GyEEm of the Future

Time is no longer a valid excuse to avoid working out.

Ignacio Aranguren
GianPaul Handal
Nicholas Yulan

Table of Contents

3 Introduction

The GyEEm of the Future is the answer to the problems of gym patrons everywhere. Many gym patrons ask themselves, “When is a good time to go to the gym?” or “Will the gym be busy right now?” Knowing if the gym is crowded is a complete guessing game. The only way to determine if the gym is crowded is to get in your car and drive there. By then you’ve already committed to going the gym and you can only hope and pray that the machines and weights are not being used by a grunting 400 pound Mr. Olympia wanna-be or a feeble 80 year old man. If health and fitness is important to you, you have just accepted the fact that there will be days in your future that you will have work out in a crowded gym. Until now…

The GyEEm of the future has the ability to inform gym members of how congested the gym is from the comfort of their home. The only thing they will have to do is type in their gyms web address in the device of their choosing and a detailed list of equipment and their availability will fill their screens. Aside from the status of the equipment there will be information available that will specify how many people are in the gym. Thirdly, once the GyEEm of the future system has been in place for enough time the website will begin to develop trends of when the gym will be busy and when it will not.

All of this information will be invaluable to many gym members. Especially those with busy schedule and those that like to plan every minute of their day. Gym patrons who do not like working out around many people will be able to find a time of the day that is just right for them. This is also applicable to serious athletes and aspiring body builders. The gym itself will also prosper from this system. The GyEEm of the future system is a feature that could help generate new clientele and perhaps be the deciding factor between gyms of otherwise equal quality.

Next we will discuss the high level description of the GyEEm of the future. There are several distinct systems that need to be discussed. These systems are the sensor system, the main hub reception system, the microcontroller system and the website system. The

interaction of one system to the next is essential to the product we are creating.

The sensor unit uses infrared motion sensors to determine the status of gym equipment. The sensor will be placed in different locations depending on the type of equipment that it is monitoring and depending on how vulnerable the sensor is to damage. The sensor will be connected to a microcontroller which will analyze the data and send it via a Zigbee transceiver to the main hub reception system.

The main hub will be receiving data packets from all of the sensors in the gym. The microcontroller of the main hub will read the data packets and addresses of the sensors and use that information to update the website.

The software aspect of our project plays a key role in taking the data that our sensors are gathering and uploading them to the website periodically. We decided to use a Cron Job because it would allow us to consistently update the website data without user interference. It is also important to be able to store all of the data that our sensors generated. We need this data to develop gym congestion trends. We decided that MySQL server would be our best option. With MySQL we are able to retain our data while also being able to access it from a remote location.

We can successfully say that we have achieved what we had expected. Our sensor works as we had intended and our reception hub is able to receive and accurately read the data packets that are sent to it. The Zigbee communication was successful and we were able to specify once a machine's availability changed. The communication between the MainHub and the laptop using the ComPort communication also worked, although we would have liked for it to be automatic.

Our website also worked as planned. We were able to setup the MySQL server. We were able to set it up for the number of sensors we constructed, which made our demo successful. For a real world application, we would have liked to include more sensors. Moreover, being able to have data to perform the analysis would have been preferable.

All of the difficult aspects of the project were accomplished. We were able to tie the entire system together and to deliver the GyEEM of the future that we had promised.

4 Detailed System Requirements

In order to have a our project functioning as we had expected. We had to be able to determine if a machine is being occupied at the gym and to transfer this information to the web. To achieve this goal there were different things that we had to do:

4.1 Sensors

Checking the availability of a machine at the gym was done using infrared sensors.To be able to physicially determine if a machine was being used we had to choose a sensor and learn how to use it. We decided to use an infrared sensor that outputs a voltage depending on how far an object is from its sensing “eye.”

4.2 Analog to digital conversion

The output that the sensor produced was analog but in order for it to be helpful in our code we had to make it digital. To complete this task we had to learn how to use the Analog to Digital conversion of the microcontroller.

4.3 Communicate through Zigbee

Once the Analog to digital conversion is completed, our next goal is to be able to transmit the message of whether a machine is being occupied or not from our sensor to the main hub. The message was sent using Zigbee protocol. To be able to send a message using Zigbee there was a protocol that we had to follow. Using SPI we had to follow a series of steps to write a frame into the Zigbee transceiver. Moreover we had to read the message on our mainHub and determine specifically what parts of the message we wanted to interpret.

4.4 MainHUB to Comport and UART

For the sake of our demonstration we decided that we wanted to have our MainHub directly connected via USB to a computer.For this communication to take place we had to talk to the ComPort on the computer using the UART. We decided that we wanted to directly talk to putty and from there create txt file with what our code was outputting to the Comport.

Another approach for this section could have been to have been to use a razzleberry pie that directly sent the data to a server where it could be used to update the website and stored for data management.

4.5 Upload of data onto MySQL

The upload of data onto MySQL to track gym usage over time requires a text file containing information to be constantly updated, according to what the different sensors are detecting. This depends principally on a connection between the sensors and the computer in which the MySQL server is managed, but also requires a series of scripts to first pull data from the serial comport (as reported by the different sensors), and then to turn that data onto a text file amenable to MySQL.

Uploading to the website

The requirements of the subsystem to upload information onto the gym website are in large part the same as the subsystems for the upload of data onto MySQL. That is, for the data to be uploaded onto the website, we need to have effective communication amongst the sensors and the main hub (connected to the computer), as well as a series of scripts to pull information and turn it into the format necessary for upload onto the web. In addition to this, however, we need a way to upload documents onto the web; to do this, we need an ftp tool (which may be built into the computer, or may be obtained via the installation of a third party ftp client such as Cyberduck), we need a server to host our web pages, and we need a series of scripts to take care of first creating the updated web page as an .html file and then uploading that file through ftp.

5 Detailed project description

5.1 System theory of operation

Our system has several subsystems that mesh together to perform a task. Our hardware system uses the sensor to detect motion within a certain range and return a voltage accordingly. The microcontroller on the sensor board essentially transmits the sensor data to the main hub via a zigbee transceiver. The main hub gathers sensor data from several sensors and uploads it to our computer.

Next we use several programs that we have designed and downloaded into the microcontroller to evaluate the voltage reading from the sensor. Once the information has been evaluated it is uploaded to the website. This is made possible using several shell

scripts that use the sensor information to modify the status of the gym equipment listed on our website.

All the data that the sensor collects is uploaded onto a MySQL server and used to create gym congestion trends.

This process happens several times a minute during the operating hours of the gym. This ensures that the patrons are informed.

5.2 System Block diagram

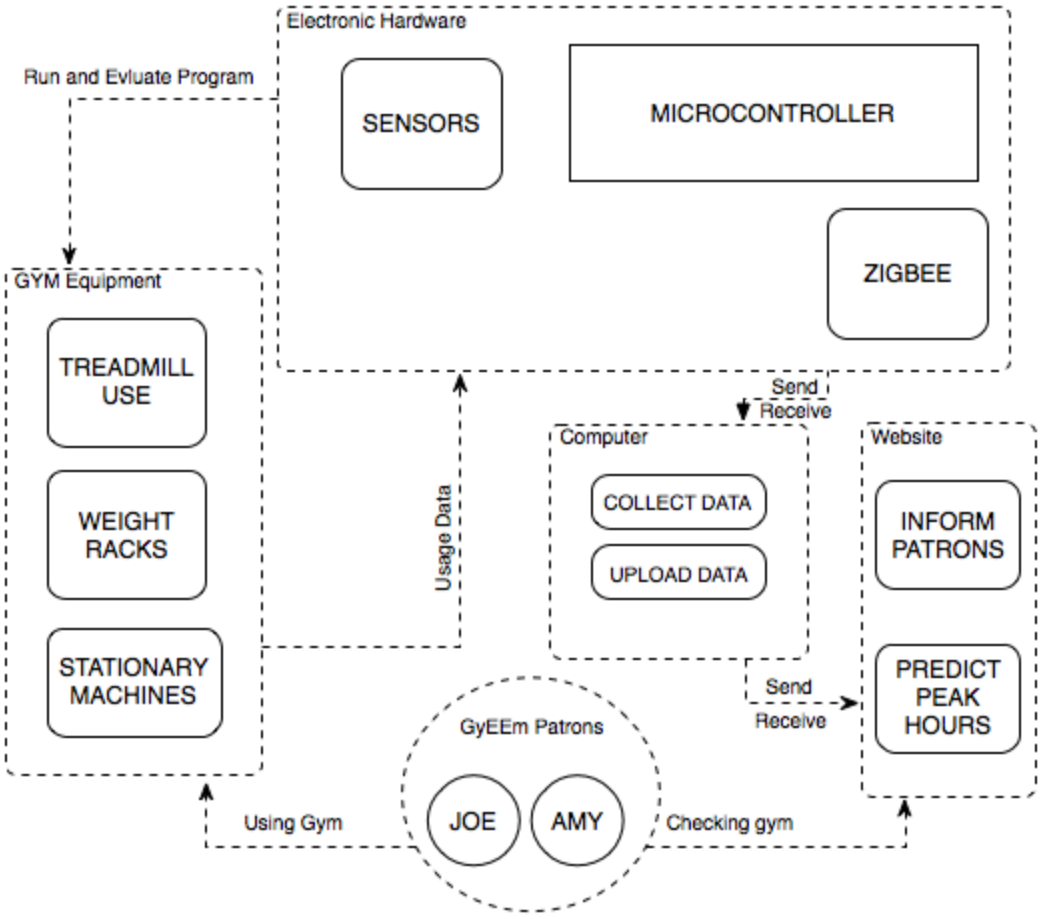


Figure 1: System Block Diagram

This block diagram shows how the different systems interact. We can see that there is a loop between the gym equipment system and the hardware. This loops ensures that the sensors are continuously updating the microcontroller with data that it can evaluate and make available to the gym users via the website.

5.3. Sensor UNIT subsystem

The sensor unit of our project consists of a microcontroller, a ZigBee Transceiver AT86RF231 and the sensor itself. The program used to analyze the information received by the sensor varies depending on the machine where the sensor is going to be attached to. Figure 1 is an example on how this subsystem works.

