Zigbee Wireless Microcontroller Network
Spring Semester Final Report

Senior Design Project ECE492, Cornell University

Scott Bingham – STB25@cornell.edu
Yunfan Donald Zhang – YDZ2@cornell.edu
Project Advisor: Bruce Land – BRL4@cornell.edu

Project Overview

The growth of wireless network technology has led to large demand for self-controlled, scalable wireless sensor networks. Remote reporting of data from many nodes to a central control and data center has many applications, from battlefield sensor networks to animal tracking and monitoring systems. Wireless nodes provide the network with the ability to reconfigure on the fly without being tied down by signal cables. The goal of our project is to implement such a network using microcontrollers connected by 802.15.4 (Zigbee) transceivers to a central control microcontroller that interfaces with a database accessible through TCP/IP. The four major components consist of 8-bit Atmel Mega32 Microcontrollers, Freescale MC13192 2.4 GHz Low Power Transceivers for 802.15.4, a Lantronix XPort Embedded Device Server contained in an RJ-45 connector package, and a MySQL database hosted on a remote computer. A block diagram of the high level design is provided in figure 1 below.

High Level Design

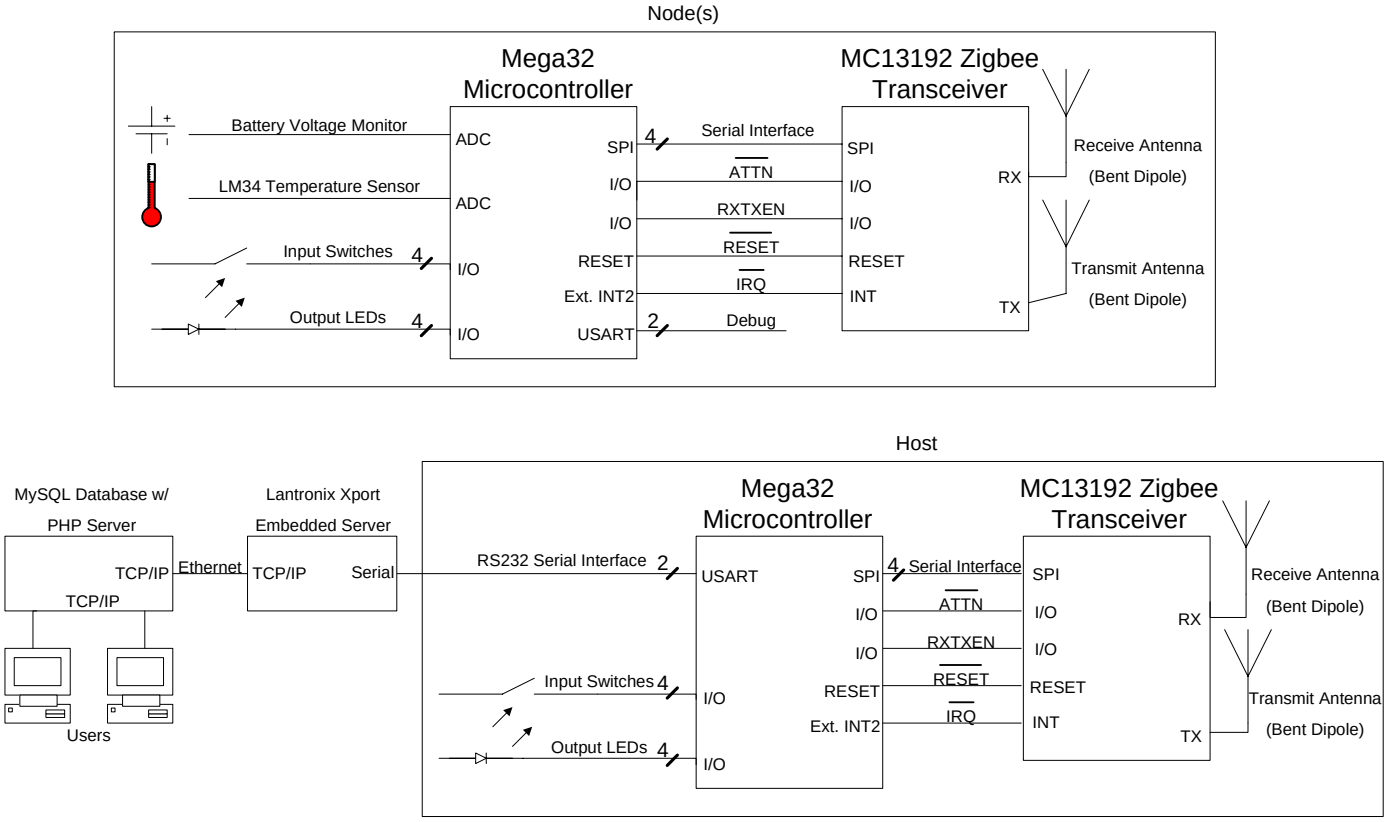

Figure 1 Hardware Block Diagram

Each node consists of a microcontroller, a transceiver and support hardware. The host consists of the same hardware but is attached to the embedded Xport server which connects to the database. All boards are capable of being either a host or a node and the same code is loaded on each board because of excessive functionality sharing. While only four nodes and one host were used in this implementation due to cost reasons, this is by no means the limit.

Node and host communication is handled through time division multiplexing to ensure only one node is transmitting at a give point in time. Also, synchronization among nodes and with the database happens at regularly scheduled intervals. The host must frequently offload data to the database so that its buffers do not overflow. By default the network is arranged in a star pattern with the host in the middle receiving packets from all nodes who disregard traffic from other nodes. However, if a node falls out of range of the host but is in range of another node, the network can transform and implement a store and forward algorithm where nodes pass on the information of other nodes to the host. While this may lead to excess or duplicate data sets, it ensures that all samples will reach the host and therefore the database if a path through other nodes exists. Duplication of data can be easily remedied at the database level.

Network Hardware

Atmel Mega32 Microcontroller

The Mega32 is an 8-bit RISC microcontroller with 2KB of SRAM and 32KB of Flash memory for program storage and is C programmable through Code Vision AVR. The microcontroller is capable of running at 16MHz with a supply voltage of 5V DC. The Mega 32 also features two 8-bit counters, one 16-bit counter, pulse width modulation, an eight channel ten bit ADC, a serial USART, a master/slave SPI interface, and 32 I/O pins. These features make this microcontroller a very robust and highly configurable platform for the network nodes. Port configurations can be seen in appendix A.

