Solar-Powered Wireless Sensor Network

E90: Senior Design Project
FINAL REPORT

Department of Engineering
Swarthmore College
May 4, 2006

Authors:
Brian Park
Simeon Realov

Advisor:
Prof. Erik Cheever

TABLE OF CONTENTS:

INTRODUCTION

The goal of this project was to build an autonomous wireless data acquisition system that offers a seamless and cost-effective solution to the problem of gathering remote sensory data. A good example of where such a system would be particularly useful is environmental monitoring, but this system can extend to any non-time-critical application. The network system is autonomous and requires minimal human interaction. Since all of the data is transferred wirelessly and the power is harvested from the sun, all one needs to do in order to install our remote sensory system is ensure access to solar energy. Beyond that, installation consists merely of setting up the nodes, attaching the sensor outputs to the module, and collecting the data on the other end. As a result, engineers using our system require very little knowledge of how the system operates in order to install it. The network topology and data acquisition algorithms are pre-programmed onto a microcontroller. The user is also allowed the flexibility to modify the system parameters to better suit his or her specific needs by reprogramming the microcontroller. Overall, our system is reasonably priced, versatile, and easy to use.

Throughout this paper we describe the design and performance of our solar-powered wireless sensor network. We begin by outlining the hardware design of a single Sensor Node along with a Computer Host, then we proceed to describe the network topology and networking protocol, and finally we demonstrate the proper functioning of our network and discuss suggested improvements. One of the main attributes of this project is to make all of the hardware and software designed over the course of this semester available in an accessible form to anyone who wishes to implement our network for a particular sensory data gathering application. Specific instructions on how to implement our network are included as part of this report.

GENRAL SYSTEM DESCRIPTION

Two different circuit designs are implemented in our network: a Sensor Node circuit and a Computer Host circuit (Figure 1). The Sensor Node gathers data from sensors and forwards the data to a Computer Host. The Computer Host interfaces with a PC and receives data from all of the Sensor Nodes. The circuits were designed to be small in size in order to be relatively inconspicuous. Since power is a critical issue for our project, all of the chips and devices used in our system have low power consumptions.

Figure 1. Circuits used in our network: Sensor Node (top) and Computer Host (bottom).

A tree topology was used to structure our network, an example of which is given in Figure 2 below. The tree topology was implemented as part of the low-power RTS (Receive-Transmit-Sleep) protocol we designed for this project. In our network implementation, data from all Sensor Nodes is propagated up to the main parent node, which is represented by the Computer Host.

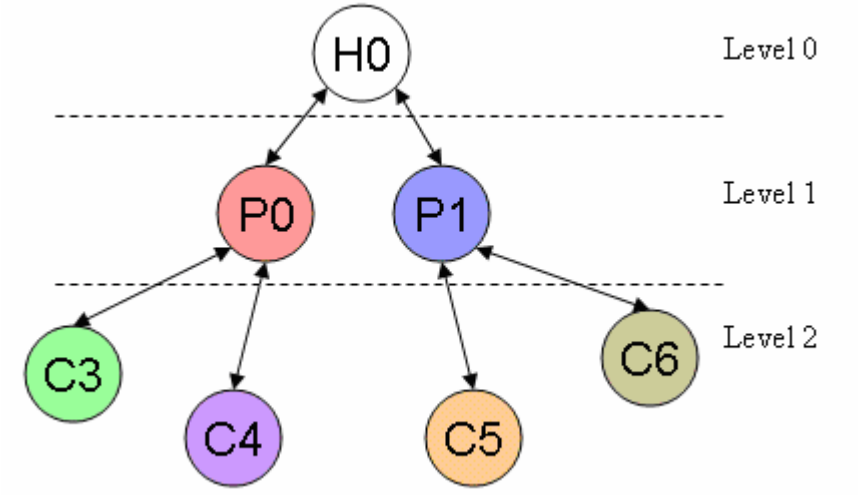


Figure 2. Example of a tree network topology with a host H0 as the root.

HARDWARE

The hardware portion of this project involved designing and constructing the Sensor Node and Computer Host circuits. The Sensor Node mainly consists of a microcontroller, an external EEPROM memory chip, and a wireless RF module. The nodes are powered by rechargeable batteries and a solar panel. Since all of our circuits run on 3.3V, a voltage regulation stage was necessary to regulate down the battery

voltage. The Computer Host also includes a wireless RF module, but does not need a microcontroller. The host communicates with a PC through the serial port. These devices, along with others, will be described in more detail in the following paragraphs.

Microcontroller

The microcontroller used to operate our circuits is Microchip's PIC16LF876A. The CMOS flash-based 8-bit microcontroller is a high-performance Reduced Instruction Set Computer (RISC) consisting of only 35 single word instructions. The microcontroller supports both digital and analog inputs, allowing easy interfacing with digital and analog sensors. The programming language used to program the PIC is very similar to standard (ANSI) C. This made developing the software an easier task since we have taken prior coursework involving the C programming language. A summary of features given by the manufacturer include: 256 bytes of EEPROM data memory, self programming, an In-Circuit Debugger (ICD), 2 Comparators, 5 channels of a 10-bit Analog-to-Digital (A/D) converter, 2 capture/compare/PWM functions, a synchronous serial port that can be configured as either 3-wire Serial Peripheral Interface (SPI) or the 2-wire Inter-Integrated Circuit (I²C) bus, and a Universal Asynchronous Receiver Transmitter (USART).[1]

Serial Communication

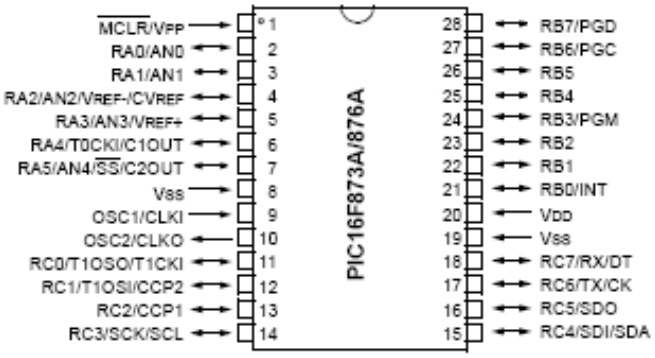
The PIC16F87XA microcontrollers have a variety of built-in features that were very useful for our particular project. The Universal Asynchronous Receiver Transmitter (USART) feature is used to establish RS-232 serial communications with our wireless RF modules. The PIC communicates with the wireless modules asynchronously through pins TX and RX (Figure 3). The Master Synchronous Serial Port (MSSP) feature supports Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) communication, providing seamless integration for digital devices and sensors.

Serial Peripheral Interface (SPI) is a simple four-wire serial interface standard defined by Motorola. The four lines required for operation are a clock, data in, data out, and chip select. Eight bits of data can be synchronously transmitted and received simultaneously. The chip select line allows multiple devices to be connected to the SPI bus in parallel. Pins SCK, SDI, and SDO are allocated for SPI (Figure 3). An external EEPROM chip is interfaced to the PIC through the SPI bus and will be described in more detail later. The additional memory greatly increases the amount of data that can be measured and stored.

Inter-Integrated Circuit is a two-wire serial interface standard defined by Philips. Only two bidirectional lines are required: clock and data. I^2C uses a 7-bit addressing scheme to establish serial communication with multiple devices. Pins SCL and SDA on the PIC are allocated for I^2C (Figure 3). Setting the MSSP for SPI or I^2C communication is done in software. For example, the `setup_spi()` function is used to setup SPI using pins SCK, SDI, and SDO. `spi_read()` and `spi_write()` are used to receive and send data over the SPI interface, respectively. `#USE I2C` is used to setup I^2C using pins SCL and SDA. `I2C_read()` and `I2C_write()` are used to receive and send a single byte over

[1] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1335&dDocName=en010240

the I^2C interface, respectively. See the PIC C Compiler Reference Manual for more information.



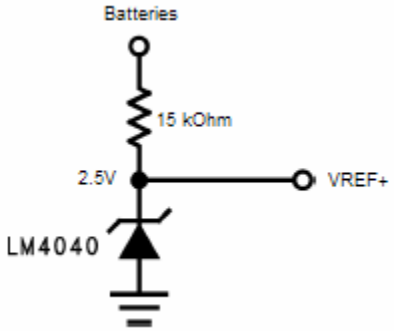
http://ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf

Figure 3. Pin diagram of 28-pin PDIP.

Analog-to-Digital Converter

Analog sensors can also be easily interfaced with the PIC. The microcontrollers are equipped with up to 5 channels of a 10-bit analog-to-digital (A/D) converter. A/D converters convert continuous signals to discrete digital values. Port A is allocated for analog input or digital input/output. The measurement range of the A/D converter is, by default, V_{DD} (supply voltage) to V_{SS} (ground). However, since the PIC is powered by batteries, the supply voltage is not a reliable reference for accurate analog-to-digital conversion. Therefore an external 2.5V shunt voltage reference (LM4040) is used to set the higher end of the measurement range: 2.5V to V_{SS} (ground). The external reference is connected to pin VREF+ of the PIC (Figure 4). The code below sets up the A/D converter to use an external reference and use all other pins on Port A for analog inputs. The last line sets up pin AN0 to be the analog pin that the PIC reads when `read_adc()` is called.

```
//Setup adc port to read from channel 0 (pin AN0)
setup_port_a( ALL_ANALOG );
setup_adc( ADC_CLOCK_INTERNAL );
setup_adc_ports( ANALOG_RA3_Ref );
set_adc_channel(0);
```

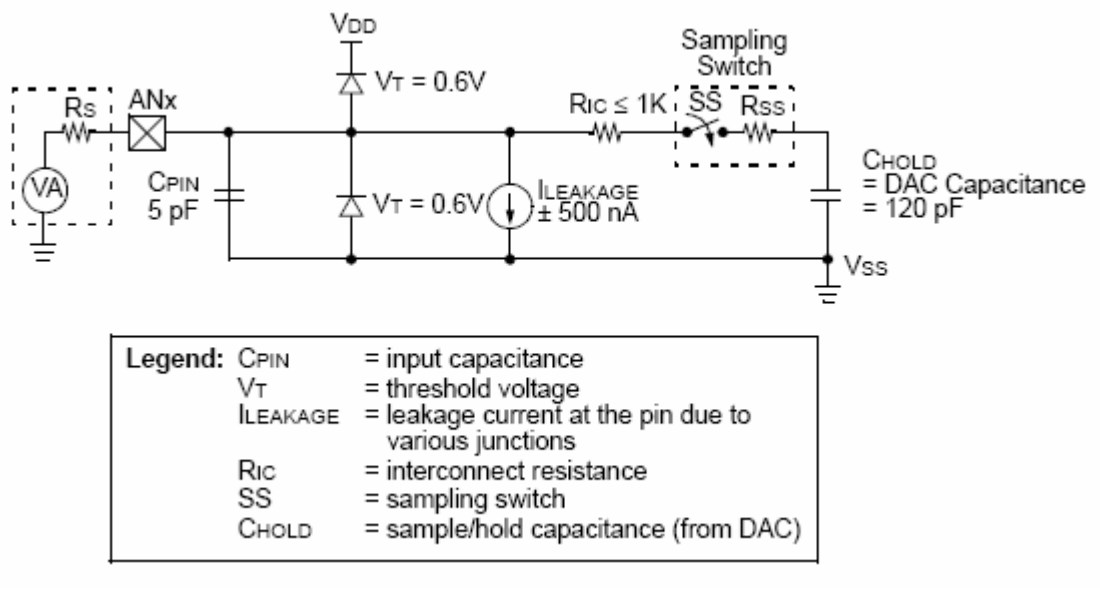


http://www.national.com/ds/LM/LM4040.pdf

Figure 4. Setup for external voltage reference.

Measuring Battery Voltage

The battery voltage is monitored to keep track of the operating capacity of our circuits. The battery voltage is measured using the PIC A/D converter. A voltage divider is used to decrease the voltage down to the measurable range of the A/D converter. Large resistor values in the MΩs were chosen to limit the amount of current sunk by the voltage divider. Since the A/D converter uses a 120pF sample and hold capacitor, the maximum recommended impedance for analog sources given by the manufacturer is 2.5kΩ (Figure 5). This is to ensure that the time constant for charging the sample and hold capacitor is small enough for accurate operation.



http://ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf

Figure 5. Analog input model for PIC A/D converter.

In order to overcome the large impedance from the output of the voltage divider, a simple 10µF charge-sharing capacitor was used (Figure 6). The charge-sharing capacitor stores charge from the voltage divider and provides the A/D converter with sufficient current for accurate operation. When the analog channel is sampled, charge from the 10µF capacitor transports to the 120pF capacitor. Since the battery voltage does not change at a very fast rate, the large time constant, $\tau = RC = (3.0M\Omega||2.2M\Omega)*10\mu F = 12.7s$, involved with charging the 10µF capacitor can be neglected.

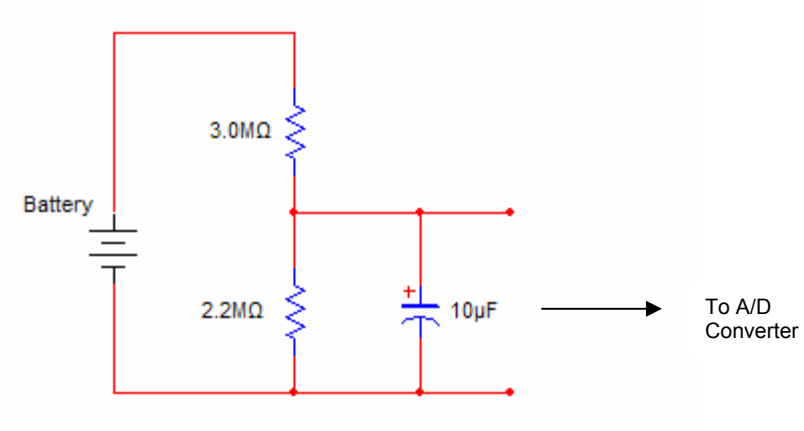


Figure 6. Battery voltage divider.

Real-Time Clock

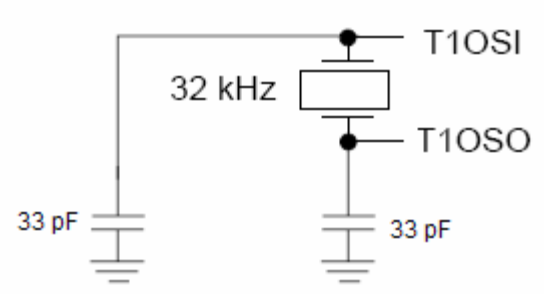
The PIC Timer1 module is used to implement a real-time clock. Timer1 is a 16-bit timer/counter with a prescaler. A timer interrupt flag is enabled to trigger the execution of an interrupt service routine when the timer overflows. Overflow occurs when the timer reaches 2^{16}. An external 32.768 kHz crystal oscillator is used to operate the timer. The oscillator is connected to pins T1OSO and T1OSI (Figure 7). Two 33 pF stabilizing capacitors are included on both sides of the oscillator. The external oscillator is passed through an 8-bit prescaler which is set to divide the frequency by 4, causing the timer to increment 8192 times per second. In order to make the timer overflow in 1 second, the timer is initialized to 57344. Inside of the interrupt service routine, global variables are incremented to keep track of real time. The code below initializes Timer1 to increment 8192 times per second.

```
setup_timer_1(T1_EXTERNAL|T1_EXTERNAL_SYNC|T1_DIV_BY_4|T1_CLK_OUT);
```

Code for enabling the Timer1 interrupt flag along with a simple interrupt service routine is shown below.

```
#INT_TIMER1          // interrupt triggered when timer overflows
void timer1_isr() {
    //increment a global counter variable once every second
    seconds++;

    // 2^16-32768/4 = 57344; overflow once every 32768/4 = 8192 counts
    set_timer1(57344);
}
```



http://ww1.microchip.com/downloads/en/DeviceDoc/33023A.pdf

Figure 7. Setup for Timer1 external oscillator.

The PIC16LF876A was chosen, in particular, because of its low power consumption, small size, and memory size. This microcontroller has a wide operating voltage range of 2.0V to 5.5V. The LF model typically draws a supply current of only 1.6mA. Also, the available 28-pin PDIP package allowed the overall circuit design to be small and compact. Initial work was done with the PIC16LF873A, which was used extensively in Electronic Circuit Applications (ENGR 72). However, due to memory constraints, the 873A was replaced with the 876A model to provide additional memory to accompany our software and algorithms. The 876A has twice the memory capacity of the 873A, as can be seen in Table 1. More information about the PIC can be found in Appendix A1.

