Smart Dashboard

CS 362 Final Project

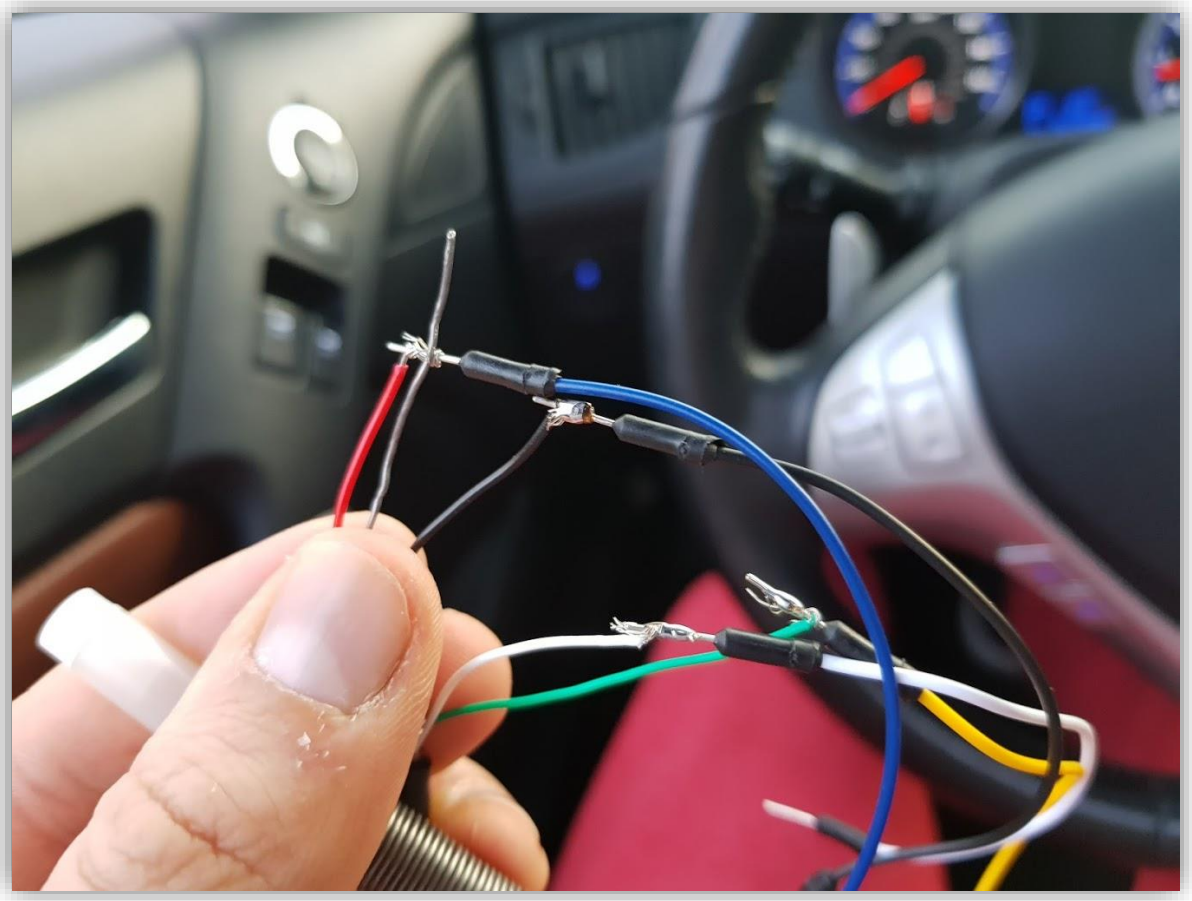
Corey Habel
Simon Rankov
Michael Koutsostamatis
James Warda

Summary

A dashboard-mounted device displaying GPS data and altitude, acceleration and orientation, and information about the vehicle using the On-Board Diagnostics (OBD-II) standard. The vehicle information includes RPM, speed, current gear, throttle and fuel level. The GPS and accelerometer data is processed by one Arduino, while the OBD data is received through an adapter and processed by a second Arduino. Both Arduinos send their data to an Arduino Mega via serial, where the data is sorted into packets and logged to an SD card. The data packets are then sent to a Raspberry Pi, which parses them and displays the information on the LCD. It also has an interactive UI for selecting certain bits of information to view.