Timer0

Timer0 is used to maintain a real time clock in our network. Using a prescalar of 1/64 and a 16MHz clock, every 250 ticks equates to one millisecond. By triggering a compare interrupt every millisecond, the interrupt service routine updates the node's software clock. By maintaining the current time, accurate to the millisecond, a timestamp is generated each time an ADC conversion has completed, and the time when a sample is received at the host microcontroller. An accurate time base is crucial to the operation and synchronization of the network as communication is based upon time division multiplexing as will be discussed in more detail later.

Analog to Digital Converter

Node microcontrollers generate sample data using the ADC. A single ended conversion on channel zero is triggered at predetermined intervals by the software. With a prescalar of 1/128 and a 16MHz clock, each analog to digital conversion takes 8 microseconds. Upon completion an interrupt is triggered and the data can be read from the result register. While the ADC produces a ten bit result, the eight most significant bits produce adequate accuracy for now. It also acts as a simple noise filter. The result of the conversion represents the range GND to VREF minus one LSB. The binary conversion is translated to a real voltage through the formula V_{IN} = ADCH * V_{REF} / 256. If higher resolution was needed, all ten ADC bits from the ADCH and ADCL registers would be combined before converting with a scaling factor of 1024 instead of 256. In this network implementation, analog to digital conversions are preformed on the LM34 temperature sensor and on the battery power supply voltage in order to provide an accurate presentation of network state.

Universal Synchronous and Asynchronous serial Receiver and Transmitter

The USART device is used to communicate with devices outside the microcontroller. It is easily connected to a Telnet terminal using an RS232 driver and socket, with the USART in asynchronous mode with a baud rate of 115.2kbps, eight data bits, one stop bit, and no parity bit. Each byte sent from the UART represents an ASCII character so that the debugging or data information can be displayed in the Telnet terminal or server without any data format conversions. Such a packet format makes it easier to parse data packets that arrive at the database interface. Control and time update packets produced by the data are also sent in ASCII characters upon request from the host microcontroller and are subsequently disseminated throughout the network and converted to internal formats.

The USART triggers interrupts upon sending and receiving a byte. There are two possible interrupts that can be triggered upon sending a character. The interrupt that triggers when the data register has shifted into the transmit register is used, as opposed to the one triggered when the transmit register is empty, for purposes of speed. The data register empty interrupt expects the interrupt to be turned off or the data register to be written to immediately upon triggering. Therefore, we incrementally send all the bytes of a buffer until to the end of the null-terminated string, at which point we disable the interrupt until another string is ready to be sent.

Upon receipt of a byte, the USART also triggers an interrupt. This can be ignored if it is know that the Telnet terminal is simply echoing the received character back. However, it is very useful to send commands to the microcontrollers from the Telnet terminal over the USART and so, this interrupt allows it to fill a string buffer until an end of transmission symbol is received, at which point the string can be processed. As mentioned previously, incoming control and time update packet transactions are initiated by the host to ensure prompt and correct processing of such critical information. The USART also generates error flags for problems encountered during an incoming or outgoing transaction, but these are not processed because as software is used to supervise such.

Serial Peripheral Interface

The serial peripheral interface provides full duplex, synchronous data transfer between the microcontroller and the transceiver or any other device equipped with an SPI port. The SPI connection model consists of a master and a slave device with four lines between them. The master-out / slave-in (MOSI) serial line is the connection over which data bits are sent from the master to the slave. The master-in / slave-out (MISO) line provides data flow in the opposite direction. A clock for these serial transfers is generated by the master and sent over the serial clock line (SCK). The clock edges on this line control the shift registers of the master and slave. The SPI clock is run at 4MHz, the fastest sustainable speed of the microcontroller and transceiver. Finally, an active low slave select line ($\overline{SS}$) allows the master to enable a slave device's SPI port for the transaction. This could potentially allow multiple slaves controlled by a single master, but in our model, there is initially only one slave and one master device. Figure 2 illustrates the master/slave interconnect in the SPI model. Both the microcontroller and transceiver sample data on a leading, rising clock edge and set up the next bit on the falling edge.

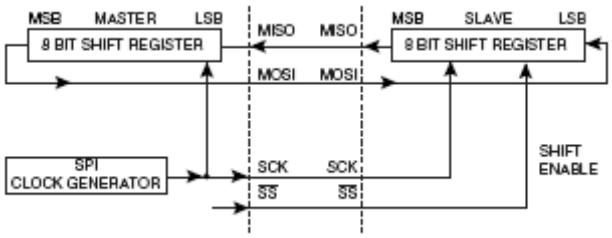

Figure 2 SPI Master/Slave Interconnect

An SPI transaction from master to slave has several steps. First, the master pulls the $\overline{SS}$ line low to inform the slave of its intent to initiate a transaction. The master then sends out a byte over the MOSI line as well as well as the corresponding clock burst on the SCK line. In the Mega32, writing to the shift register generates the required clock pulses. The slave clocks these bits in and simultaneously clocks the contents of its shift register out onto the MISO line. This byte is ignored unless the devices allow for simultaneous input and output. When the entire byte is shifted out of the master, an interrupt is generated. If there are more bytes to be sent, the master can continue to write to the shift register. When the transaction is complete, the master drives the $\overline{SS}$ line high.

An SPI transaction initiated from slave to master is similar but has some key differences. Because the transceiver can only act as a slave, the microcontroller must always remain the master. Therefore, in order to serially send data into the microcontroller, the transceiver must inform the microcontroller of its intent to send. This is accomplished through an external interrupt on the microcontroller. When pulled low, an interrupt is generated and the microcontroller reads from the transceiver through an SPI transaction. The master again sets the enable line low to begin. Because the master must generate the clock bursts, dummy data is written to the microcontroller's data register which is ignored by the slave. The master then reads the data shifted over the MISO line upon completion of the transfer. If more data is to be sent, the master again writes to the data register to create more clock bursts. When the transaction is completed, the $\overline{SS}$ line is driven high.