Table 1. Memory Capabilities of PIC16F873A/876A

Model	*Program Flash*	*Data Memory*	*Data EEPROM*
16LF873A	4K words	192 bytes	128 bytes
16LF876A	8K words	368 bytes	256 bytes

http://ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf

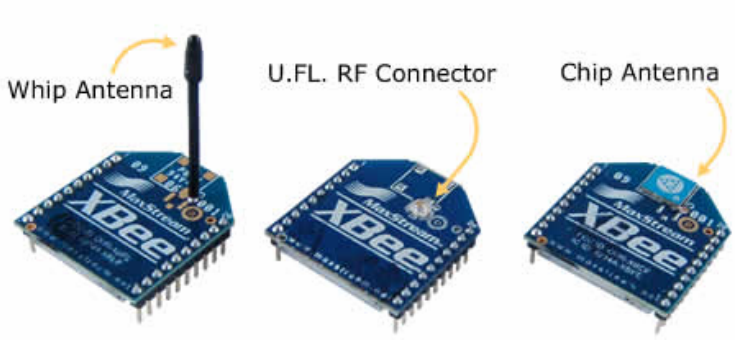
RF Communication

The selection of the RF communication modules used for the wireless data transmission portion of our design was based on a number of different criteria such as range of communication, power consumption, ease of integration, and cost. The wireless transceivers that we chose for our design are the XBee and XBeePRO, which conform to the IEEE 802.15.4 standard and are offered by MaxStream, Inc. The IEEE 802.15.4 wireless standard, more commonly known as ZigBee, is ideally suited for our project. Similar to the more popular and established IEEE 802.11b and Bluetooth standards, it operates in the commercial 2.4GHz (ISM) radio band. The specification allows for up to 255 network nodes and maximum transfer rates of 250Kbps at a range of 30 meters. ZigBee technology is slower than 802.11b (11Mbps) and Bluetooth (1Mbps) but consumes significantly less power. This makes the IEEE 802.15.4 standard particularly suitable for our project, since it was specifically designed for data gathering applications with relatively low transfer rates and limited power resources.

XBee vs. XBeePRO

Maxstream's ZigBee-based RF module comes in two varieties: the XBee and the XBeePRO. Apart from the price, the main difference between the two modules is the range of communication and the power consumption during transmission (refer to data sheet, Appendix A6). The XBeePRO is the more powerful of the two, consuming approximately 891mW of power during transmission and covering a line-of-sight range of up to 1.6km. The XBee, on the other hand, has more modest power consumption during transmission of approximately 149mW, but it also has a much more restricted range of communication of only 100m line-of-sight. Both modules have similar power consumption during reception (165mW for the XBee and 182mW for the XBeePRO). In terms of pricing, the XBeePRO costs $32.00, while the XBee costs only $19.00. Clearly, the two flavors offer a choice between power consumption and range of communication. Since the two modules have identical footprints, however, they are completely interchangeable, which meant that we could use either one in our finished design without making any modifications whatsoever. As a result, we decided to order and use both the XBee and XBeePRO for our project, and assess directly the capabilities of each one.

Antennas



http://www.maxstream.net/products/xbee/xbee-oem-rf-module-zigbee.php

Figure 8. Antenna options for XBee modules.

The RF modules come with three different antenna options: a whip antenna, a U. FL RF connector, or a chip antenna. The three different antenna options are demonstrated in Figure 8 above. The whip and chip antennas come integrated onto the actual modules, whereas the U. FL. RF connector can be used for connecting an off-chip dipole or other external antenna. Table 2 below gives a comparison between the performance of the whip antenna and the chip antenna in different settings. Connecting a dipole to the U. FL RF connector gives similar performance to the whip antenna option.

Table 2. Performance of Antennas

Module	Antenna Type	Outdoor Distance (Visual Line-of-Sight)	Indoor Distance (Office Building)	Indoor Distance (Warehouse)
XBee	Chip	470 ft. (143 m)	80 ft. (24 m)	-
	Whip	845 ft. (258 m)	80 ft. (24 m)	84 ft. (26 m)
XBee-PRO	Chip	1690 ft. (515 m)	140 ft. (43 m)	-
	Whip	4382 ft. (1335 m)	140 ft. (43 m)	355 ft. (108 m)

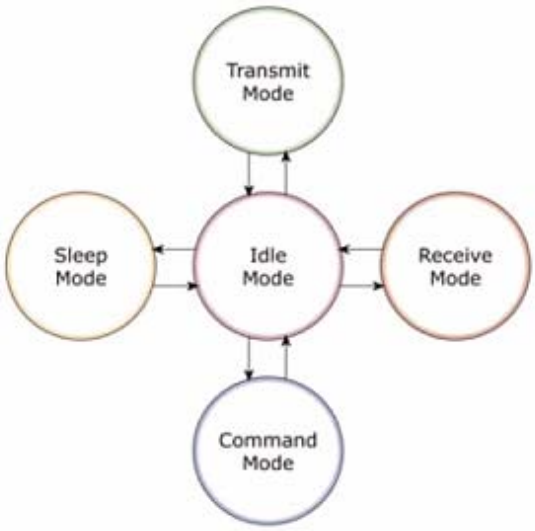
http://www.maxstream.net/support/knowledgebase/files/XST-AN019a_XBeeAntennas.pdf?PHPSESSID=2e9f6eb44b2dbe8a2089473c02c0b141

Even though the whip antenna gives better performance than the chip antenna, we suspected that it might be harder to work with. Due to problems with antenna selection in previous wireless communication projects we have worked on, we decided to go with the option that seemed the least problematic. Consequently, we chose to order the modules with integrated on-chip antennas.

UART

One of the main advantages of the XBee module is its UART (Universal Asynchronous Receive Transmit) serial interface. This interface makes it ideal for communication with a PC, as well as a PIC microcontroller. Essentially, when operating in its normal receive/transmit mode, the XBee serves as a wireless serial communications extension, and in simple applications it can be used as a serial cable replacement, handling baud rates as high as 115,200. This feature made the XBee modules particularly appealing due to our familiarity with asynchronous serial communications.

Modes of Operation



http://www.maxstream.net/products/xbee/product-manual_XBee_OEM_RF-Modules.pdf

Figure 9. XBee modes of operation.

The XBee has five modes of operation as seen in Figure 9 above: idle mode, receive mode, transmit mode, command mode, and sleep mode. When powered up, the XBee automatically goes into idle mode. If RF data comes in through the wireless port,

the modules goes into receive mode, buffering in the data and then forwarding it to the host via the serial port. When data is sent to the XBee from the host through the serial port, the XBee automatically switches to transmit mode and transmits the data wirelessly. The third mode the XBee can enter is a command mode, which allows it to receive simple AT commands from the host through the serial port. These commands are similar to the commands used in old internet modems and can be used to change configurations such as destination address, baud rate, packet size, transmission channel, to name a few (for a full list of available commands, please refer to Appendix A6). The command mode makes the XBee module very easy to configure. Finally, the XBee has a sleep mode, which comes in a number of different variations. The sleep mode we chose to use is Sleep Mode 1, which is pin-controlled and allows us to reduce the XBee's power consumption to less than 33μW.

External Memory

EEPROM vs. SRAM

We decided to use EEPROM external memory for our design, which would serve the purpose of providing temporary storage for the sensory data collected at each node. We chose EEPROM over SRAM because EEPROM memory is non-volatile. Thus, in the unlikely event of complete power shutdown, the data already gathered would not be compromised. Also, EEPROM chips need to be powered only during a read or write cycle, and can spend most of the time in a very low-power stand-by mode, consuming much less power than SRAM chips, which need to be powered at all times. Of course, using EEPROM comes at the price of slower read/write cycles. However, due to the relatively small network throughput we require for our suggested applications, this would not be a problem. In essence, our choice ensures higher reliability and lower power consumption at the price of slower data access speed.

EEPROM Selection

The EEPROM chip we decided to use is Microchip's 25AA160A 16Kbit SPI Bus Serial EEPROM. This chip consumes about 10mW during read/write cycles, and about 3.3μW in low-power stand-by mode. It has a lifetime of 1,000,000 erase/write cycles, which translates to about 2 years of operation under the conditions specified by our networking protocol.

The main reason we chose the 25AA160A EEPROM was the fact that we could interface it using the SPI (Serial Peripheral Interface) bus of our microcontroller, which is completely separate from its UART module. Thus, we were able to simultaneously communicate with both the external EEPROM memory and the XBee module. As can be seen in the later section describing our communications protocol, this functionality is crucial for our network's proper operation. Finally, it should be noted that even though the EEPROM uses the SPI interface, this interface could also easily be used to communicate with other peripherals as well, such as digital sensors for example, by using a simple addressing scheme which enables and disables devices connected to the same SPI bus.

Memory Organization

Microchip's 25AA160A 16Kbit SPI Bus Serial EEPROM is organized into 128 16-byte pages. Only one page can be written per write cycle, each of which takes approximately 5ms, but the entire memory can be read sequentially. Since there weren't any available drivers to use with this particular chip, we had to write our own driver code (see later section on Hardware Drivers). Fortunately, apart from some minor difficulties associated with the low-to-high data triggering for the PIC's SPI port, this was not a problem. Finally, using this chip had the advantage of allowing us to expand the memory of our sensory network nodes without modifying the hardware (including the driver code), since there are identical chips in its family that go up to 256Kbits (the maximum addressable memory using a 16-bit addressing scheme).

Sensors

In order to test the performance of our system, simple integrated circuit (IC) temperature sensors were used. The LM19 by National Semiconductor is a precision analog integrated circuit (IC) Centigrade temperature sensor. The sensor has an operating range of -55°C to +130°C. The input voltage range is from 2.4V to 5.5V. The sensor also has a maximum drain current of only 10μA. In addition, the sensor does not require any other external components. The output of the sensor is ideal for use with the PIC's A/D converter with a 2.5V reference. The output voltage varies inversely with increasing temperature, ranging from 2.5 V to 0 V. The maximum error in the output is ±3.8°C. The temperature can be calculated by:

$$T = -1481.96 + \sqrt{2.1962 \times 10^{6} + \frac{(1.8639 - V_O)}{3.88 \times 10^{-6}}}$$

http://www.national.com/ds/LM/LM19.pdf

Power

As with any other solar-powered system, power management was one of the most significant hardware design problems we had to face. The basic idea behind our design is to use a solar panel to trickle charge four 1.2V AA rechargeable batteries connected in series, which would in turn power the rest of our circuit. Since the four AA batteries in series have a nominal voltage of 4.8V, we would also require a voltage regulation stage that would bring down the supply voltage to the desired 3.3V. Also, we decided to provide the option of skipping the solar power stage completely and power the module using ground power through a simple wall transformer. Thus, wherever regular ground power is available, it could be used in place of solar power as a more reliable source.

Solar Panels

The solar panels we used in our design were 4.5"x3.0" and 4.5"x6.0" 6V weatherproof flexible solar panels from Silicon Solar Inc. The 4.5"x3.0" solar panel provides a maximum current of 50mA at 6V, and the 4.5"x6.0" provides a maximum

current of 100mA at the same voltage. We decided to use solar panels of different sizes and capacities since the nodes using the XBeePRO would have a higher demand for power than nodes using the XBee module. It should be noted that even though the XBeePRO requires more than five times more power for transmission than the XBee, the RF modules spend most of their "on" time in idle or receive modes, in which they both consume relatively the same amount of power (see section on XBee modules above). Consequently, a 2:1 ratio in the power provided by the two different solar panels would be more than sufficient to offset the difference in the power consumption of nodes using the two different RF modules.

Batteries

The main power source/storage unit for our system consists of four 1.2V AA NiMH Energizer batteries rated at 2500mAh connected in series. Thus, the nominal voltage of the power source is 4.8V and it can provide as much as 2.5A of current for 1 hour, or equivalently, 250mA for 10 hours (see discharge characteristic in Appendix A5). We decided to use NiMH rechargeable batteries as opposed to any other variety of rechargeable batteries because NiMH batteries are widely available and have relatively high capacities as compared to other types of rechargeable batteries. Also, NiMH batteries can be trickle-charged continuously at rates of up to 1/10 of their capacity without damaging the battery. Thus, we could charge our 2500mAh batteries with currents of up to 250mA without any problems. This greatly simplified our charging circuit, since we could simply connect the solar panels across the batteries, confident that the current they provide would not exceed 250mA even when using the large solar panels. The only problem with this configuration we had to be conscious of was that when the solar panel is not getting enough light, the batteries could discharge through it, since the voltage across the solar panel would be lower than the voltage across the batteries. To prevent this from happening, we simply added a diode that would let current flow only in the direction from the solar charger to the batteries (see Figure 10 below).

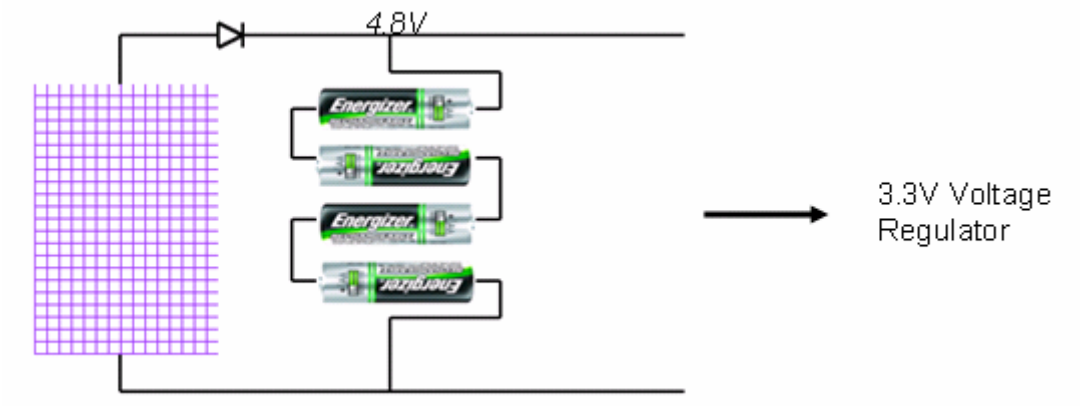
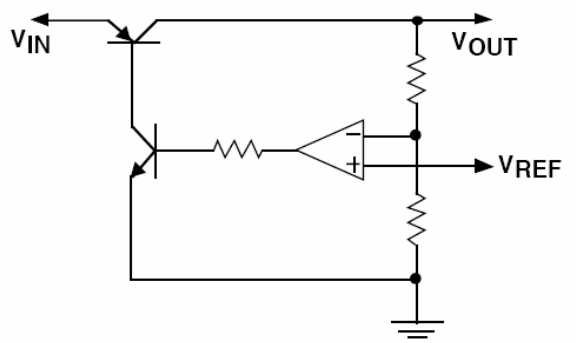


Figure 10. Solar battery charging circuit.

Voltage Regulation

After some research, we were able to establish that there are two generally accepted ways of stepping down voltage in battery powered applications, where efficiency is a primary concern. The first one is using a low-dropout linear voltage regulator, and the second one is using a buck switching regulator.

Linear Regulator



http://www.national.com/appinfo/power/files/f4.pdf

Figure 11. A simplified schematic of a LDO linear voltage regulator.

The linear regulator is by far the simpler of the two options. As can be seen from the simplified schematic above (Figure 11), the linear voltage regulator senses the voltage at its output and compares it to a set reference voltage. If there is a difference between the two voltages, the differential amplifier adjusts the current through the pass transistor as to compensate for the difference. Essentially, the excess voltage from the source is dissipated in the pass transistor, bringing the supply voltage down to the desired value, while maintaining the required load current. Thus, the efficiency of the linear regulator is equal to the output voltage over the input voltage, as seen in the equation below:

$$Efficiency = \frac{P_{out}}{P_{in}} = \frac{V_{out}}{V_{in}}$$

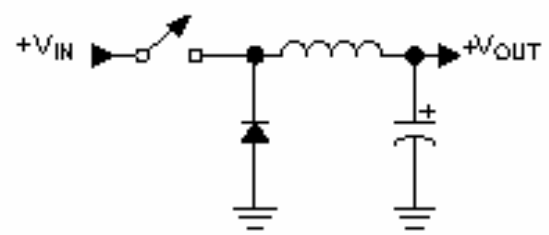
Substituting in our values of V_{in} = 4.8 V and V_{out} = 3.3 V, we get an efficiency of less than 70%. Obviously, in an application where power is scavenged from the sun in very limited amounts, 70% efficiency is not optimal.