OBD Interface

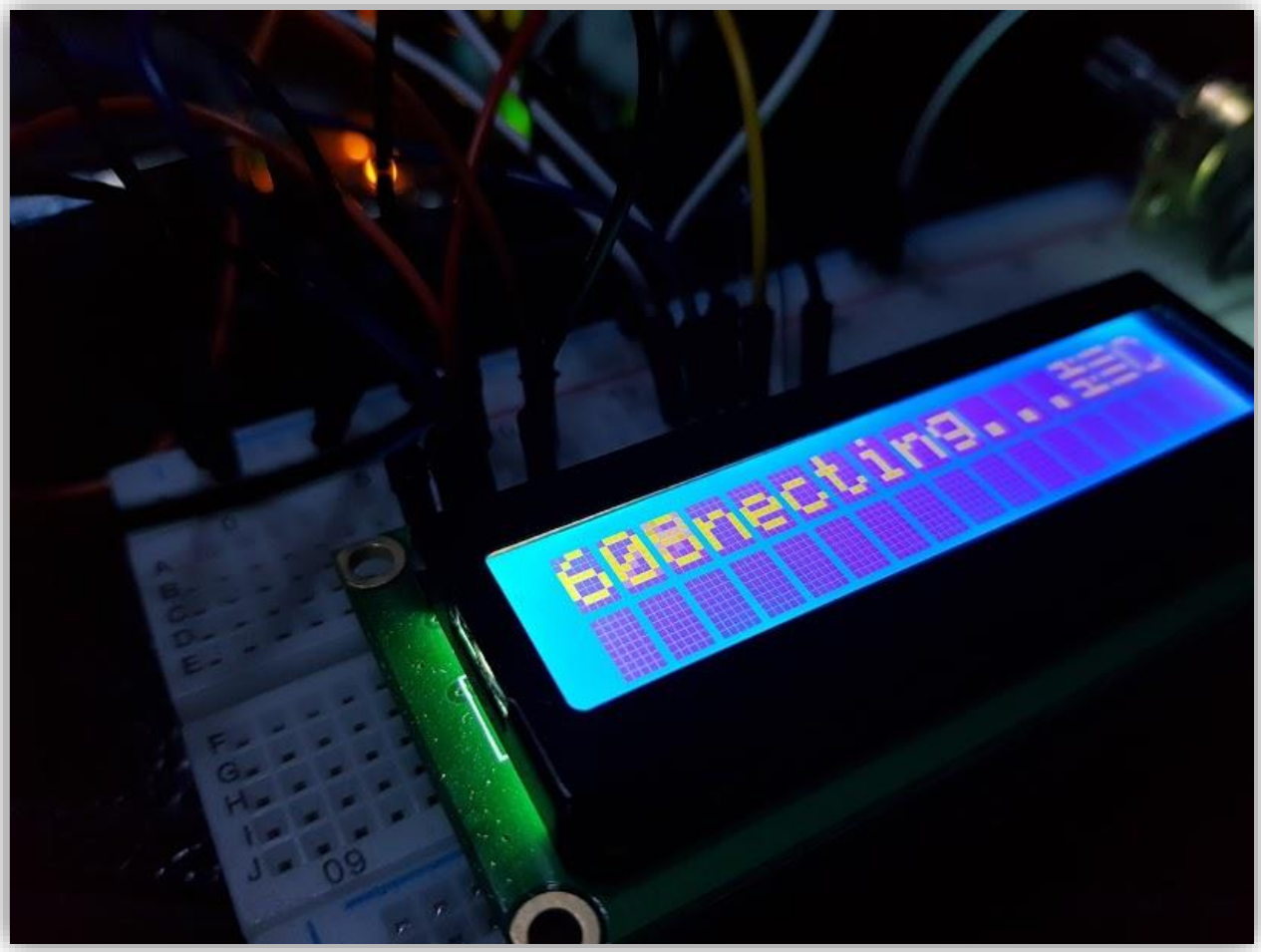
Interfacing with a car through OBD has proven to be mostly straightforward yet interesting and at times utterly infuriating. Ordering the OBD adapter was straightforward and it arrived quickly, but it came with a termination that was likely meant to fit onto an Arduino shield rather than into my board itself (or the fan pins on a PC motherboard). Fearing the need to buy another part for our project when it was almost crunch time, I did some research to find that the 4 wires from OBD were TX, RX, 5v and ground. This made it easy to solder and shrink wrap the final connector for easy integration into our Arduino setup with no extra hardware needed. I also wired up an LCD for debugging purposes.