Power Saving Sleep-Mode

The microcontroller has six power-saving sleep modes with varying levels of hardware disabled. Because it was necessary to maintain an accurate time base through the use of timer0, the only sleep mode usable was the idle mode that turns of the clock to the core and flash while leaving other clocks and functional blocks running (including timers). To further save power, the analog comparator and analog to digital converters were disabled during sleep mode. The microcontroller exits sleep mode a few cycles after an interrupt is raised, typically the timer interrupt maintaining the millisecond time base in this case. Because it takes several cycles for the processor to exit sleep mode and service the responsible interrupt, there is a strong possibility of clock skew over many sleep iterations. To compensate for this, the time across all nodes in the network is periodically resynchronized with the host, who never sleeps, and the database which provides the real time.

EEPROM

Because the flash is not writable by the microcontroller, the only non-volatile memory available is the EEPROM, which is used to store the data set, microcontroller status, and timestamp of the node after every conversion. This is done so that the event before a node

failure can be retrieved when the node is brought back to life. This is particularly useful in debugging the network, but also provides a way to recover the last data set taken before a node turned off for any reason, for example, battery failure.

Freescale MC13192 2.4 GHz Transceivers for 802.15.4

The MC13192 is a short range, low power transceiver operating in the 802.15.4 physical layer as specified by the IEEE standard. While the transceiver was designed to interface with the HCS08 family of microcontrollers, it is programmable by the Mega32 through the SPI port. The Zigbee transceiver supports buffered transmit and receive data packets through sixteen 5.0MHz channels at a data rate of 250kbps per channel. The transceiver supports streaming and packet modes; however, we use the packet mode to conserve microcontroller resources. The packet structure of the MC13192 consists of a four byte preamble, a one byte start frame delimiter, a one byte frame length indicator, a payload of up to 125 bytes, and a two byte frame check sequence for cyclic redundancy checks of the data as shown in figure 2. The physical transmission and reception of data is transparent to our project and was a key factor in deciding to use this transceiver.

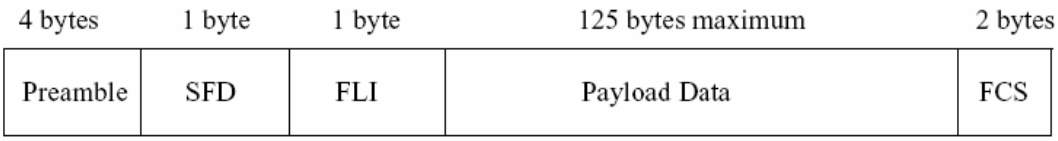

Figure 2 Wireless Packet Format

The transceiver is controlled entirely by writing to several internal registers through the SPI port. There are four register values that must be initialized for proper operation. Their function is never disclosed other than that they fall under the reserved field classification. There are also a number of user selectable registers that must be initialized for the operation of our network. Because of the presence of reserved fields in the user configurable registers, all transactions must take place in the read – modify – write format in order to preserve their values. All SPI transactions consist of a control byte followed by an even number of data bytes. The command byte consists of a read/write bit and a six bit address. Figure 3 illustrates the contents of an SPI packet. Singular read transactions consist of sending a command byte to the transceiver followed by two data bytes being sent back to the microcontroller. All of the required clock bursts are generated by the microcontroller as previously described. Singular write transactions consist of a command byte and two data bytes being sent from the microcontroller. Singular transactions are used to read and write control and status registers. Singular read and write transactions are depicted in Figures 4 and 5, where CE corresponds to $\overline{SS}$ on the microcontroller. Recursive reads and writes are needed to access the transmit and receive packet RAMs, however. A recursive transaction means that one command byte is sent followed by more than two data bytes. In order to write or read from the packet RAMs, the RAM register address must be specified in the command byte and then all subsequent data comes from or is written to the RAM and not the register. Every clock burst will read or write from the next byte in RAM and is auto-incremented. It is the responsibility of the microcontroller to determine how many clock bursts to produce.