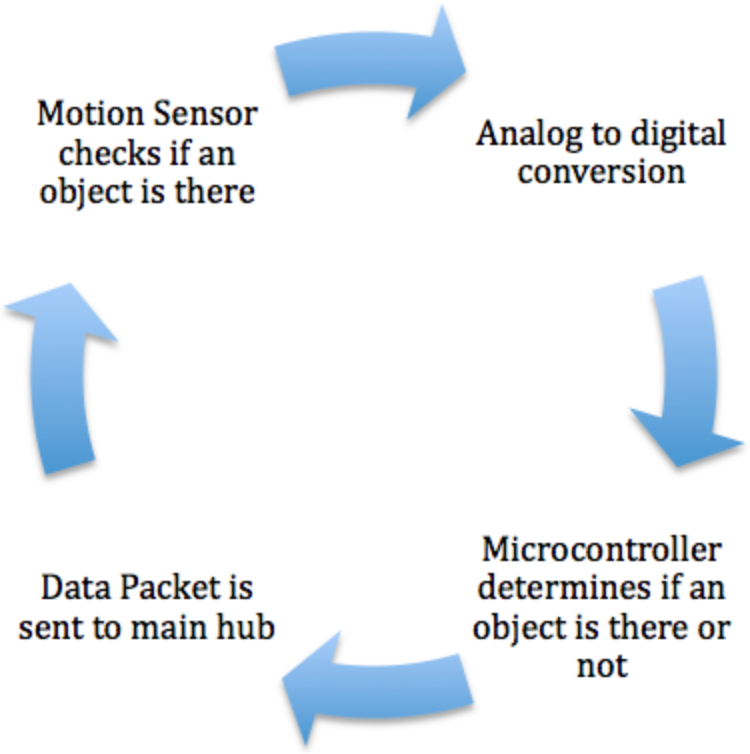


FIGURE 2: Sensor Unit Cycle

The sensor we decided to use for our project is the Sharp infrared motion sensor GP2Y0A02YK0F. We decided to use this sensor because it has the capability to detect objects that are 20 to 150 cm from it. This sensor produces an output analog signal of 0 to 3.3 V when an object is there. The optimal range for the sensor is between 10cm and 30 cm, which is more than suitable to fulfill our project specs.

The analog to digital conversion is needed to analyze the analog signal that the board receives from the sensor. This conversion is used to determine if a machine or a piece of equipment is being occupied. The digital signal after conversion was a number between 0 and 1024. This was converted using a factor of 3.3/1024 and then analyzed to know what to send through the ZigBee Transceiver. If a certain voltage was outputted by the sensor that meant that there was someone occupying the machine and therefore a

certain message was sent.

Sending the data packet to the main computer hub was one of the most complex parts of this subsystem. The data packet was sent to the transceiver using SPI. For the transceiver to send a data packet we had to write to the frame of the transceiver using 802.15.4 standards. The data packet consists on several sections. Figure 3 describes this process.

FIGURE 3: Procedure to Send a Data Packet

The first part consisted on a byte that signalizes that a packet is going to be written to the frame to be sent. The second byte or PHR consists on the number of bytes that are going to be sent. In the PHR you have to take in consideration the FCF bytes and the addresses. This is then followed by the Frame Control Field (FCF), which gives several information about the package including security, data type, destination addressing mode,etc. Then we had to specify the address of the main hub. Least but last, we actually sent our information based on the analysis that was done on the sensor information. We setup our system in order to send a data package from each machine every time there was a change of status. In simpler words, a message was only sent once a machine changed from available to unavailable.

5.3.1 Important functions

void init_adc3(void)

This function goes through the protocol of initializing the A/D conversion. The process consisted on following 12 steps in which we had to decide the analog inputs, the format of the output, the sample clock, the clock source, the Mux, etc.

void init_SPI(unsigned long Fsck)

The initialization of the is very important. In this function we make sure that we have the right SPI, set the edge and polarity clock, enable master mode and make the mode 8-bit.
unsigned char do_stuff(unsigned char data)

This is by far our most used function. This function cleared the SPI interrupt, wrote to the SPI buffer and waited for the interrupt flag to be set again. Once it was set it outputted the data it received in hex.

unsigned char write(unsigned long add , unsigned char data)

This function is an extension of the do_stuff. In this function we set the change in Chip selesct in order to write to a register. We use the do_stuff function to write a command, the address and after that we write the data we want to that specific register.

unsigned char read(unsigned add)

This function is similar to the write function. In this function, we also have to trigger the chip select, then we use do_stuff to write the read command to SPI and the address. Then dummy values are sent in order to be able to receive the data we want from that specific package.

void reset_128()

This function is used to make sure we reset our microcontroller. Also, we needed to trigger the chip select. Usually for the Zigbee transceiver to send a message reset ha to go from low to high but because or reset was set up backwards in our board, we had to move reset digitally from high to low with a delay in between.

void Map_2()

This mapping function was used because the microcontroller that we decided to place in our board was different to the one we used for testing. We had to make sure we set up the data in and the data out accordingly for SPI.

void weight_check()

This function was important because it was constantly checking the value of he sensor and doing the Analog to Digital conversion. After the Analog to digital conversion was done this function sent the value of this conversion for the function *send_frame_someone_there(int ADCValue)* to analyze.

void send_frame_someone_there(int ADCValue)

This function analyzed the value of the Analog to Digital conversion and compared it to the value before it. If there was a change in occupancy then a message is sent using the function *frame_write()*.

unsigned char frame_write(unsigned PHR, unsigned destination, unsigned source, unsigned data)

This function completed the sensor module by sending a message when a change of occupancy occurred. It is important to note that the message we sent in hex was used base on the ASCII code that will later be read by our ComPort.

5.4 ZigBee subsystem

There were many options that we could have chosen when deciding on what type of network our GyEEm of the future would run on. The first restrain that we used in selecting a chipset was price. Our project has a budget that we need to respect and ZigBee's low cost fits our budget nicely. ZigBee is less than $15 per unit. However, do not let the low cost of ZigBee mislead you into thinking that it is not capable.

Aside from its low cost, ZigBee has full 802.11 functionality and also has the ability to save a lot of power which allows for less maintenance and lower long term costs. A battery-powered node can wake up, check in, send data, and shut down in less than 30 ms. That being said the shelf life of the battery will run out before the battery capacity is used up. The low chipset cost and energy saving ability was perfect for our low cost project. Another aspect of the ZigBee that was attractive to our project was that it is very easy to integrate into our microcontroller. This saves us a lot of time in effort because we do not need to do RF engineering to implement ZigBee into our microcontroller. We can plug it right in to our design.

Lastly, ZigBee has the possibility to support extremely large networks. A larger gym will require many more ZigBee nodes. The increased number of nodes will create a mesh network that will allow data to travel through intermediate nodes to reach its destination. This will be essential for implementing the GyEEm of the future into much larger commercial gyms.

5.5 Main Hub Reception subsystem

This subsystem is highly based on his predecessor, with the difference that several messages are received. This subsystem consists of another Zigbee Transceiver AT86RF231, a pic-32 Microcontroller and a computer.

The Zigbee transceiver received a data packet in a similar fashion to how we sent one. The received data packets are written to the frame of the transceiver and this are analyzed by the microcontroller.

An interrupt service routine was set up waiting for a change on an external interrupt. Once a change was noticed,a register (TRX_STATUS) that signals that a message is being received was read using SPI and when this event occurred our microcontroller proceeded to read the frame. In order to read a frame we had to send a SPI signal that gave us access to read the frame.

Once the frame was completely read we decided to use the UART to communicate to the ComPort of a Laptop. The UART was used to talk to the Comport using Putty or the Console in our case because we decided to use a MacBook Pro. A message was written to the ComPort using the putu() function that we worked with last semester.

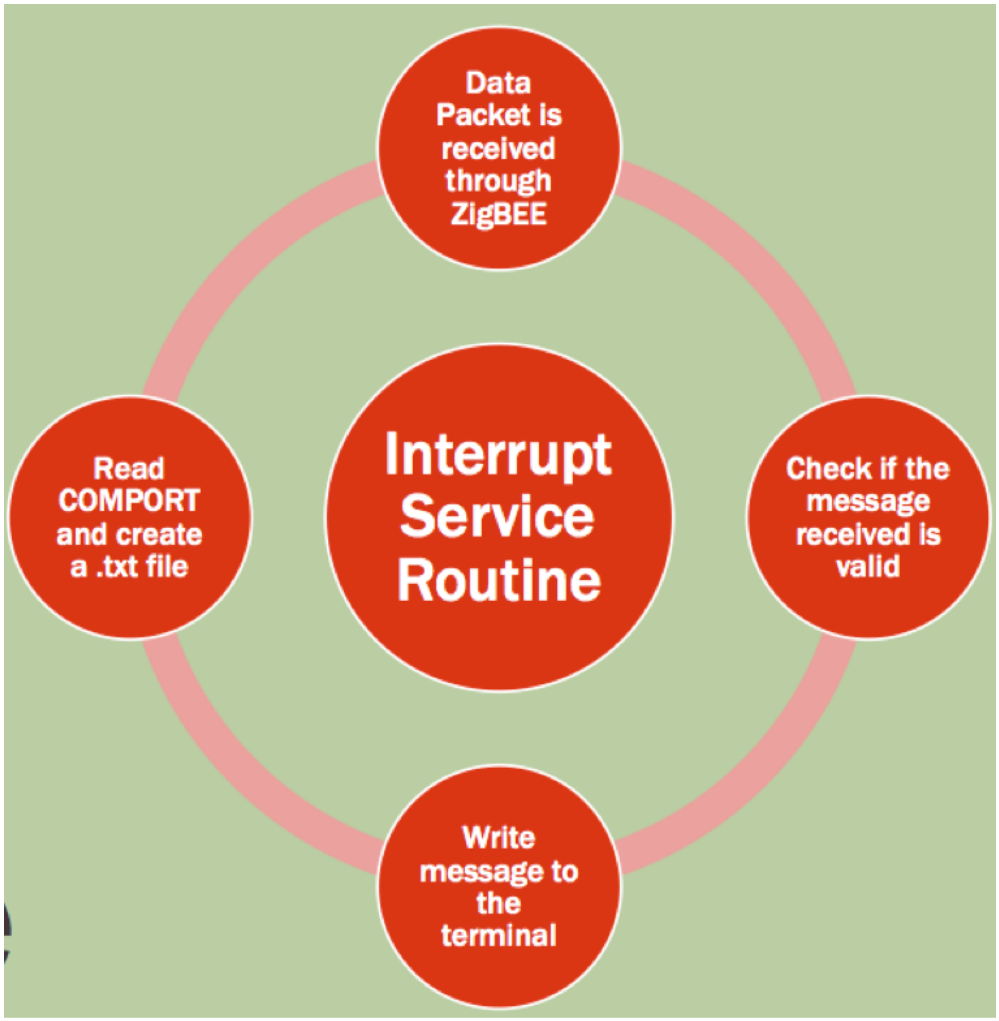


FIGURE 4: Procedure to Read a Data Packet

5.5.1 Important Functions

unsigned char do_stuff(unsigned char data)

This is the same as the function we described in the previous section for the sensor module.

unsigned char do_stuff_put(unsigned char data)

This function is a modified do_stuff function. This version of the function does the same of his predecessor nut with the capability of writing the message to the ComPort. This version directly writes the hex message to the ComPort.

void init_SPI(unsigned long Fsck)

Same function as the one in the previous module. It is needed to have the SPI working correctly.

unsigned char write(unsigned long add , unsigned char data)

Same function as the one in the previous module.

unsigned char read(unsigned add)

Same function as the one in the previous module.

void sensor_read()

This is the function that we used to read the frame that is received. The read command is sent first, the PHR is read, and then the source and data packet. The source and data packet are what is written to the ComPort to create a txt file.

void set_reception()

The purpose of this function is to prepare the Zigbee transceiver to receive a message. Inside this function we make sure that status of the TRX is RX_ready, or in simpler words ready to receive.

void check_reception()

This is the function that is used to make sure that a complete frame is received once the interrupt service routine is triggered.

void Enable_INT0(void)

This is the function needed to enable the interrupt and signal to it what to trigger for. In our specific case our interrupt is just triggered once a frame is completely received.

void __ISR(_EXTERNAL_0_VECTOR,IPL7AUTO) int0_ISR(void)

This is the most important function of this module. This routine does not go inside our main code but is accessed once our interrupt is triggered. This same routine is where we go and read the frame that is received.

void set_promiscous()

This function was needed to make sure that we received all the messages that were sent. This was done to make sure that we were not facing any problems due to protocol or security.