It should be noted here that a low-dropout (LDO) linear regulator does have the potential of reaching higher efficiencies if the difference between the input voltage and the output voltage is not as great. For example, if we were to use 3 batteries instead of 4, then V_{in} = 3.6V and V_{out} = 3.3V, and the overall efficiency would be higher than 90%. However, in this case we would be losing a fourth of our power storage capacity. Of course, such high efficiency levels can only be achieved using a low-dropout (LDO) voltage regulator. The dropout voltage is defined as the smallest difference between the input voltage and the output voltage that ensures proper operation of the linear voltage regulator.

Finally, one of the great advantages of the linear regulator as compared to the switching regulator is that it can maintain a very steady power supply. As it turns out, while digital circuitry is not very sensitive to noise in the power supply, analog circuitry certainly is. Although our XBee modems are digitally interfaced, they depend on analog circuitry for RF communication. Thus, a steady power supply would be preferable for their proper operation.

Buck Switching Regulator

The second option for step-down voltage regulation involves using a buck switching regulator. A simplified topology of such a regulator is presented in Figure 12 below.



http://www.maxim-ic.com/appnotes.cfm/appnote_number/2031

Figure 12. Simplified schematic of a step-down buck switching regulator.

The controller for the buck switching regulator essentially sets the duty cycle of the switch that connects the input power supply to the inductor. A higher duty cycle corresponds to a higher current, and a lower duty cycle corresponds to a lower current. The inductor in the circuit maintains a relatively constant current through the regulator, while the capacitor maintains a relatively constant voltage at the output. In order to support a changing load, the controller senses the voltage at the output of the switching regulator and adjusts the duty cycle as to maintain the output voltage constant with respect to a reference voltage (3.3V in out case). Thus, if the load requires a higher current, it would essentially begin to bring down the output voltage of the regulator, which in turn would cause the controller to adjust the duty cycle, so that the output voltage would return back to its proper level. As a result, using a buck switching regulator to regulate supply voltages introduces noise in the power supply that results from the changing power demands of the load. In our particular application this would turn out to be somewhat problematic when using a switching regulator to power the XBeePRO, since it changes its power consumption almost five-fold when switching between transmit and receive modes.

In terms of efficiency, however, the switching regulator is by far the more efficient one of the two options. In fact, if working with ideal components, a buck switching regulator would theoretically have a 100% efficiency rating, since voltage is converted by storing energy in the inductor and capacitor, rather than by simply burning it out, as is the case with the linear regulator. Of course, no real circuit is ever composed of ideal components, so some losses should be expected in a real application of the switching regulator. Still, if the regulator is limited to operating at the low power levels we require, there are circuit techniques that allow a designer to overcome some of the

imperfections inherent in the design. As a result, a switching regulator used for our application could achieve efficiencies between 90% and 95%, which are practically independent of the ratio between the input and output voltages.

Final Design Decisions (for Voltage Regulation)

For our final design, we decided to include both a linear and a buck switching voltage regulator in our system. The linear voltage regulator we chose was the 3.3V LDO linear voltage regulator from STMicroelectronics's L4931 series. It has a relatively low typical dropout voltage of 0.4 V, and can operate with input voltages as high as 20V with output currents of up to 300mA (see datasheet in Appendix A8). We added the linear regulator so that it can be used to regulate the supply voltage when the sensor nodes are plugged into ground power via the wall transformer and efficiency is not a concern. In the case when the sensor nodes are powered through solar power, however, efficiency is a major issue. In this case, it is best to use a switching regulator instead. The one we used in our design is based on Texas Instruments' TPS6220x family of step-down, DC-DC converter controllers. In the range of input voltages and output currents our circuit requires, this step-down converter gives us efficiencies between 90% and 95% (see datasheet in Appendix A10).

Circuit Design

Computer Host

The block diagram below (Figure 13) shows a high level view of the Computer Host, which gathers and logs all of the sensory data collected by the network.

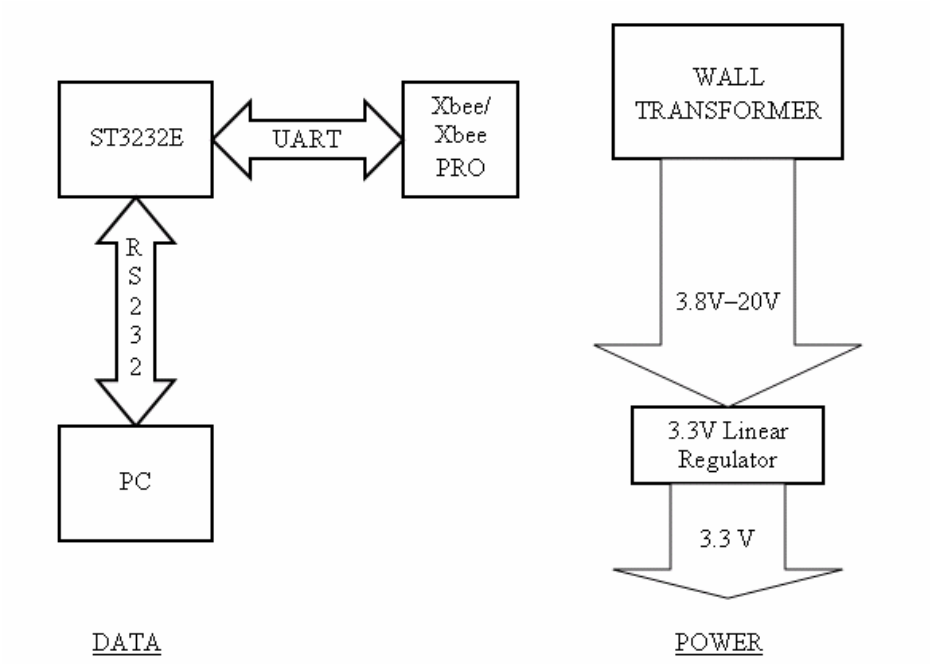


Figure 13. High-level block diagram of Computer Host.

The Computer Host interfaces with a PC through the serial port. Since the Computer Host must be physically connected to a PC, there is no need to power the circuit using batteries. As a result, the circuit is powered by a DC adaptor (wall transformer). Also, the Computer Host is operated by the PC, and does not require a microcontroller. Power from the DC adaptor is regulated by a linear voltage regulator. The ST3232E chip is a RS-232 receiver and transmitter. The device boosts the signals from the XBee to levels that can be interpreted by a PC. This is performed by a dual charge pump using four 0.1µF capacitors. An inventory list of the parts used to construct the Computer Host is shown in Table 3. The total price is an underestimation since some of the parts were already available to us by the college. The list also does not account for sockets, headers, jumpers, or the cost of the PCB. Figure 14 shows a picture of the Computer Host circuit with important features highlighted. Figure 15 is a circuit schematic of the Computer Host.

Table 3. Parts List for Computer Host

Quantity	*Part #*	*Description*	*Price ($)*
1	182-009-212-531	CONN DB9 FEM .318" RA MET SHELL	1.68
1	B3F-1000	SWITCH TACT 6MM MOM 100GF	0.20
5	Capacitor	100 nF	
1	Capacitor	10 µF	
1	DPD030040-P7P-TK	TRANSFORMER 3VDC 400MA P7 PLUG	5.43
1	L4931ABV33	ST Low Drop Voltage Regulators/Drivers TO-220AB 3.3V 0.25A Positive	0.70
1	LED		
1	PJ-007	CONN PWR JACK RT ANG 1.3MM I.D.	0.45
1	Resistor	200 Ω	
1	Resistor	10 kΩ	
1	XBP24-ACI-001	XBee24 PRO-2.4GHz-CHIP - 1mw -I	32.00
		TOTAL	40.46

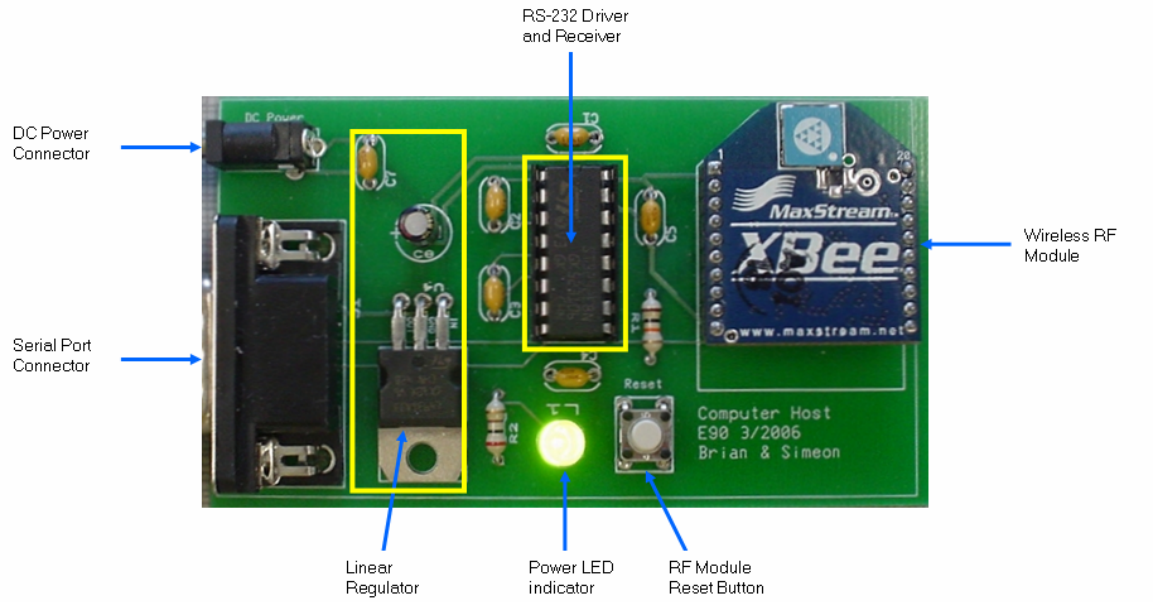


Figure 14. Picture of Computer Host circuit board.

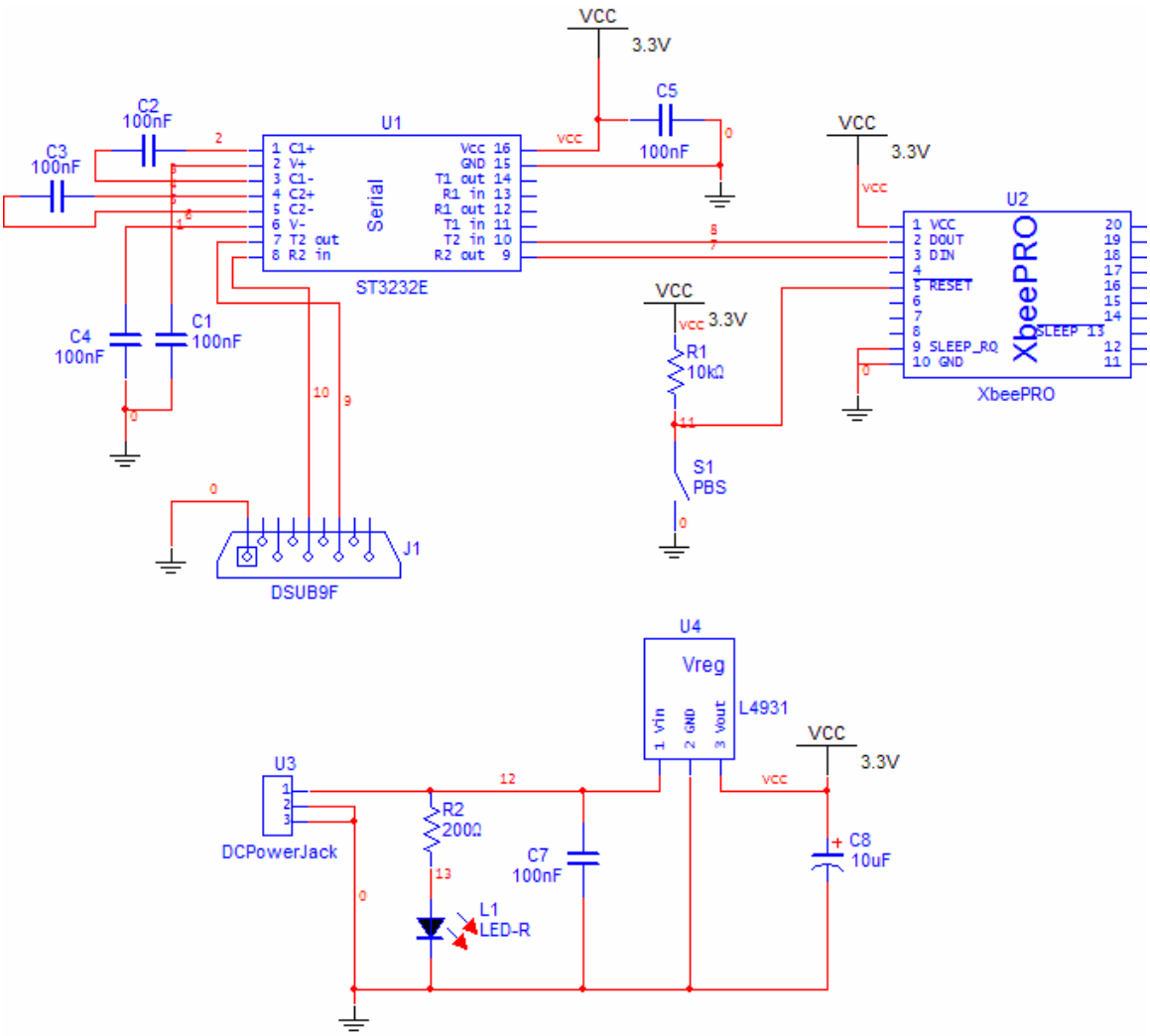


Figure 15. Circuit schematic for Computer Host.

Sensor Node

The Sensor Node gathers data from sensors and forwards the data to the host. The block diagram (Figure 16) below shows a high-level view of a single network node with two alternative power stages.

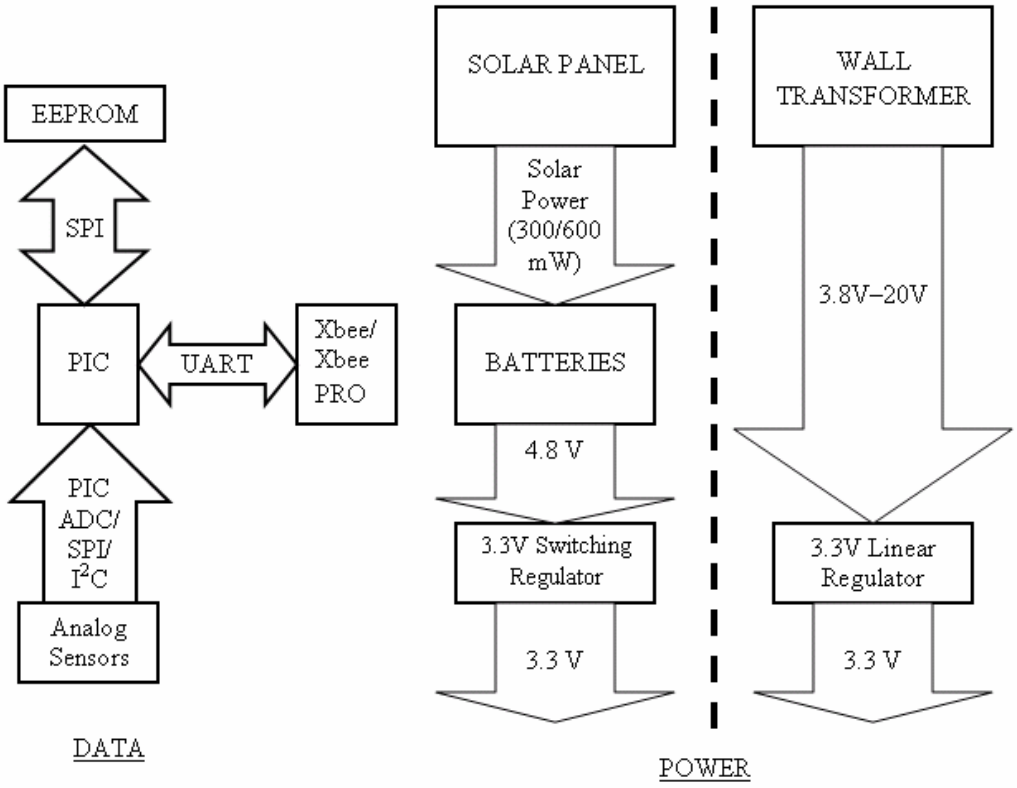


Figure 16. High-level block diagram of Sensor Node with alternative power stages.

