SHAMBLES

Student **HA**ndheld **M**o**B**i**L**e r**E**sponse sy**S**tem

Electrical Engineering Senior Design Project 2007-2008

The Dream Team

Andrew Daniel Carter
Matthew Paul Elliott
Herbert Andrew Harms
Benjamin Edward Keller

Table of Contents

1 Introduction

Effective education is an ever increasing concern in the United States today. Each group of students learns at a different pace and responds positively to different teaching methods. Measuring student understanding allows an educator to tailor his or her own style to a particular group of students, improving the learning experience for everyone involved. In the typical classroom, this measurement is accomplished through examination, periodic homework assignments, and targeting single students with questions. Utilizing these methods exclusively, several lectures might pass before the teacher notices a gap in student understanding. The problem is even more severe in the large classroom settings, which have become common among campuses of higher learning as it is extremely difficult for instructors to gauge the comprehension of hundreds of students during a class. Asking individuals verbal questions is an arguably sufficient means of in class feedback for 20 or 25 students, but this approach is simply not feasible in a large lecture hall. There is serious demand for a real-time, feedback system, which would increase teaching efficiency. Instructors using a system such a system would have access to information allowing them to spend more time on concepts which are difficult for a particular class and less time on concepts which are more easily understood.

It is possible for a small engineering team to develop a cheap and effective product to fill this need. We propose Shambles, a wireless classroom response system. Shambles uses open-source standards and technology to reduce cost which makes the system accessible to a wide range of instructors and students. Currently available devices that attempt to address this issue are flawed in key areas. A commonly used solution, einstruction(tm)'s classroom performance system (CPS), suffers from slow response, non-rechargeable batteries, and an expensive recurring subscription fee.

During the preliminary design stages, we identified system requirements common to any satisfactory solution. Students must be able to register their nodes for a particular class and then join a network. An instructor must be able to propose a question (multiple choice or free numerical response) to students. The students then must be able to answer the question. Answers corresponding to a particular question must be stored, manipulated in any way required, and then displayed to the instructor and possibly the students as well. The system must operate quickly and reliably for small, medium, and large class sizes. After speaking with several local experts on the subject of classroom response systems, we have identified several more requirements including one-touch message sending, system status feedback for the instructor, and expanded student input functionality. One-touch message sending is a change from our originally planned "send button" system, while the other two main suggestions are simply added functionality.

The student response system contains two main system types: the node and base station. The node consists of a microcontroller, Zigbee chip, battery, battery charger, USB controller, a keypad, and a LCD. The base station consists of a microcontroller, Zigbee chip, DC Adapter, and a USB controller. The base station and nodes communicate with computers via USB. The nodes and base station communicate with each other wirelessly via Zigbee MAC protocol. Each IC block (Zigbee, battery charger, etc.) within the node and base station devices must communicate with a microcontroller. The microcontroller will manage all the components of the student response system. It orchestrates the radio transmitting and receiving performed by the Zigbee chips on the node and base station. It controls the operation of the battery charger as well as wired data transfer from nodes and base station to computer. The microcontroller also handles user input (keypad) as well as user output (LCD) transferred by the user interface. This block will include writing a significant amount of software which controls all the components of each board. Zigbee wireless communication and network protocol is used to link each individual node to the base station. A battery charger IC is used to charge a battery pack that is built into each node. Power must also be provided to the base station via USB and DC adapter. Wired communication between each system device and computer will be implemented with USB. Nodes need to communicate with computers in order to register for classes in which the student response system is used. This interface will also be used to power the battery charger. USB is used by the response receiver to transmit student response data to the computer. The user interface of the response pads consists of two components. Users input information to be sent by the node to the base station and eventually to the instructor. The user interface will also display information back to the user such as the question to be answered, the answer to be sent, and network status. Status modes include no network and network joining accomplished. User data will be entered via keypads. The node and base station will also utilize a reset button. Inter-chip communication is implemented with SPI in order that each IC can communicate with the microcontroller. In this way the microcontroller can manage all the chips in each node. Zigbee and USB chips may not communicate with the microcontroller at once and must operate on an interrupt basis.

Software is written for the computer in order to interface with the nodes and base station through USB. The computer acts as a charging station for the nodes and allows them to register for classes online. The computer is used to collect, analyze, and display student response.

Our current design meets almost all of our basic expectations we held for SHAMBLES last semester. The basic functionality highlighted above for a classroom response system is certainly present. Node and base station communication is implemented wirelessly through a Zigbee network protocol. Base station and computer communication is implemented though USB. The batteries sufficiently power our handhelds while USB and a DC adapter will power our base station. We can charge the batteries with either USB or DC power. The keypad and LCD both function as expected and the registration/quiz software for the pc is enough to meet basic class feedback needs. The node system is small enough to be considered a true handheld. We have also stayed true to the open source nature of our project which should keep costs low. We have software prepared that should able to register nodes through USB, but as of today we do not have a USB chip on the node board and so are unable to demonstrate this ability. While we had multiple nodes join the classroom network before the demo, due to hardware failure we only had one functional node board. We were able to test three nodes at once in lab with the quiz program. They were able to connect to the base station network, receive the question messages, and send the answer messages. However, even with three, we were unable to successfully test our system under normal classroom conditions. It was impossible to really stress test our system to determine the maximum number of nodes we could handle. We could have written code to send packets as fast as possible from a node or even see how many times we could connect to the network with different identification numbers with our hardware. This would have given us a little better idea of the limitations of our system, but unfortunately during our designated testing time, the design team was focused on simply producing one reliably working node because of rampant hardware failure.

2 Project Description

2.1 System Theory of Operation

SHAMBLES consists of a node for each student and one base station for the instructor which both use Zigbee for wireless communication. At the start of class, Students must join a network managed by the base station. During class, the instructor may pose a question through the quiz software program on his or her computer. The question is transferred by USB to the base station. The base station uses the network to send the question which appears on each node's LCD. Students answers, provided via keypad, are sent back to the base station. The responses are stored in the computer and displayed as text in real-time. After the instructor closes the question, all response data is presented in graphical form.

The microcontroller controls the node and base station boards. Zigbee and USB require SPI communicate and are controlled via SPI. Zigbee and USB cannot communicate with the microcontroller at the same time and so our integration software relies on interrupts to keep the two separate.

Zigbee is a wireless communication industry standard based on the IEEE802.15.4 standard designed to be a low data-rate communication link between wireless devices. The advantage of Zigbee is that it can be easily implemented in situations requiring low-power and relatively low-cost processing. The Zigbee chipset we chose from Atmel implements the transceiver in an IC package that communicates with the microcontroller. The chip provides an inexpensive and robust design that works well in our intended environment, a large classroom. We use a limited subset of the Zigbee MAC implemented by Atmel to achieve the three basic functions that are needed to set up and maintain the network: 1) the base station announces that a network is available, 2) the nodes ask the base station to join the network, and 3) the base station allows the nodes to join and keeps track of who is present on the network, i.e. who can submit answers. The amount of data to be relayed through our network is very small: the base station sends out a question and the nodes return an answer.

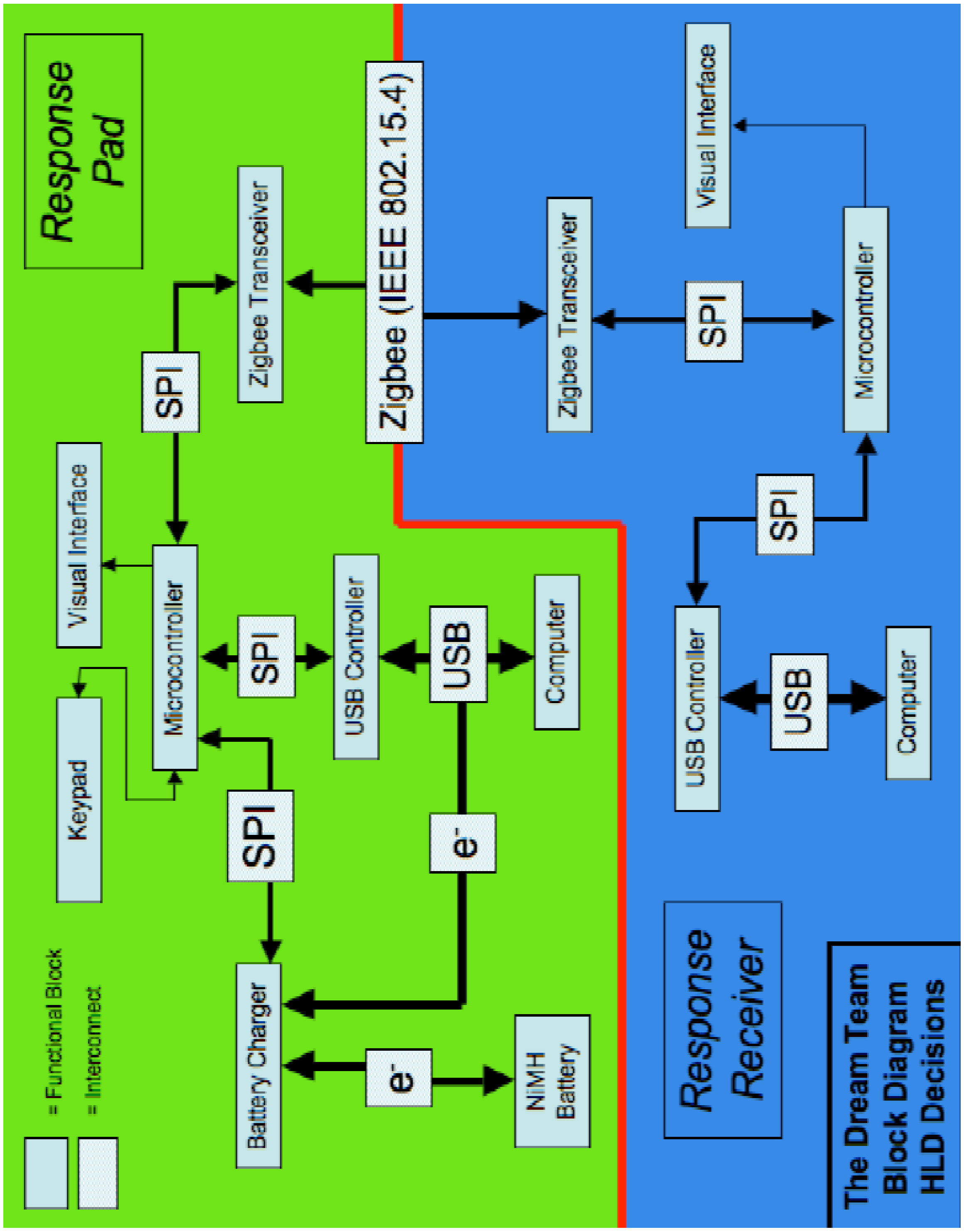
Universal Serial Bus (USB) is a serial bus standard to interface devices. The SHAMBLES devices use USB because it allows hotplugging, it can power low consumption devices, the standard is implemented by all PC vendors, its maximum bandwidth of 480 Mbits per second exceeds our requirements, and most OS's come packaged with generic device drivers. A USB controller chip handles the low level USB communication protocol on the device side, using an SPI link with the micro controller to ferry data packets between the computer and device. During device enumeration, the SHAMBLES devices declare themselves as USB devices of the Human Interface Device (HID) class. The OS then hands the device to its generic HID driver, giving our software access to the raw USB data packets without writing a custom driver.