5.6 Software subsystem

How we dealt with each problem and why:
(1) Cron Job because it allows us to do things automatically
(2) MySQL server because it makes our data easy to manage, and because it allows us to keep all of it and to access it remotely - we want to use this data for time series analysis to forecast future GyEEm usage.

A bit of background on MySQL:
MySQL is an open source relational database management system (RDBMS) that runs as a server providing multi-user access to a number of databases. The Relational Database

Management System (RDBMS) is a DBMS based on the relational model. RDBMSs have become the predominant choice for the storage of information in new databases (for example, DBs used for financial records, manufacturing and logistical information, personnel data, and much more). The leading commercial RDBMS vendor is Oracle, which is the owner of MySQL (MySQL originally belonged to the Swedish company MySQL AB, which was acquired by Sun Microsystems in 2008, which was then bought by Oracle in 2010). Other leading commercial RDBMS vendors are: IBM, Microsoft, SAP, and Teradata. The relational Model is a database model (i.e. a model for database management) based on first-order predicate logic, first formulated and proposed by Edgar F. Codd. In this model, all data is represented in terms of tuples, grouped into relations. The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the DBMS software take care of describing data structures for storing the data and retrieval procedures for answering queries. (So basically the purpose is to make it such that users can search for what they need without having to worry about the details of how the data is stored and archived (the DBMS software takes care of that). Most implementations of the relational model (i.e. RDBMS software) use the SQL data definition and query language.

The input of our 'software subsystem' was the information that the main hub received through Zigbee. This information was logged onto a computer by first identifying the corresponding serial port by using *screen* - a tool that is built into most unix systems to communicate with serial ports. To display devices that are connected to the computer's ports, we the following command:

ls /dev/tty.*.

We then identify which port is the one to which the main hub is connected to, and then use *screen* to communicate with that port. If, for example, the port was /dev/tty.usbserial-A401501O, then we would use *screen* as follows*:*

screen -L /dev/tty.usbserial-A401501O 57600.

Note that the third argument (57600) specifies that baud rate, which should be the same as established earlier in order to operate properly. In addition to that, the first argument (-L) is what lets the *screen* tool that we want to log whatever pieces of information are passed from that serial port. The way *screen* operates is that it creates a text file called screenlog.0 on whichever directory it was launched from, and this text file is simply a transcription of what is being communicated to the main hub. The key to making this subsystem connect to the previous ones was to log information using *screen*, and to turn that information into an amenable form to use in filling out templates, in preparation for data upload onto the website and onto our team's database.

A .txt data file was then created from the logged information that was passed from Zigbee to the main hub and transcribed onto screenlog.0. This .txt data file was created by

a script called parser.sh, which takes information from screenlog.0 and fills a template file called report_from_zigbee_template accordingly. More specifically, parser.sh reads the contents of screenlog.0 and compares them to what has been seen on that same file previously; thus, parser.sh determines what the new contents of screelog.0 are (that is, what is the newly reported information attained from Zigbee). To do this, parser.sh keeps track of the number of messages previously seen on screenlog.0 in a separate text file called num_messages_register.txt, and uses it as reference every time it reads the contents of screen log.0 (every minute), to determine whether there are new messages or not and, if there are, how many; this is what allows for determining what the new messages are.

The resulting .txt data file (which is named report_from_zigbee.txt) lists the different machines from the GyEEm, followed by either a '1' if the sensor reports that the machine is currently in use, and a '0' if the machine is available. This data file is read by a two shell scripts, which write different files that are outputs of this subsystem. One of these is a .html file (which is what will be uploaded to the GyEEm website), edited according to the information in the .txt data file from Zigbee. The other output is a timestamped datafile, with information that will be uploaded to a MySQL server.

To get the information from Zigbee onto the website, we needed a way to have the .html file (that controls the content of our website) be updated constantly and automatically. The fundamental element in our solution to the problem of how to get the information from Zigbee to the website and to the MySQL server is a cron job that we set up using crontab. crontab is a Unix/Linux utility that enables the user to set up a cron job - that is, crontab lets the user make the computer execute a particular command at a particular time (on a particular date or dates, or even every *x* hours or minutes).

To set up a cron job, the crontab utility provides three options for the user: -l, -e, and -r. These allow the user to list, edit, and remove the contents of the current cron table file for the current userid (the user account on the machine). Each user account on the machine has its own cron table files; the table files are constantly polled by the machine's cron daemon, and each have a different function in the system. One of these files - the /var/spool/cron file of each account - is what is edited when the user enters crontab -e in the command line terminal, and it is the file that is used to set up personalized cron jobs. Upon entering the edit option, the user has to set up a cron job in accordance to the following structure:

```
# * * * * *  command to execute
# ┬ ┬ ┬ ┬ ┬
# │ │ │ │ │
# │ │ │ │ │
```

│ │ │ │ └── day of week (0 - 7) (0 to 6 are Sunday to Saturday, or use names; 7 is Sunday, the same as 0)
│ │ │ └──── month (1 - 12)
│ │ └────── day of month (1 - 31)
│ └──────── hour (0 - 23)
└────────── min (0 - 59)

A note about using crontabs: each time a cron job is executed, whatever messages might result in addition to the operations that a script carries out (such as messages updating the status of a particular job, or anything that would be printed on the command line upon execution of a particular script) is sent as a 'mail' to the user's system mail box. To delete all mails from the mail box (and thus avoid unnecessarily filling the computer's storage with junk messages), one should access the system mailbox as follows:
mail
and then type
d 1
to delete the first message,
d 2
to delete the second message,
and
d *
to delete all messages. To avoid cluttering the system mailbox, we set up another cron job to delete all messages in it periodically (every 10 minutes).

We wrote a bash script called HTML_editor.sh, which pulls information from a data file and edits an HTML template accordingly. This template lists the GyEEm equipment, followed by 'XXXX's that are replaced depending on the input from Zigbee. We also wrote a bash script called mysql_friendly_text_writer.sh, which prepares data for database upload. To set up a cron job to execute this file every 2 minutes, we used a bash script called do.sh which ran the series of bash scripts to execute necesary tasks. We put our files on our Desktop, and then entered crontab -e in the command line terminal, which enabled us to edit the /var/spool/cron file in a command line editor similar to vi. Through this editor, we did set up our cron job by adding
*/2 * * * * ~/Desktop/do.sh
onto the file. This finalized the cron job setup, and so the automatization of the upload of GyEEm availability information onto the website was complete. Figure 4 (shown on next page)outlines the flow of the data through our system.

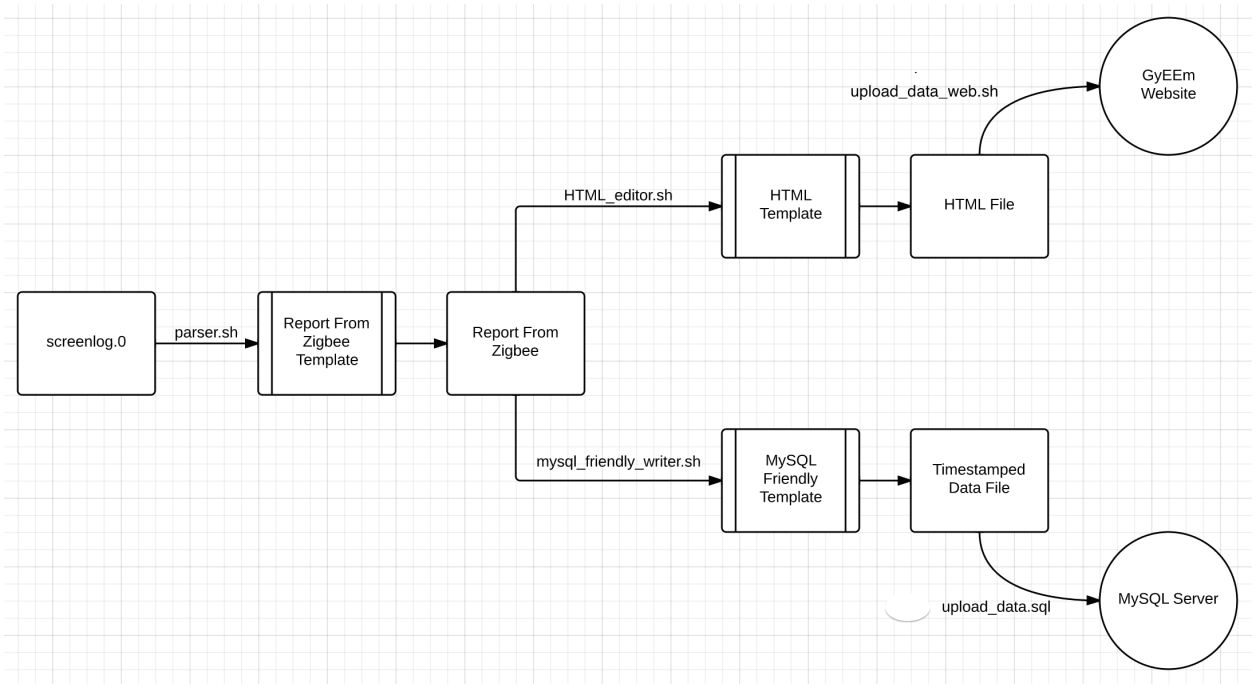


Figure 5: The flow of data through our system

To accomplish the second task of the software subsystem - namely, information management using a MySQL database - we first had to set up a server. This was done by downloading the MySQL server software; from their website (http://dev.mysql.com/downloads/mysql/), we downloaded mysql-5.6.16-osx10.7-x86_64.dmg and later used this package to install the software. Once the MySQL server software was installed, we had to do a couple of administrative mini-tasks to set up the server properly (setting up a password for root access of our database, creating a path for quick access through the command line, and setting up environment variables). We then created a database in our server called gotf (CREATE DATABASE gotf;), and then created a table called gyeem_info within gotf (USE gotf; CREATE TABLE gyeem_info;). We then had to find a way to have our data be uploaded onto that table.