The Sensor Node is operated by a PIC16LF876A microcontroller. The microcontroller runs off of a 4MHz external resonator. The PIC can be programmed through the ICD interface. Sensors can be conveniently attached to available pins on the PIC, which are located on a header at the bottom of the board (Figure 17). Up to 10 digital pins and a maximum of 4 analog pins are available as inputs (Table 4). The circuit can either be powered by batteries and solar panel or a DC adaptor (wall transformer). The power can be chosen to be regulated by either the linear regulator or switching regulator through settings of jumpers. For increased performance, the switching regulator should be used when the circuit is battery powered since the switching regulator has a higher efficiency compared to the linear regulator. The linear regulator should be used when the circuit is powered by a DC adaptor since the input voltage range of the linear regulator is higher than the switching regulator. Setting the jumpers for linear regulation also enables a power LED indicator. An inventory list of the parts used to construct the Sensor Node is shown in Table 5. The total price is an underestimation since some of the parts were already available to us by the college. The list also does not

account for sockets, headers, jumpers, or the cost of the PCB. Figure 17 shows a picture of the Sensor Node circuit with important features highlighted. Figure 18 is a circuit schematic of the Sensor Node.

Table 4. Available Pins on the PIC

Pin Name	*Description*
RA1/ AN1	Digital I/O. Analog input 1.
RA2/ AN2/ VREF-/ CVREF	Digital I/O. Analog input 2. A/D reference voltage (Low) input. Comparator VREF output.
RA4/ T0CKI/ C1OUT	Digital I/O – Open-drain when configured as output. Timer0 external clock input. Comparator 1 output.
RA5/ AN4/ SS/ C2OUT	Digital I/O. Analog input 4. SPI slave select input. Comparator 2 output.
RB1	Digital I/O.
RB5	Digital I/O.
RC2/ CCP1	Digital I/O. Capture1 input, Compare1 output, PWM1 output.
RC3/ SCK/ SCL	Digital I/O. Synchronous serial clock input/output for SPI mode. Synchronous serial clock input/output for I2C mode.
RC4/ SDI/ SDA	Digital I/O. SPI data in. I2C data I/O.
RC5/ SDO	Digital I/O. SPI data out.

Table 5. Parts List for Sensor Node			
Quantity	*Part #*	*Description*	*Price ($)*
1	05-1291	Flexible Solar Panels (6v Modules) 100mA 5.0”x6.0”	26.65
1	1N4001	DIODE GEN PURPOSE 50V 1A DO41	
1	25AA160A-I/P	IC SEEPROM 16K 1.8V 8DIP	0.99
1	B3F-1000	SWITCH TACT 6MM MOM 100GF	0.20
2	Capacitor	10 µF	
1	Capacitor	100 nF	
2	Capacitor	33 pF	
1	CDRH5D18-100NC	INDUCTOR 10UH SHIELDED SMD	0.92
1	Connector	RJ-12	
2	JMK316BJ106KL-T	CAP CER 10UF 6.3V X5R 1206	0.44
1	L4931CZ33	ST Low Drop Voltage Regulators/Drivers TO-92 3.3V 0.25A Positive	0.56
1	LED		
1	LM4040CIZ-2.5	IC VOLT REF PREC MICROPWR TO-92	1.15
4	NH15-2500	AA Rechargeable NiMH -1.2 Volts - 2500 mAh	9.99
1	PIC16LF876A-I/SP	Microchip PICmicro - PIC16LFxxx SPDIP-28 14KB 368 RAM 22 I/O	4.75
1	PJ-007	CONN PWR JACK RT ANG 1.3MM I.D.	0.45
4	Resistor	1 MΩ	
1	Resistor	15 kΩ	
1	Resistor	2.2 MΩ	
1	Resistor	200 Ω	
1	Resistor	3.0 MΩ	
1	Resonator	4.00 MHz	
1	SBH-341AS	HOLDER BATT W/COVR 4AA ON/OFF SW	1.08
1	SE3201-ND	C-001R 32.7680K-A	0.27
1	TPS62203DBVT	IC DC-DC CONV STEPDOWN SOT-23-5	2.00
1	XB24-ACI-001	XBee24 -2.4GHz-CHIP – 1mw -I	19.00
		TOTAL	68.45

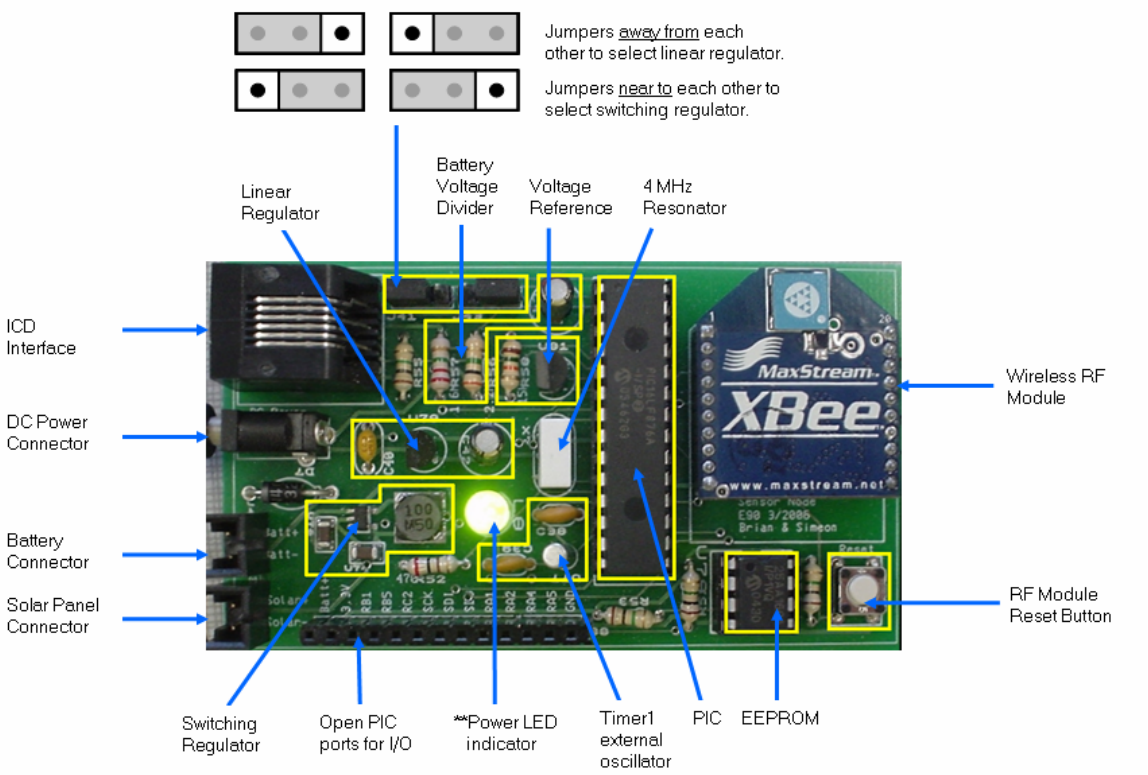


Figure 17. Picture of Sensor Node circuit board.

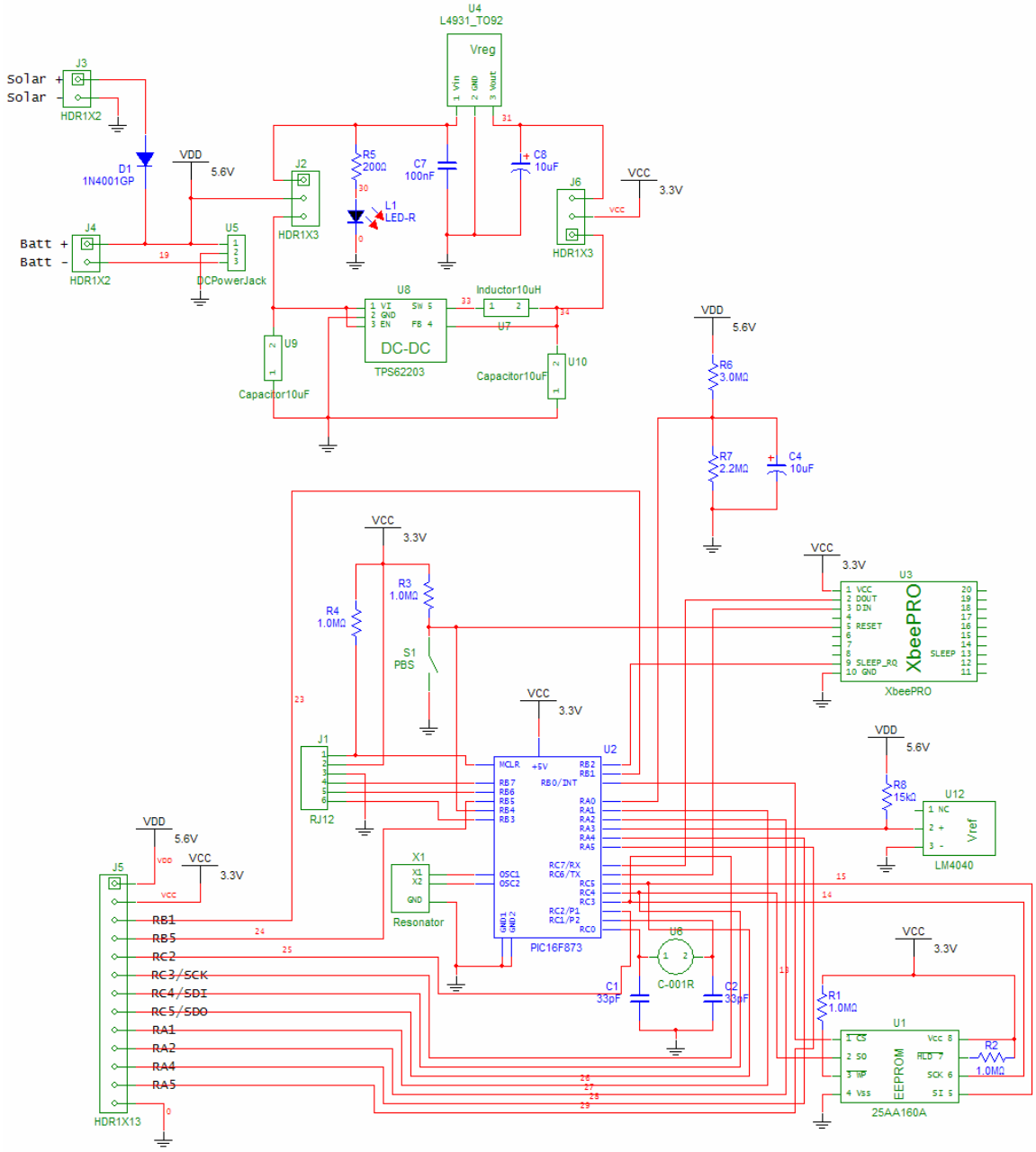


Figure 18. Circuit Schematic for Sensor Node.

SOFTWARE

The software portion of this project consisted of writing driver codes to interface all the peripheral devices with the microcontroller and the PC, designing and implementing a low-power networking protocol, and designing a computer host program that would gather all of the data forwarded to the PC host and save it to a web server in an easily accessible format. Each of these steps is described in the sections below.

Hardware Drivers and Interface

EEPROM Driver

The driver we wrote for interfacing the external EEPROM with the PIC microcontrollers through the SPI serial bus is contained in the file named "EEPROM_SPI.c" (see Appendix B4). We wrote this driver based on the functional description of the EEPROM chip's operation featured in its datasheet (see Appendix A7). We used a driver file for a different family of EEPROM chips that comes with the PIC C compiler as a template for writing our own driver code. Although we did not end up using much of the actual code, we preserved the general structure of the driver, which was still helpful.

The following functions were implemented:

- `void init_ext_eeprom()`: This function is used to initialize the EEPROM chip and configure the PIC's SPI port for proper communication with the chip. It needs to be called before the external EEPROM memory is accessed for the first time and every time after the SPI gets reconfigured for communication with a different peripheral device, such as a digitally interfaced sensor, for example.

- `void write_ext_eeprom(EEPROM_ADDRESS address, BYTE data)`: This function is used to write a single byte of data to a specified location in memory.

- `void write_page_ext_eeprom(EEPROM_ADDRESS address, int8* data)`: This function is used to write a whole page (16 bytes) of data starting at a specified location in memory. It should be noted here that `data` needs to be an array of exactly 16 bytes, and `address` needs to be an address at the beginning of a page on the EEPROM chip. Valid page addresses are all multiples of the page size (in this case 16 bytes), such as 0, 16, 32, …, 2032. If the page address passed to this function is not a multiple of 16, the whole page will not be written to memory.

- `BYTE read_ext_eeprom(EEPROM_ADDRESS address)`: This function is used to read a single byte of memory stored at the specified `address`. The data read is returned by the function.

- `void read_page_ext_eeprom(EEPROM_ADDRESS address, int8* data)`: This function is used to read an entire page from external EEPROM memory. The page read begins at the location specified by `address` and is stored in `data`, a 16-byte array, which needs to be passed by reference to this function. Unlike the `write_page_ext_eeprom` function, in this case `address` does not necessarily need to be a multiple of the page size. However, in our implementation it always is.

- `void erase_ext_EEPROM()`: This function erases the external EEPROM memory by overwriting all previous data with 0's.

All functions above leave the EEPROM in its standby mode, thus making sure that the external EEPROM chip operates in its low-power state when idle. Also, if a chip of different size is to be used, all that needs to be altered are the constants `EEPROM_PAGE_SIZE` and `EEPROM_NUM_PAGES`, which hold the page size and the number of pages, respectively. For more information on how the functions described above were implemented, please refer to the commented code in Appendix B4.

XBee Driver

We wrote a driver to interface with the XBee modules as well. It is contained in the file named "xbee.h" (see Appendix B3). This driver utilizes only a limited portion of the RF modem's functionality, to the extent that is required by our particular application. Each of these functions puts the XBee module in Command Mode before sending the appropriate commands, and exits Command Mode upon completion. Before exiting Command Mode, each of these functions sends a WR instruction to the XBee, which ensures that all of the changes in configuration are permanently stored to non-volatile memory. For more information on entering Command Mode, refer to the XBee Product Manual (Appendix A6).

The following functions were implemented:

- `void set_MY_add(int16 add)`: This function is used to set the MY address for the node. The MY address is the 16-bit address used to address data to the node.

- `void set_dest_add(int16 add)`: This function is used to set the 16-bit destination address for the node.

- `void set_channel(int8 channel)`: This function is used to set the channel, on which this node is going to communicate. It is important that the source node and the destination node are configured to communicate on the same channel; otherwise, communication would be unsuccessful, even if all addresses are set properly. The allowable channels for communication differ slightly for the XBee and the XBeePRO modules, and can be found in the XBee product manual in Appendix A6.

- `void set_dest_add_and_ch(int16 add, int8 channel)`: This function is used to set the destination address and the transmission channel of a node simultaneously. The advantage of using this function in place of using `set_dest_add` and `set_channel` consecutively is that it is faster since Command Mode is entered only once.

- `void xbee_sleep()`: This function is used to put the XBee in a low-power Sleep Mode.

- `void xbee_wake_up()`: This function is used to wake-up the XBee when in Sleep Mode.

- `void xbee_init()`: This function is used to initialize the XBee by setting a number of parameters (see actual code for mode details). It should be called before the XBee is used for the first time. Also, the XBee needs to be out of Sleep Mode when this function is called, which implies that a call to `xbee_init` needs to be preceded by a call to `xbee_wake_up`. This is also true for the rest of the functions mentioned above. If the XBee is in Sleep Mode, then it does not respond to any commands.

The PIC interfaces with the XBee module through its UART serial port. The baud rate used for communication varies depending on the position of the node in the network as described in a later section of this report. The allowable baud rates for the nodes are 9600, 4800, 2400, and 1200. All four of these baud rates are supported by both the PIC and the XBee modules.

The XBee's baud rate needs to be set to the appropriate value prior to using any of the functions above. This can be done by placing the XBee module on the Computer Host and using Maxstream's proprietary software X-CTU to set the baud rate (see Figure 19). X-CTU can be downloaded from Maxsteam's website (www.maxstream.net).