The quiz software maps pre-registered device ID's to student names and loads arbitrary questions from a user selected file. The instructor begins and ends each question period using either the menu buttons or keyboard shortcuts. A screen printout delivers real-time student responses which are timestamped and saved for later processing, as well as a graph of all user responses at the end of each question period.

The in-class student-node interface consists of a 4x4 button matrix keypad and a 128x64 Crystalfontz series graphic liquid crystal display module. A keypad is utilized to input student responses into our system. Each question and answer choice are displayed to the student on the node's LCD as well as the student's response. Node state information such as network status is also displayed. An LED backlight allows the student to see the screen clearly even in a low-light classroom. Data is sent from the microcontroller to be displayed on the LCD through an 8-bit parallel bus controlled by the Samsung KS0713 Controller.

A single cell rechargeable battery powers our nodes. DC power can be supplied from the wall through a 5V AC adapter for recharging. USB can also provide 5V power from a computer to power the base station or recharge a node battery. Along with some supporting circuitry and logic inputs from the microcontroller, a battery charge and power controller chip controls power flow for our units. The supporting circuitry is designed for separate USB and DC inputs, so that we may use either (DC takes precedence when both connected) to power the node. If a battery is connected, the chip automatically charges the battery when external power is connected and then powers the node from the battery when external power is removed. The chip implements a number of safety features including thermal voltage regulation and monitoring for both chip and battery as well as input voltage limiting and over-current protection. Since the Maxim chip only regulates voltage to 5V, a low-dropout, low quiescent current, voltage regulator chip is attached to the battery charger chip output to drop the voltage to the 3.3 V required to power the node and base station.

2.2 System Block Diagram



2.3 Microcontroller

The Atmel Mega1281 is an 8-bit microcontroller capable of sophisticated functionality in a small package. It includes 128KB of program memory, 8KB of RAM, and is clockable to 16MHz. The chip also includes a multitude of timers, counters, PWM signal generators, and UARTs. Most of our subsystems require the Serial Peripheral Interface (SPI); the Mega family of microcontrollers includes this in hardware, making it trivial to communicate with our integrated circuits. The devices can run on 3.3V logic, reducing power consumption for the handhelds. To aid our open source initiative, the Amtel microcontroller supports a free integrated development kit and C complier, making programming relatively easy. Extensive online technical support is available for Amtel microcontrollers, along with example code, allowing rapid development of new software. The choice of Mega1281 allows the system to scale if the current configuration is insufficient.

Programming the device is done through the In-System Programmer, using the STK500 and STK501 Module. AVR Studio (http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=2725) is a free Integrated Development Kit that allows developers to write in assembly or C code for their AVR. Included with this document is our code base, specifically “base station” and “node” AVR Studio project files. Atmel provides a good introduction to their software: http://www.atmel.com/dyn/resources/prod_documents/novice.pdf. The software is quite easy to use and similar to other microcontroller IDEs.

The microcontroller, of course, controls the other devices on the board. The two programs we wrote, one for the base station and one for the node, both start with initialization functions. We initialize each piece of the board, the Zigbee transceiver chip, the USB controller chip, the LCD controller, and the keypad. The main program execution then begins after the initial address for use with Zigbee has been set. The program for the base station checks for incoming packets over the Zigbee connection or the USB connection. If a packet is received over either, then the program parses the message for the type of message received and handles it accordingly. If a “question” packet is received from USB, then the question is sent out to the nodes using a MAC data frame. If an “Association Request” function is received over the Zigbee link, the base station stores the 64-bit address of the node and assigns a temporary 16-bit short address to be used with the current network. It then sends back a response to the node that it has been accepted to the network. The last type of message that is received is a data response from the node. This answer is received over Zigbee and then sent over USB to the computer. Finally, the base station sends beacons at a specified time interval to announce that a network is present and available for association.

The node program is simpler than the base station. The node program begins with initialization of all the components. It then waits to receive a network beacon from a base station. If the network beacon is received, it then tries to associate by sending an “Association Request.” If the association was successful it receives an “Association Response” from the base station. After it has joined a network, it waits for a question to be broadcast by the base station, and then waits for a response from the keypad input. Once the keypad input is received, the data is sent out immediately on the Zigbee link. A flowchart for each program is included in Section 2.4 Zigbee for clarification.

2.4 Zigbee

The Zigbee subsystem provides the wireless communication connection between the base station and the nodes. Zigbee is a wireless communication industry standard based on the IEEE802.15.4 standard. It is designed to be a low data-rate communication link between wireless devices requiring lower power than other wireless standards such as WiFi or Bluetooth. For these reasons, we chose Zigbee for our system because the devices we are designing need to get good battery life to last through a day of classes, and the amount of data needed to be transferred from nodes to the base station is very minimal (one or two bytes of data). The Zigbee chipset we chose from Atmel implements the transceiver in an IC package that communicates with the microcontroller. The chip provides an inexpensive and robust design that works well in our intended environment, a large classroom. The chip requires very little in terms of external components. All that is needed is two capacitors, a crystal, an antenna circuit, and connections to the microcontroller. The chip also has very good signal characteristics, detecting signals down to -110 dBm, providing more than enough range to cover a classroom.

The Zigbee subsystem is divided into two components, the hardware board and software program. The hardware is very simple because of our use of Atmel's transceiver chip. It is an integrated-circuit (IC) that handles the reception and transmission of packets over the RF interface. The chip implements the physical layer as defined by the IEEE802.15.4 standard. It also includes a buffer to store frames that have been received or that are in the process of being transmitted. The received frames are uploaded to and downloaded from the chip through an SPI interface. This interface is interrupt driven because of the limited space available in the chip's buffers. The chip will send an interrupt to the microcontroller to download a frame as soon as it is received. The chip also can be set to an automatic acknowledgement state that includes an address filter. Several registers in the chip are reserved to store the PANid and device addresses as specified by the 802.15.4 standard. These addresses are used for the address filter when it is in the appropriate state to save on interrupts and using the microcontroller for these tasks, which are easily handled in hardware much faster than in software. The chip also has an adjustable output power, which ranges from 0dBm to 15dBm. The transceiver chip functions by operating in one of several 'states' depending on the desired operation. The chip must be set to the transmit state when it is ready to transmit a frame. Normally it is set to the receive state while it is listening for a frame to come over the RF interface. These states are controlled by the microcontroller, and the transceiver chip returns the current state upon request.

The transceiver IC chip comprises most of the functionality of the physical Zigbee board. The other components needed are an external crystal to provide timing to the chip and to the oscillator for the RF circuitry and two capacitors for the crystal. The last component needed is an antenna circuit, which includes a balun and an antenna. The balun takes a 100 Ohm balanced load from the Zigbee chip to a 50 Ohm unbalanced antenna. We used two different antenna for our boards, and both seemed to work well. One antenna is a chip antenna and the other is an external antenna connected to an SMA connector. Both worked well enough for our application and could transmit packets across a classroom.

The other side of the Zigbee subsystem is the software written for the microcontroller. We used the API provided by Atmel for accessing the transceiver chip. The API is divided into two parts, a hardware independent Transceiver Access Toolbox (TAT) and a hardware dependent Hardware Abstraction Layer (HAL). The TAT functions are the API calls that are made from the microcontroller. These functions are contained in the tat.h and hal.h source code in Appendix 6.2.