To upload our data onto table gyeem_info from our gotf database within MySQL, we again relied on crontab to set up another cronjob, as well as a bash script to create a MySQL-friendly .txt file, and a .sql script to take the contents of the new .txt file and upload them onto our MySQL server. The first step of our database-upload task was to read the input .txt file using a bash script called mysql_friendly_text_writer.sh, and to have that same bash script use a template called mysql_friendly_text_template.txt to write a timestamped .txt file called mysql_friendly_text_YYYYMMDD_HHMM.txt, where the suffix represents the time. After that, a .sql file called data_uploader.sql loads data from the newly created mysql_friendly_text_YYYYMMDD_HHMM.txt file to the gyeem_info table on MySQL.

The decision to use MySQL to manage and store our data was a wise one, because it allowed us to posteriorly access specific pieces of information through custom queries, depending on what our objectives were. To look at the usage of the Elliptical on May 2nd, at 2AM in the morning, we'd type:

select * from gyeem_info where machine_name = 'Elliptical' && month = 5 && day = 2 && hour = 2;

The result is as shown in Figure 6.

```
mysql> select * from gyeem_info where machine_name = 'Elliptical' && month =5 && day = 2 && hour = 2;
+--------------+------+-------+------+------+--------+---------------+
| machine_name | year | month | day  | hour | minute | machine_usage |
+--------------+------+-------+------+------+--------+---------------+
| Elliptical   | 2014 |     5 |    2 |    2 |     43 |             1 |
| Elliptical   | 2014 |     5 |    2 |    2 |     44 |             1 |
| Elliptical   | 2014 |     5 |    2 |    2 |     45 |             0 |
| Elliptical   | 2014 |     5 |    2 |    2 |     53 |             0 |
| Elliptical   | 2014 |     5 |    2 |    2 |     57 |             1 |
| Elliptical   | 2014 |     5 |    2 |    2 |     58 |             1 |
| Elliptical   | 2014 |     5 |    2 |    2 |     59 |             0 |
+--------------+------+-------+------+------+--------+---------------+
```

Figure 6: Example of a MySQL query and result.

Lastly, we needed to figure out a way to get our .html files to actually be uploaded to the internet every minute, upon being created. That is, we'd already figured out a way for our files to be created appropriately (through the use of our shell scripts), and for this to be done both automatically and periodically (through the use of cron jobs); we still needed a way to open up a socket and update our website by uploading the .html file containing the latest information about machine statuses. To do this, we had to use ftp. By typing 'ftp' onto the command line, we opened up an ftp tool which is built onto unix systems. We then used the command

ftp seniordesign.ee.nd.edu

to connect to the appropriate host. After that, we specify our username and password. We then can explore the contents of our website using ftp commands. In particular, we use the command

put ~/Desktop/gyeem_of_the_future_machine_statuses.html Gymhomepage.html

to replace the website's Gymnhomepage.html file with the gyeem_of_the_future_machine_statuses.html file that contains the latest information (and is currently located in our Desktop). Once we'd established the logistics of this routine, we wrote a script called upload_data_web.sh, which would be executed automatically as part of the cron job to get the information onto the website.

6 System Integration Testing

6.1 Describe how the integrated set of subsystems was tested.

6.1.1 Testing the board

Once the board we designed arrived we then soldered all of the components that were going to be permanently fixed to the board. Then we connected the zigbee component and the sensor.

Next we compared our board file to the board we had soldered to check for any errors in soldering. We used a voltmeter to make sure the connections were working. Once we did this we plugged in our board and gave it power. The first thing we did after connecting power was to make sure none of our components were heating up. They were not. we proceeded to the next step. We then used the voltmeter again to make sure our pins and other components were receiving the proper amount of voltage and in the case of the regulators emitting the proper voltage. If your pins are not receiving the proper amount of voltage you may have to cut a trace and feed a wire from a voltage source to the component that is lacking voltage.

Once the connections are tested and all the voltages are correct then we can move onto testing the microcontroller and sensor.

6.1.2. Testing the Microcontroller code & sensor

In the code we need to make sure that our analog to digital pins on our microcontroller match the ones specified in the code. We also made sure than any board changes were accounted for within the code.

Once these precautions were taken we compiled the code and downloaded it into out microcontroller. If our sensor did not detect anything we looked back at the code and usually found a typo or a mislabeled pin. One problem to be careful of is to make sure your code deletes the previous information that was on the microcontroller before you download new code. This may affect the success of your testing.

6.1.3 Testing the website

Once the code and the board were functioning properly we moved on to test the website. We were reading the data that our sensor was generating via a program called “Putty.” When we tested the website we accessed this information and uploaded it. If our shell scripts were functioning correctly then the sensor information would display as “available” or “unavailable” on our website.

6.2 Show how the testing demonstrates that the overall system meets the design requirements

The concept of our project was to create a system that would allow people to check the availability of equipment at their gym from the comfort of their home. The idea behind this was that our system would help prevent gyms from becoming too crowded.

We designed a board that would allow our sensor to interact with our microcontroller. We then created code that would be able to determine whether a certain piece of equipment was or was not in use. Once this code was downloaded into the microcontroller we would be able to successfully determine the availability of equipment.

After having one functional sensor we built another and were also able to use it to monitor the status of equipment. We then uploaded this information to the internet for all to see. Once on the internet website was running all the information from our sensors was viewable to the public.

We were able to create a system that accomplished the goals we set for our project at the beginning of the year.

7 Users Manual/Installation manual

7.1 How to install your product

Installation of our product will largely depend on the layout of your gym, as well as the specific machines that you intend to keep track of with sensors. This is because of the different shapes of each machine. Our sensors are built to be packaged in small boxes, and are thus very much amenable to pieces of gym equipment with dashboards such as treadmills, ellipticals, or stationary bikes. Installation of sensors on weight racks weight machines, though, might be a bit more tricky; what one has to focus on when installing the sensors is to make sure that they are both out of the way (won't get 'bumped into'), are pointed at wherever a person using a particular piece of equipment would be standing, and are close enough to detect presence. Once the sensors are properly installed, one must plug the 'main hub' onto the computer that will be responsible for collecting data and making it publicly available via the gym's website. After this, pressing the 'reset' button on each sensor would leave the system installed.

How to setup your product

Once everything is properly installed, the setup is done through a window that will pop up when the main hub is plugged into the computer. This window will be a GUI that will permit the user to establish the identity of each sensor, and to define the distance range he/she might want that particular sensor to detect (which will vary, depending on the piece of gym equipment that the sensor corresponds to). As part of this set up process, the user will be prompted to specify information about the gym's website server, as well as to decide on a password to protect the gym's MySQL database. Once this is done, the setup software will take care of putting all the scripts in place and of installing MySQL on the computer, as well as creating the appropriate database and tables on it, depending on which machines are being tracked by the sensors installed at the gym.

How the user can tell if the product is working

The user can tell if the product is working by looking at the website, and seeing if changes are being updated accordingly as someone comes in and out of the range of a particular sensor.

How the user can troubleshoot the product

Our program will have a troubleshooting feature as part of the software package that becomes installed on your gym's computer at setup. This troubleshooting feature will work by testing different subsystems independently, thus identifying the source of the problem. To do this, the communication between the sensors and the main hub via Zigbee is tested first (by checking the contents of screenlog.0), and then the efficacy of each step of data flow through our system is assessed by running the different scripts that catalyze each step. Once this source is identified, the administrator will be prompted to make changes accordingly, and everything will be tested again. Depending on the performance of subsystems after changes have been made, system functionality will be re-evaluated and the administrator will be prompted to make changes once more, if necessary.

8 To-Market Design Changes

In designing our prototype there were some financial and scheduling constraints that restrained us from producing a device ready to be sold on the market. To be able to have this product sold on the shelves of any store or made easy to install on any gym there would be three different changes we would make. We would make the sensor module smaller, add the capability of multi machine and include Main hub as a Razzleberry pie.

To be able to sell this product and make it easy to install we need to make it smaller. The sensor needs to be basically be 'hidden' on the machine and it should not make the usage of the machine uncomfortable. In order to do this, we would have needed to use a different, less consuming and smaller sensor or have designed our own. Obviously there were time and monetary constraints that did not allow us to do this.

The second change we would make on our product would be to make the sensor capable to be used on any different machine (elliptical, bicycles, treadmills,etc.) using the same program. With the addition of LEDs and Buttons the user would be able to change the type of machine usage the sensor would be analyzing using the same board. Moreover, this would make the product more user friendly.

The third, would be to make a razzleberry pie capable of analyzing all the sensors. For our project we used a computer from where we uploaded the data to the web, but with a razzleberry pie, the later will be able to upload the information to the web as well as utilizing less space at the gym.

The fourth and last changed, would be to improve our website and improve our data analysis tools. In order to do this we would require more testing at certain gyms to be able to obtain more information and prove our design to potential customers. One of the most important features of our product, is its capability to predict trends at the gym in order to reduce costs and increase profits for the gym owner. This can only be achieved and proven once we have done some testing.

9 Conclusions (and future work)

The GyEEm of the future project was demonstrated successfully. We have been able to demonstrate complete the functionality of our system. This was accomplished owing to that we were able to work together as a group and individually on each of the subsystems.

We successfully chose a microcontroller that provided the functionality that we need. The pic32 family microcontroller is best for our sensor system. It allows for easy interaction between our main hub and sensors.

We were able to design a board that was less than half the size of our main hub. We were also able to place the pins for the zigbee so that it would be able to be stacked and conserve space. This will allow for easier integration into gym equipment. A bulky hardware package would be more vulnerable to damage and cause the GyEEm of the future to be less reliable.