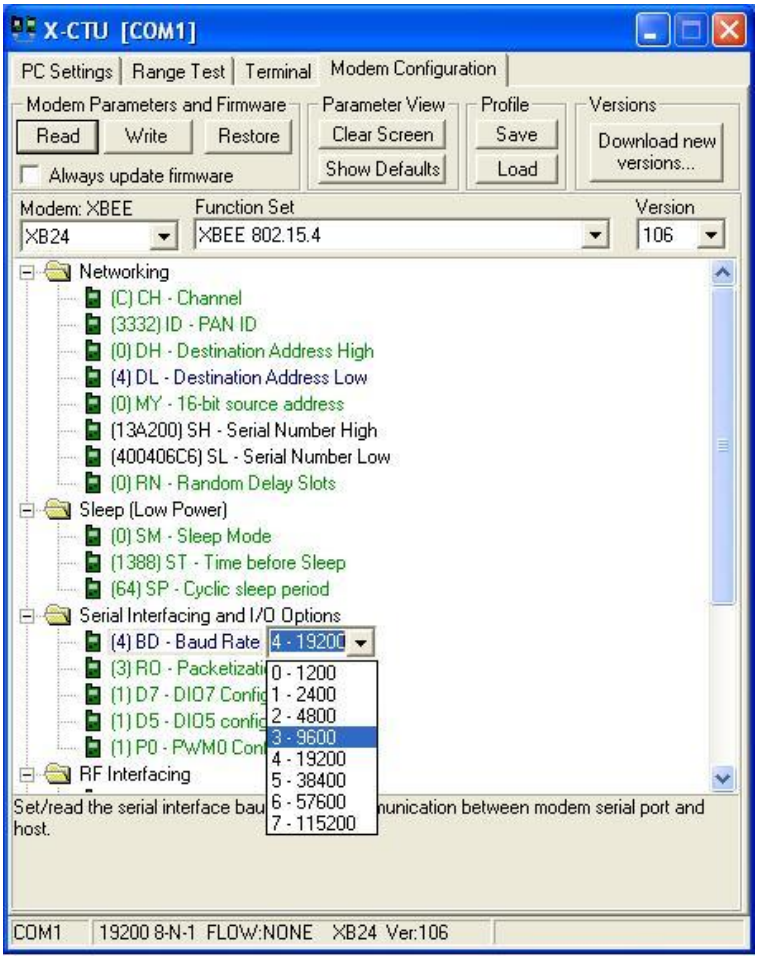


Figure 19. Setting the XBee's baud rate using X-CTU program by Maxstream.

For more information regarding the XBee driver, please refer to the commented code attached in Appendix B3. For more information on using X-CTU to configure the XBee modules, please refer to the XBee product manual attached in Appendix A6.

Network

For the networking part of our project we first had to choose a network topology, and then implement it as part of a specifically designed network protocol. The network topology needed to be suited for the particular needs of this project (i.e. collecting data at a centralized node) and the network protocol needed to be designed as to take advantage of the low-power Sleep Mode of the XBee modules.

Network Topology

After careful examination of existing wireless network topologies, we decided that a tree network topology would be ideally suited for the purposes of a sensory data-gathering wireless network. In a tree topology, all of the data gets propagated up to one main root node, which in our case represents the Computer Host. The Computer Host gathers all of the data and logs it into files stored on a web server to be analyzed at a later

time. The actual network consists of parent and child nodes. A child node is connected to a parent node and sends all of its data to its parent. A parent node is connected to a number of children, as well as its own parent node. Its purpose is to propagate its own data up the network, as well as forward all of the data gathered by its children. The Computer Host serves as the top level parent, gathering the data from all nodes. An example network configuration is shown in the Figure 20 below.

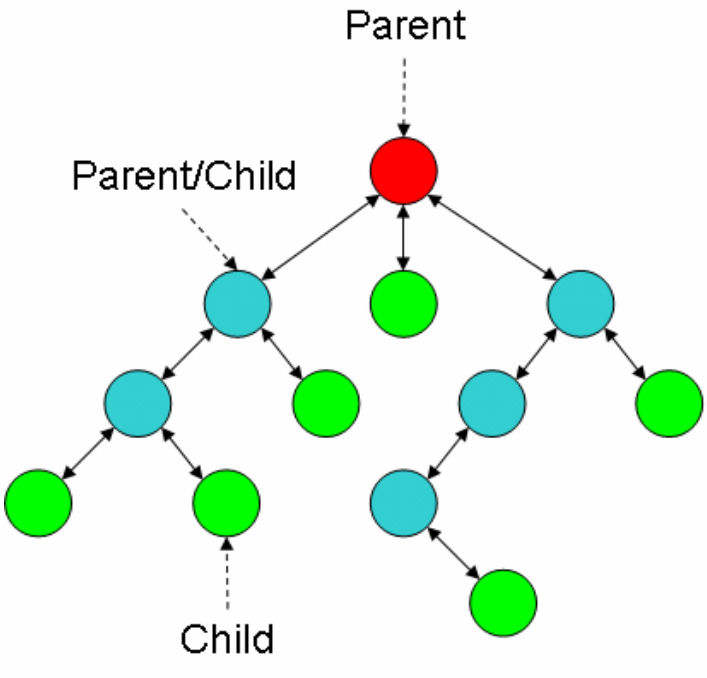


Figure 20. Example of a tree network topology.

It should be noted here that while a tree topology lends itself naturally to a network whose aim is to propagate data to a central node, it suffers from one main disadvantage. If a parent loses power and is unable to sustain wireless transmission, all of its children lose their connection to the PC host. As a result, when designing our wireless protocol, and in particular, when determining the ratio of wireless transmission to sleep time, we had to be extremely conservative as to make such a scenario as unlikely as possible. In turn, this reduced the throughput of our network. Naturally, we still allowed for the possibility of a parent node losing power and devised ways to handle such a scenario with minimal loss of information. For more details, please refer to the following section describing the RTS wireless networking protocol.

RTS (Receive-Transmit-Sleep) Network Protocol

The RTS protocol was developed especially for the purposes of this project. It allows for synchronized communication between wireless network nodes with reduced average power consumption. Each node goes through three cycles: a Receive Cycle (R), during which the node receives data from one of its children, a Transmit Cycle (T), during which the node transmits either its own data or that of one of its children to its parent, and a Sleep Cycle (S), during which the node turns off its RF module and logs

data from the sensors. The development of this protocol was necessary due to the fact that even though the XBee wireless modems require relatively small amounts of power for transmission and reception, their continuous operation in receive/transmit mode cannot possibly be sustained by the power supplied by our batteries and solar panels.

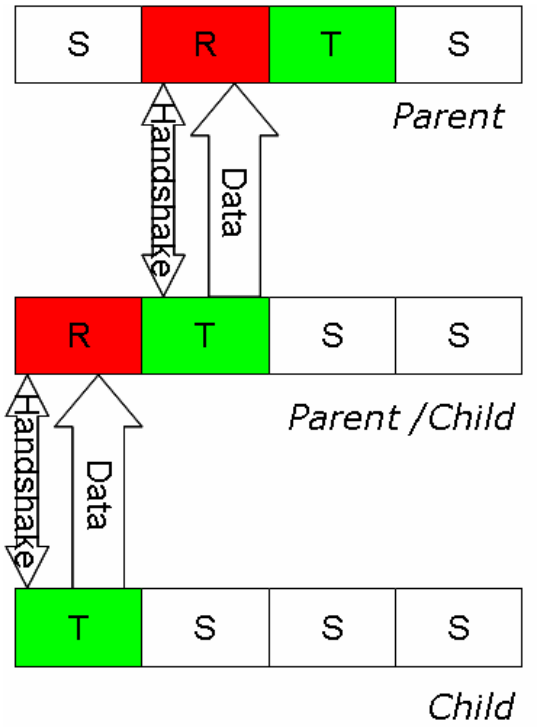


Figure 21. Propagation of data in the network.

In order to maintain reliable network operation, the Receive and Transmit Cycles of each node have to be synchronized in such a way as to allow data to propagate up the network (see Figure 21). Essentially, this means that we had to make sure the receive cycle of a parent node coincides perfectly with the transmit cycle of any one of its children nodes. Fortunately, this was easily accomplishable with the help of the real-time clock we implemented on our PIC microcontrollers. It allows us to set precise windows for reception and transmission of data, as well as define pre-calculated set periods of sleep for each node.

Data Packets

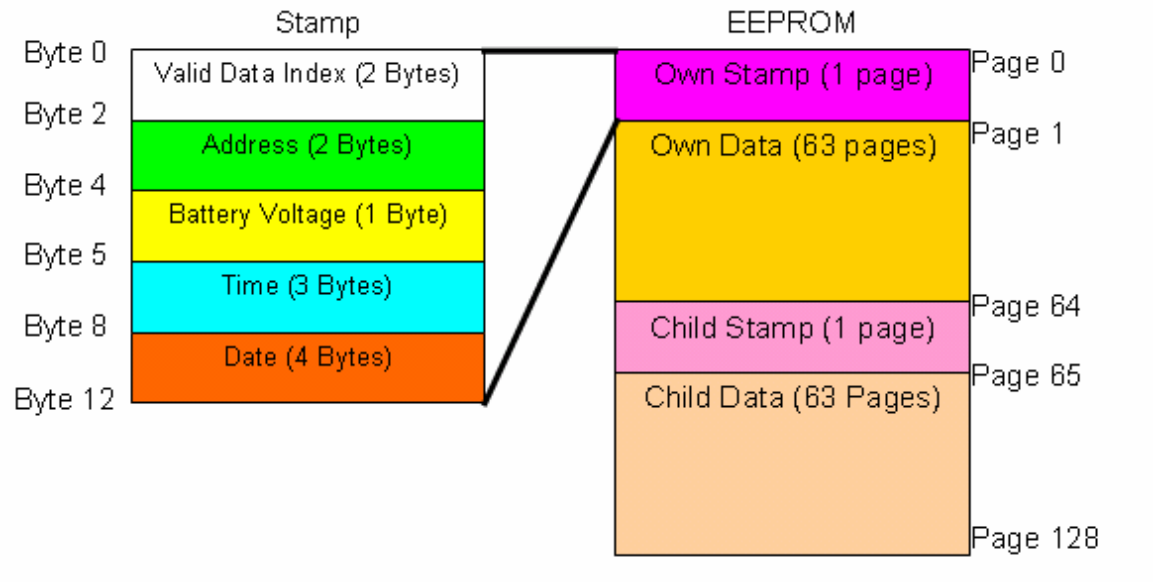


Figure 22. External EEPROM data structure.

As shown in Figure 22 above, external memory is divided into two segments of equal size (1024 bytes each in the case of a 16Kbit EEPROM). Each segment contains a header, called a stamp, which is 16 bytes long (1 page) and starts at the beginning of the segment. Bytes 0-1 represent the index of the last valid data entry. This index is necessary, since all 1024 bytes of data are transmitted each time, and very often not all 1024 bytes of data contain useful information. Bytes 2-3 represent the network address of the node which gathered the data contained in the segment. Byte 5 holds an 8-bit value, which represents the battery voltage at the time the data was gathered. Bytes 6-8 contain the time when the node started gathering data in the format hh:mm:ss. Finally, bytes 9-12 contain the date when data was gathered in the format mm:dd:yyyy. The remaining four bytes of data in the first page of a segment are left blank and can be used to store any other pertinent information regarding the node. For more information on the stamp, please refer to the comments related to the function `stamp_EEPROM`. [2]

The first of the two segments contains all of the data gathered by the node. The second segment is used for temporary storage of data coming from a child node; this data will be forwarded up the network until it reaches the PC host. A whole segment of length 1024 bytes is sent during each RTS cycle regardless of whether enough data was actually gathered to fill up the entire segment.

Adding Nodes to the Network

Before a node can be added to the network, the following constants contained in the main node file, "node.c" (Appendix B1), need to be set accordingly:

[2] All functions referred to in the *RTS Network Protocol* Section can be found in the file "rts.h" attached in Appendix B2

- `MY_ADDRESS:` This constant contains the address of the node. It is essential that this address is unique to this node, so track of all addresses currently in use in the network needs to be kept.

- `NUM_CHILDREN:` This constant contains the number of children this node will be able to accommodate. These are the parent slots. Each parent can have up to 12 slots (0-11). The limitation in the number of slots comes from the limited number of initialization channels that can be used by both the XBee and the XBeePRO. The initialization channels are the channels on which new children request to be added to the network. These channels (0x0D – 0x18) need to be different from the regular transmission channel (0x0C), so that attempts to add oneself to the network do not disturb regular network operation.

- `PARENT_ADDRESS:` This constant contains the address of the parent node that the node is trying to attach itself to. As a result, careful track needs to be kept of all parents in the network and the slots they have available.

- `PARENT_SLOT:` This constant contains the slot of the parent the node is trying to use for communication. The parent slot should be unoccupied at the time the node attempts to join the network. Slots are counted from 0 up, so the first slot is slot 0, the second one is slot 1 and so on.

- `BAUD_RATE:` This constant contains the baud rate which the node is going to use to communicate with the XBee module. The baud rate of a child has to be set to half the baud rate of the parent node. This is done in order to avoid data loss in transmission between parent and child.

Once all of the constants have been set properly, the node can proceed to add itself to the network. As soon as a new node gets turned on, it goes into an initialization transmit cycle.

Initialization Transmit Cycle

```
int8 init_handshake_req()
void T_cycle_init()
```

During an Initialization Transmit Cycle (T_init Cycle), the node continuously calls the `init_handshake_req` function, in which it requests a handshake using its network address as a token. Enough delay is introduced between the handshake requests as to allow the parent, once it picks up the handshake request, to change its destination address accordingly and respond to the new child. The parent responds to the new child by giving it a sleep time, which ensures that the child will transmit data only when the parent is expecting to receive it. The parent also sends the child the time and date, so that the child can set its real-time clock appropriately. Finally, the child waits until the parent signals the end of a time slice, at which point both parent and child go into their Sleep Cycles with synchronized timers.

Initialization Receive Cycle

```
int8 init_handshake_resp(int16 sleep_time, int16 i)
int8 R_cycle_init(int16 i)
```

A parent goes into an Initialization Receive Cycle for each one of its free slots. The number of the free slot determines the initialization channel the parent will listen on. The parent continuously calls the `init_handshake_resp` function, which will respond to an initialization handshake request by a new child. If this happens, the parent sends the appropriate handshake information to the child as described above and then waits for the time slice to end. The sleep time for each child is determined by the following formula:

$$SleepTime = (NumberOfChildren + 1) * (ThisNodeSleepTime + 2) - 2$$

By setting the child's sleep time according to this formula, the parent guarantees that the child will sleep for the appropriate amount of time and will be ready to transmit data exactly when the parent is ready to receive it. Once the time slice ends, the parent signals the child that the time slice has ended and goes into a Sleep Cycle. The node initialization procedure is illustrated in Figure 23 below.

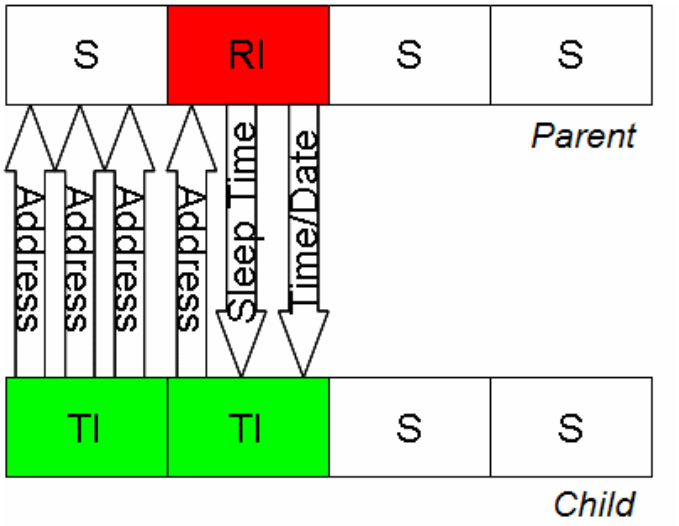


Figure 23. Initialization Transmit/Receive Cycle.