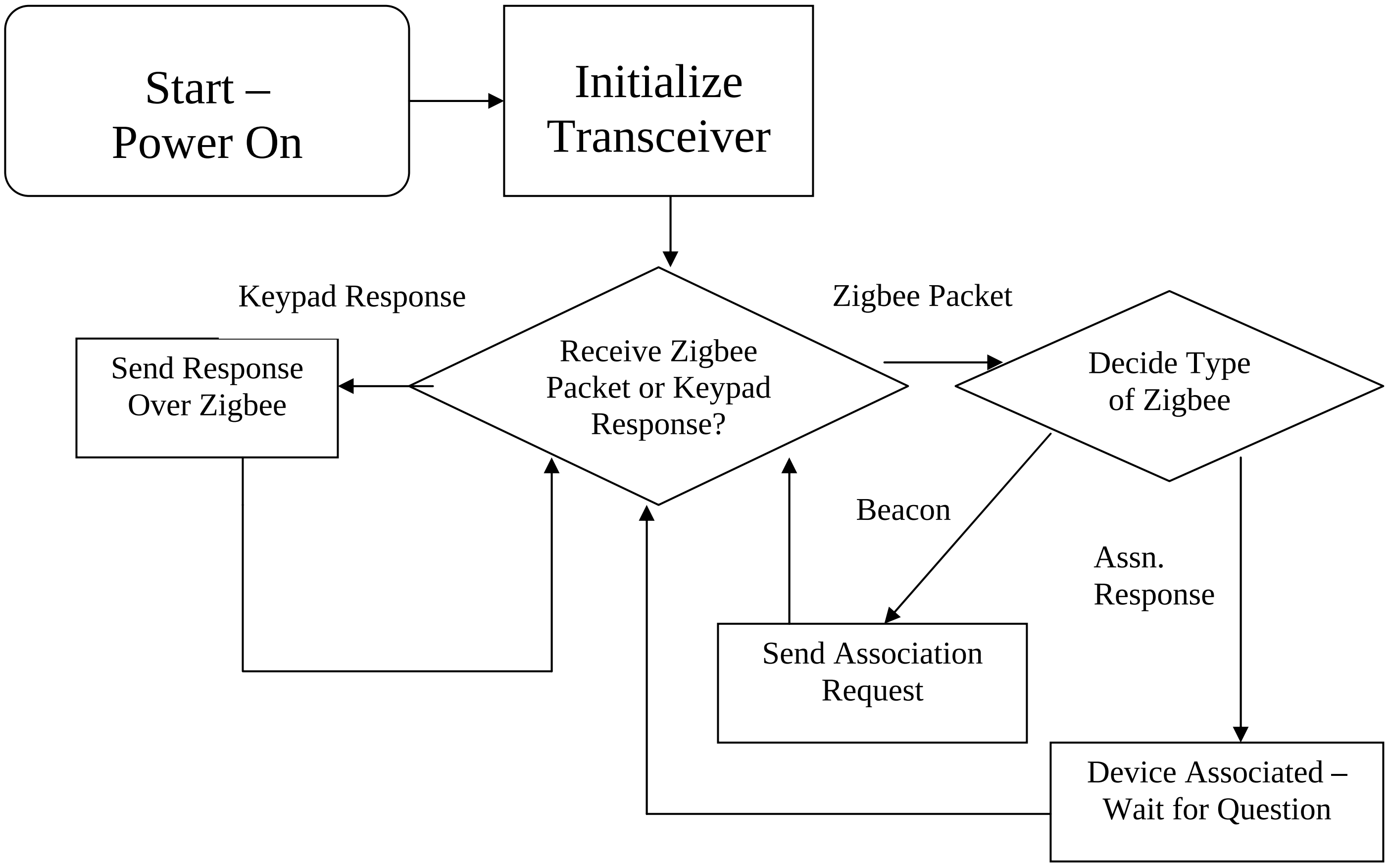
With the API functions in hand, we still had to implement a MAC protocol stack that conformed to the 802.15.4 standard. Atmel also provided one of these, but we were not able to use it as is for two reasons. First, the stack is too large and takes up too much of the data memory (RAM) on the microcontroller. Second, the MAC written by Atmel assumes the board has been wired in a specific way, which we did not follow. Therefore, we had to write a MAC layer protocol to implement the required functionality.

We use only a limited subset of the Zigbee MAC implemented by Atmel because our application does not require all the functionality provided by Atmel's Zigbee stack. This allows us to simplify the necessary programming and reduce the size of the application. The networking layer on top of the MAC is also very simple because our system only requires a small amount of overhead for set up. Three basic functions are needed to set up and maintain the network: 1) the base station announces that a network is available, 2) the nodes ask the base station to join the network, and 3) the base station allows the nodes to join and keeps track of who is present on the network, i.e. who can submit answers. Each of these functions is done through a different type of frame. The network announces that a network is available through beacon frames transmitted at specified intervals. The nodes and base station establish communication through set-up messages sent using MAC command frames. Finally, the data is sent using MAC data frames. The details of these frames are specified in the standard and explained in the code attached in Appendix 6.2.

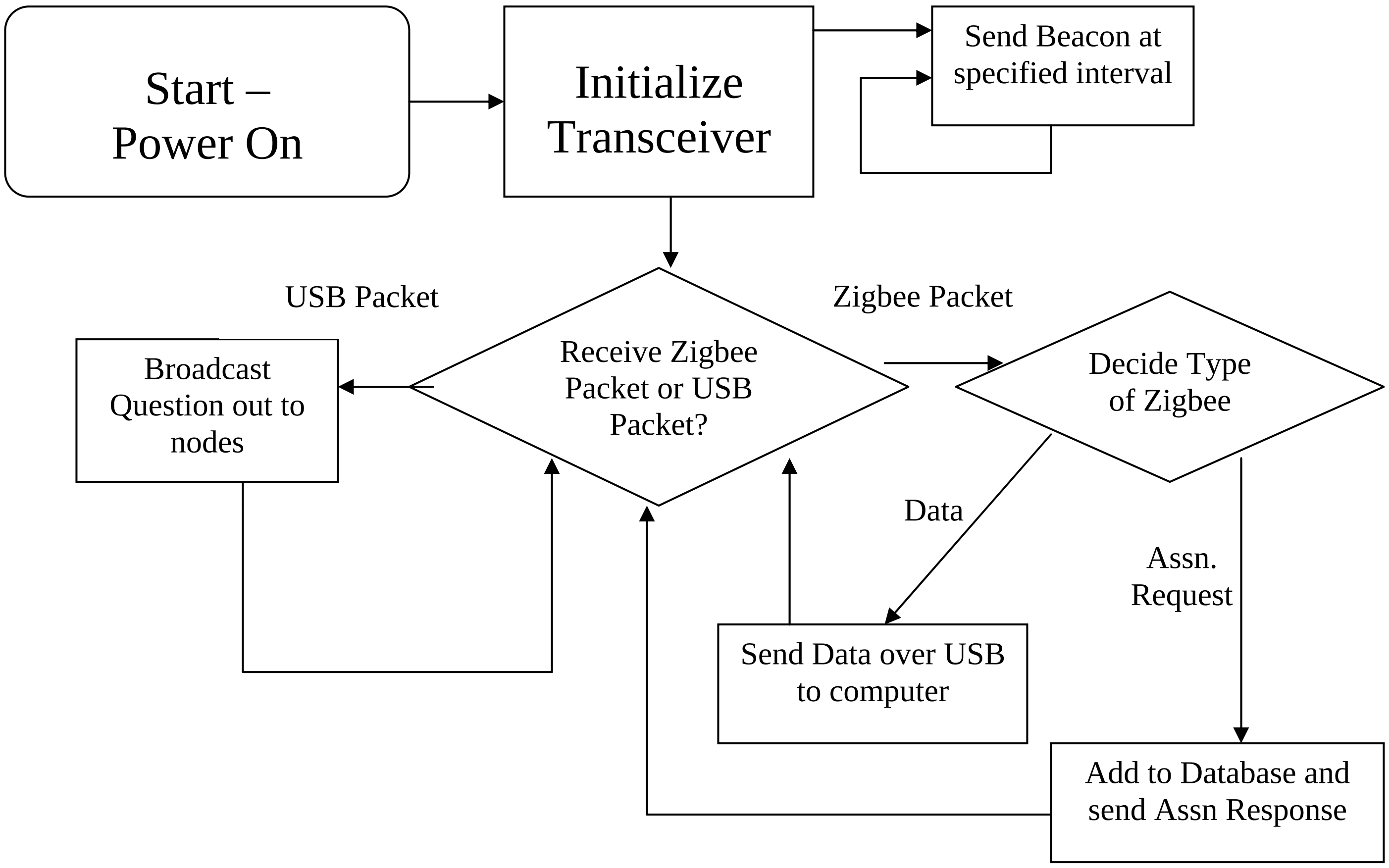
The overall flow of the program is simple and straightforward. The first step is to initialize the transceiver to put it into a well know state. This ensures the receive path is set up to receive and transmit at the appropriate frequencies. The subsequent details for the node and base station are specified under Section 2.3 Microcontroller. The Zigbee functionality is mostly contained in the file mac_node.h, mac_node.c, mac.h, and mac.c. These files contain functions that create the required MAC header for the frames. The program then fills in any subsequent network layer headers and payloads depending on the type of frame being sent. Whenever a frame needs to be sent, the appropriate function is called to build the frame. The program then fills in the information passed to it that needs to be sent. Upon reception of a frame that passes the built address filtering, the program parses the frame to determine the frame type, the information that it contains, and any response that needs to be sent back. It then performs the required processing and alerts the user if need be and replies to the sender if required. The flowcharts for the base station and node are given below.

The development environment we used for the programming is AVR Studio from Atmel. It uses a gcc compiler, so all the code is written in C. The development environment provided an easy way to write the programs, compile it specifically for our microcontroller, and load the program onto the microcontroller.

