet will include an immediate history of all received messages that the mote heard.
- c:\cygwin\tos4.3\env_router.comp
    - Component file for “c:\cygwin\tos4.3\env_router.c”.
- c:\cygwin\tos4.3\system\photo.c
    - OS component abstraction of the analog photo sensor and associated A/D support. It provides an asynchronous interface to the photo sensor.

- c:\cygwin\tos4.3\system\photo.comp
    - This module encapsulates the PHOTO sensor. It automatically gives the correct commands to the ADC and then returns the sensor reading to the application with the DATA_READY event. It is partially implemented by the PHOTO.c file and also the PHOTO.desc file. PHOTO.desc file includes the ADC and the wiring between the ADC and the PHOTO component. Most sensor files will look very similar except for names and channels and ports.
- c:\cygwin\tos4.3\system\photo.desc
    - Description file for "c:\cygwin\tos4.3\photo.c".
- c:\cygwin\tos4.3\system\temp.c
    - OS component abstraction of the analog temp sensor and associated A/D support. It provides an asynchronous interface to the temp sensor.
- c:\cygwin\tos4.3\system\temp.comp
    - This module encapsulates the TEMP sensor. It automatically gives the correct commands to the ADC and then returns the sensor reading to the application with the DATA_READY event. It is partially implemented by the TEMP.c file and also the TEMP.desc file. TEMP.desc file includes the ADC and the wiring between the ADC and the TEMP component.
- c:\cygwin\tos4.3\system\temp.desc
    - Description file for "c:\cygwin\tos4.3\temp.c".
- c:\cygwin\tos4.3\system\include\basic.h
    - Sensor information, such as channels and pin IDs
- c:\cygwin\tos4.3\system\include\msg.h
    - Packet details, such as member variables, packet length, etc…
- c:\cygwin\tos4.3\system\include\hardware.h
    - Specific to the Rene mote, it defines the hardware layout in terms of pin IDs, ports, etc…

5.2 Example on adding sensors

Example on adding a third sensor, and for ease of explanation, lets say it is a humidity sensor:

- Define a series of files called "humidity.c", "humidity.desc", and "humidity.comp".
- Start by importing all data from another sensor ("photo.*")
- Replace anything that says photo with humidity within all variables in all 3 files
- Open "basic.h" and add (make sure you have connected the sensor to PW3 and ADC channel 3; this can be done by referring to Figure 5.2 on the next page:
    - ALIAS_PIN(HUMIDITY_CTL, PW3);

- #define HUMIDITY_CHANNEL 3

- Open “humidity.desc”, and change the port to the right one as defined in the “basic.h”, in our case it would be 3.

Using the steps above, you will have the foundation to be able to read this new sensor. If you open any existing application (“.desc”) from the “C:\cygwin\tos4.3\apps” directory, and just replace either anything that contains say photo with humidity, the new application should read the new sensor just like it used to read the old one.

If it is desired to make all sensors work at the same time, the process becomes much more involved, however it can be figured out rather simple if some logic is used and one is familiar with the TinyOS setup. Lets say I wanted to adapt the simple mote program “c:\cygwin\tos4.3\apps\envmon_crc_signal.desc“ to read a third sensor humidity. Once the foundation of “humidity.*” is laid, the files that will have to be modified are:

- c:\cygwin\tos4.3\apps\envmon_crc_signal.desc
- c:\cygwin\tos4.3\env.c
- c:\cygwin\tos4.3\env.comp

In these 3 files, you have to replicate the functionality of one of the old sensors (photo or temp) and rename it to humidity.