Data Transmission

```
int8 handshake_req()
int8 transmit_page(int16 address)
int8 handshake_resp(int16 i)
int8 receive_page(int16 address)
```

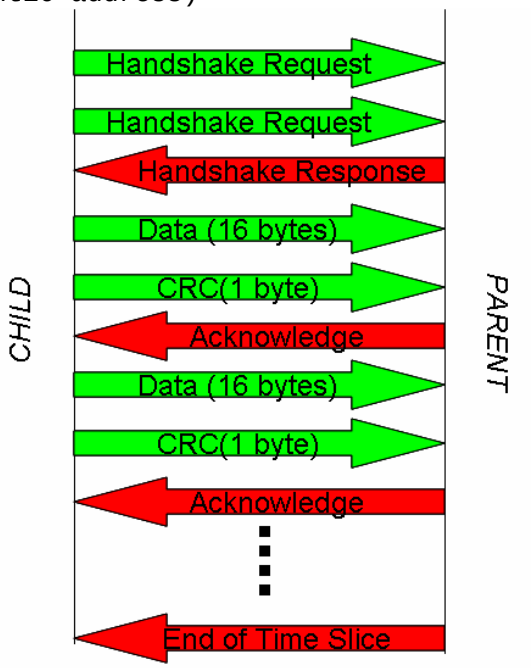


Figure 24. Data transmission between child and parent.

Figure 24 above shows how data is transferred between a parent node and a child node. The child begins the process by sending a handshake request to the parent (`handshake_req` function). The request is repeated at set intervals until the parent responds (`handshake_resp` function) or the data transmission window runs out. The data transmission window is determined by the `TIME_SLICE`, which in our case was set to 12 seconds. Once the parent responds to the handshake, the child begins data transmission by sending the first 16 bytes (1 page) of its data segment, followed by an 8-bit CRC (Cyclic Redundancy Check) checksum (`transmit_page` function). If the CRC byte the parent receives corresponds to the CRC byte calculated based on the 16 bytes of data it received, then the parent sends a positive acknowledge signal, which prompts the child to send the next page of data (`receive_page` function). If the received and calculated CRC bytes do not correspond, however, this implies an error occurred during transmission, so the parent sends a negative acknowledge signal to the child asking it to resend the data. Once all 64 pages of the child's data segment have been successfully transmitted, the child waits for the parent to signal the end of the time slice. This is done in order to resynchronize the child and parent clocks and avoid misalignment caused by slight discrepancies between the clock rates of the two nodes.

Calculating an 8-bit CRC Checksum

```
int8 CRC_calc(int8* page)
```

For our 8-bit CRC checksum calculation we used a table of 256 pre-calculated CRC entries (stored in the PIC's read-only memory) based on the polynomial $x^8+x^5+x^4$. This polynomial has been proven effective in determining a wide range of errors in transmission. The algorithm for calculating CRC checksums based on the entries in this table is the following:

1. Set the checksum equal to zero
2. Do a bitwise exclusive OR (XOR) operation between the first byte and the checksum
3. Use the result to index into the table
4. Set the checksum equal to the value returned from the table
5. Repeat steps 2 – 4 for each byte in the 16-byte sequence
6. The value of the checksum after the 16^{th} byte is the CRC byte

The CRC data table as well as the algorithm for calculating the CRC are described in much more detail in Appendix A12, which contains the reference material we used as a guide to implementing our CRC calculation.

Receive Cycle

```
int8 R_cycle(int16 i)
```

During a receive cycle the node waits for one of its children to initiate a handshake on the regular transmission channel. If the handshake terminates successfully, the node receives data from its child and writes it to the bottom half of its external EEPROM. When the cycle is over, the `R_cycle` function returns 0 to indicate success. The data is received in the manner specified by our data transmission algorithm described above. If the handshake doesn't terminate successfully, or if not all of the data is received successfully, the `R_cycle` function returns 1 to indicate failure. In this case, the parent does not attempt to forward any data up the network, since no valid data was received.

Transmit Cycle

```
int8 T_cycle(int16 D_address)
```

During a transmit cycle the node attempts to initiate a handshake with its parent. If the handshake is successful, the node either transmits its own data up to the parent or it forwards the data just received from one of its children. Again, transmission is done according to the data transmission algorithm described above. `T_cycle` returns 0 if the transmission is successful and it returns 1 if the handshake is unsuccessful or not all 64 pages are transmitted successfully. If `T_cycle` fails, the child node realizes that it has lost connection to its parent. In this case, it sleeps until it reaches the exact same part of

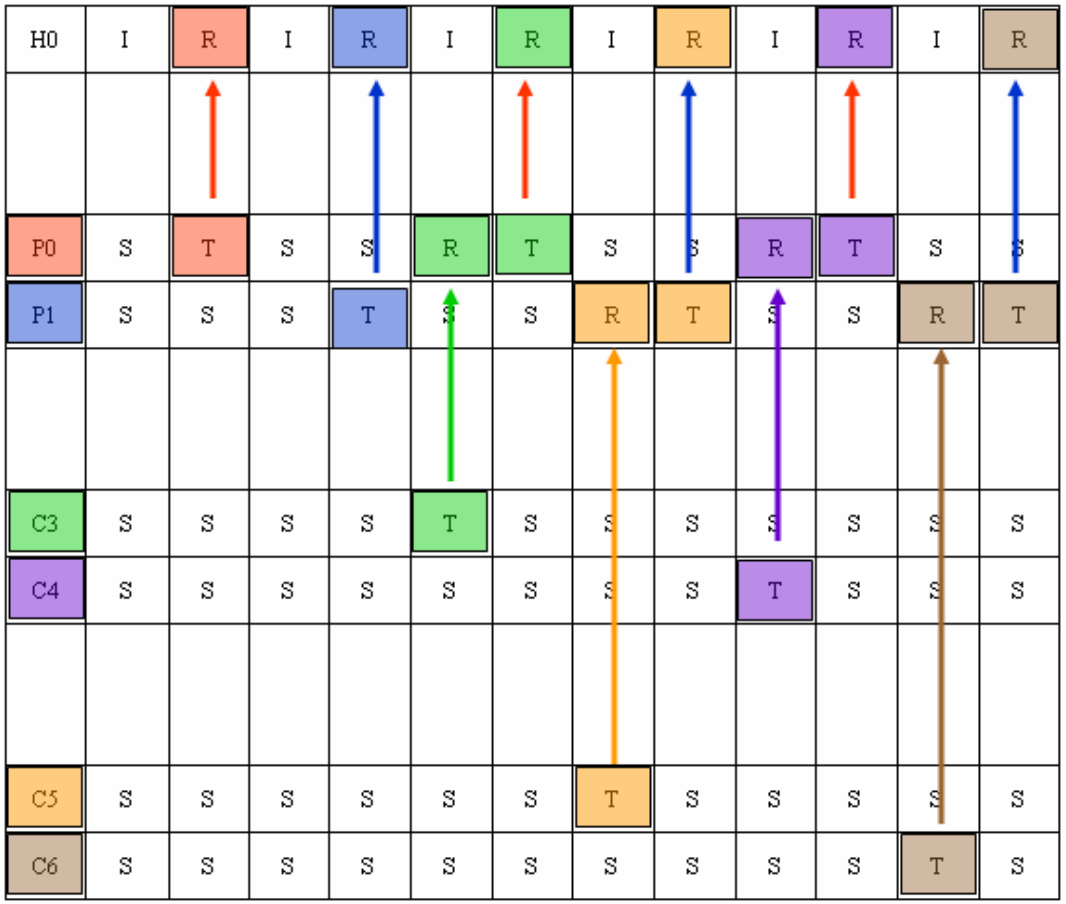
its full cycle where it realized it had lost the connection to the parent. At this point it wakes up and tries to retransmit the data it was unable to transmit previously. This can be either its own data, or a data that needs to be forwarded from a child. For more information on when this scenario might occur, please refer to section on failure mode below.

Sleep Cycle

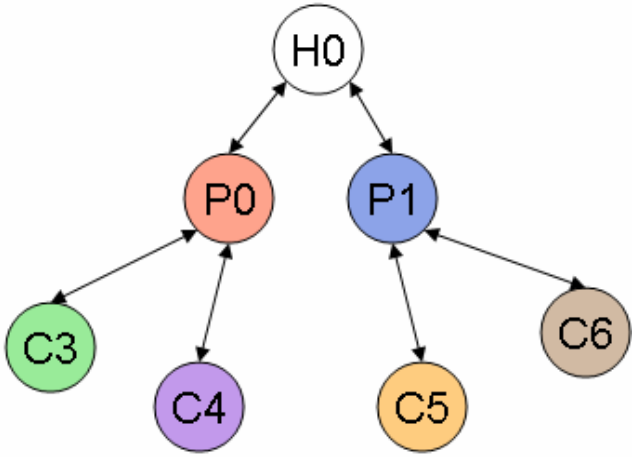
```
void S_cycle(int16 sleep_time)
void delay_RTS(int32 count, int16 time_out)
```

During the sleep cycle, the node puts the XBee module to sleep and goes into low-power mode operation. Then it begins gathering data from the sensors in set time intervals. The delay in-between gathering data points is set by defining the `LOOP_DELAY` constant. The `LOOP_DELAY` constant holds the delay between data points in hundreds of microseconds and can go up to 2^{32}, which corresponds to about 119 hours. It is essential that no additional delays are added to the `get_data` function, since they might offset the precise timing of the RTS cycles. The `LOOP_DELAY` constant is used with the function `delay_RTS`, which delays for the set period of time determined by `LOOP_DELAY`, but also times out when the Sleep Cycle is over.

An example timing diagram of a functioning RTS network is shown on the next page (Figure 25). Each square represents a time slice of 12 seconds. The color coding is used to help trace the data packets. The arrows are used to show how data propagates up the network. The color of an arrow indicates the node that sent the data packet. Note that instead of having sleep cycles the PC host, H0, has idle cycles, since it is connected to an unlimited power supply and it does not need to put the XBee to sleep.



Timing Diagram



Tree Structure

Figure 25. Example time diagraming of the RTS protocol.

Network Throughput

One of the biggest inherent tradeoffs of the RTS protocol is that the data throughput for each node is dependent on its position in the network. The throughput of each node is a function of its parent's sleep time, the number of the parent's children, and the number of the node's children. It is given by the following formula:

$$DataRate = \frac{1024}{(NumberOfChildren + 1) * (SleepTime + 2) * TimeSlice}$$

The origins of this formula can clearly be seen in the timing diagram on the previous page (Figure 25). The longer the sleep time of the parent, the longer a node has to wait before the parent enters a receive cycle and accepts data transmission from the child. Also, the more children a node has, the longer it has to wait before it can transmit its own data. Finally, the more siblings a node has, the longer it has to wait for its turn to communicate with the parent. As a result, only the nodes at Level 1 can operate at maximum capacity, and all other nodes have to operate at a lower capacity. For a time slice of 12 seconds and a data packet size of 1024 bytes, the throughput for a Level 1 parent with two children is approximately 3.5 bytes/sec.

In the actual implementation of our network the ratio of sleep time to receive/transmit time for Level 1 is 3:1 (R:T:S = 1:1:6), which is as low as it can be if proper operation of the nodes is to be guaranteed with a reasonable certainty. With this ratio an XBee node on Level 1 consumes approximately 60mW on average and an XBeePRO node on Level 1 consumes approximately 80mW on average in one day. Assuming that about 5 hours of sunlight shines on a regular day, the small solar panel supplies about 62.5mW on average and the large solar panel supplies about 125mW on average in one day. Thus, according to these calculations, an XBee node powered by a small solar panel and an XBeePRO node powered by a large solar panel should be able to sustain infinite operation. Of course, this is only valid if the assumption of 5 hours of sunshine a day is valid. Still, since the nodes can operate up to almost 7 days without any solar power, the 5 hours of sunshine a day needs to average over one week. A lower sleep time to receive/transmit time ratio would not be able to sustain operation for prolonged periods of time under these assumptions, since the nodes would not be able to scavenge enough solar power to maintain operation.

Since the nodes on Level 2 have no children in the example above, their throughput is essentially the same as that of their parents. If a node on Level 2 had 2 children, however, then its average throughput would be 3 times as low, or approximately 1.2 bytes/sec, because in addition to transmitting its own data, it also has to forward the data coming from both of its children.

Failure Modes

```
void failure_mode()
```

If the battery voltage of a node goes below a certain critical value set by the constant `FAIL_LOW`, then it goes into a low-power mode, where it essentially cuts off

communication with all other nodes, and replaces all of its cycles with low-power Sleep Cycles. This means that the node continues logging data from the sensors, but does not send any data up the network. The node maintains this mode of operation until the battery voltage goes above a second threshold value, `FAIL_HIGH`, when it goes back to its normal mode of operation. `FAIL_LOW` and `FAIL_HIGH` are set so there is a difference between them which assures us that once the node goes back into normal mode of operation, it will remain in normal mode of operation for some time, rather than go back immediately into low-power mode as soon as it transmits once. It should be noted that while in low-power mode, the node continues to keep track of its RTS schedule, so that once it is ready to exit low-power mode, it can get back to its normal schedule for communicating with its children.

If a parent node goes into low-power mode of operation, its children will be unable to forward data up the network. Consequently, they need to be able to handle `T_cycle` failure gracefully. If a call to `T_cycle` results in a non-zero return value, a node instantly goes to sleep for a period of time given by:

$$SleepTime = (NumberOfChildren + 1) * (ThisNodeSleepTime + 2) - 1$$

This sleep time is 1 time slice bigger than the sleep time the node gives to each of its children. It is set in such a way that the node essentially skips through an entire rotation, and wakes up exactly in time to try to retransmit the data it was trying to transmit during the last cycle. This sequence is repeated until the node's parent finally comes back up and is ready to accept the data.

Notice that when a node loses the connection with its parent, it automatically stops accepting any connection requests from its children. As a result, the children lose communication with their parent as well. Thus, every node below the parent that went down goes into failure mode, where it continues gathering data and periodically checks to see if the connection with the parent is back up again. This scenario ensures that no data will be lost due to a parent node going into a low-power mode. Instead, the data gathered by nodes below a parent that goes temporarily down will only be delayed until the parent is ready to come back. Of course, this is only valid until the nodes that are in failure mode have enough free memory to keep logging their data. If a parent is down for prolonged periods of time, it is possible that its children will run out of room for logging data and data will be lost.

All of the functionality of the node as a part of the RTS network is contained in the main function of the file "node.c" attached in Appendix B1. Please refer to the commented code in this file to get a better understanding of how the different parts of the RTS protocol come together.

Computer Host Program

The Computer Host program communicates with a XBee module using the computer's serial port. Since most computers run on Microsoft Windows, the program was intended to be compatible with Windows machines. The host program was programmed using Microsoft Visual Basic. Visual Basic offers an easy GUI development environment for software applications. The serial port communications is

performed using the MSComm Control component. The MSComm communicates with the XBee module on COM1 port with the settings of 19200 baud, 8 data bits, No parity, and 1 stop bit. The program reads data from the XBee's buffer one byte at a time. The code below shows how the serial port can be opened using the settings mentioned above.

```
' use COM1 port
MSComm1.CommPort = 1
' 19200 baud, no parity, 8 data, and 1 stop bit
MSComm1.Settings = "19200,N,8,1"
' the control reads one character at a
' time from the hardware buffer
MSComm1.InputLen = 1
' open the serial port
MSComm1.PortOpen = True
```

The main purpose of the host program is to collect data from the network and save the data to a file. When data from a node is successfully collected, the data is appended to a text file. Data from a node is written on one line in a comma delimited format. This text file can be written to a web accessible location to make the data visible using a web browser. To do this, the entire filename and path must be inputted. If only a filename is specified, then the data will be saved to a file that is located in the same directory as the executable file.

When the program starts, it enters into Command Mode. Commands to the XBee can be sent in this mode. The program will start looking for children and logging data when a filename is entered and the "Save" button is pressed. At this stage, the program enters Saving Mode. Currently the program is setup to support only three children under the Host. The program can be recompiled to support any number of children under the host by setting `NUM_CHILDREN` appropriately in "global_variables.bas." When the "Stop" button is pressed, the program stops appending data to the file and enters Debugging Mode. At this point, the program keeps running as usual but no longer saves data; only commands can be sent to the XBee.

The host program conforms to the RTS network protocol described earlier. Much of the functions in the host program are the same as for the nodes (`R_cycle_init`, `R_cycle`). However, since the host is at the top of the network and is plugged into a power outlet, it does not need transmit or sleep cycles. Instead, the host simply alternates between receive and idle cycles. Consequently, the sleep times for children under the host are calculated differently compared to sleep times for children under nodes. The sleep time is calculated by:

$$\text{sleep time} = (\text{number of children} \times 2) - 2$$

The multiplication by two accounts for the receive and idle cycles for the host. The subtraction of two accounts for a receive cycle and transmit cycle for the child. For example, having 6 children under the host will result with a child having a sleep time of 10 cycles:

```
HOST      R I R I R I R I R I R I R
NODE    R T S S S S S S S S S S R T
```

See detailed comments included in the files in Appendix C for more information about the host program.