Handheld Node Flowchart – Program Flow



Base station Flowchart – Program Flow


Start – Power On
Initialize Transceiver
Send Beacon at specified interval
USB Packet
Zigbee Packet
Receive Zigbee Packet or USB Packet?
Broadcast Question out to nodes
Decide Type of Zigbee
Data
Assn. Request
Send Data over USB to computer
Add to Database and send Assn Response

2.5 Universal Serial Bus

The Universal Serial Bus subsystem constitutes the link between the base station and the computer. Through it, the base station updates the computer with network changes and node responses. The computer uses it to notify the base station of illegal nodes and of the current state of the quiz software.

The Universal Serial Bus subsystem interfaces with the microcontroller subsystem and the computer software subsystem. The physical interface between the USB subsystem and the microcontroller is established with the SPI link. The software interface between the USB subsystem and the microcontroller is carried out through the use of global variables. The software interface between the USB subsystem and the computer software is the libusb c USB library and the Python PyUSB wrapper for libusb.

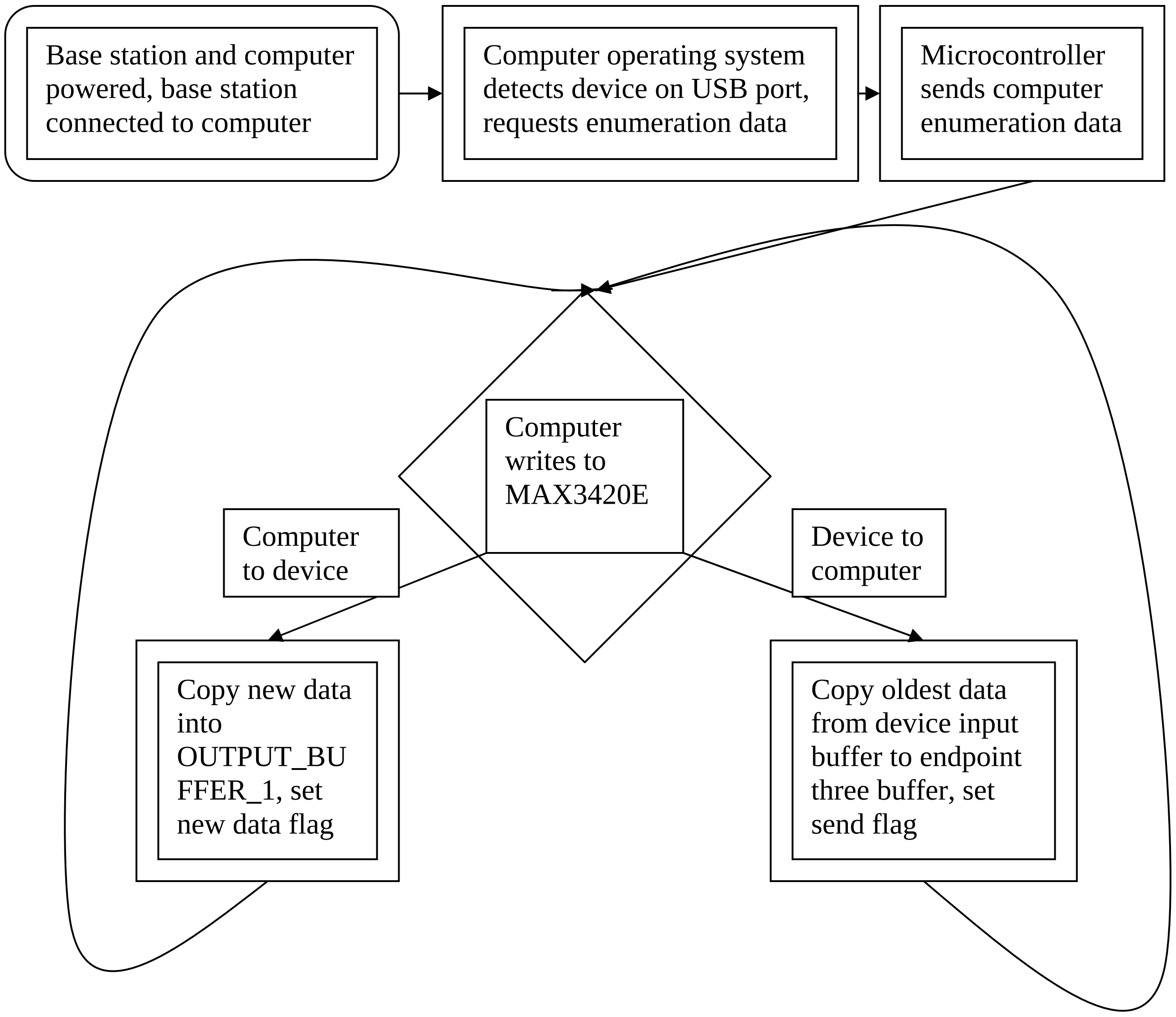
To test the correct operation of the USB subsystem, a test program was written, with software for both the microcontroller and computer, which would continuously transmit data between the microcontroller and computer. The data was manipulated on both the computer and microcontroller, and the payloads were verified on both ends. The system averaged forty minutes of uptime before an error occurred.

The chip chosen for the USB subsystem is the Maxtor MAX3420E. It implements a pure USB device (not a serial port over USB). It handles all of the low level USB protocol, including line activity detection, Host NAK and ACK communications, and checksums. The chip then transfers the raw data to the microcontroller over the easy to program SPI interface.

The USB code on the microcontroller is driven by the state of the input pin PB4. When the MAX3420E chip has a new message for the microcontroller it drives PB4 low. The main loop in the microcontroller contains and if statement that checks the state of PB4. When PB4 is low, a handling function is called. This function reads the IRQ buffer of the USB chip. If setup data is available, the microcontroller reads in the data and makes the appropriate changes to the device setup. If setup data is requested by the host, the controller loads endpoint zero with the specified enumeration data and toggles a flag in the USB chip register signaling that the data is ready to be transmitted. If output data has arrived in endpoint 1, the microcontroller reads the data into a 65 byte global buffer and sets a global variable the indicates to the main loop that new data has arrived. If an input request has been made on endpoint three, the microcontroller shifts the next message to be sent from the circular global buffer into the output buffer on the USB chip.

To send data from the microcontroller to the computer, the function device_to_computer() is called. The first argument is the pointer to the first byte to send and the second input is the number of bytes to be loaded. The function returns zero if the load was successful and one if the global circular buffer is full. Since USB is host driven, this function only loads the data into a holding buffer, the data is not sent to the computer until the computer requests more data from the device.

To receive data from the computer, the global flag OUTPUT_1 is checked. If this flag is non-zero, then the buffer OUTPUT_BUFFER_1 contains new information from the computer. After reading this data, the flag should be set to zero


Base station and computer powered, base station connected to computer
Computer operating system detects device on USB port, requests enumeration data
Microcontroller sends computer enumeration data
Computer writes to MAX3420E
Computer to device
Device to computer
Copy new data into OUTPUT_BUFFER_1, set new data flag
Copy oldest data from device input buffer to endpoint three buffer, set send flag

2.6 Power

We chose the single cell Powerizer Polymer Li-Ion rechargeable battery to power our nodes. The battery more than meets our voltage and current requirements while still conforming to the desired handheld form factor size limitations. The maximum charge rated at 1050 mAh is sufficient to power nodes for up to 10 hours of continuous use. DC power can be supplied from the wall through a 5V AC adapter for recharging. USB can also provide 5V power from a computer to power the base station or recharge a node’s battery.

Along with some supporting circuitry and logical inputs from the microcontroller, Maxim's MAX8677A 1.5 A Dual-Input USB/AC Adapter Charger and Smart Power Selector controls power flow for our units. The supporting circuitry is designed for separate USB and DC inputs, so that we may use either (DC takes precedence when both connected) to power the node. If a battery is connected, the MAX8677A automatically charges the battery when external power is connected and then powers the node from the battery when external power is removed. The chip implements a number of safety features including thermal voltage regulation and monitoring for both chip and battery as well as input voltage limiting and over-current protection.

Since the Maxim chip only regulates voltage to 5V, the MCP1700 3-Pin SOT 23 low-dropout, low quiescent current, voltage regulator chip is attached to the Maxim chip output to drop the voltage to 3.3V. This voltage regulator output powers the entire board.

The supporting circuitry includes capacitors, resistors, and a voltage regulator chip. Decoupling capacitors are placed on the battery, DC input, USB input, SYS power output, VL output, as close as possible to the MAX8677A chip to eliminate any AC signal superimposed on the DC power (and logic) lines. A charging timer capacitor sets the fast charge and prequal fault timers which are involved in a safety function that will not let the battery charge for too long. Logic output pull-up resistors are used to ensure that inputs to logic systems settle at expected 3.3 V logic levels. A thermistor is used to monitor the battery or system temperature. Charging is suspended when the thermistor temperature is out of range. A pull up resistor equal to the thermistor resistance is also used. Resistors are used to set the DC input current limit as well as the fast charge current. The MCP1700 3-Pin SOT 23 low-dropout, low quiescent current, voltage regulator chip will be attached to the SYS output to drop and regulate the voltage at 3.3 V. The voltage regulator output will power the entire board. The voltage regulator circuit requires input and output bypass capacitors. Please refer to schematic in Appendix 6.2.

Logic inputs from the microcontroller include charge enable, USB suspend, and maximum USB input current (either 100mA or 500mA). These logic inputs will be controlled by logic output pins on the USB chip (which are in turn controlled by the microcontroller through SPI). Logic outputs include active low fault (battery timer expires before charging is complete), USB power, DC power, charging, and charge complete indicators. Outputs will be indicated by LEDs.