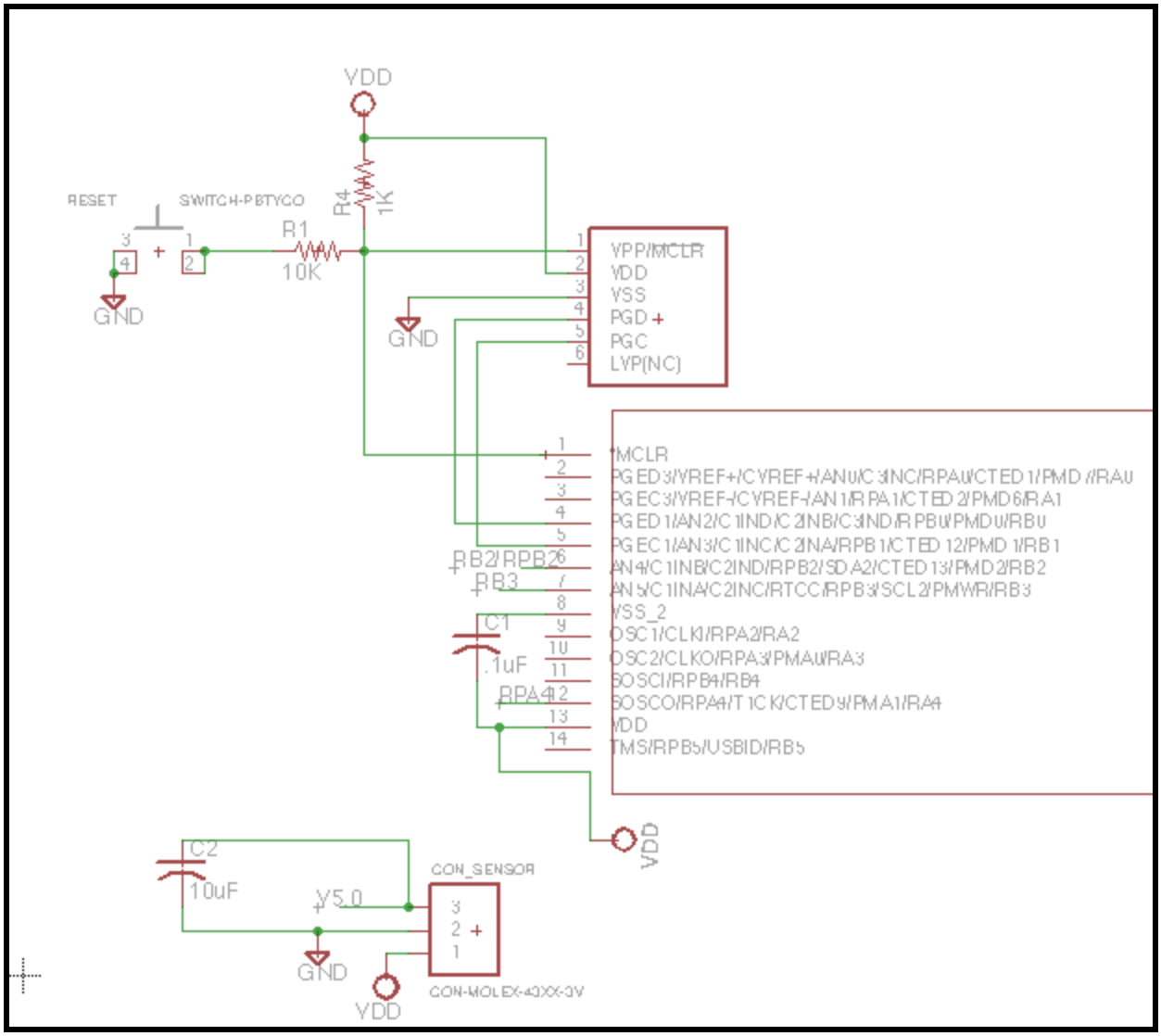
We observed successful communication between all of our sensors to the main hub and vice versa. We also were able to successfully upload this information to our website for all gym patrons to see.

We are beginning to create the congestion trend subsection of the gym website. We can accomplish this once we collect enough data on our MySQL server. We will then evaluate this data and upload it back into the website. Once our project is running as expected we will be able to provide of the previous week of gym activity. Our long term goal will be to provide gym trend information on holidays based on the last year of data. This data will supplement the regular weekly data that will be displayed.

10 Appendices

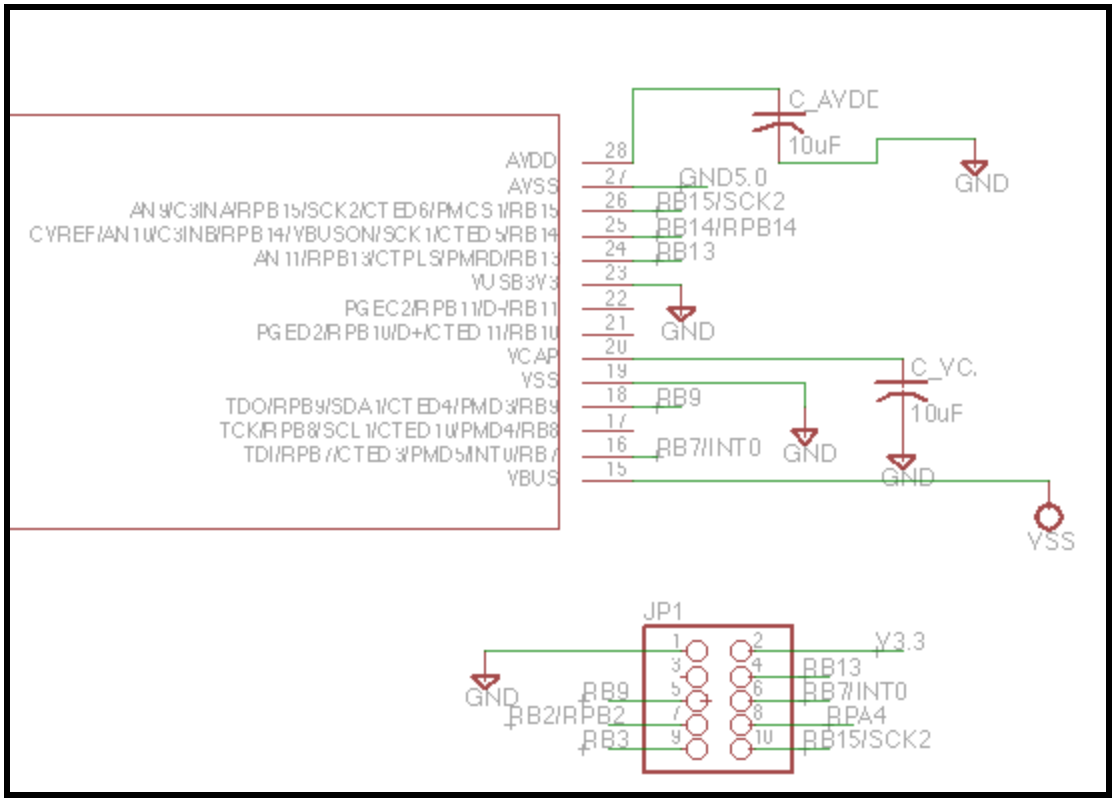
10.1 Complete hardware schematics

The board figure had to divided into three images so that all of its components could be seen.



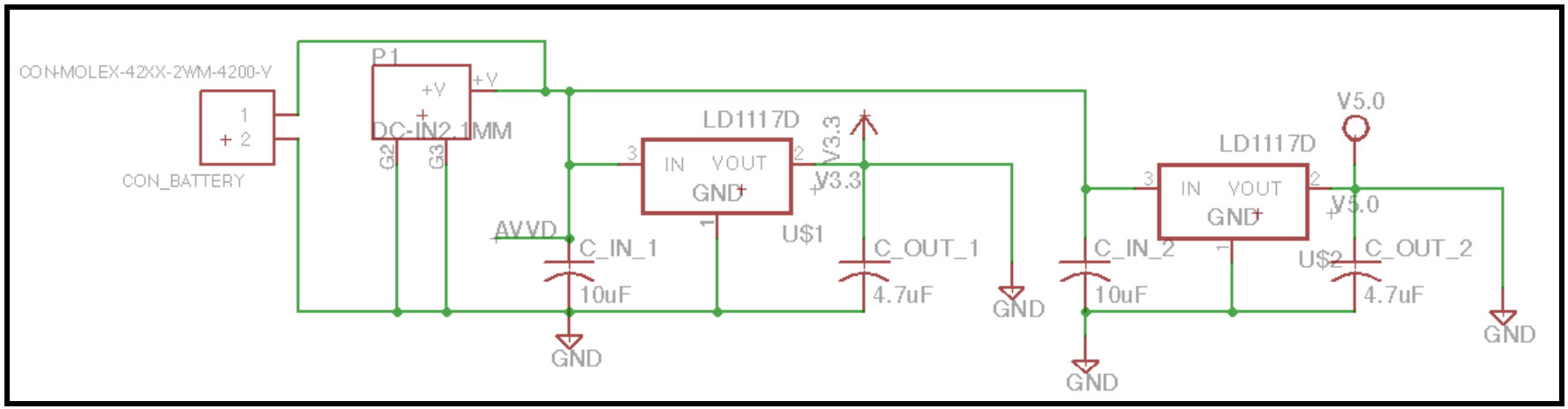
Board Figure 1: Microcontroller, Sensor connector, Pickit3

Here we see half of the pins of the Microcontroller and where they are connected. We can also see the Sensor connector which is lables “CON_SENSOR” on the schematic. The Sensor connector has a 10uF capacitor connected to the VDD and Ground to allow the sensor to make more accurate measurements. Lastly, we can see the Pickit3 at the upper left of Figure 1 which has a reset connected to MCLR.