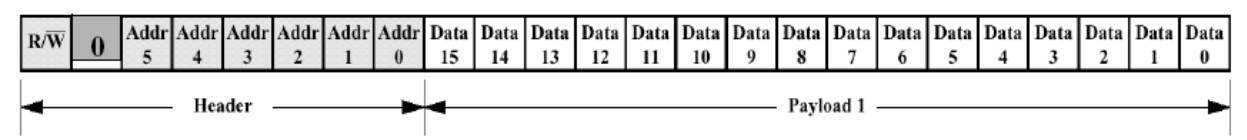

Figure 3 SPI Packet Format

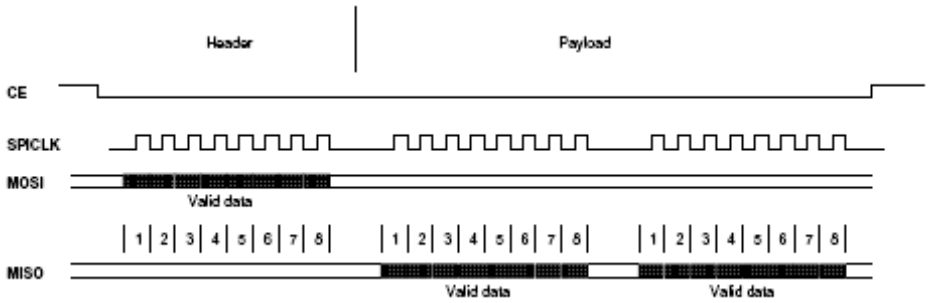

Figure 4 Singular Read Transaction

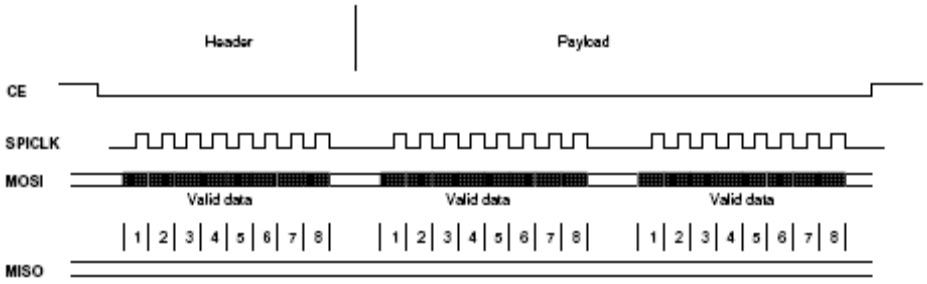

Figure 5 Singular Write Transaction

The transmit packet control register selects which packet RAM is to be used and the length of the packet to be sent. The length of the data to be sent is written before each transmission. All interrupts in the IRQ mask register are turned off because the only concern is with the successful transmission or reception of a packet for the time being. For error correction purposes, it may require enabling some of these interrupts if testing shows a significant number of the transmissions being corrupted or not sent. To date, this has not presented a problem. The control registers allows selection of packet transmit or receive modes. These modes must be reinitialized after every reception or transmission as the transceiver returns to idle mode after these events. This also means that each transceiver is either a transmitter or receiver and never both at any point in time. This is very useful in filtering out packets not intended for that transceiver. The transceiver also allows for several low power states when not in use. The hibernate state is used in conjunction the idle state of a node's microcontroller when the transceiver is not needed. While all register and packet RAM data is preserved in hibernation, this extremely low power state has the side effect of being slow to resume normal operation. However, by awakening the transceiver in sufficient time to react to microcontroller activity, we are able to reap massive power savings without sacrificing performance. In order to awaken the transceiver, the $\overline{\text{ATTN}}$ pin is driven low, and an interrupt request is generated once the transceiver returns to idle mode if the interrupt mask is set appropriately. All general purpose I/O pins on the transceiver are also disabled, as is the clock output, which helps reduce power consumption. Timers on the transceiver are also turned off because the microcontroller controls

the transmission of packets by setting transmit mode and asserting the transmit/receive enable line.

Upon successful transmission or reception of a packet, the $\overline{\text{IRQ}}$ line is pulled low by the transceiver to inform the microcontroller of an internal interrupt. At this point, the microcontroller will read the IRQ status register and perform the appropriate function, described in detail later. If the interrupt indicates a packet was received, the microcontroller then reads the RX status register to determine the length of the packet received. Finally, the microcontroller reads the appropriate number of bytes from the received packet RAM.

Lantronix XPort Embedded Device Server

The XPort is a self contained TCP/IP server with the ability to both store dynamic web pages and act as a serial to Ethernet converter. The device is configurable through a web interface as well as through a serial connection compatible with our USART. All network overhead is handled by the XPort transparent to the user. Currently, the XPort is used as a transparent serial connection between the host microcontroller and a server connected to the XPort over the internet. The decision was made to implement this function of the XPort before storing web pages on the XPort, as the memory of a computer is virtually unlimited in comparison to the XPort. The data flowing from the USART is passed to the server running a PHP script which then stores the data into a MySQL database. MySQL offers much more functionality for storing and analyzing data than an embedded Java Applet would.

Custom PCB Implementation

Because of the difficulties and performance considerations encountered when attempting to connect separate boards for the transceiver and microcontroller (discussed in more detail later), it was decided that a new printed circuit board should be designed and fabricated to accommodate all necessary hardware in one location. The PCB's main features, as shown in figure 6, will be discussed briefly. Bypassing capacitors exist between power and ground on all integrated circuits and will not be discussed beyond the power circuit. The PCB consists of four electrical layers, the top signal layer, the bottom signal layer, the power plane, and the ground plane. In figure 6, the left blue box encloses the transceiver, and the right blue box encloses the microcontroller. Each board is capable of being either a host or node and can be powered by a 9V battery as shown.

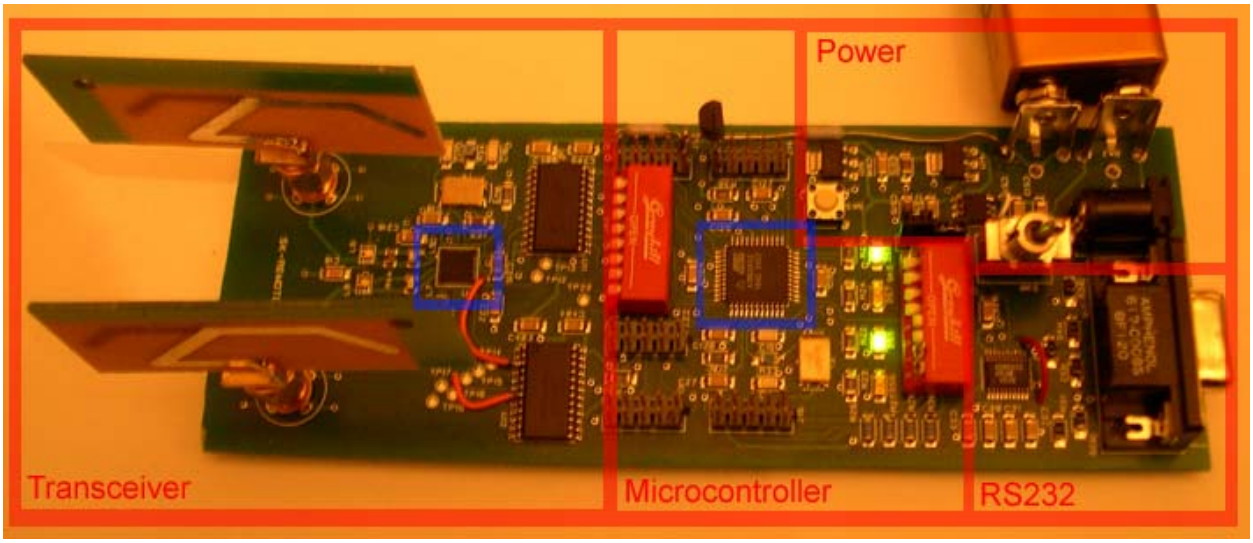

Figure 6 Node on Custom Printed Circuit Board

Transceiver Circuit

The transceiver circuit consists of the MC13192 Zigbee transceiver, a 16MHz crystal, two level converters, and two antennae with their associated hardware. The transceiver is housed in a QFN-32 package which is practically impossible to solder by hand, due to the leads being located entirely underneath the casing. This was one of the main reasons for manufacturing our own board as connecting wires to the transceiver proved very difficult. While both the transceiver and microcontroller operate at 16MHz, their locations are sufficiently far apart that it was easier and more noise-free to allocate a separate crystal to each, rather than snake around the traces in between them. Because the transceiver operates at 3.3V and the microcontroller operates at 5V, it was necessary to place voltage level converters between the two chips for the SPI and related control signals. Communication is possible between the transceiver and microcontroller without level converters; however, we felt it best not to over or under-voltage a high speed serial line to ensure data integrity. The final components of the transceiver circuit are the bent dipole antennae. Their design and that of their impedance matched circuitry was derived from reference designs provided by Freescale on the MC13192's product webpage. The circuit for the transmit antenna varies slightly from that of the receive antenna. It is important to note that the shielding ground plane is present inside the board for all places except around the antenna traces and related circuitry.