The MAX8667A implements a number of safety features including thermal voltage regulation and monitoring for both chip and battery, input voltage limiting and over-current protection.

The chip will be controlled through the general purpose input and output pins on the USB controller chip to conserve microcontroller pins.

The MCP1700 voltage regulator is tested first by measuring output voltage. and current. The output should be a constant 3.3 V. The voltage regulator connected to the SYS output and testing of the MAX8766A begins by measuring voltage and current from the SYS , logic, and voltage regulator outputs with 5 V DC power connected through an AC adapter but without a battery. A battery will then be connected along with DC power. The system will be monitored while the battery charges. SYS, logic, and voltage regulator output voltages and currents are measured along with the battery voltage. Charging is suspended using the logic input charge enable. After the battery is charged (checking automatic charge stop), DC power is removed and the system voltages and currents are be monitored. Next, USB is tested. All input power configurations is tested including USB, USB and battery, USB and DC, and lastly USB, DC, and battery. All logic inputs are tested. Before any power input is connected to the chip, the voltage is measured. All results are matched with the operating characteristics given in the specification sheet. After the system is built, microcontroller control of the logic inputs and power to other subsystems are tested.

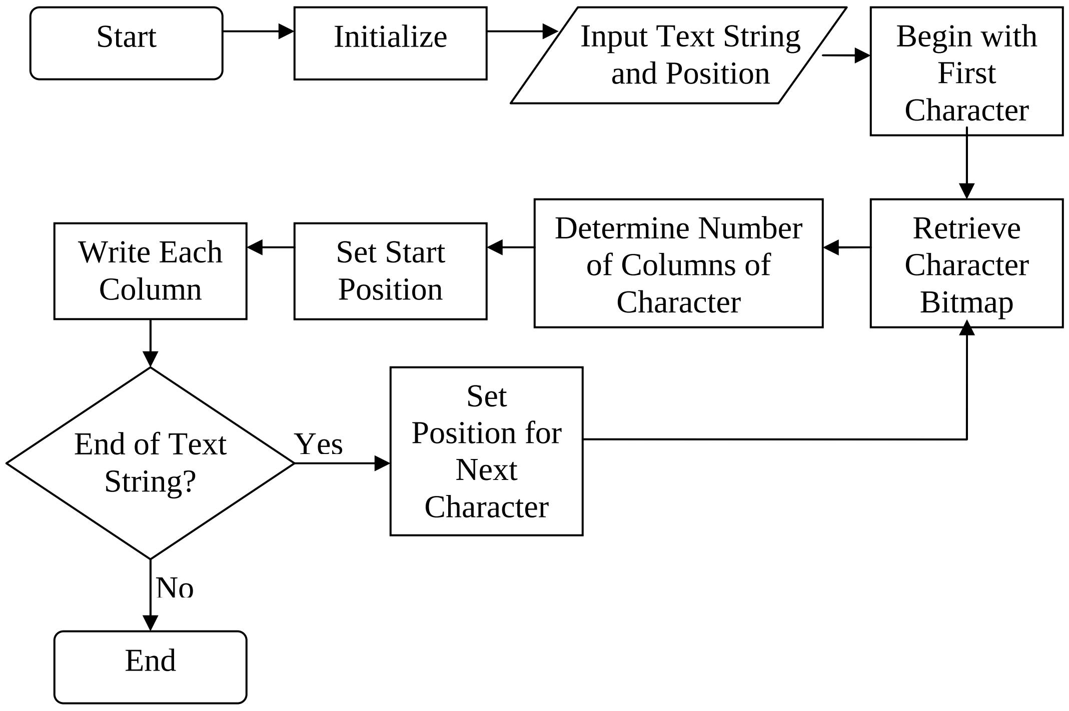
2.7 Liquid Crystal Display

The in-class student node output interface consists of a 128x64 Crystalfontz graphic liquid crystal display module. The LCD was chosen for its small size and the flexibility of its high resolution display. Each question and answer choice are displayed as well as the student's response. Node state information such as network status is also displayed. An LED backlight allows the student to see the screen clearly even in a low light classroom.

There is no real hardware behind this subsystem besides the LCD, its controller, and the connections from the microcontroller. This subsystem is dominated by software. First a program is used to generate bitmaps for each character that is to be displayed on the LCD. There is a long initialization procedure which sets the pins necessary to turn on the LCD and chooses various settings such as contrast and data entry type. We chose to use parallel instead of series data entry and positive enable. We used a string type of structure to hold the information to pass to the LCD. Functions were written to display an entire string, setting the starting position (column is automatically updated when an byte is written), that is the column and the page (8 pages for 64 individual rows), justify text, clear the entire LCD, clear one page, write small and large characters, find the length of a character, write bytes, and read an LCD output byte.

In order to display text on the screen, both the starting position and the text must be inputted. A function takes these inputs and matches each character of the string to a bitmap image. The text writing function loops through a call of the write character function for each character in the string. The write character function finds the length of the character and then loops through a call of the write byte function for each byte column of the character bitmap. The write byte function prepares the LCD for data write and then writes one byte along a parallel bus. Setting the starting position is easily accomplished although there are some considerations which must be taken into account for our LCD. For example, the first 3 columns do not appear on the screen. So a column call of zero is translates to a call to column 4. Clearing pages and columns are simply accomplished by writing blank bits and updating the column and page. Justification of text includes several options. This is calculated based on screen size and the length of the text string in pixels. Writing bytes is simply loading the 8 bit parallel bus with your byte to display and having the correct chip selects. Data is written on the positive edge of an enable bit. Reading a byte is similar, but the microcontroller's pins must be set to input instead of output. Again, a certain chip select arrangement will result in the LCD outputting status data. This was written but is not actually used in our code. It was used sparingly initially during debugging. Please refer to lcdfunctions.c in Appendix 6.1.

LCD Flow Chart - Writing Text to LCD


Start
Initialize
Input Text String and Position
Begin with First Character
Retrieve Character Bitmap
Determine Number of Columns of Character
Set Start Position
Write Each Column
End of Text String?
Yes
Set Position for Next Character
No
End

The keypad interfaces with both the microcontroller and the user. It is one way a student may get the necessary question and answer choice information from the response system. It also informs the user of the node status. The information is carried along logic lines from the microcontroller. LCD control is performed by various logic bits from the microcontroller. Data is sent from the microcontroller to be displayed on the LCD through an 8-bit parallel bus controlled by the Samsung KS0713 controller.

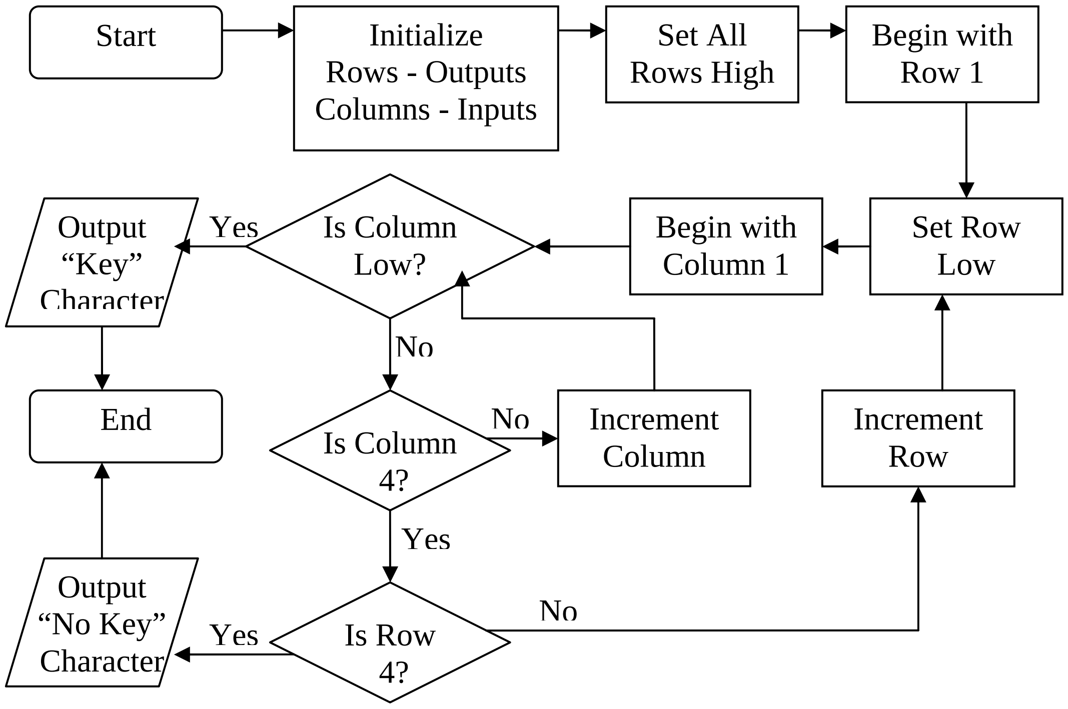
The system is easily tested by various means. We displayed text at all positions of the LCD. If you can read the text, the LCD works. We implemented both large and small font. We also tested the built in functionality such as all pixels on and all pixels off.

2.8 Keypad

The in-class student-node input interface consists of a 4x4 button matrix. A keypad is utilized for student response because it provides the necessary input functionality in a familiar and compact fashion. With out keypad, a student can answer true and false, multiple choice, and numerical type questions. Keypad buttons include numerals, plus and minus signs, scientific notation, and four programmable soft buttons. Key depression is detected by periodically polling the keypad with the microcontroller. An external keypad is attached to the first revision of the node board shown, but the final node revision has a keypad implemented with buttons directly onto the PCB. The change was made in order to decrease the size for a handheld form factor.