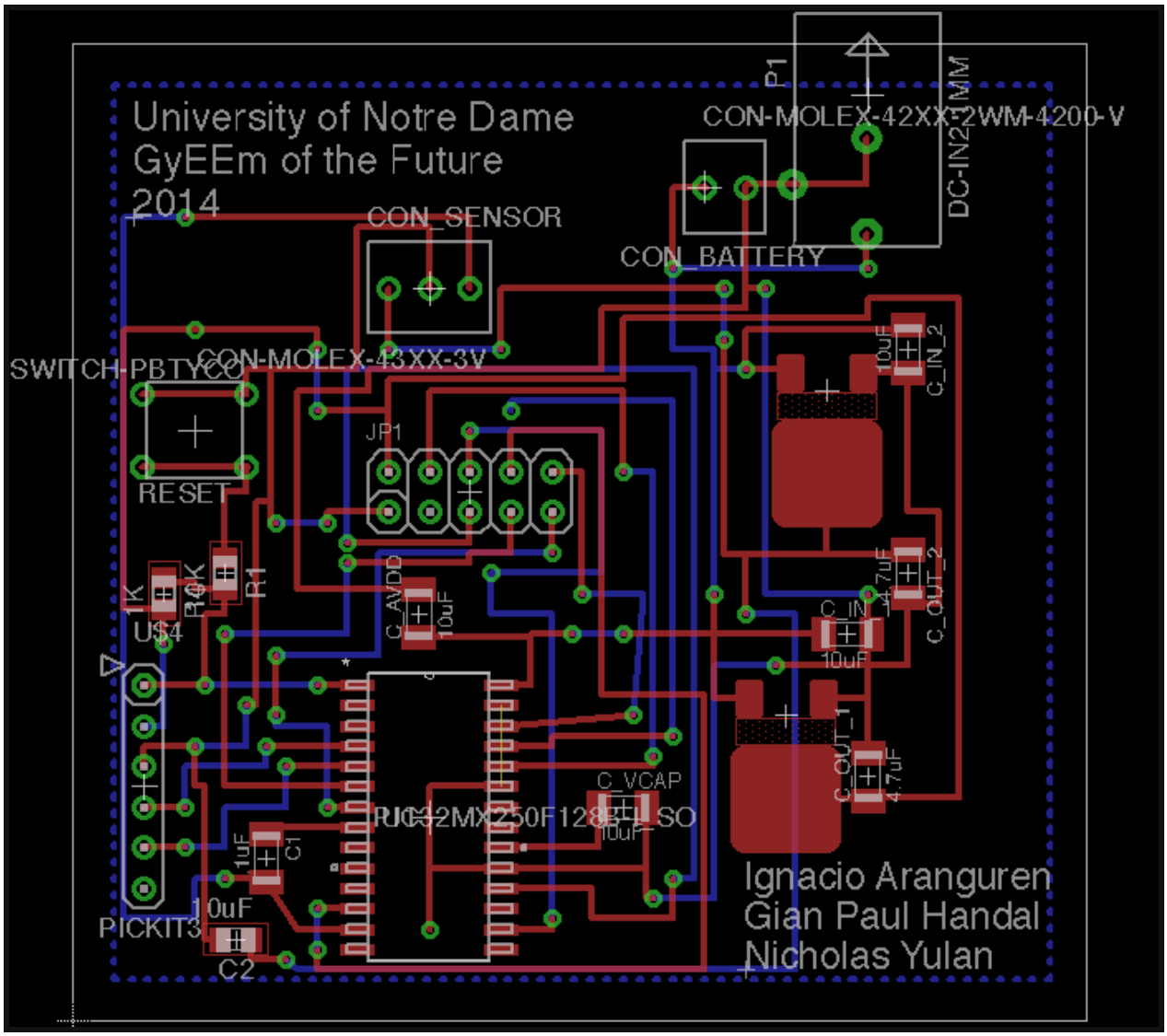
Board Figure 2: Microcontroller, and Zigbee Connector

In this image we can see the pins on the right side of the microcontroller. We can see which pins are connected to the Zigbee Connector, which pins are connected to ground and which pins require a capacitor. The capacitor values are listed in the image.


CON-MOLEX-42XX-2WM-4200-V
1
2
CON_BATTERY
P1
+V
+V
DC-IN2.1MM
G2
G3
LD1117D
3
IN
VOUT
GND
2
U$1
V3.3
V3.3
AVVD
C_IN_1
10uF
C_OUT_1
4.7uF
GND
GND
LD1117D
3
IN
VOUT
GND
2
U$2
V5.0
V5.0
C_IN_2
10uF
C_OUT_2
4.7uF
GND
GND

Schematic Figure 3 (pg 27) - shown vertically to enhance image:

The image on the previous page shows how the two voltage regulators are connected to each other and to the power source. In this image we have a connector for a battery and another connector for an in wall power source. In our demonstration we used a battery connected to our battery connector. While we were testing we powered our device from the wall to conserve batteries.



Board Figure 1

This is an image of the updated layout of our board. We can see how all the components are attached and which connections are above and below the board. All the components have names and values to ensure that anyone could refer back to the document in the future. Our initial board had several errors that we cut and reconnected manually. This is the final version of the board.

Complete Software listings

Relevant parts or component data sheets (do NOT include the data sheets for the microcontroller or other huge files but give good links to where they may be found.)

10.1 Data Management Scripts

gyeem_of_future_machine_statuses.html

```
<!DOCTYPE html>
<html>
   <body>

      <h1><font face="Monaco" color="black">GyEEm of the Future Machine Availability Page</font></h1>

      <p>Bicycle: available</p>
      <p>Elliptical: available</p>
      <p>Treadmill: available</p>

      <img src="arnold.jpg" alt="arnie" width="256" height="192">


      <p>Last Updated: May 01, 2014 at 15:53</p>


   </body>
</html>
```

template.html

```
<!DOCTYPE html>
<html>
   <body>

      <h1><font face="Monaco" color="black">GyEEm of the Future Machine Availability Page</font></h1>

      <p>Bicycle: <font face="Monaco" color="bp1color">bp1status</font></p>
      <p>Elliptical: <font face="Monaco" color="ell1color">ell1status</font></p>
      <p>Treadmill: <font face="Monaco" color="tm1color">tm1status</font></p>
```

```
        <img src="arnold.jpg" alt="arnie" width="256" height="192">


            <p>Last Updated: month day, year at hour, minute</p>


            </body>
</html>


mysql_friendly_text.txt
Elliptical      2014    05      01      15      53      0
Treadmill       2014    05      01      15      53      0
Bench_Press     2014    05      01      15      53      0


mysql_friendly_text_template.txt
Elliptical      year    month   day     hour    minute  Elliptical:
Treadmill       year    month   day     hour    minute  Treadmill:
Bench_Press     year    month   day     hour    minute  Bench_Press:

num_messages_register.txt
143
report_from_zigbee_template.txt
Bench_Press     bp_status
Elliptical      el_status
Treadmill       tm_status

report_from_zigbee.txt
Bench_Press     0
Elliptical      0
Treadmill       0

do.sh
#!/bin/sh

#  Created by nacho on 2/9/14.


# First, let's pull the data from Zigbee
~/Desktop/parser.sh

# Now, let's prepare that data to be uploaded onto the website
```

~/Desktop/HTML_editor.sh

Now, let's prepare that data to be uploaded to our database
~/Desktop/mysql_friendly_text_writer.sh

Let's upload that data to our database then
`/usr/local/mysql/bin/mysql -u root -ptigersofthenorthEEswag85 < ~/Desktop/upload_data.sql`

Let's upload that data to our website now
~/Desktop/upload_data_web.sh

HTML_editor.sh
```
#!/bin/sh

#  HTML_editor.sh
#
#
#  Created by nacho on 2/9/14.
#

TIME=`date '+%F %X'`


BENCH_PRESS_STATUS=`awk '/Bench_Press/ {print $2}' ~/Desktop/report_from_zigbee.txt`
ELLIPTICAL_STATUS=`awk '/Elliptical/ {print $2}' ~/Desktop/report_from_zigbee.txt`
TREADMILL_STATUS=`awk '/Treadmill/ {print $2}' ~/Desktop/report_from_zigbee.txt`


# bench_press
if [ "$BENCH_PRESS_STATUS" == 1 ]
then
bp1status=unavailable
bp1color=red
elif [ "$BENCH_PRESS_STATUS" == 0 ]
then
bp1status=available
bp1color=green
else
bp1status="no_info"
fi

# treadmill
```

```
if [ "$TREADMILL_STATUS" == 1 ]
then
tm1status=unavailable
tm1color=red
elif [ "$TREADMILL_STATUS" == 0 ]
then
tm1status=available
tm1color=green
else
tm1tatus="no_info"
fi

# elliptical
if [ "$ELLIPTICAL_STATUS" == 1 ]
then
ell1status=unavailable
ell1color=red
elif [ "$ELLIPTICAL_STATUS" == 0 ]
then
ell1status=available
ell1color=green
else
ell1status="no_info"
fi

year=`echo ${TIME:0:4}`
month=`echo ${TIME:5:2}`
day=`echo ${TIME:8:2}`
hour=`echo ${TIME:11:2}`
minute=`echo ${TIME:14:2}`

if [ "$month" == 01 ]
then
month_text="January"
elif [ "$month" == 02 ]
then
month_text="February"
elif [ "$month" == 03 ]
then
month_text="March"
elif [ "$month" == 04 ]
then
month_text="April"
```

```
elif [ "$month" == 05 ]
then
month_text="May"
elif [ "$month" == 06 ]
then
month_text="June"
elif [ "$month" == 07 ]
then
month_text="July"
elif [ "$month" == 08 ]
then
month_text="August"
elif [ "$month" == 09 ]
then
month_text="September"
elif [ "$month" == 10 ]
then
month_text="October"
elif [ "$month" == 11 ]
then
month_text="November"
elif [ "$month" == 12 ]
then
month_text="December"
else
month_text="N/A"
fi


`sed
's/bp1status/'$bp1status'/;s/bp1color/'$bp1color'/;s/ell1status/'$ell1status'/;s/ell1color/'$ell1color'/
;s/tm1status/'$tm1status'/;s/tm1color/'$tm1color'/;' ~/Desktop/template.html >
~/Desktop/gyeem_of_future_machine_statuses.html`


mail_deleter.sh
#!/bin/sh

#  HTML_editor.sh
#
#
#  Created by nacho on 4/19/14.
```

```
> /var/mail/nacho


mysql_friendly_text_writer.sh
#!/bin/sh

#  HTML_editor.sh
#
#
#  Created by nacho on 2/9/14.
#

TIME=`date '+%F %X'`

BENCH_PRESS_STATUS=`awk '/Bench_Press/ {print $2}' ~/Desktop/report_from_zigbee.txt`
ELLIPTICAL_STATUS=`awk '/Elliptical/ {print $2}' ~/Desktop/report_from_zigbee.txt`
TREADMILL_STATUS=`awk '/Treadmill/ {print $2}' ~/Desktop/report_from_zigbee.txt`

# bench_press
if [ "$BENCH_PRESS_STATUS" == 1 ]
then
bp1status="1"
elif [ "$BENCH_PRESS_STATUS" == 0 ]
then
bp1status="0"
else
bp1status="no_info"
fi

# treadmill
if [ "$TREADMILL_STATUS" == 1 ]
then
tm1status="1"
elif [ "$TREADMILL_STATUS" == 0 ]
then
tm1status="0"
else
tm1tatus="no_info"
fi

# elliptical
if [ "$ELLIPTICAL_STATUS" == 1 ]
then
```

```
ell1status="1"
elif [ "$ELLIPTICAL_STATUS" == 0 ]
then
ell1status="0"
else
ell1status="no_info"
fi

year=`echo ${TIME:0:4}`
month=`echo ${TIME:5:2}`
day=`echo ${TIME:8:2}`
hour=`echo ${TIME:11:2}`
minute=`echo ${TIME:14:2}`

`sed
's/Elliptical:/'$ell1status'/;s/Treadmill:/'$tm1status'/;s/Bench_Press:/'$bp1status'/;s/year/'$year'/;s
/month/'$month';s/day/'$day'/;s/hour/'$hour'/;s/minute/'$minute'/'
~/Desktop/mysql_friendly_text_template.txt > ~/Desktop/mysql_friendly_text.txt`

parser.sh
#!/bin/sh

LAST_NUM_MESSAGES=`awk '{print $0}' ~/Desktop/num_messages_register.txt`
LAST_KNOWN_BP_STATUS=`awk 'NR==1{print$2}' ~/Desktop/report_from_zigbee.txt`
LAST_KNOWN_EL_STATUS=`awk 'NR==2{print$2}' ~/Desktop/report_from_zigbee.txt`
LAST_KNOWN_TM_STATUS=`awk 'NR==3{print$2}' ~/Desktop/report_from_zigbee.txt`

MESSAGE_LENGTH=2

FILE_TEXT=`cat ~/Desktop/screenlog.0`
echo '\nThe text in the file is:\n\n'$FILE_TEXT
TEXT_LENGTH=`echo "${#FILE_TEXT}"`
echo '\nThe number of characters in the file is: '$TEXT_LENGTH
NUM_MESSAGES=`expr $TEXT_LENGTH / $MESSAGE_LENGTH`
echo '\nThe number of messages in the file is: '$NUM_MESSAGES
echo '\nThe number of messages in the last log was: '$LAST_NUM_MESSAGES
NUM_NEW_MESSAGES=`expr $NUM_MESSAGES - $LAST_NUM_MESSAGES`
echo '\nThe number of new messages is therefore: '$NUM_NEW_MESSAGES

if [ "$NUM_NEW_MESSAGES" != 0 ]
then
## begin for loop
for i in `seq 1 $NUM_NEW_MESSAGES`
```

```
do
   CURRENT_MESSAGE=`echo $FILE_TEXT | awk -v i=$i -v
last_num_messages=$LAST_NUM_MESSAGES '{ print substr( $0,
2*last_num_messages+2*i-1, 2 ) }'`
   echo '\n\nMessage number '$i' is: '$CURRENT_MESSAGE

   CURRENT_MACHINE=`echo $CURRENT_MESSAGE | awk '{ print substr( $0, 0, 1 ) }'`

   ## machine identification
   if [ "$CURRENT_MACHINE" == "B" ]
   then
      machine_reported="Bench_Press"
     echo 'The machine being reported is: '$machine_reported
   elif [ "$CURRENT_MACHINE" == "E" ]
   then
      machine_reported="Elliptical"
     echo 'The machine being reported is: '$machine_reported
   elif [ "$CURRENT_MACHINE" == "T" ]
   then
      machine_reported="Treadmill"
     echo 'The machine being reported is: '$machine_reported
   fi

   ## status retrieval
   CURRENT_MACHINE_STATUS=`echo $CURRENT_MESSAGE | awk '{ print substr( $0, 2, 1
) }'`
   echo 'The machine status reported is: '$CURRENT_MACHINE_STATUS

   ## status pairing
   if [ "$CURRENT_MACHINE" == "B" ]
   then
      LAST_KNOWN_BP_STATUS=$CURRENT_MACHINE_STATUS
      echo 'The last known bp status has been updated as: '$CURRENT_MACHINE_STATUS
   elif [ "$CURRENT_MACHINE" == "E" ]
   then
      LAST_KNOWN_EL_STATUS=$CURRENT_MACHINE_STATUS
      echo 'The last known el status has been updated as: '$CURRENT_MACHINE_STATUS
   elif [ "$CURRENT_MACHINE" == "T" ]
   then
      LAST_KNOWN_TM_STATUS=$CURRENT_MACHINE_STATUS
      echo 'The last known tm status has been updated as: '$CURRENT_MACHINE_STATUS
   fi
```

```
    `sed
's/bp_status/'$LAST_KNOWN_BP_STATUS'/;s/el_status/'$LAST_KNOWN_EL_STATUS'/;s/tm
_status/'$LAST_KNOWN_TM_STATUS'/' ~/Desktop/report_from_zigbee_template.txt >
~/Desktop/report_from_zigbee.txt`

done
## end for loop
fi

`echo $NUM_MESSAGES > ~/Desktop/num_messages_register.txt`

upload_data_web.sh
#!/bin/sh

#  Created by nacho on 4/9/14.

HOST=seniordesign.ee.nd.edu
USER=gotf
PASS=tlr9456xd

ftp -inv $HOST << EOF
user $USER $PASS
put ~/Desktop/gyeem_of_future_machine_statuses.html Gymhomepage.html
bye
EOF
exit 0

upload_data.sql

USE gotf;
LOAD DATA LOCAL INFILE '~/Desktop/mysql_friendly_text.txt' INTO TABLE gyeem_info;
```