Microcontroller Circuit

The microcontroller is isolated from the transceiver circuit and the RS232 circuit via DIP switches so that the microcontroller can be used independently of the other components and so that the microcontroller can be programmed properly. Because the microcontroller is connected to the transceiver via the SPI port and is programmed via the SPI port, it was necessary to provide a mechanism to isolate the two chips in order to program the microcontroller. Without these isolation switches, the STK500 would drive both the transceiver and the microcontroller simultaneously and be unable to program the latter. Also included in that DIP switch package is isolation from the reset controller, to be discussed shortly, because the STK500 must have control of the reset pin in order to program the processor. All I/O ports are connected to headers on the edge of the board so that peripherals may be easily attached. The pin layout is identical to

that of the STK500's port headers where ground and 5V are present as the ninth and tenth pins. The header in the middle of the board closest to the microcontroller is the programming header. Connected to four of the pins of port C are LEDs which are turned on by sinking current into the microcontroller. They indicate when a node is transmitting, receiving, sending UART communication, and the current status of the node (whether it is sleeping, waiting for the database, etc.). While LEDs may not be useful in every application, they can be easily unsoldered from the board and port C will revert to its normal operation. Next to the LEDs is another DIP switch package. Half of the switches act as input switches for the microcontroller and two of those input switches are connected to the external interrupts. The remaining four switches isolate the microcontroller from the RS232 driver on the transmit, receiver, clear to send, and ready to send lines. As we did not use ready to send and clear to send, they are temporarily hardwired together in the RS232 circuit. On the top middle of figure 6, the LM34 and the low battery detection circuits can be seen attached to port A's headers for the analog to digital converters.

RS232 Circuit

The RS232 circuit consists of the RS232 driver chip and the RS232 connector. The RS232 driver pumps up the voltage to the levels required by the serial communication standard on the connector side of the chip and bring the voltage back down to suitable levels for the microcontroller on the other side of the chip. Capacitors located next to the driver allow for the charge pumping needed to reach these high voltage levels. The RS232 connector has zero Ohm jumpers available to connect different pins of the socket together if needed for a given protocol. In this implementation, all jumpers are lifted as only the send and receive data lines are needed.

Power Circuit

In order to provide flexibility, this PCB is capable of being powered by both battery and AC adapter. A common switch is shared between the two power sources where turning one on turns the other off so that they can never be connected directly to each other, though they shared a common ground. Next to the main power switch are two headers that act as switches for the power to the RS232 circuit and the transceiver circuit. Because these are unlikely to be switched often, they were implemented with headers and jumpers to save space and cost. The main components of the power circuit are the 5V and 3.3V regulators. Both are connected to 1uF and 10uF capacitors on their input and output sides to smooth any ripples in the voltage sources. The final components of the power circuit are the reset controller and the reset switch. The reset controller is used so that the processor and transceiver chips do not bounce on and off while the source voltages are brought up to the proper levels. The push button reset switch provides a quick way to reset the microcontroller and transceiver without powering down the entire board.

Network/Software Implementation

When a microcontroller turns on, the first step it takes is to read in the input switches. These switches determine whether or not a board is a node or the host. For nodes, the remaining input switches determine what the node's number is so that it knows when to transmit and the database knows which node a data sample originated in. The next step is to initialize the microcontroller. The data direction of all of the ports is set first with the functionality defined in appendix A. Next, timer0 is configured to trigger an interrupt every millisecond to update the

clock. As each lower unit of time overflows, the next higher level is incremented. This is handled in the interrupt to maintain as accurate a clock as possible without additional hardware. Most other interrupts simply set a flag which is processed in the main program loop. Next, the USART is initialized and messages are displayed on the Telnet terminal or server indicating the program running and its version number. After the USART, the SPI port is initialized and if the microcontroller is a node, the ADC is initialized. Finally, state variables and flags are initialized.

Once the microcontroller is initialized, then the transceiver is initialized. Registers 0x08, 0x11, 0x05, 0x06, 0x03, 0x09, 0x0C, 0x1B, 0x1D, 0x1F, and 0x21 are initialized through a read – modify – write transaction. This is to preserve the data of reserved fields. The final value of each register is also read back and displayed over the USART to ensure proper initialization of and connection with the transceiver if debugging messages are enabled. These control registers setup the transceiver to act in packet mode as opposed to streaming mode, mask out unwanted interrupts, disable unneeded hardware such as I/O ports and timers, and set other control fields necessary for correct operation. Once the host has initialized its microcontroller and transceiver it queries the database for the current time and commands, which will be discussed in a later section. Upon successfully receiving the current time and commands, the host broadcasts the current time and commands so that waiting nodes can receive the information. Because nodes are not connected to the database, they must wait until the host comes alive and broadcasts this information to the network. Upon receiving this information, both the host and nodes proceed into their scheduled tasks.

It is important to note that there are five commands that can be broadcast to the nodes to reconfigure the operation of the network. They are char sample_enable, store_forward_enable, host_burst_enable, low_batt_enable, and analog_enable. The default behavior for network is to stream analog samples to the database and turn off if the battery drops too low. This requires nodes to send their data, and only their data, to the host as it is sampled; the host immediately forwards this data to the database for processing.

After the initialization, if the microcontroller is the host, the RXTXEN line is set high to begin receiving packets. External interupt0 is now enabled. The host transceiver waits in receive mode whenever it is not transmitting the time/commands or servicing a read flag. Whenever the interrupt line goes low, meaning the transceiver requests an interrupt be handled, the host reads the IRQ status register, 0x24, on the transceiver to clear the interrupt and service it. If the register indicates a packet was received, then the RX status register is read to determine the length of the packet. A recursive read for the remaining data is then performed on the receive packet RAM. The data read from the transceiver is filtered at this point and if the packet is valid, the result is then inserted into the USART buffer and written out the USART port if host burst mode is disabled. If burst mode is enabled, the host buffers the data (in another buffer discussed later) until just before it queries the database for an updated time/command and sends all collected data packets at once. This reduces the number of TCP/IP packets created and reduces the overhead by making packets longer. The host then returns to receive mode and waits for the next packet. Every ten seconds the host performs the database query and updates the network (if there is no reply in the short request window, the host proceeds with the last commands, broadcasts, and returns to receiving mode). It is done at this frequency to allow command changes to happen quickly and to inform waiting nodes of the network status so they may begin taking data samples. Because the host is always listening for packets, it is not allowed to enter sleep mode and therefore consumes significantly more power than the nodes if sleep mode is enabled. By recording current and voltage levels for several network cycles, the

data in table 1 was collected for a host board using a *Measuring Pad* data acquisition instrument (see appendix B for pictures). The conditions for the maximum power consumption for the host are when nodes are implementing store and forward and the host is still streaming packets to the database as they come in.