The hardware of the keypad is a simple matrix of 16 keys. Each row and each column of keys are connected to each other. A button press unites row and column. (See Board schematic) The C++ software programmed into the microcontroller uses this functionality to isolate each key. The microcontroller first sets the rows to outputs and columns to inputs. In this way, the microcontroller can set the rows and read the columns. We use pull-up resistors on the columns to make sure the values do not float and are pulled high unless otherwise grounded. The microcontroller holds each row high but one, which is held low. Then it reads each column to see if it is low. If a low is detected, you know a key has been pressed. Each key is an index in a character matrix which correspond to the numerals 0-9 as well as T, F, and A-D. The character can then be output to the node program. The code, mainly several functions in lcdfunctions.c, is attached in Appendix 6.1.

Keypad Flow Chart - Input Via Keypad



The keypad interfaces with both the microcontroller and the user. It is how the student enters input into the system. The information is carried along logic lines to the microcontroller.

The system is easily tested via the LCD. The characters associated with each key may be programmed to appear on the LCD when each key is pressed.

2.9 Computer Software

The computer software subsystem implements the quiz, as well as displays the results to the professor and stores the results for later manipulation.

The computer software interfaces with the USB subsystem on the base station. The software uses the PyUSB wrapper to the c USB library libusb. PyUSB uses the operating system's generic HID drivers transparently to the user because the base station declares itself as a HID class USB device. The interfaces let the computer software treat the base station as a stream input and output device.

The software was tested by running through many practice quizzes. The one working node was allowed to act as many different nodes to test the bar graph generating routine.

The communication protocol between the computer and base station is as follows:

Computer to Base station:

User not in class
[0][2 byte local user address]

Question begin
[1][Question length in bytes (max 62)][Question string (including null char.)]

Question end
[2]

Base station to Computer

User has registered
[0][64 byte ID][2 byte temp ID]

User has answered
[1][2 byte user temp ID][answer]

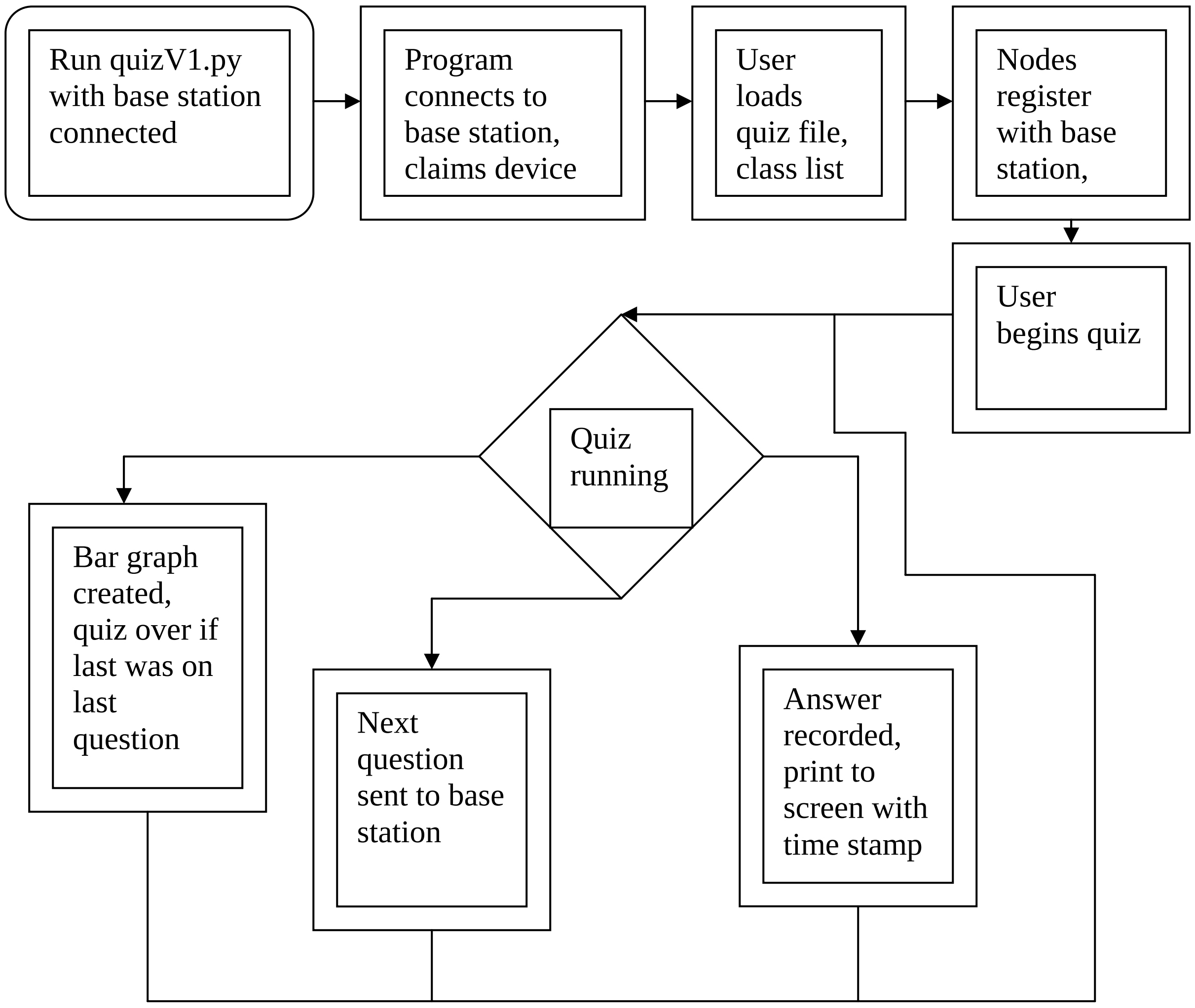
When the computer software, quizV1.py is run, a window is created. Next, the user tells the program to connect to the base station. The program searches the USB bus for a base station device and returns a handle to it if one is found. Then the program claims the base station and configures it for normal operation. The user then loads a quiz file and class file. The quiz file defines the questions and answers to be broadcast to the nodes. The class file contains the listing of student first and last names and 64 bit permanent IDs. Finally, the quiz is begun which puts the program in quiz mode and sends the first question to the base station. In quiz mode, if a question is in progress, user responses are displayed on the window. The question is ended by pressing 'e' which generates a bar graph of the student responses. If no question is running, the next question of the quiz is asked by pressing ‘n’. When the last question of the quiz is ended, the quiz is automatically ended. The screen output can be saved to a text file at any time by pressing ‘s’.

The format for the class list file is as follows:

Each line contains [student ID]:[student first name]:[student last name]

The format for the quiz file is as follows:

Each line contains [question (max 62 chars)]:[answer1]*[answer2]*….[answerN]


Run quizV1.py with base station connected
Program connects to base station, claims device
User loads quiz file, class list
Nodes register with base station,
User begins quiz
Quiz running
Bar graph created, quiz over if last was on last question
Next question sent to base station
Answer recorded, print to screen with time stamp

3 System Integration Testing

3.1 Testing

Testing of the system will consist of a quiz in which all basic functionality is demonstrated. A node joins a network managed by the base station. A question from the GUI is transferred from computer via USB to base station and from base station via Zigbee to node. The node displays the question on the LCD. A user inputs an answer on the keypad which is then transferred back to the computer where it is stored and displayed in text with a timestamp. After the question is closed in the GUI, the results are then displayed in graphical form. Power of the system is tested throughout the quiz. This functionality was tested with three nodes in the lab, but we demonstrated just one node on demo day.

3.2 Meeting Requirements

Our quiz software testing shows that our design fulfills all basic functionality-related product requirements, but on demonstration day we were only able to demonstrate one node. The base station was successfully powered through USB and a DC adaptor. The node was powered by a battery which was rechargeable through USB and DC power. The node is a handheld. USB and wireless communication, keypad input, LCD and computer graphics output were all demonstrated. On demonstration day, we were able to set up a Zigbee network, send an arbitrary question from the computer wirelessly to the node, display the question on the node, send an answer back to the computer, store the answer and display it in graphical form. The basic quiz functionality which would be used in a classroom was successfully demonstrated.

Because we only had three/one nodes to test with, it was impossible for us to simulate actual large classroom usage where hundreds of students would be attempting to connect to the network and sending answers back to the base station simultaneously. We were planning on developing some code which would transmit answers and attempt to connect to the network as fast as possible, but we did not have time. The only other requirement that was not satisfied was online registration of the nodes for a class.

4 User Manuals

Users / Installation Manual

The system consists of three parts: a node, a base station, and the computer software. The computer requires Python to be installed. Some familiarity of Python is required. This includes:

python 2.5 (http://www.python.org/download/releases/2.5/) or greater,
pylab (http://matplotlib.sourceforge.net/) module,
libusb (http://libusb.wiki.sourceforge.net/) version 0.1.12,
PyUSB (http://pyusb.berlios.de/) module, latest version,
wxPython (http://www.wxpython.org/) module, latest version.

And of course our software, Quizv1.py, included in the Appendix.

The hardware is easy to setup and use. Take a USB B-connector cable and plug it into the base station's B receptacle. Requires an external 6V DC source into the power jack. With or without Quizv1.py running, plug the other end of the USB cable into the computer. If the computer does not acknowledge with a tone, please check the power of the board, or try a reset. If problems persist, there may have been a hardware malfunction. After successful recognition of the board, run Quizv1.py, if it not already running.