RESULTS AND TESTING

XBeePRO Test

After completing our final design, we noticed an alarmingly high transmission error rate when using sensor nodes with the XBeePRO powered through the switching regulator. Since we had anticipated the noise in the supply voltage might cause such problems, we set off to determine whether this was indeed the case. We powered an XBeePRO node from an independent, steady 3.3 V power supply and noticed that in this case there were no errors in the transmission. As a result, we could be sure that the problem was coming from our voltage regulation power stage. Since the XBee nodes did not exhibit similar behavior, we concluded that the five-fold increase in power consumption of the XBeePRO's during transmission caused the DC-DC converter to output significantly more noise in the supply voltage, and in turn affect the performance of the analog circuitry in the RF modules.

In order to overcome this problem, we decided to cascade the switching regulator with a 3.0 V LDO linear regulator. In this way, we could reduce the noise in the regulated supply voltage while maintaining a relatively high efficiency of about 86% (a 95% efficiency for the switching regulator followed by a 91% efficiency for the LDO linear regulator). Scaling down the supply voltage to 3.0 V was not a problem, since this supply voltage was within the operating range of all of our circuitry. The only difficulty we had was finding a 3.0 V linear voltage regulator with a low enough drop-out voltage that would allow it to operate with a supply voltage of 3.3 V. Fortunately, we were able to find such a regulator. The Si9181 Micropower 350-mA CMOS LDO Regulator from Vishay Siliconix has a typical drop-out voltage of 150 mV (see data sheet in Appendix A11).

We placed the new 3.0 V LDO voltage regulator in place of the old 3.3 V voltage regulator, and we used the jumpers to connect the batteries to the input of the switching regulator, the output of the switching regulator to the input of the 3.0 V LDO linear regulator, and the output of the 3.0 V LDO linear regulator to the supply voltage for our circuit (see Figure 26 below).

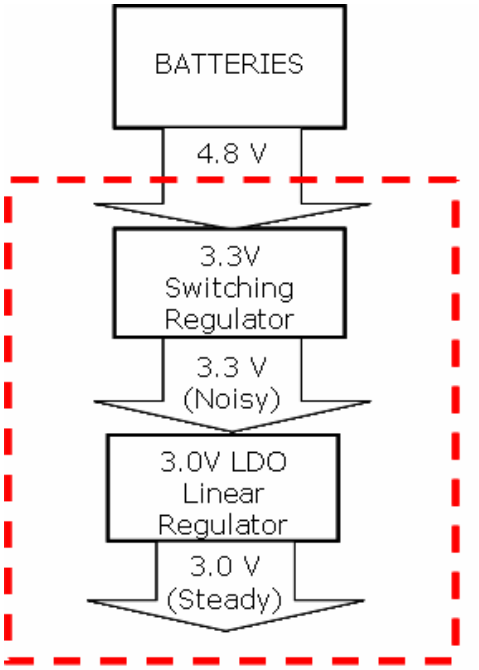


Figure 26. Cascading the 3.3 V switching regulator with a 3.0 V LDO linear regulator to reduce noise in the supply voltage while maintaining a high efficiency.

This successfully reduced the noise in the supply voltage, and in turn dramatically decreased the error rate of the XBeePRO nodes, as can be see in Figure 27 below. The error rate was decreased to the very acceptable level of approximately 1/10000. Since errors in transmission are detected and eliminated in our data transmission algorithm through the use of an 8-bit CRC checksum and re-transmission of invalid data, the XBeePRO nodes with this improved power stage could now reliably be used as part of our wireless network.

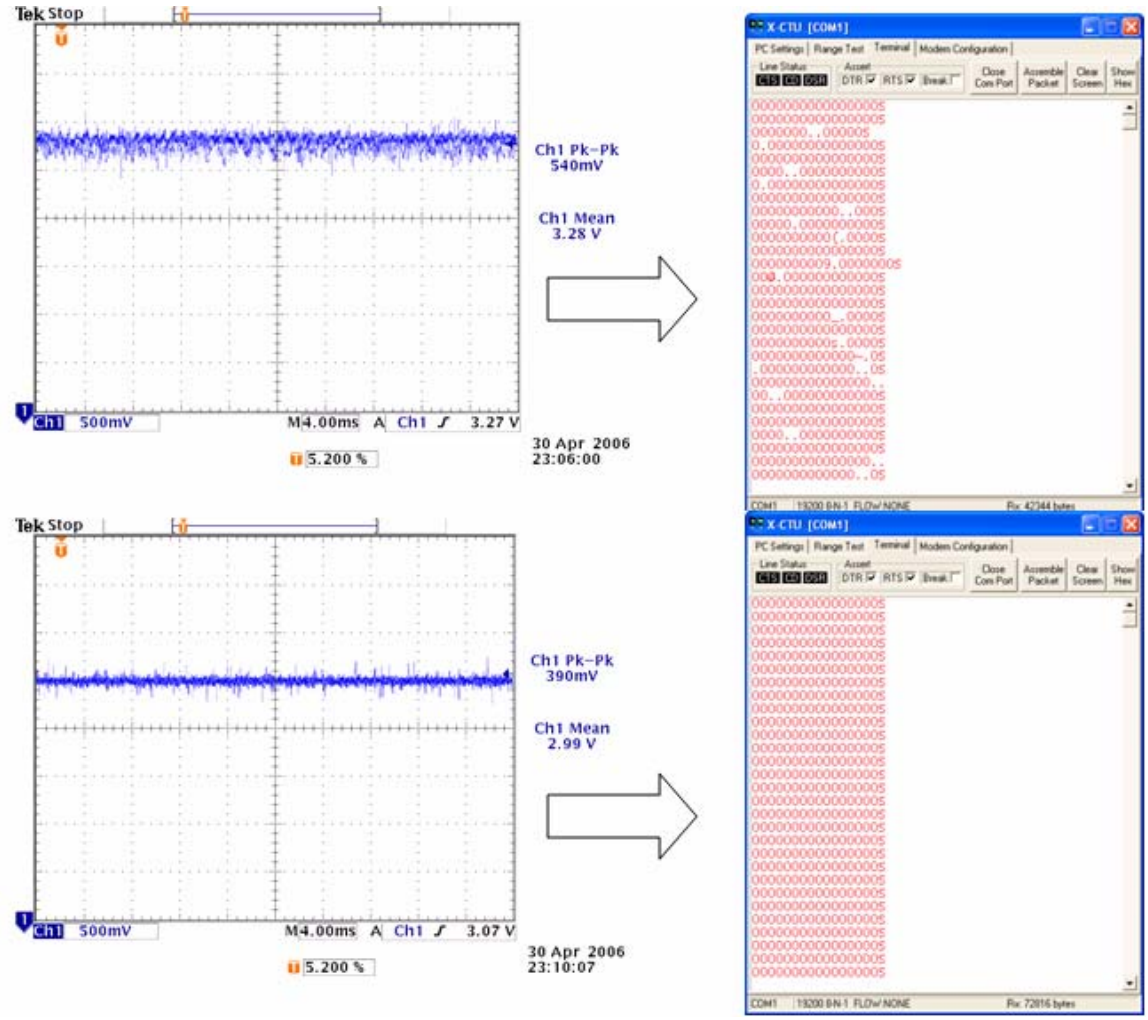


Figure 27. *Top:* Noisy supply voltage outputted by switching regulator results in numerous transmission errors. *Bottom:* Reduced noise in supply voltage by adding a 3.0 LDO linear regulator significantly reduces error rate.

Temperature Data Test

The network was tested using the LM19 temperature sensors. The temperature and battery voltages were analyzed over time. The network layout of the test is shown in Figure 28 along with placement of the nodes in Figure 29. The test was performed in Hicks Hall at Swarthmore College. The nodes were placed in Tupperware containers for protection from dust and debris.

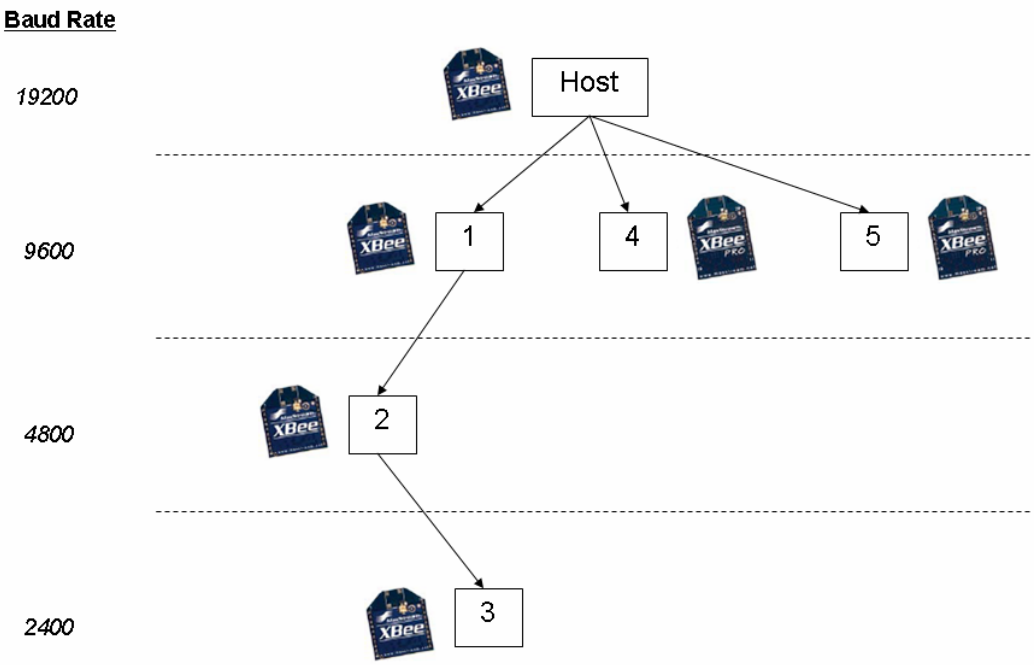
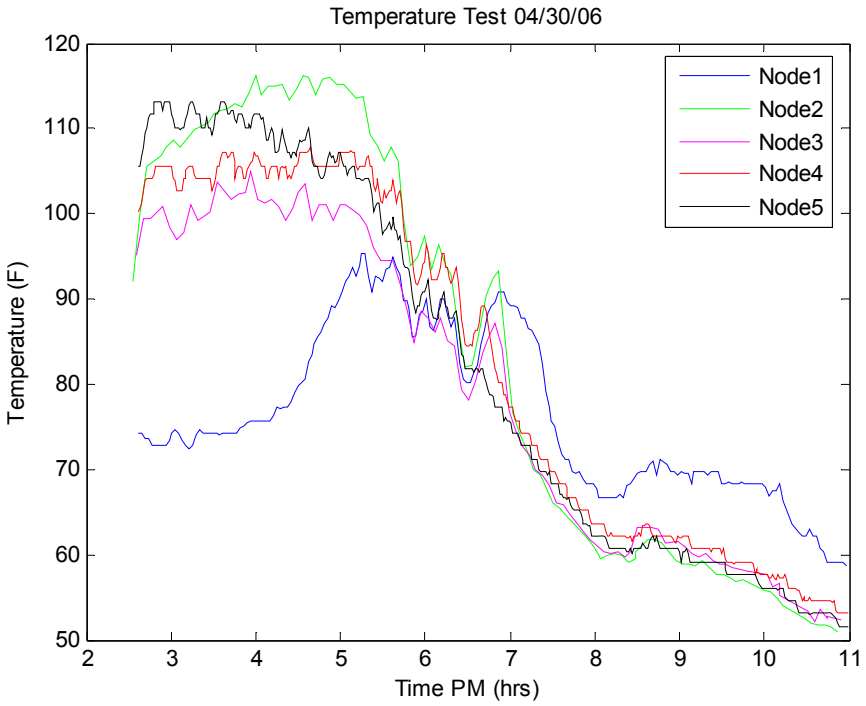


Figure 28. Temperature test network layout.

Figure 29. Temperature test node locations.

Graph 1 shows the variations of the temperature at each node over time. The high temperatures recorded between 3 and 6 PM for Node 2, 3, 4, 5 are due to the fact that the sensors were concealed in Tupperware containers and positioned in direct sunlight. The Tupperware essentially acts as an insulating layer, trapping heat inside of the container. The trend of Node 1 is different from the other nodes because it was placed outside of a

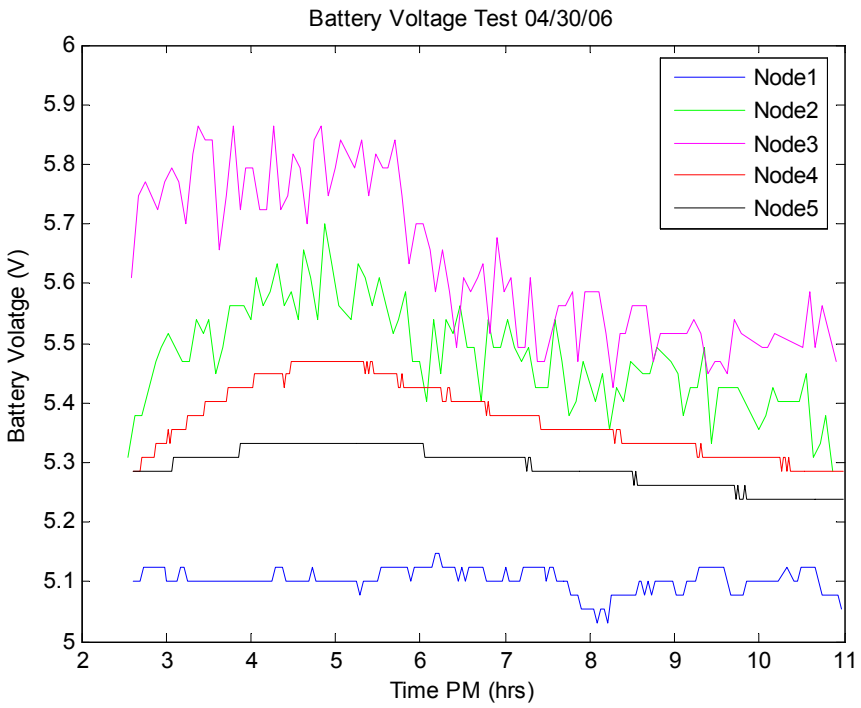
room window and was not exposed to direct sunlight. As the day turned into night, the temperatures of all of the nodes decreased as was expected.



Graph 1. Temperature vs. Time

The battery voltage trends also agree with our expectations, as shown in Graph 2. As a note, only changes, not absolute values, of the battery voltages can be discussed due to the lack of precision in the resistors used for the battery voltage divider. The voltage of Node 1 remains relatively constant since it was plugged into a power outlet. Both Node 4 and Node 5 were equipped with XBeePRO modules and placed on Level 1 with no children (Figure 28). Both of these nodes consume the same amount of power transmitting data. As can be seen in the graph, the battery voltage of Node 4 increases more than Node 5. This can be explained by the fact that Node 4 was equipped with a large solar panel (4.5”x6.0”) and Node 5 was equipped with a small solar panel (4.5”x3.0”).

The battery voltage of Node 2 increased the most, from 5.3V to 5.7V. Node 2 was equipped with a large solar panel and was placed on Level 2 of the tree network (Figure 28). Nodes on Level 2 spend less time transmitting data and consume less power compared to nodes on Level 1. As a result, Node 2 was able to store more of the energy it harvested from the Sun. Part of this explanation can also be attributable to the fact that Node 2 used an XBee module, whereas Node 4 and Node 5 used XBeePRO modules which consume more power. This same reasoning can be used to explain the battery voltage increase of Node 3.



Graph 2. Battery Voltage vs. Time

Power Efficiency

Some preliminary testing of the power stage indicated that the observed efficiencies were very close to the efficiencies stated by the manufactures. In particular, the efficiency of the switching regulator was shown to be close to 95%, and the efficiency of the linear regulators was shown to conform to the formula:

$$Efficiency = \frac{P_{out}}{P_{in}} = \frac{V_{out}}{V_{in}}.$$

The efficiency of the switching regulator cascaded with the 3.0 V LDO linear regulator was also close to the predicted value of 86%. However, all of the efficiency tests were preliminary and were based on a very limited data set. For more conclusive efficiency results, more data needs to be gathered.