Table 1 Host Board Electrical Characteristics Under Maximum Load (4 Nodes)

	Supply Voltage (Volts)	Current (Amps)	Power (Watts)
Maximum	10.02	0.077826	0.705104
Minimum	8.99	0.033913	0.339469
Average	9.06	0.075369	0.682786

After a node completes initialization, it proceeds to wait for the host to broadcast the current time and commands. This listening is performed every ten seconds in synch with the host so that the network can adapt rapidly to new commands. If store and forward mode is enabled, nodes broadcast the commands and time sequentially before exiting the update code so that nodes out of range of the host will also receive the commands and time. In normal operation, the node will then put itself and the transceiver in sleep mode until it is time to take an A/D conversion. When the sample is complete, the node sends the data sample to the host and again goes to sleep until the synchronization period. If sampling is disabled, the node does not perform the conversion, does not send a packet, and simply goes back to sleep. If the analog enable bit is not set, the microcontroller reads the voltage on the input pin and reports whether it read a digital true or false. If low battery mode is enabled and the node has battery voltage reading circuitry attached (verified at startup), the node also reads the voltage of its battery. If the voltage falls below 6V, the node warns the host that its battery is low. If the voltage falls further to 5.2V, the regulator will no longer be able to guarantee a 5V output and so the node informs the host and then goes to sleep permanently until its battery is changed. This is done to prevent the microcontroller from turning on and off as the load on the battery varies the voltage. This also protects data in the EEPROM as a power failure during writing would cause corrupt data. Power consumption data was also gathered for nodes during normal operation, sleeping when not transmitting or synching, and is presented in table 2. It is shown that sleep mode significantly helps power consumption over the worst case. While the previous table shows the maximum power consumption of the host, the worst power consumption for nodes is when they are in store-and-forward mode and behave almost identically to the host, with the exception of not talking to the database. Note that the power consumption of the node could be lowered further if power was disconnected from the RS232 driver which is only used in debugging the nodes.

Table 2 Node Board Electrical Characteristics Under Normal Load

	Supply Voltage (Volts)	Current (Amps)	Power (Watts)
Maximum	10.36	0.080435	0.720696
Minimum	8.91	0.020435	0.210274
Average	9.87	0.041105	0.401479

Before store-and-forward can be fully explained, the buffers on board the microcontrollers must be briefly explored. The first is the USART buffer which contains the strings to be written out to the USART port. Every string is an ASCII null terminated string and

is written when the send function is called. The function enables the sent interrupt and begins the transaction. The interrupt sends bytes until there are no more and then disables itself and the program continues. This buffer is large enough to hold eight nodes data packets for transmitting to the database when burst mode is enabled. A smaller UART buffer exists for receiving time/command packets from the database and is filled during synchronization time until a stop character is received. The next buffer is a transmit buffer for the SPI port. The first byte is always the command byte for the SPI transaction followed by the data byte pairs to be sent. The send function continues to send bytes via the SPI port to the transceiver until it reaches the number of bytes specified by the function parameter. This allows for recursive writes to the transmit packet RAM. When the transaction is complete, the $\overline{SS}$ line is pulled high. Another buffer is a receive buffer for the SPI port. The command byte is initialized in the buffer and the number of bytes to read is passed to the get function. This function fills the receive buffer with the specified number of bytes to be read and then pulls the $\overline{SS}$ line high. To generate the required clock bursts from the microcontroller, a dummy variable must be written, the loop counter, to the SPI port, and on the following interrupt, read the value returned from the transceiver. Finally a matrix was created to hold data specifically for store and forward operations and bursty host transmissions. Each node is assigned a row in the matrix that stores their data as it is received, in store and forward for the nodes and both store and forward and burst mode for the host. Packets to be sent out are then created by iterating through the valid data in the matrix and invalidating data as it is sent.

To display data flowing from the Xport, a PHP script was written to act as a TCP client to talk to the Xport, which acts as the TCP server. The script first creates an IPv4 TCP socket and then tries to connect to the XPort at a specified IP address and port number. Once the connection is established, it goes into an infinite loop that will continue to read from the server for any incoming TCP packets. The XPort dumps serial data from its buffer into a TCP packet and sends it to the specified address upon receiving a specified character. Every time the script sees a packet, the data is stored to a table in the MySQL database, along with the time that the packet is received. The script should run indefinitely or until the webpage is stopped by the user. Since the script will be running endlessly, a flush function is necessary to output the data to the browser as soon as it arrives. This avoids any hold up either in the Apache server buffer or in the web browser's own buffer. This script displays the raw data as it comes in; another script is used to query the database to display the processed/stored data. With this script, options can be set, such as to only display all data from node 1, or to only display data for the past hour, etc.

Store and forward is implemented in the following manner. When in store and forward mode, all nodes listen for other nodes to transmit and store the data into the row for that node in the previously mentioned matrix. Because nodes are constantly listening, they are not allowed to sleep and act very much like hosts. Nodes continue buffering this data until it is their turn to sample and transmit, at which point they send their entire matrix and invalidate it. All listening nodes will possibly see multiple sets of data in that packet which is then parsed and stored in the matrix. This cascades up to the host who either sends the multiple data packets every time they are received or buffers them until synchronization. If a host or node already has valid, as yet un-transmitted data for a node, it does not overwrite the previous data in the matrix until it has been transmitted. However, the data should still exist somewhere in the network and eventually make its way to the host. While this method can create many duplicate packets when many nodes listen to each other, it greatly increases the odds of an out of range node's data making it to the

host. Duplicate packets that read the database are detected and eliminated so as not to pollute the database. As mentioned before, time and command data is forwarded by nodes at synchronization time so a node never has to be in direct communication with the host. The value of this practice was verified by placing a node in a metal refrigerator. Without store and forward, the node could not be heard from. However, after enabling store and forward and placing another node close to the refrigerator, the temperature data from inside was able to reach the database.