After Quizv1.py loads, go to File -> Connect to the device, or use the hotkey c. This connects the software to the base station. After this, load a class list, using the hotkey o or File -> Load a class. An example class list is included in the Appendix. At this point, node devices may be turned on, and can be turned on at any time after this has been done. They will automatically connect themselves to the base station and the network. If this is not acknowledged on the LCD screen, check the connection to the Zigbee board. If there is no text on the LCD screen, make sure the screen is securely connected to the ZIF socket. Text should appear on the screen as soon as the device is turned on. If problems persist, the battery may require a charge. Simply plug the node into any computer and the battery will be charged.

After nodes join the network, this is acknowledged in the software window. After a satisfactorily number of nodes have been added to the network, press Quiz -> Load a quiz, or use the hotkey L. An example quiz listing is included in the Appendix. Then start the quiz with Quiz -> Begin quiz, or press the hotkey q. The first question will be pushed to each node's screen, with possible answers. The 12 white press buttons are laid out like a telephone pad; nodes press the button corresponding to the number of their response. The base station will receive all the responses and record multiple presses by each node. After pressing e, or Quiz -> End current question, the question is closed out and a bar graph is printed of node responses to each answer. Only the last response of each node will be used to generate the bar graph. Press Quiz -> Next question, or hotkey n to move on to the next question in the quiz list. You can save the output of the software screen at any time to a text file by going to

File -> save. Quit the software by going to File -> quit.

5 Conclusions

The Dream Team senior design project, SHAMBLES, is a certain success. We have designed a working prototype of an inexpensive, open-source classroom response system. SHAMBLES is currently several iterations from a usable product, but much progress has been made and the basic functionality has been thoroughly demonstrated. Scaling has not been tested, but the support of several hundred nodes was a key factor in hardware and software design. Along the way to producing SHAMBLES, we have developed a valuable open-source hardware platform which includes Zigbee wireless and USB communication, LCD text output, Keypad entry, and rechargeable battery power capability. Hardware reliability proved to be the most difficult aspect of this project. We are proud of the work that has been accomplished this year in senior design. SHAMBLES is, for us, the design experience integral to any electrical engineering education.

6 Appendices

6.1 Board Schematics
All board schematics are available on the team's senior design website in the EAGLE format.

6.2 Code
All microcontroller code is available on our website. Included here is computer-side code for running the base station.