Range

The preliminary tests of the range of the sensor nodes indicated that the ranges conformed to the specifications provided by the XBee manufacturer for the on-chip antenna modules. These ranges can be obtained from Table 2 in the section on RF communications. Again, for a more conclusive range test, more data needs to be gathered.

Failure Mode

Failure Mode has not been thoroughly tested due to time limitations of the project. The response of the sensor nodes to unsuccessful transmission cycles was observed to be according to our specifications. However, we were unable to test the full failure mode functionality, and in particular setting the constants `FAIL_LOW` and `FAIL_HIGH` to values that would ensure the desired failure mode behavior. Therefore, if failure mode is to be used, the contents of the `fail_mode` function contained in the file "rts.h" need to be uncommented and the constants associated with it have to be tested and refined by the user.

APPLICATION NOTES

Programming the Sensor Nodes

In order to program a sensor node, all the user needs to do is set the five constants defined at the beginning of the file "node.c" as described in the section on adding new nodes above. The only reserved address in a newly created network is the address 0, which is used by the computer host. It should also be kept in mind that the computer host has only 3 slots (0–2), so it can accommodate only up to 3 children on Level 1. For example, a node with an address of 1 that is set to use slot 0 of the host and have 2 children is defined in the following way:

```
//All pertinent node information that can be stored as constants
#define MY_ADDRESS 1 //this node's address
#define NUM_CHILDREN 2 //number of children for this node
#define PARENT_ADDRESS 0 //parent address
#define PARENT_SLOT 0 //the parent slot
#define BAUD_RATE 9600 //the baud rate for Level 1
```

Before a child can be added to Node 1, we must wait for Node 1 to get accepted by its parent (Computer Host 0). It is essential that no children nodes are trying to join Node 1 before it has become a part of the network, because their attempts to join will interfere with Node 1 joining the network. Essentially, this means that all nodes on Level 1 need to be added to the network successfully before nodes on Level 2 are turned on. The same goes for all other levels as well. The only sure way to know that a node has been successfully added to the network and is ready to accept children is the PC host receiving the first data packet from that node.

Consequently, if a child node is to be added to Node 1 from the example above, we first have to wait for the first data packet from Node 1 to arrive. Once this happens, we can program a second node with the following parameters:

```
//All pertinent node information that can be stored as constants
#define MY_ADDRESS 2 //this node's address
#define NUM_CHILDREN 1 //number of children for this node
#define PARENT_ADDRESS 1 //parent address
#define PARENT_SLOT 0 //the parent slot
#define BAUD_RATE 4800 //the baud rate for Level 2
```

Since Node 1 has two slots, we can also program a second child with the following parameters:

```
//All pertinent node information that can be stored as constants
#define MY_ADDRESS 3 //this node's address
#define NUM_CHILDREN 1 //number of children for this node
#define PARENT_ADDRESS 1 //parent address
#define PARENT_SLOT 1 //the parent slot
#define BAUD_RATE 4800 //the baud rate for Level 2
```

Notice that Node 2 and Node 3 are trying to add themselves to Node 1 using two different slots (Slot 0 for Node 2 and Slot 1 for Node 3). If they we both trying to add themselves using Slot 0, then only one would get added and the other one would remain in its Initialization Transmit Cycle forever. Thus, when setting up a network, it is crucial to make sure that each node is trying to add itself to a parent using an open slot. Also, notice that the baud rate for Level 2 goes down to 4800, since it needs to be half of that on the Level 1 above, which is 9600.

Once Node 2 and Node 3 have been programmed properly, they can both be turned on at the same time. One will not interfere with the other, since they are trying to communicate with Node 1 on different initialization channels.

As soon as data is received from Node 2, we can add a child to it as well. Here are the parameters for such a child:

```
//All pertinent node information that can be stored as constants
#define MY_ADDRESS 4 //this node's address
#define NUM_CHILDREN 0 //number of children for this node
#define PARENT_ADDRESS 2 //parent address
#define PARENT_SLOT 0 //the parent slot
#define BAUD_RATE 2400 //the baud rate for Level 3
```

Again, the baud rate for this node goes down to 2400, which is the appropriate baud rate for Level 3. This child adds itself using Slot 0, since this is the only available slot on Node 2 (Node 2 has only 1 slot). Also, this new child is set up to have 0 number of children, which means that no children can be added to it (it has no open slots).

The network that we end up with in this example is illustrated in Figure 30 below:

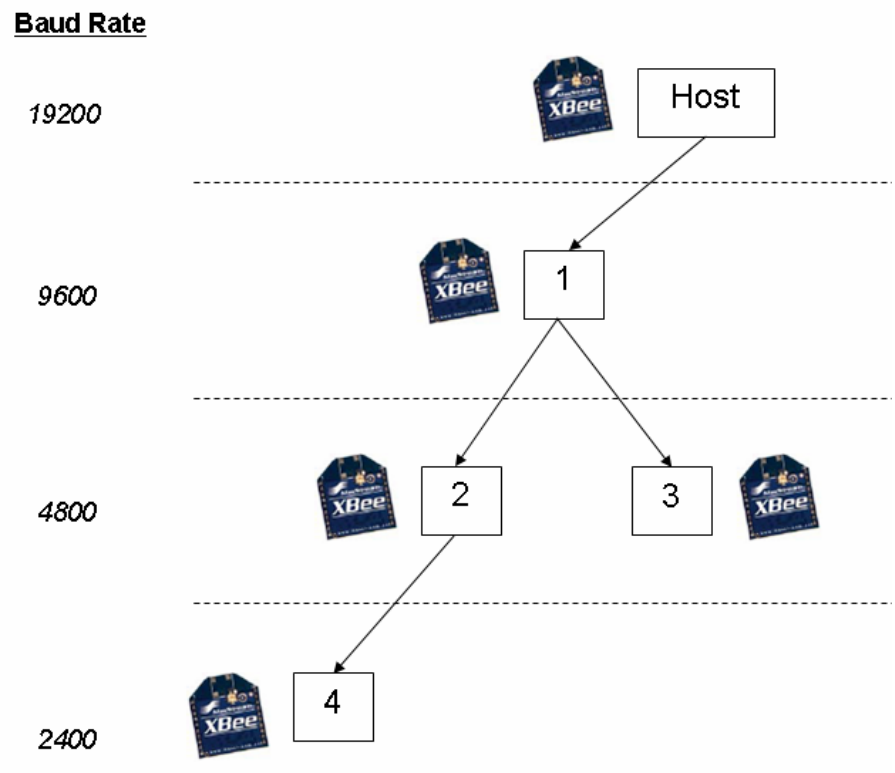


Figure 30. Network configuration resulting from example above.

Using the Computer Host Application

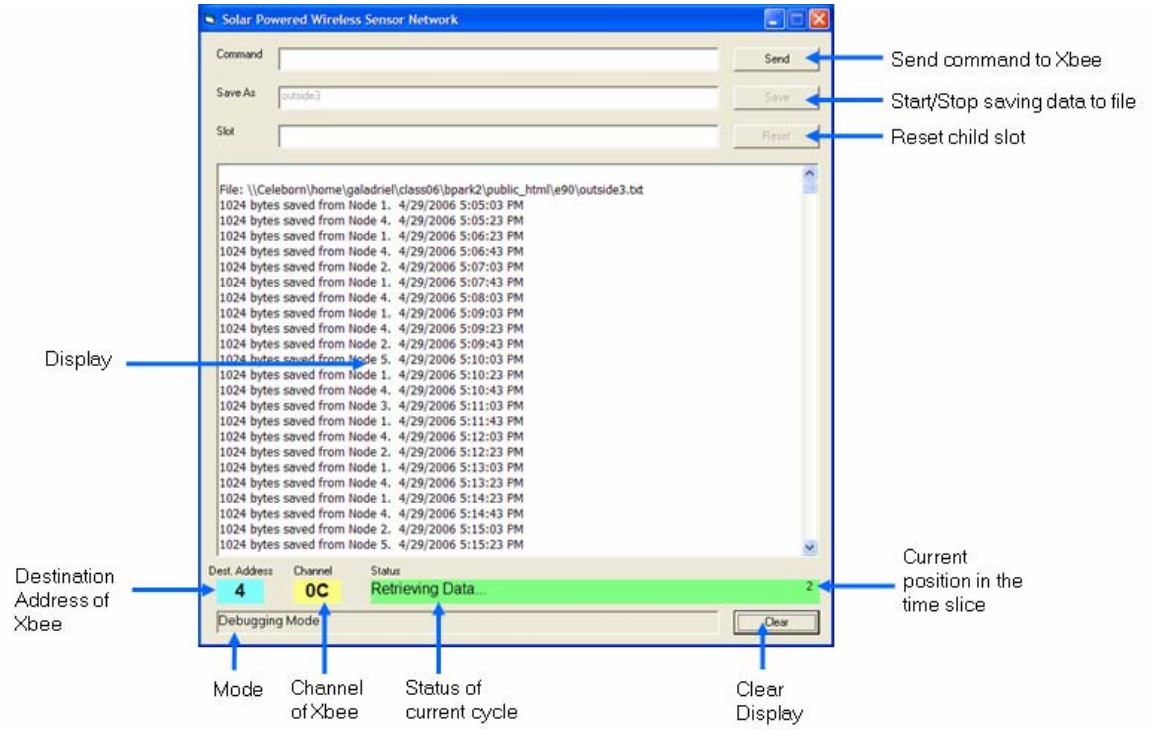


Figure 31. Screenshot of Computer Host program.

1. Open the program (host.exe)
2. If serial port settings are correct, the program starts in Command Mode. The COM1 port must be available and the XBee/XBeePRO module must be set at 19200 baud.
3. Commands can be sent to the XBee by entering a command and clicking "Send." This can be used to make sure the program works properly.
4. Turn on all Level 1 Nodes.
5. Enter a filename and click "Save" to start the RTS protocol and enter Saving Mode. The file will be saved in the same directory that host.exe is located in. To save the data to another file or into a web accessible location, the entire file path must be entered. For example, saving the file to \\Celeborn\home\galadriel\class06\bpark2\public_html\e90\example.txt will make the file available online at http://www.engin.swarthmore.edu/~bpark2/e90/example.txt
6. Data will be saved to the file in a comma delimited format. Refer to the Data Packets section under the RTS Network Protocol topic in this report for more information on how the data is organized. New 1024 bytes of data will be written to the file in one long string of numbers. Each byte is separated by a comma.
7. Add a Level 2, 3, or 4 child to the network only after the display has printed that it has saved data from the child's parent.
8. If a Level 1 node goes down or consistently fails to send data, reset the host slot by entering the slot number and clicking Reset. Troubleshoot the node and add it back to the network for re-initialization. This feature allows you to fix a Level 1 node without stopping the data gathering process altogether.
9. Click Stop to stop saving data and enter Debugging Mode. In Debugging Mode, only commands can be sent to the XBee and the program no longer lets you save data. The program cannot go back into Saving Mode.
10. Exit the program.

Gathering Data Using Analog and Digital Sensors

In order to gather desired sensory data using our system, all the user needs to do is rewrite the `void get_data()` function in the "rts.h" file to interface with any sensors that are going to be connected to the sensor nodes. The current implementation of this function is shown below:

```
//get_data function used to gather data
void get_data(){
   int8 i = 0;

   write_ext_eeprom(node.next_data, temp_sense());
   //write_ext_eeprom(node.next_data, 'A');
   node.next_data++;

}
```

This implementation calls the `temp_sense` function from the file "LM19.h" (see Appendix B5), which simply reads the value of analog pin AN1, which is connected to

the analog output of the LM19 temperature sensor we used for our temperature test. However, the `get_data` function can be rewritten so that it reads analog values from any one of the available analog pins, or digital values from any one of the available digital pins. In addition to this, the `get_data` function can also be written so that it reads data from digitally interfaced analog sensors through the SPI or the I^2C bus.

Also, the delay between data points needs to be defined by the `LOOP_DELAY` constant as mentioned earlier. The `LOOP_DELAY` constant sets the delay between data points in hundreds of microseconds and it is defined in "rts.h" (see Appendix B2). For example, consider:

```
#define LOOP_DELAY 100000
```

This definition results in a delay of 100000x100μs = 10s.

System Specifications

Table 6. System Specifications

Operating Time	
Without Sunlight	approx. 160 hrs for Level 1, R:T:S=1:1:6
With Sunlight	Infinite?
Data Throughput	$DataRate = \frac{1024}{(NumberOfChildren+1)*(SleepTime+2)*TimeSlice}$
Range	Specified by XBee/XBeePRO modules
Maximum Number of Nodes on Level 1	3
Maximum Number of Levels	4
Maximum Number of Children	12
Maximum Number of Nodes	3x12x12x12 = 5184

SUGGESTED IMPROVEMENTS

Host Program

Visual Basic provides very little control over background processes and actual hardware communications with the serial port. Although the program gets the job done, it is very inefficient and takes up all of the CPU's processing power during data collection. Using the computer while the program is collecting data can cause the program to hang or cause the network to become out of synch. A more efficient program would allow the user to multitask and work with other applications on the computer while collecting data from the network.

Measuring Battery Voltage

The resistors used in the battery voltage divider were 5% resistors. These errors can provide misleading measurements of the batteries. Therefore, 1% resistor values should be used to obtain a more accurate measurement.

Encasing

Tupperware containers are not very robust and are susceptible to leaks. The development of a weatherproof encasing that is permeable to 2.4Ghz frequencies is needed to test the network in all types of weather conditions.

Testing

The entire system needs to be tested further in field conditions. In particular, the efficiency of the power stage, the range of communication, and the proper functioning of nodes in failure mode need to be examined in more detail.

ACKNOWLEDGEMENTS

We would like to thank our adviser Professor Erik Cheever for his advice and support throughout the development of this project. Our gratitude also extends to Edmond Jaoudi for his help and resourcefulness when it came down to finding suitable hardware solutions for our circuits, and to Professor Carr Everbach for helping us come up with and develop the idea for this project.

APPENDIX A: *Data Sheets*

A1. PIC16F87XA Data Sheet
http://ww1.microchip.com/downloads/en/DeviceDoc/39582b.pdf

A2. LM4040 Precision Micropower Shunt Voltage Reference
http://www.national.com/ds/LM/LM4040.pdf

A3. PICmicro™ Mid-Range MCU Family Reference Manual
http://ww1.microchip.com/downloads/en/DeviceDoc/33023A.pdf

A4. LM19 2.4V, 10µA, TO-92 Temperature Sensor
http://www.national.com/ds/LM/LM19.pdf

A5. ENERGIZER NH15-2500
http://data.energizer.com/PDFs/nh15-2500.pdf

A6. XBee™/XBee-PRO™ OEM RF Modules
http://www.maxstream.net/products/xbee/product-manual_XBee_OEM_RF-Modules.pdf

A7. 25AA160A 16K SPI™ Bus Serial EEPROM
http://ww1.microchip.com/downloads/en/DeviceDoc/21807b.pdf

A8. L4931 VERY LOW DROP VOLTAGE REGULATORS WITH INHIBIT
http://www.ortodoxism.ro/datasheets/stmicroelectronics/4340.pdf

A9. ST3232E ±15KV ESD-PROTECTED, 3 TO 5.5V, LOW POWER, UP TO 250KBPS, RS-232 DRIVERS AND RECEIVERS
http://www.ortodoxism.ro/datasheets/SGSThomsonMicroelectronics/mXytwqz.pdf

A10. TPS62203 HIGH-EFFICIENCY, SOT23 STEP-DOWN, DC-DC CONVERTER
http://www.ortodoxism.ro/datasheets/texasinstruments/tps62203.pdf

A11. Si9181 Micropower 350-mA CMOS LDO Regulator With Error Flag/Power-On-Reset
http://www.ortodoxism.ro/datasheets/vishay/71119.pdf

A12. CRC Calculation Humidity Sensors (Revision 1.03)
http://www.sensirion.com/images/getFile?id=80

APPENDIX B: *PIC C Files*

B1. node.c
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/node.c

B2. rts.h
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/rts.h

B3. xbee.h
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/xbee.h

B4. EEPROM_SPI.c
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/EEPROM_SPI.c

B5. LM19.h
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/LM19.h

B6. tick.h
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/b/tick.h

APPENDIX C: *Computer Host Visual Basic Files*

C1. host.vbp - Project
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/c/host.vbp

C2. host.frm - Form
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/c/host.frm

C3. rts.bas - Code Module
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/c/rts.bas

C4. global_variables.bas - Code Module
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/c/global_variables.bas

C5. host.exe - Compiled Version
http://www.engin.swarthmore.edu/~bpark2/e90/appendix/c/host.exe