10.2 The Sensor Module Code

```
/*
 * File:   Source_Sender.c
 * Author: ghandal
```

```
 *
 * Created on March 26, 2014, 7:03 PM
 */

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include "configbitsrev8.h"
#include "lib_delay.h"
#include "lib_spilcd.h"
#include "lib_serial6.h"

// New settings for our custom board
#define CS PORTBbits.RB9
#define SPIint IFS1bits.SPI2RXIF
#define Reset PORTBbits.RB13
#define Sleep PORTBbits.RB3
#define FCF_write1 0x44
#define FCF_write2 0x88
#define int0 PORTDbits.D0
int voltage_test = 0;
void init_adc3(void)
{

   // To configure the ADC module, perform the following steps:
//1. Configure the analog port pins in AD1PCFG<15:0> (see 17.4.1).

    ANSELA= 0;

    ANSELAbits.ANSA0 = 1 ; //Make A0 analog for conversion
    ANSELBbits.ANSB13 = 0;
    ANSELBbits.ANSB3 = 0;
    ANSELBbits.ANSB2 = 0;
    //ANSELBbits = 0;
    //ANSELAbits.ANSA4 = 0;
//2. Select the analog inputs to the ADC multiplexers in AD1CHS<32:0> (see 17.4.2).
    AD1CHSbits.CH0SA = 0 ;
   //AD1CHS = 0x00020000; // Connect RB2/AN2 as CH0 input
//3. Select the format of the ADC result using FORM<2:0> (AD1CON1<10:8>) (see 17.4.3).
    AD1CON1bits.FORM = 000 ; // Format as 16 bit integer
//4. Select the sample clock source using SSRC<2:0> (AD1CON1<7:5>) (see 17.4.4).
    AD1CON1bits.SSRC= 000; // Continous conversions
//5. Select the voltage reference source using VCFG<2:0> (AD1CON2<15:13>) (see 17.4.7).
```

```
   AD1CON2bits.VCFG= 000 ;  //Avdd
//6. Select the Scan mode using CSCNA (AD1CON2<10>) (see 17.4.8).
   AD1CON2bits.CSCNA = 0;
//7. Set the number of conversions per interrupt SMP<3:0> (AD1CON2<5:2>), if interrupts are to
be used (see 17.4.9).
   AD1CON2bits.SMPI = 0000; // only stored in ADIBUF0
//8. Set Buffer Fill mode using BUFM (AD1CON2<1>) (see 17.4.10).
   AD1CON2bits.BUFM= 0 ; // we
//9. Select the MUX to be connected to the ADC in ALTS AD1CON2<0> (see 17.4.11).
   AD1CON2bits.ALTS = 0 ; // clear ALTS
   AD1CHSbits.CH0NA = 0; // set to VR-
//10. Select the ADC clock source using ADRC (AD1CON3<15>) (see 17.4.12).
   AD1CON3bits.ADRC = 0 ; // PBCLK is used as the conversion clock source
//11. Select the sample time using SAMC<4:0> (AD1CON3<12:8>), if auto-convert is to be used

//12. Select the ADC clock prescaler using ADCS<7:0> (AD1CON3<7:0>) (see 17.4.12).
   AD1CON3bits.ADCS = 10; // set the ADCS to 10?
//13. Turn the ADC module on using AD1CON1<15> (see 17.4.14).
   AD1CON1bits.ON = 1;
//14. To configure ADC interrupt (if required):
//a) Clear the AD1IF bit (IFS1<1>) (see 17.7).
//b) Select ADC interrupt priority AD1IP<2:0> (IPC<28:26>) and subpriority AD1IS<1:0>
   //Do I need this?
//IFS1CLR = 0x0002; // Ensure the interrupt flag is clear
//IEC1SET = 0x0002; // Enable ADC interrupts
//IPC<24:24>) if interrupts are to be used (see 17.7).
//15. Start the conversion sequence by initiating sampling (see 17.4.15).
    //Done in our function

}

void init_SPI(unsigned long Fsck){  // function to initialize SPI

  SPI2CONbits.SMP = 0;// maybe is 1 but we can play with this later
  SPI2CONbits.CKE = 1; // edge clock
  SPI2CONbits.CKP = 0;  // polarity clock
  SPI2CONbits.MSTEN = 1;// set on master mode enable
  SPI2CONbits.MODE32=0; // 8-bit disable 32 // Do we still need to disable more 32 and 16?
  SPI2CONbits.MODE16=0; // 16 bit disable
  // SPI4CON = 0b00000000000000000000000100100000; //CKP (bit 8) and CKE
  //need to be correct
```

```
    unsigned long Fpb = 40000000; // see page 29 of PIC32 Reference SPI.pdf
    //unsigned long Fsck = Fpb/(2*(SPI3BRG + 1));
    //SCK3 = Fsck;
    // make it to 2MHz
    SPI2BRG = Fpb/(2*Fsck)-1;

    // not sure if we have to enable buffer mode
    // int rData;
    // rData=SPI3BUF;
    //SPI3BUF=1;

    SPI2CONbits.ON = 1;
}
unsigned char do_stuff(unsigned char data)
{
   SPIint = 0; // set interrupt to 0
   SPI2BUF = data; // send the buffer to data
   while(!SPIint);

   return SPI2BUF;
}
unsigned char write(unsigned long add , unsigned char data) // function that goes to an adresss
and writes the data to that point
{
    CS = 0;
    do_stuff(0xC0+add);

    do_stuff(data);

    CS = 1;
}

unsigned char read(unsigned add) // adress is the argument we are calling
{
    CS = 0;
    do_stuff(0x80+add);
    //do_stuff(add);

    do_stuff(0x00);// dummy value
    // return Data?
    CS = 1;

}
```

```
unsigned char frame_write( unsigned PHR, unsigned destination, unsigned source, unsigned
data)
{
   write(0x02,0x09); // PLL_ON state
   delay_ms(1);
   read(0x01);
   write(0x02,0x02); // TX-ready to send
   delay_ms(0.5);
   CS=0;
   do_stuff(0x60); // write command
   do_stuff(PHR); // PHR has to be a number between 9 and 127 in hex, unless it is ack(2+data)
   do_stuff(FCF_write1); //lower bit of FCF
   do_stuff(FCF_write2);// higher bit of the FCF

   do_stuff(0x45); // Sequence number  -  Figure out how to setup ramdomly
   //Destination Pan
   do_stuff(0x67);
   do_stuff(0x45);
// Destination ID
   do_stuff(0x01);
   do_stuff(0x23);
// Actual Source Adress
   do_stuff(source);
   do_stuff(source);
   // The stuff we want to send
   do_stuff(data);//actual stuff we want to send
   do_stuff(data);//actual stuff we want to send
   do_stuff(data);//actual stuff we want to send
   do_stuff(data);//actual stuff we want to send
   do_stuff(data);//actual stuff we want to send
    do_stuff(data);//actual stuff we want to send
   do_stuff(data);//actual stuff we want to send
   CS=1;
   delay_ms(1);
   write(0x02,0x08); // GO BACK TO trx STATE
   delay_ms(1);
}
void reset()
{
   CS= 0 ;
   Reset = 0;
   delay_ms(1);
   Reset = 1;
```

```
    CS=1;
}
void reset_128() // inverted reset
{
    CS= 0 ;
    Reset = 1;
    delay_ms(1);
    Reset = 0;
    CS=1;
}
void wakeup()
{
    CS = 0 ;
    Sleep = 1 ;
    delay_ms(1);
    Sleep = 0 ;
    delay_ms(1);

    CS = 1;
}

void send_frame_someone_there( int ADCValue)
{

    double voltage = ADCValue*3.3/1024;

    int x  ;

   if(voltage> 1.5  ) // if object is from 10 to 20cm of sensor
   {
       x=1 ;
  // frame_write( 0x11, 0x20, 0x45, 0x31); // 31 is a 1 in ASCII,45 is an E for elliptical

   }
   else if (voltage< 1.5  )
   {
       x=0;
  // frame_write( 0x11, 0x20, 0x45, 0x30); // 30 is a 0 in ASCII
   }
   if ( x != voltage_test )
   {
       if (x == 1)
       {
```

```
        frame_write( 0x11, 0x20, 0x42, 0x30); // 30 is a 0 in ASCII,42 is an B for Bike
       // frame_write( 0x11, 0x20, 0x45, 0x30); // 30 is a 0 in ASCII,45 is an E for elliptical
      }
      else if( x == 0)
      {
      frame_write( 0x11, 0x20, 0x42, 0x31); // 31 is a 1 in ASCII,42 is an B for Bike
        // frame_write( 0x11, 0x20, 0x45, 0x31); // 31 is a 1 in ASCII
      }
  }


  voltage_test = x;




  delay_ms(100);


}
void weight_check()
{
 while(1){
AD1CON1bits.SAMP =  1  ;
delay_ms(10); // sample for 10 mS
AD1CON1bits.SAMP = 0 ;
while (AD1CON1bits.DONE == 0 ); // conversion done?
int ADCValue = ADC1BUF0;//ADCValue we obtain from conversion


 send_frame_someone_there(ADCValue);
}
}
void Mapping()
{
   SDI2R = 0x10;

   RPA4R = 0x10;


}
void Map_2()
{
```

```
   SDI2R = 4;
   RPA4R = 4;
}
/*
 *
 */
int main(int argc, char** argv) {
// make all pins analog
    //AD1PCFG = 0xFFFF;
    DDPCONbits.JTAGEN = 0; // disable JTAGEN


    Map_2();
    init_adc3();

// Make the specific pins digital
    TRISBbits.TRISB9 = 0;
    TRISBbits.TRISB13 = 0;
    TRISBbits.TRISB3 = 0;
    TRISAbits.TRISA4 = 0;
    TRISBbits.TRISB2 = 0;


   init_SPI(10);// making the SPI4BRG large enough so the clock does not take toolon g



   reset_128();
   reset_128();
   reset_128();
   delay_ms(1);
   wakeup();
   wakeup();
   wakeup();
   wakeup();

   //weight_check();
   weight_check();
   delay_ms(1);

    return (EXIT_SUCCESS);
}
```