One of the pivotal steps in my assignment was registering information from OBD. If I could get even one thing from the car to show up on a debug LCD, then I was in the green and could easily work the rest out.

This proved to be more trial and error that I hoped. The libraries that I was using were sparsely documented, and I had to determine that I was using a serial UART connection to interface with OBD and not direct OBD. After importing the correct libraries and some more trial and error determining PID codes, I was able to get my RPM on the LCD. An OBD PID is a value that can be read from a car, such as PID_RPM or PID_ENGINE_LOAD. I initially coded for obtaining a single value first, making it easy for myself to add more later on through the use of a switch statement. This allowed me to figure out the quirks of OBD and then learn what the display was to output later on from my group mates.

I used the debug LCD constantly from then on, making sure that values were read in to the Arduino properly without the need for my groups' hardware. It would also be instrumental in troubleshooting



misread values and connection issues when we integrated all of our parts. An OBD connection is established by setting up a serial connection and calling the obd.begin() function. This is checked with an

obd.init() flag that indicates the presence of a connection with the car. As long as the connection is present, a program can obd.readPID() to obtain a specific code from the car, and if the connection drops, a reconnect function halts data collection and attempts to reestablish a connection. Aside from some common oddities, (the car outputs speed in kilometers per hour and all temperatures in Celsius), the main issue with OBD is that there are manufacturer specific codes that control the large part of the readouts on the car. For example, the current gear PID is unknown to me and I have spent hours looking through Hyundai forums to find it. I am familiar with a function that lists all available PIDs built in to the OBD libraries I was using, but that was discovered far too late and we were not able to get a gear readout ready in time for filming and the due date. Aside from OBD frustrations, my part of the program gathered the needed data to local variables, formatted them into an array and sent the array off using software serial to the data logger. My only gripe with this project was that I was not able to get proprietary codes working, but otherwise this project was a huge learning experience that allowed me to use my electronic toys to talk to my mechanical toys for fun and utility.

Resources:

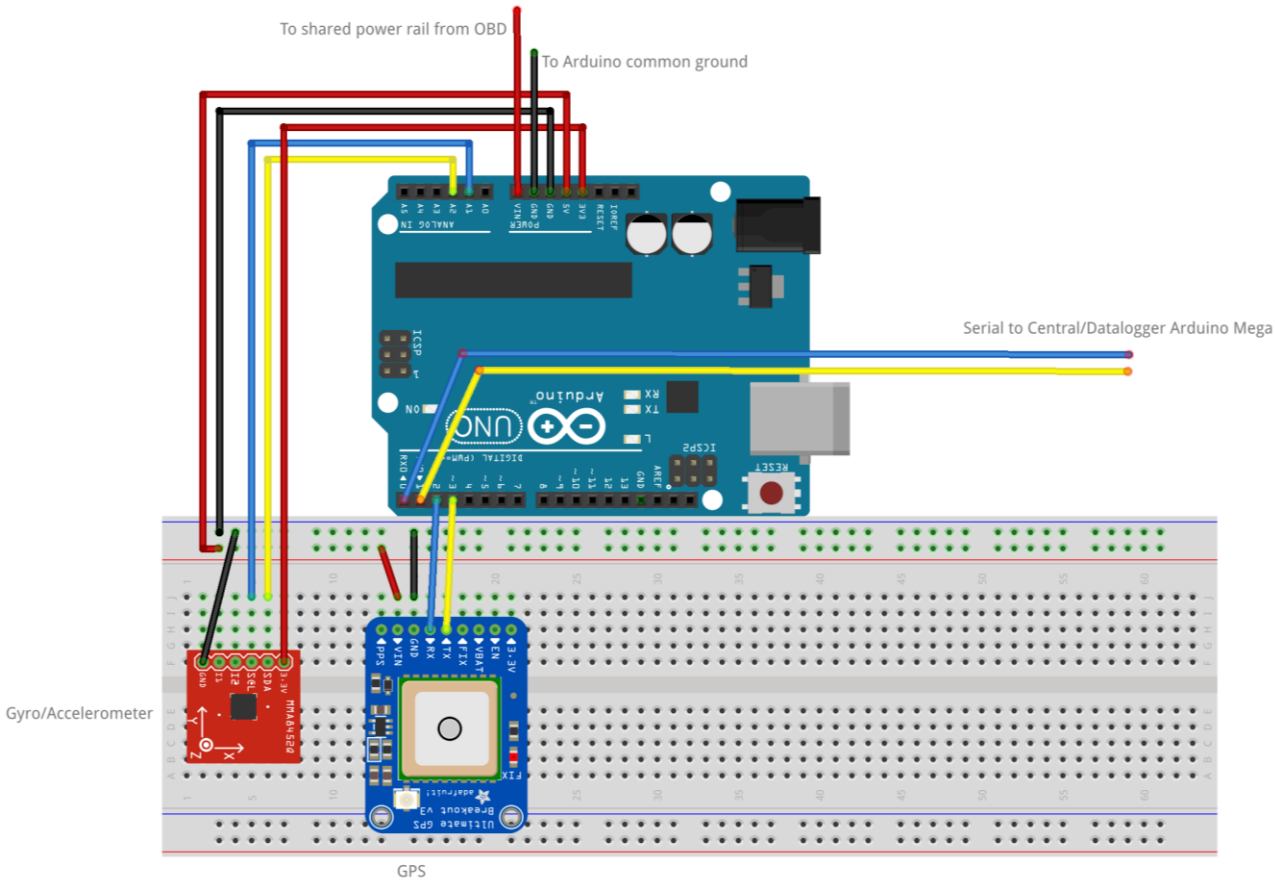
https://freematics.com/ (this entire site was very useful)

https://github.com/stanleyhuangyc/ArduinoOBD (also part of Freematics)

http://Arduinodev.com/Arduino-obd-data-indicator-rgb-matrix/

GPS / Accelerometer

To receive GPS coordinates and acceleration values relative to the car, an Arduino Uno was used, connected to an Adafruit Ultimate GPS and an Adafruit Triple-Axis Accelerometer. Using libraries provided by Adafruit, this Arduino constantly fetched acceleration data from the accelerometer and was fed GPS coordinates from the GPS module. Both modules are 5V tolerant, so it was easy to power them from the 5V regulated output of the Arduino. The Arduino received power from the OBD power output, which was also used to power the other Arduino Uno. The hot wire from the OBD cable went into the power rail on a breadboard, where the Arduinos could each connect to via their Vin pin. The built-in regulator of the Arduino regulated this down to 5V. The Arduinos communicating via serial also needed a common ground, so they were each wired into the ground rail of the breadboard. Below is a diagram of how these two devices connected to the Arduino.



The gyroscope/accelerometer module uses I2C protocol to communicate with the Arduino. The Arduino Uno does not have dedicated I2C pins, so the Arduino Wire library was used for emulating I2C on analog pins A4 and A5. After initializing the gyroscope object, the object is read every loop iteration which stores the values internally in the library. The library functions to return acceleration data on all three axes are then called, which are stored into a char array to be sent to the central Arduino.

The GPS module connects to the Arduino via serial. As the Arduino Uno only has one dedicated serial line, we had to use the SoftwareSerial library to emulate a second serial line. The GPS used this emulated line while the dedicated one was used for communication with the central Arduino. Adafruit provides libraries for the GPS, which do a lot of the hard work pertaining to fetching updated information. After the GPS is set up in the code, it must be read every loop iteration. This works in a similar way to the Accelerometer, as we can simply call library functions to return the values we want. The Arduino calls the functions for latitude and longitude, then converts these values to char arrays. These arrays are then added to the char array which will be send to the central Arduino.

The data string being sent to the central Arduino is a char array of all desired values separated by semicolons. The central Arduino combines this string with the string from the OBD Arduino, then sends both as one long char array to the Raspberry Pi for display.