6.2.1: quizv1.py

```
import wx
import time
import thread
import usb
import sys
import pylab
import string

ID_EXIT = 1
ID_OPENCLASS = 2
ID_CONNECTUSB = 3
ID_SAVE = 4

ID_STARTQUIZ = 5
ID_LOADQUIZ = 6
ID_ENDQUESTION = 7
ID_NEXTQUESTION = 8

ID_TEXTPANEL = 9

class mainFrame(wx.Frame):

    def __init__(self, parent = None):

        wx.Frame.__init__(self, parent, -1)

        self.CreateStatusBar()

        fileMenu = wx.Menu()
        fileMenu.Append(ID_CONNECTUSB, "&Connect to Base station", "Establish USB
link with base station")
        fileMenu.Append(ID_OPENCLASS, "&Open Class", "Load class listing")
        fileMenu.Append(ID_SAVE, "&Save Output", "Write text to file for later parsing")
        fileMenu.Append(ID_EXIT, "&Exit", "Close Program")
        quizMenu = wx.Menu()
        quizMenu.Append(ID_LOADQUIZ, "&Load Quiz Questions", "Load a quiz from
file")
        quizMenu.Append(ID_STARTQUIZ, "&Begin Quiz", "Begin current quiz")
        quizMenu.Append(ID_ENDQUESTION, "&End Question", "End response period
for current question")
        quizMenu.Append(ID_NEXTQUESTION, "&Start Next Question", "Ask students
next question in quiz")


        menuBar = wx.MenuBar()
        menuBar.Append(fileMenu, "&File")
        menuBar.Append(quizMenu, "&Quiz")
        self.SetMenuBar(menuBar)

        wx.EVT_MENU(self, ID_CONNECTUSB, self.onConnectUsb)
        wx.EVT_MENU(self, ID_OPENCLASS, self.onOpenClass)
        wx.EVT_MENU(self, ID_EXIT, self.onExit)
        wx.EVT_MENU(self, ID_STARTQUIZ, self.onStartQuiz)
        wx.EVT_MENU(self, ID_LOADQUIZ, self.onLoadQuiz)
        wx.EVT_MENU(self, ID_ENDQUESTION, self.onEndQuestion)
        wx.EVT_MENU(self, ID_NEXTQUESTION, self.onNextQuestion)
        wx.EVT_MENU(self, ID_SAVE, self.onSave)

        self.textPanel = wx.TextCtrl(self, ID_TEXTPANEL, style = wx.TE_MULTILINE |
wx.TE_READONLY)

        wx.EVT_CHAR(self.textPanel, self.onChar)
        wx.EVT_CLOSE(self, self.onExit)

        self.SetTitle('DreamTeam Quiz')
        self.Show(True)

    def onChar(self, event = None):

        if event.GetKeyCode() == ord('e'):
            self.onEndQuestion()
        elif event.GetKeyCode() == ord('n'):
            self.onNextQuestion()
        elif event.GetKeyCode() == ord('l'):
            self.onLoadQuiz()
        elif event.GetKeyCode() == ord('q'):
            self.onStartQuiz()
        elif event.GetKeyCode() == ord('c'):
            self.onConnectUsb()
        elif event.GetKeyCode() == ord('s'):
            self.onSave()
        elif event.GetKeyCode() == ord('o'):
            self.onOpenClass()

    def onSave(self, event = None):

        saveFrame = wx.FileDialog(None, style = wx.FD_SAVE)
        if saveFrame.ShowModal() == wx.ID_OK:
            self.textPanel.SaveFile(saveFrame.GetPath())

    def onOpenClass(self, event = None):

        global studentList, studentListLock

        openFrame = wx.FileDialog(None, message = "Choose a Class List", )
        if openFrame.ShowModal() == wx.ID_OK:
            studentListLock.acquire()
            studentList = []
            textFile = open(openFrame.GetPath())
            for line in textFile:
                line = line[0:-1]
                temp = line.split(':')
                studentList.append(student(temp[0], temp[1], temp[2]))

            self.textPanel.AppendText(''.join(['Class loaded from file: 
',openFrame.GetPath()]))
            self.textPanel.AppendText('\n')

            studentListLock.release()


    def onExit(self, event = None):

        global parentRunning
        parentRunning = False
        self.Destroy()

    def onStartQuiz(self, event = None):

        global quizRunning, quizRunningLock, curQuestionIndex, curQuestionIndexLock, 
questionList, questionListLock, questionInProgress, questionInProgressLock

        quizRunningLock.acquire()
        if not quizRunning:
            curQuestionIndexLock.acquire()
            questionListLock.acquire()
            questionInProgressLock.acquire()
            quizRunning = True
            questionInProgress = True
            curQuestionIndex = 0
            self.textPanel.AppendText('Quiz has begun\n')
            self.textPanel.AppendText(''.join(['Question ', str(curQuestionIndex+1), ': ',
questionList[curQuestionIndex]]))
            self.textPanel.AppendText('\n')

            curQuestionIndexLock.release()
            questionListLock.release()
            questionInProgressLock.release()
        quizRunningLock.release()


    def onLoadQuiz(self, event = None):

        global questionList, questionListLock, quizRunning, quizRunningLock, answerList,
answerListLock

        quizRunningLock.acquire()
        if not quizRunning:
            openFrame = wx.FileDialog(None, message = "Choose a Quiz List", )
            if openFrame.ShowModal() == wx.ID_OK:
                questionListLock.acquire()
                answerListLock.acquire()
                questionList = []
                answerList = []
                textFile = open(openFrame.GetPath())
                for line in textFile:
                    line = line[0:-1]
                    formatted = line.split(':')
                    if len(formatted[0]) > 62 or len(formatted[1]) > 30:
                        print 'Questions and answers can only be 62 characters long'
                        break
                    questionList.append(''.join([formatted[0], '\0']))
                    answerList.append(formatted[1])

                self.textPanel.AppendText(''.join(['Quiz loaded from file:
',openFrame.GetPath()]))
                self.textPanel.AppendText('\n')

                questionListLock.release()
                answerListLock.release()
        quizRunningLock.release()
    def onEndQuestion(self, event = None):

        global quizRunning, quizRunningLock, curQuestionIndex, curQuestionIndexLock,
questionList, questionListLock, questionInProgress, questionInProgressLock

        quizRunningLock.acquire()
        questionInProgressLock.acquire()
        if quizRunning and questionInProgress:
            curQuestionIndexLock.acquire()
            questionListLock.acquire()
            questionInProgress = False
            self.textPanel.AppendText('Response period over\n')
            if curQuestionIndex == len(questionList)-1:
                curQuestionIndex = -1
                quizRunning = False
                self.textPanel.AppendText('Quiz Over')
                self.textPanel.AppendText('\n')

            curQuestionIndexLock.release()
            questionListLock.release()
        quizRunningLock.release()
        questionInProgressLock.release()


    def onNextQuestion(self, event = None):

        global quizRunning, quizRunningLock, curQuestionIndex, curQuestionIndexLock,
questionList, questionListLock, questionInProgress, questionInProgressLock

        quizRunningLock.acquire()
        questionInProgressLock.acquire()

        if quizRunning and not questionInProgress:

            questionListLock.acquire()
            curQuestionIndexLock.acquire()

            curQuestionIndex += 1
            questionInProgress = True

            self.textPanel.AppendText(''.join(['Question ', str(curQuestionIndex+1), ': ',
questionList[curQuestionIndex]]))
            self.textPanel.AppendText('\n')

            questionListLock.release()
            curQuestionIndexLock.release()
        quizRunningLock.release()
        questionInProgressLock.release()

    def onConnectUsb(self, event = None):

        global usbThread, usbThreadLock

        usbThreadLock.acquire()

        if usbThread == None:

            busses = usb.busses()
            found = False
            for bus in busses:
                for device in bus.devices:
                    if device.idVendor == 2922:
                        myDevice = device
                        found = True

            if found == False:
                print 'USB device not found'
                usbThreadLock.release()
                return

            handle = myDevice.open()
            handle.setConfiguration(1)
            handle.claimInterface(0)

            usbThread = thread.start_new_thread(controlUsb, (handle,))
            self.textPanel.AppendText('Base station successfully connected')
            self.textPanel.AppendText('\n')

        usbThreadLock.release()



class mainApp(wx.App):

    def __init__(self):

        wx.App.__init__(self)

        self.frame = mainFrame()
        self.SetTopWindow(self.frame)
        self.frame.Show(True)
```

```
class student:

    def __init__(self, idNumber, firstName, lastName):

        self.idNumber = idNumber
        self.firstName = firstName
        self.lastName = lastName
        self.answer = None
        self.localId = None

def behind(plot):
    p = plot
    pylab.show()
    sys.exit()

def controlUsb(handle):

        global app, quizRunning, quizRunningLock, questionInProgress,
questionInProgressLock, curQuestionIndex, curQuestionIndexLock, questionList,
questionListLock, answerList, answerListLock, studentList, studentListLock,
parentRunning, parentRunningLock


        studentsPresent = []

        localQuestionInProgress = False

        localIndex = -1

        while True:

            # Check if GUI was closed
            if not parentRunning:
                sys.exit()

            # Check if GUI has ended current question
            if localQuestionInProgress == False and questionInProgress == True:
                localQuestionInProgress = True

                ###############
                # HACKERY FOR GOOD GRAPH, ONE NODE
                choices = answerList[curQuestionIndex].split('*')
                N = len(choices)
                width = 0.5
                ind = pylab.arange(N)
            answers = N*[0]
            # END HACKERY
            #################

        elif localQuestionInProgress == True and questionInProgress == False:
            localQuestionInProgress = False

##              ################
##              # LOST TO HACKERY
##              choices = answerList[localIndex].split('*')
##              N = len(choices)
##              width = 0.5
##              ind = pylab.arange(N)
##              answers = N*[0]
##              # END LOST
##              #################

            for curStud in studentsPresent:
##                  ###########
##                  # LOST TO HACKERY
##                  if curStud.answer.isdigit():
##                      index = string.atoi(curStud.answer)
##                      if index < N+1:
##                          answers[index-1] += 1
##                  # END LOST
##                  ##################
                curStud.answer = None

            plot = pylab.bar(ind, answers, width)
            pylab.ylabel('Number of student responses')
            pylab.title((questionList[localIndex])[0:-1])
            pylab.xticks(ind+width/2, tuple(choices))

            #pylab.show()
            thread.start_new_thread(behind, (plot,))



        # Check if GUI has advanced the question
        if localIndex != curQuestionIndex:

            localIndex = curQuestionIndex

            if localIndex != -1:
                #print 'Question to USB'
                text = ''.join([chr(1), chr(len(questionList[localIndex])),
questionList[localIndex], chr(0)])
                handle.interruptWrite(0x1L, text)
                #print len(text), text

                #print 'Answer to USB'


                temp = answerList[localIndex].split('*')
                text = []
                for i in range(len(temp)):
                    text.append(''.join([chr(i+49), ')', temp[i], ' ']))
                text.append('\0')
                text = ''.join(text)
                text = ''.join([chr(2), chr(len(text)), text])
                handle.interruptWrite(0x1L, text)
                #print len(text), text


        # Check for messages from base station
        buff = handle.interruptRead(0x83L, 64)
        # Ignore the nothing messages
        if len(buff) == 1:
            time.sleep(.5)
##          else:
##
##              app.frame.textPanel.AppendText('Received usb message: ')
##              for i in buff:
##                  if i > 47 and i < 123:
##                      app.frame.textPanel.AppendText(chr(i))
##                      app.frame.textPanel.AppendText(' ')
##                  else:
##                      app.frame.textPanel.AppendText(str(i))
##                      app.frame.textPanel.AppendText(' : ')
##
##
##              app.frame.textPanel.AppendText('\n')
        # Association messages
        elif buff[0] == 0 and len(buff) == 11:

            # convert tuple of longs for Perm ID to single string of digits
            tempId = []
            for i in range(1,9):
                adder = 0
                if buff[i] < 0:
                    adder = 256
                tempId.append(buff[i]+adder)
            for i in range(len(tempId)):
                tempId[i] = str(tempId[i])
                while len(tempId[i]) < 3:
                    tempId[i] = '0'+tempId[i]

            tempId = ''.join(tempId)

            # convert tuple of longs for Temp ID to single string of digits
            tempLocalId = []
            for i in range(9,11):
                adder = 0
                if buff[i] < 0:
                    adder = 256
                tempLocalId.append(buff[i]+adder)

            for i in range(len(tempLocalId)):
                tempLocalId[i] = str(tempLocalId[i])
                while len(tempLocalId[i]) < 3:
                    tempLocalId[i] = '0'+tempLocalId[i]

            tempLocalId = ''.join(tempLocalId)

            # check for student in current class list
            for curStud in studentsPresent:

                if curStud.idNumber == tempId:
                    app.frame.textPanel.AppendText(''.join(['Re-joining: ', curStud.firstName,
' ', curStud.lastName, ' ', str(time.time())]))
                    app.frame.textPanel.AppendText('\n')
                    curStud.localId = tempLocalId
                    break

            # if new member was not in class before
            else:

                for curStud in studentList:
                    if curStud.idNumber == tempId:
                        curStud.localId = tempLocalId
                        app.frame.textPanel.AppendText(''.join(['Now joining: ',
curStud.firstName, ' ', curStud.lastName, ' ', str(time.time())]))
                        app.frame.textPanel.AppendText('\n')
                        studentsPresent.append(curStud)
                        break

        # Response messages
```

```
elif buff[0] == 1 and len(buff) == 4 and localQuestionInProgress:

    tempLocalId = []
    for i in range(1,3):
        adder = 0
        if buff[i] < 0:
            adder = 256
        tempLocalId.append(buff[i]+adder)

    for i in range(len(tempLocalId)):
        tempLocalId[i] = str(tempLocalId[i])
        while len(tempLocalId[i]) < 3:
            tempLocalId[i] = '0'+tempLocalId[i]

    tempLocalId = ''.join(tempLocalId)

    for curStud in studentsPresent:
        if curStud.localId == tempLocalId:
            curStud.answer = chr(buff[3])

            ##############
            # GOOD GRAPH ONE NODE HACKERY
            if curStud.answer.isdigit():
                index = string.atoi(curStud.answer)
                if index < N+1:
                    answers[index-1] += 1
            # END HACKERY
            #################



            app.frame.textPanel.AppendText(''.join([curStud.firstName, ' ',
curStud.lastName, ' ', curStud.answer, ' ', str(time.time())]))
            app.frame.textPanel.AppendText('\n')

            break
```

```
if __name__ == '__main__':

    # Create globals for inter-thread communication
    studentList = []
    studentListLock = thread.allocate_lock()
    questionList = []
    questionListLock = thread.allocate_lock()
    curQuestionIndex = -1
    curQuestionIndexLock = thread.allocate_lock()
    quizRunning = False
    quizRunningLock = thread.allocate_lock()
    questionInProgress = False
    questionInProgressLock = thread.allocate_lock()
    answerList = []
    answerListLock = thread.allocate_lock()

    usbThread = None
    usbThreadLock = thread.allocate_lock()
    parentRunning = True
    parentRunningLock = thread.allocate_lock()

    app = mainApp()
    test = 'chim'

    app.MainLoop()
```

6.2.2 Example Class List
Add the following to.txt file:

```
035135018171216196137027:Generic:Student
78287324:Andy:Carter
7732:Sweet:Billy
1:Matt:Elliott
```

6.2.3 Example Quiz Format
Add the following to.txt file:

```
How much wood would a woodchuck chuck?:some*more*most*half
Favorite member of Dreamteam?:Ben*Matt*Andy*Bill
When the wipple will wipples shorter?:yes*no*maybe*whether?
```

6.3 Parts Inventory

Parts Inventory (Beyond assigned kit)

3 PIC18F4620 Microcontrollers(DIP)
1 ATMEGA8515L Microcontroller(DIP)
1 ATMEGA16 Microcontroller(DIP)
4 CFAX1264AP1 Graphic LCDs
2 MAX8677A Battery Chargers (QFN)
2 ATMEGA1281 Microcontroller(QFP)

An assortment of(Rs and Cs are 0603):
12MHz Crystals
16MHz Crystals
8MHz ceramic resonator
10pf cap
18pf cap
5.6pf cap
22pf cap
LCD ZIF socket
Linx 2.4Ghz chip antenna
RightAngle SMA antenna
SPST switches
SMA bulkhead connector
USB AB Micro connector
BFS17 (NPN Transistor, SOT 23)
680ohm
1.5ohm
10ohm
150ohm
3kohm
33ohm
10kohm
0ohm
10k thermistor
.068uf cap
.1uf cap
1uf cap
4.7uf cap
10uf cap
red led