Database and GUI

The PC side of this project involves two php scripts and a MySQL database. The PC takes in TCP/IP packets that are sent out by the computer that is connected to the host MCU serially. It processes the packets and stores the relevant information on the database. It also responds to host MCU's request for a time stamp.

The script that listens for incoming TCP/IP connection is called server.php. Since this script will have to be run indefinitely listening for incoming packets, it cannot be run in a web browser as it will take up all the system resources doing so until the browser forces it to time out. Instead we run this script from the command line in the background. Much like a MCU code this script also contains a while(1) loop, which keeps it running forever. When the script is first run it creates a TCP/IP server at a specified IP address and port number. It then waits for any incoming socket connection. Once a connection has been established with the computer that is serially connected to the MCU, the script enters the while(1) loop in which it waits for incoming packets. The first thing the MCU does when connected to the PC is to request a time stamp as well as a five bits command. Once the server script receives this request it sends out the current time generated on the PC and depending on whether if the user has specified the commands or not, it will either send the users commands or a default commands. This time and commands sync will be done every 10 seconds between the host MCU and the database. Once the first sync has occurred, the host MCU will proceed to send packets stored in its buffer to the database. Once the script receives what it recognizes as valid data, it will unparse it and perform some simple error checking. This involves making sure there are no duplicate packets coming in (this would only occur when the network is in store and forward modes as all the nodes will transmit all the packets to everyone), and also the temperature should be within reasonable range. When checking for duplicate packets, the script will check against the entire current database as it is possible that a duplicate packet exists from a previous transmission. Once a data passes error checking, it will be entered into the MySQL database, the server script will then return to waiting for more incoming packets.

The second script is the GUI part of this project. It queries the database and displays the collected data. It also allows users to set commands to the network as well as performs some simple data analysis. Unlike the server script this display script will be accessed via a web browser, which means the user can potentially access the data collection network anywhere on the internet. The first thing on the GUI page is a short description of each of the five commands that the user can set for the network. Once the user make the selection and submits it, the new configuration is stored in the database and sent out to the host MCU during the next sync. This means there might be a maximum of 10 seconds delay in setting the commands since the MCU and the database syncs every 10 seconds. The next section on the GUI allows the user to sort the

data currently available on the database by choosing a specific nodes or a time period. To prevent long webpage and query time, the GUI will only display up to the 100 most recent data points. Once the correct data has been loaded and formatted into a table, a built-in javascript checks all the temperature values for the ones that are above a certain threshold. If an "overheating" has been detected, the script will pop a window alerting the user that which node has passed the threshold temperature.

Implementation Problems

Initially our biggest implementation problem was the physical wiring and soldering of the transceiver boards. Because the original boards had other microcontrollers attached to them to communicate with the transceivers, we had to remove them before we could connect our microcontrollers. Prof. Land removed two of the microcontrollers for us, and we ground off the remaining microcontroller with a Dremel tool. Once the microcontrollers were removed, we had the problem of connecting to tiny pads on two of the boards and via contacts on the third. The pads were difficult to connect to due to their small size and some of the pads we had to connect to fell off. This meant we had to connect to vias and one of the pins underneath the transceiver. This connection was extremely difficult to maintain electrical integrity as there was only a very limited and fragile connection with transceiver pin. On the board that we had to grind the transceiver off of, all of our connections were to vias. Once we removed all of the shorts between the microcontroller pads created by grinding off the chip, we had to solder onto vias that were often directly next to each other. This often created shorts between two vias that had to be fixed many times. Also, we had difficulty making a good electrical connection with the via due to its small size and solder resist and solder residue covering the copper ring. Also, connecting to the STK board proved difficult as we did not want to solder on it. That meant our connections consisted of plugging wires into the sockets on the end of our ten wire ribbon cables, which often fell out. These problems were rectified by building the new boards.

Connecting the XPort also caused several problems. The XPort will often lose connection and need to be reset after making configuration changes. Also, sometimes the XPort will not connect with the router and obtain an IP address. So far, we are unable to determine the causes of these problems, and tech support's answer was basically that we need to buy the development kit. It is suspected that these problems are maybe be related to overheating of the regulator or an electrostatic discharge.

Because our transceiver operates at 2.4GHz, we encounter a lot of noise with other wireless devices. Currently we filter out bad packets with software, however occasionally these bad packets will cause the host transceiver to stop working and need to be reset. It is possible that switching to a different wireless channel will reduce bad packets as well as moving farther from sources of interference.

Conclusions / Future Improvements

Overall the Zigbee network implementation works very well. By filtering out bad packets and forwarding data for nodes out of range, very few data points do not arrive at the host. While time division multiplexing may eventually limit the number of nodes this network can support, the system with the four nodes presently available demonstrated that significant expandability is possible. However, the scalability of store and forward is very limited due to the small amount of RAM present and the amount used by buffers currently. Data could be sent in binary instead of ASCII as long as the data is not trying to be read in a terminal expecting ASCII.

Although there are possible improvements, our implementation met or exceeded all of our expectations, despite our initial hardware problems. Our sensor data is displayed in nearly real-time and survives being transmitted on a very crowded frequency.

For future implementations, a few minor PCB problems need to be addressed. Firstly, two connections from the level converter to the transceiver need to be made. The assumption that test points went through the board was incorrect and so those connections had to be made manually. Also the 5V regulator that was supposed to be an equivalent part was not, as it contained an enable where the one on the schematic did not. Finally, Vref was mistakenly connected to 5V meaning that all internal references are no longer usable unless the pin is lifted. If further power reductions are needed for extended life, additional circuitry such as the sensors could be gated, using output pins on the microcontroller. A possible addition to the project may be replacing the bent dipole antennae with a more directional antenna to further increase the range when employed with the store and forward policy.

Acknowledgements

We would like to thank our project advisor Professor Bruce Land for his help on this project and his patience with our reclusive work habits. We would also like to thank Dranetz-BMI for providing funding and granting access to fabrication/design facilities so that we could design and implement our printed circuit board used in the second half of this project.