Main Hub code

```
/*
 * File:   MainHub.c
 * Author: ghandal
 *
 * Created on March 19, 2014, 11:41 PM
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/attribs.h>
#include <xc.h>
#include <plib.h>
#include "configbitsrev8.h"
#include "lib_serial6.h"
#include "kit32r7lib.h"
//define CS LATDbits.LATD5
#define CS PORTBbits.RB8
#define SPIint IFS1bits.SPI4RXIF
#define Reset PORTBbits.RB11
#define Sleep PORTBbits.RB10
#define FCF_write1 0x44
#define FCF_write2 0x88
#define int0 PORTDbits.RD0
#define INT0Flag IFS0bits.INT0IF
unsigned char test_SPI4BUF;
void init_SPI(unsigned long Fsck){  // function to initialize SPI
  SPI4CONbits.SMP = 0;// maybe is 1 but we can play with this later
  SPI4CONbits.CKE = 1; // edge clock
  SPI4CONbits.CKP = 0;  // polarity clock
  SPI4CONbits.MSTEN = 1;// set on master mode enable
  SPI4CONbits.MODE32=0; // 8-bit disable 32 // Do we still need to disable more 32 and 16?
  SPI4CONbits.MODE16=0; // 16 bit disable


        unsigned long Fpb = 40000000; // see page 29 of PIC32 Reference SPI.pdf

        // make it to 2MHz
        SPI4BRG = Fpb/(2*Fsck)-1;

        SPI4CONbits.ON = 1;
}
unsigned char do_stuff(unsigned char data)
{
```

```
  SPIint = 0; // set interrupt to 0
  SPI4BUF = data; // send the buffer to data
  while(!SPIint);

  return SPI4BUF;
}
unsigned char do_stuff_put(unsigned char data)
{
  SPIint = 0; // set interrupt to 0
  SPI4BUF = data; // send the buffer to data
  while(!SPIint);
  test_SPI4BUF = SPI4BUF;
  putu(SPI4BUF);
  return SPI4BUF;
}
unsigned char write(unsigned long add , unsigned char data) // function that goes to an adresss
and writes the data to that point
{
        CS = 0;
        do_stuff(0xC0+add);

        do_stuff(data);
        CS = 1;
}
unsigned char read(unsigned add) // adress is the argument we are calling
{
        CS = 0;
        do_stuff(0x80+add);

        do_stuff(0x00);// dummy value
        // return Data?
        CS = 1;
}

void sensor_read()
{
        CS = 0;
        do_stuff(0x20); // read command

 // dummy value to get PHR

        //int i = 1;
        int j = 0;
```

```
        int  data[j];
        int i =1;
        char  source_adr1;
        char source_adr2;
        for ( i  ; i < 16; i++)
        {
        //do_stuff(0x01);


        if(i<=9) // get higher bit of FCF
        {
        do_stuff(0x01);
        // set_output_device(1);
        // do_stuff(0x01);
        //const char* FCF_read1 = SPI4BUF;
        //tprintf(FCF_read1);

        }
//
        if (i == 13)// store data to analyze it later
        {
                do_stuff_put(0x00);
                //data[i] = SPI4BUF;
                //i++;
        }
        if(i==14) // get the source adress sexond pin
        {
        do_stuff_put(0x00);
        // tprintf(SPI4BUF);
                //source_adr2 = SPI4BUF;
        }
        if(i == 15) // get the source adress sexond pin
        {
        //do_stuff_put(0x00);
        // tprintf(SPI4BUF);
                //source_adr2 = SPI4BUF;
        }

        }
        CS = 1 ;

}
void set_reception()
```

```
{
  read(0x01);
        write(0x02,0x08); // Force TRX_OFF
        delay_ms(1);
        read(0x01);
        write(0x02,0x06); // RX_ON and ready to receive
        delay_ms(1);
        read(0x01);

}
void check_reception() // checks until some data has been recepted
{

        int reception = 0;
        //write(0x02,0x03); // Force TRX_OFF
        read(0x01); // check and see if the RX_ON is actually on
        while(reception != 1)
        {
        //while(!INT0Flag);//Interrupt is triggered
        read(0x0F); // read the IRQ_Status to check if RX_start
        if (SPI4BUF == 0x04) // iF irQ_2 or RX_StART
        //read(0x01); // Read TRX_Status
        //if(SPI4BUF == 0x01)
        {
        reception = 1; // send message to LCD that says reception has been started
        }
        else if (SPI4BUF == 0x08) // to check if TRX_END
        {
        reception = 1;
        }
        }
 }

void reset()
{
        CS= 0 ;
        Reset = 0;
        delay_ms(1);
        Reset = 1;
        CS=1;
}
void wakeup()
{
```

```
        CS = 0 ;
        Sleep = 1 ;
        delay_ms(1);
        Sleep = 0 ;
        delay_ms(1);
        CS = 1;
}
void Enable_INT0(void)
{
        write(0x0E,0x08); // write to IRQ_Mask to only signalize when interrupt RX_start and
TX_end
        INTCONbits.MVEC =1 ;//Multi vectored system
        INTCONbits.INT0EP = 1 ; // set the edge polarity to rising edge
        IEC0bits.INT0IE =1;//enabling interrupt 0
        IPC0bits.INT0IP = 7 ; //setting priority to 7
        IFS0bits.INT0IF = 0 ;
        asm("ei");
        //do_stuff(0xAA);
        //do_stuff(0xAA);
        //do_stuff(0xAA);
}
 void __ISR( _EXTERNAL_0_VECTOR,IPL7AUTO ) int0_ISR(void)
 {
        check_reception();
        sensor_read();
        IFS0bits.INT0IF = 0 ;
        LATE =0x00; // lih=ghts turn on to check
 }

 void set_promiscous()
 {
        //Set Short adresses and Pan Adreses to 00
        write(0x20,0x00);
        write(0x21,0x00);
        write(0x22,0x00);
        write(0x23,0x00);
        write(0x24,0x00);
        write(0x25,0x00);
        write(0x26,0x00);
        write(0x27,0x00);
        write(0x28,0x00);
        write(0x29,0x00);
        write(0x2A,0x00);
```

```
        write(0x2B,0x00);
        write(0x17,0x02); //Enable promiscous Mode
        // Disable generation of Acknowledgement bit 4
        // Bit 6 &7 Acknowledgement of all frames
        write(0x2E,0xD0);
}

/*
 *
 */

int main(int argc, char** argv) {
// make all pins analog
        AD1PCFG = 0xFFFF;
        DDPCONbits.JTAGEN = 0; // disable JTAGEN
// Make the specific pins digital

        TRISBbits.TRISB8 = 0;
        TRISBbits.TRISB11 = 0;
        TRISBbits.TRISB10 = 0;
  init_SPI(10);// making the SPI4BRG large enough so the clock does not take toolon g

  reset();
  Sleep = 0 ;
  //Initialize Uart at 57600
        serial_init(57600UL);
        //set output device 1 to the terminal
        set_output_device(1);
        //**tprintf("program is being initialized\n");



  wakeup();
  Enable_INT0();
  read(0x01);
  delay_ms(1);
  delay_ms(1);
  write(0x0E,0x08); // write to IRQ_Mask to only signalize when interrupt RX_start and TX_end
  set_promiscous();
  set_reception() ;


        return (EXIT_SUCCESS);
```

```
}
```