One problem encountered with reading both the Accelerometer and GPS simultaneously was the fact that they each required varying periods of time to update the data each loop. Initially, the code was written to just read both and then immediately store into the data string. Since the accelerometer was much faster to update than the GPS, the GPS would sometimes return false or garbage data since it would be in the middle of an update when it was being read. This was solved by storing the most recent GPS data in local variables and changing those variables when updating, as opposed to grabbing the values directly from the GPS. This allowed the acceleration data to be updated much faster, since it would be sent with the last GPS coordinates instead of waiting for new ones. This was an acceptable change, as the GPS coordinates will not be changing nearly as dramatically as the acceleration values. If the Arduino

was waiting for the GPS to update every iteration, spikes in the Acceleration data could be missed, such as when the car is braking or performing a sharp turn. In later analysis of the stored data, GPS coordinates can also be interpolated much more easily than acceleration values can.

Resources:

https://www.arduino.cc/en/Reference/Wire - for implementing I2C protocol on the Arduino Uno

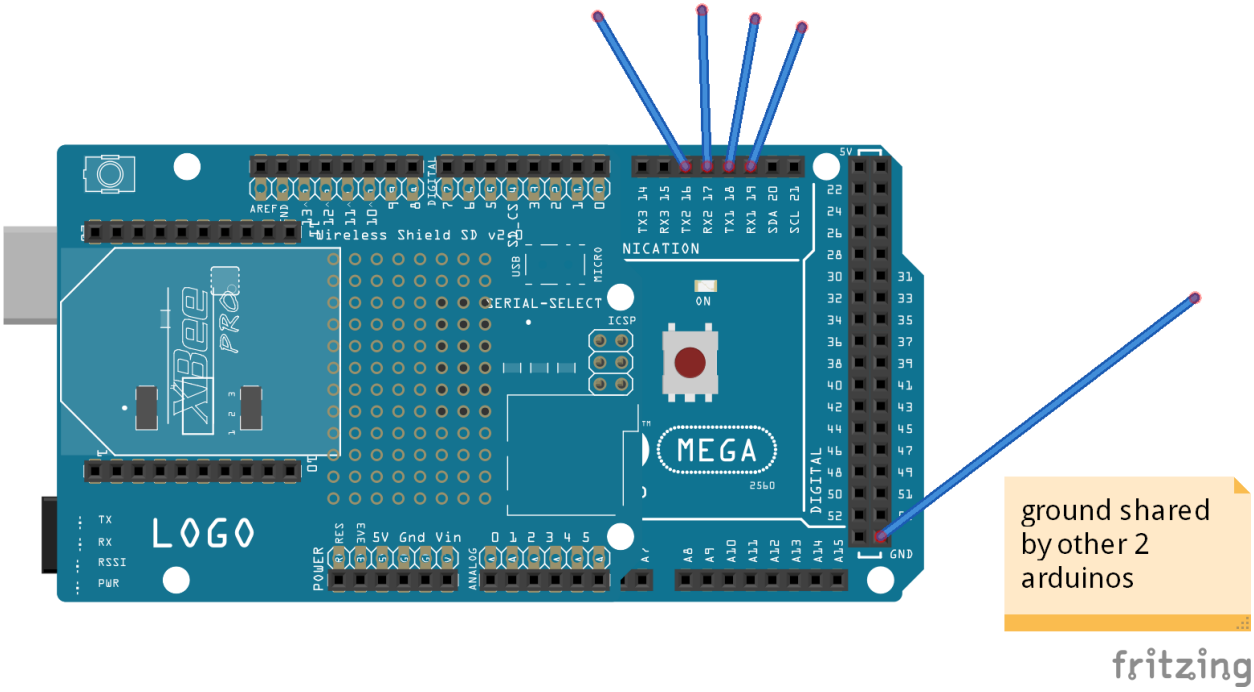
https://learn.adafruit.com/adafruit-ultimate-gps - GPS setup

https://learn.adafruit.com/adafruit-mma8451-accelerometer-breakout - Gyro/Accelerometer setup

https://www.arduino.cc/en/Reference/SoftwareSerial - for implementing SoftwareSerial on the Uno

Data Logging / Central Arduino

For the data logging portion of the project an Arduino mega was used with an attached SD shield for logging the data. To receive data from the other two Arduinos a serial connection is used. This is why we chose to use an Arduino mega instead of an Arduino Uno. The Arduino mega allows for multiple serial ports rather than just one. The Arduino connected to the OBD-II port would receive its data, form it into a string, and send the resulting string to the mega via serial. The second Arduino connected to the GPS and accelerometer did the same thing. Below is a diagram of how it looked.