Appendices

Appendix A. Port Definitions:

```
//A.0     ADC          Input Battery Voltage
//A.1     ADC          Input Sensor Voltage
//A.2-A.7 Not Used     -
// Port B Switch Isolated minus B3, RST switched in its place, not in order though
//B.0     RXTXEN       Output
//B.1     ATTN_        Output
//B.2     IRQ_         Input
//B.3     PWM          -
//B.4     SPI SS_/CE_  Output
//B.5     MOSI         Output
//B.6     MISO         Input
//B.7     SCK          Output
//C.0     LED2         Output   Xmit
//C.1     LED1         Output   UART
//C.2     LED3         Output   Rcv
//C.3     LED4         Output   Alive
//C.4-C.7 Not Used     -
//D.0 =   USART RXD          Input   Switched
//D.1 =   USART TXD          Output Switched
//D.2 =   SW6          Input
//D.3 =   SW5          Input
//D.4 =   SW4          Input
```

//D.5 = SW3 Input
//D.6-7 RTS/CTS - Not Used, Hard tied on far side of switch, dont flip those switches

Appendix B. Pictures

Figure 7 Measuring Pad used for Electrical Characterization

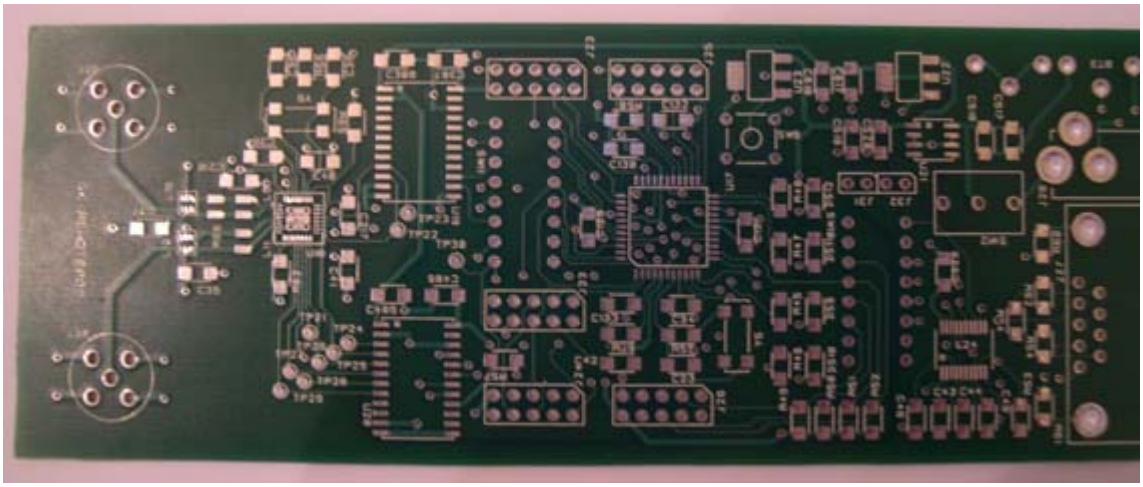

Figure 8 Unpopulated PCB

Figure 9 Populated PCB 1

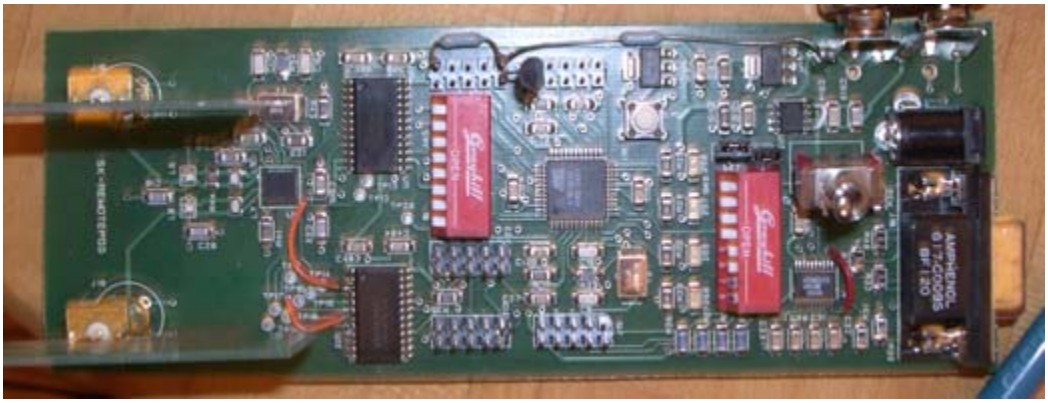

Figure 10 Populated PCB 2

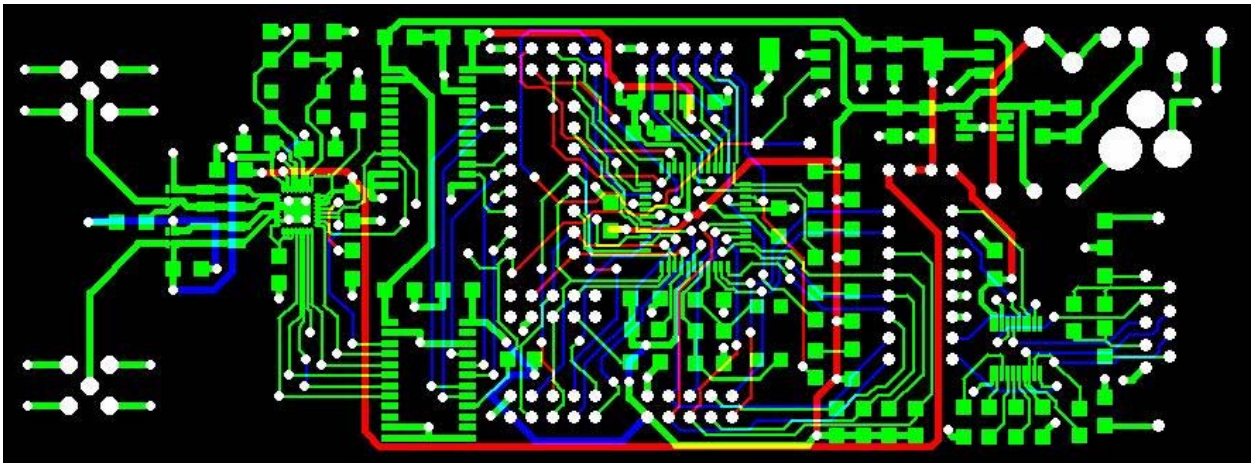

Figure 11 Electrical Layers Minus Ground Plane (Green Top, Red Power, Blue Bottom)