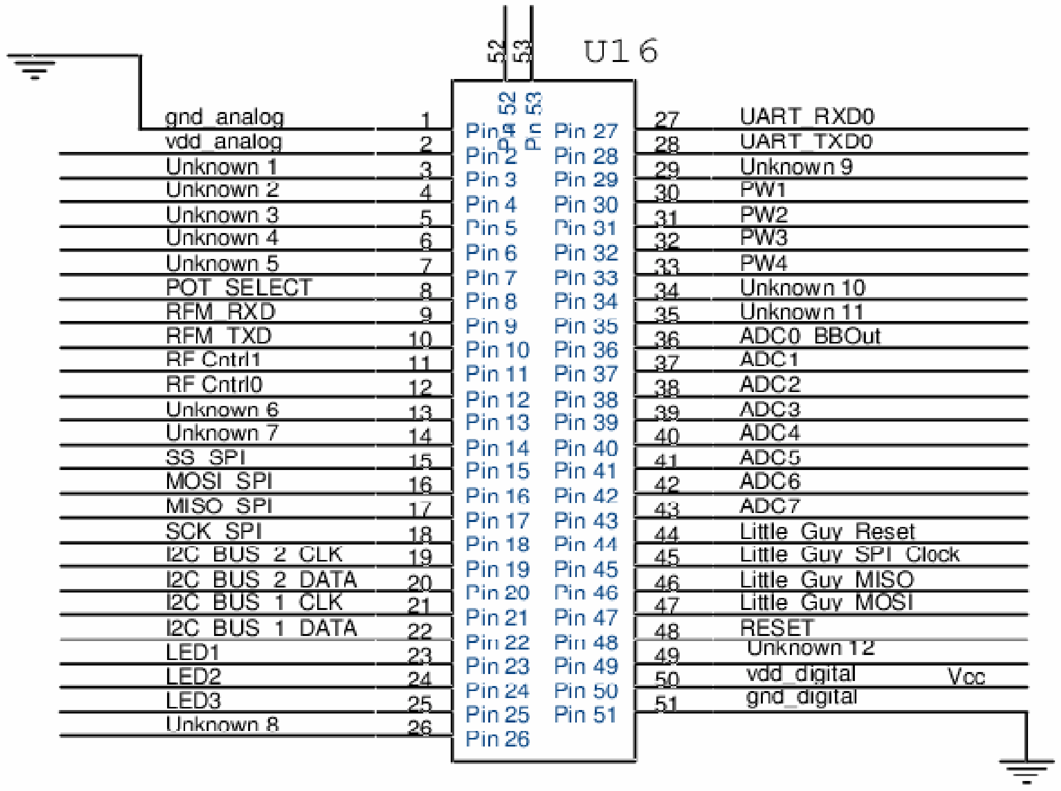


Figure 5.2

5.3 Changing parameters on existing programs

I am going to use the “c:\cygwin\tos4.3\apps\envmon_crc_signal.desc” as my example. Be aware that along with this application, there are many files that are associated with. The highest level files are “env.c” and “env.comp”, which utilize the lower levels. Most things can be customized simply by changing these few files.

5.3.1 Changing mote ID

At compile time, “make install_windows.15”, the 15 will represent the decimal representation of the mote ID. If the 15 is omitted “make install_windows”, then the mote will be assigned the default ID specified in the “c:\cygwin\tos4.3\main.c” by the variable “TOS_LOCAL_ADDRESS”. There is not way to guarantee a unique ID, and therefore more than one mote can have the same ID; the way this would behave in the network is that the data coming from both motes will be viewed as coming from on mote, and data directed towards the motes with the same ID will reach both motes as long as they are both within communication range.

5.3.2 Changing group ID

Open the file ”c:\cygwin\tos4.3\Makefile” and edit the variable “LOCAL_GROUP” to the desired HEX number. Be aware that only motes in the same group can communicate with each other. If the group number does not match, then the packets will be rejected and they will never reach the application layer.

5.3.3 Changing frequency of packet transmission

In our case, we need to look for “TOS_CALL_COMMAND(ENV_CLOCK_INIT)(tick2ps);” in the “env.c” file. Once you find this, and it is usually at the beginning as the device initializes, you can change the variable tick2ps (2 ticks per second) to whatever you want. The definition of these (tick2ps) variables can be found in “c:\cygwin\tos4.3\system\include\hardware.h”. For other files, the only thing that changes is instead of “ENV_CLOCK_INIT”, it would be “ENV_ROUTER_CLOCK_INIT”. You could always do a search for “CLOCK_INIT”, or “tick*ps” to find places where the rate of transmission is defined. By the way, these definition will most always be found in “.c” files.

5.3.4 Changing packet length

To change packet length, open “c:\cygwin\tos4.3\system\include\msg.h” and change the variable “DATA_LENGTH” to the desired length. Be aware the there is also a header and a trailer, that are defined in “struct MSG_VALS{}”, and therefore the total length of the packet will always be more than the “DATA_LENGTH”. Although the current “DATA_LENGTH” is only 30 bytes, because of the header (3 bytes) and trailer (4 bytes), the total packet length is 37 bytes. Be aware that the receiving application on the PC (the listen application written in JAVA) needs to know how long packets are so it knows how to segment the incoming data and parse it properly.

5.3.5 Adding variables to the header or trailer

This can all be done in the file “c:\cygwin\tos4.3\system\include\msg.h”, however that merely adds them to the structure and assembles it to send over the radio, however to make those new variables useful, more work needs to be done depending on their functionality and what layer they want to be utilized at.

5.3.6 Adding variables to the data portion

For out example, you would open file “c:\cygwin\tos4.3\env.c” and find the structure env_Msg.

```
typedef struct
{
        char level;
        char src;
        char temp;
        char photo;
        char  msgNum;
} env_Msg;
```

By simply adding another variable here, or deleting one, you can change the layout of the data. Be aware that the order of the variable is the same order that they will appear in the actual data packet. Obviously, you still have to do something meaningful with the new added variable as the above procedure only adds it to the data packet.

5.3.7 Changing resolution of data coming in (bit shifting)

The data as read from the sensors is recorder as a 10 bit number, however 2 bits are used for internal processing and therefore only 8 bits are usable at most. By bit shifting operations on the raw data, one can increase or decrease the sensitivity and granularity of the received data. In my example, you would have to open file “c:\cygwin\tos4.3\env.c” and find “TOS_TASK(ENV_send_sample) {}” and edit the following lines with the appropriate bit shift operation:

- envMsg->photo   = (char) (VAR(photoSum) >> 3) & 0xFF;
- envMsg->temp   = (char) (VAR(tempSum) >> 3) & 0xFF;

Depending on the bit shift operation, the resulting values will either increase or decrease its range of possible values. For example by bit shifting the light reading by 3, the total possible range of the reading is 0 to 128. To increase or decrease this range, change the bit shift operation up or down. At the extremes, the range will either be too small and it will not change enough to differentiate the readings, or it will be too much and overflow will occur in which the data that is read will not be reliable. The 0xFF is the bit value that will replace any unused bits after the bit shift operation.

5.3.8 Changing static route for base station motes that are used in the Proximity Detector

Open the file “C:\cygwin\tos4.3\env_router.c” and find the following:

```
#ifdef BASE_STATION_MOTE
        VAR(route) = 0x0f;
        VAR(set) = 12; // BS beacons every 12 seconds
        VAR(level) = 1;
        TOS_COMMAND(ENV_ROUTER_SUB_CLOCK_INIT)(tick1ps);
#endif
```

By changing the “VAR(route) to whatever is needed, you can change the route that all messages leave upon when this program transmits a message. To broadcast to everybody, use address “TOS_BCAST_ADDR”.

6.0 Glossary

1.