Once the data was received it would be logged into the SD. Afterwards, the data would be sent to the raspberry pi through a third serial connection. The connection used here was through the USB serial port and into the pi. This was also the method for powering the mega.

When receiving the data a 1 or a 0 would be added to the beginning of the strings depending on who they came from. This was done so that when the data is sent to the pi it would be easier to analyze and know what exactly you were working with. Each string received was in a specific format but still

needed a way of being differentiated for easier processing. No other changes were made to the strings or data before being sent out.

When we first started testing our project the string collection was not working properly. At first the code simply looked for any available input from either of the two Arduinos. The problem with this is it would start to collect one string, stop, start processing another, and then go back to collecting the first. To get around this issue several things were tried. Sending data at specific times, setting checks to know when to stop, and rewriting collection several times. However, none of these worked. The solution turned out to be a simple check to see if the terminating null character had been encountered when collecting the string. This was done by parsing the serial buffer bit by bit rather than collecting everything at once and storing it. When collecting strings from just one Arduino this was not a problem and the mega was able to collect everything with a simple call to the read function in the serial library.

Resources:

https://www.Arduino.cc/en/Reference/SD

https://www.Arduino.cc/en/Reference/SPI

https://www.Arduino.cc/reference/en/language/functions/communication/serial/

Raspbery Pi Dashboard Display

The raspberry Pi portion of our project was the face of the Smart Dash. Its responsibilities consisted of receiving data from the Arduino Mega, parsing the data, and outputting it to the user. I will discuss all of the parts required to get this to work.

Connections:

The connection was preformed through a USB type A to USB type B connection. The data was transferred from the Adriano Mega through serial. We simply set up a speed of 9600 and received the string in this format: "{Currentgear; CurrentSpeed; currentRPM; currentOilTemp; CurrentGPSxCoord; currentGPSyCoord; CurrentTorque; accelerometerXval; accelerometerYval; AcceleromererZval; engineLoad; Runtime; intakeTemp; }".

Data Parsing:

I parsed the data using a python command called “split” this split my input string by semicolon and put it into a list. I then went to the corresponding index to receive the data I was looking for. The trouble I ran into was my GUI freezing when I would parse the data. The GUI would not update the values or allow the user to interact with it as it received the data. In order to fix this, I had to create a separate thread that was constantly reading in data from serial and updating global variables. This thread was launched at the start of the program and ran through the entire runtime of the program.

GUI:

In order to show the user their car stats in real time we needed to use a Raspberry PI touch display that connected directly to the pins of the Raspberry PI. The display required a specific operating system and drivers that were included. After setting the display up we needed to program a GUI for the user to interact with. The GUI was programmed in python using a library called “Tkinter”. The GUI consisted of three modes: race mode, stats mode, and graph mode. When the user clicks one of the modes a separate thread was launched to update the values at a constant rate. To minimize CPU usage, the thread

was then destroyed when the user navigates to a different tab. The Pi with display is pictured below.

- Race mode displayed a live feed of the car's speed, current gear, and RPM. The screen will flash green when the driver should shift gears and flash red when the car is redlining.
- Stats mode displayed all of the current statistics of the car including: current gear, speed, RPM, oil temperature, gps coordinates, torque, engine load, runtime, intake temperature, and accelerometer data. The statistics were updated in real-time using a separate thread to read the values in.
- Graph mode displays the users speed data from the start of the application. The speed data is stored in to an array as it is received from serial. When the graph tab is selected, the speed data is graphed using a python library called "matplotlib". The graph is updated every 5 seconds so that the user can see real time results as the car is running. A separate thread is launched to create the graph and destroyed when the graph is done plotting the user's speed.

Resources:

https://www.raspberrypi.org/blog/the-eagerly-awaited-raspberry-pi-display/ - Raspberry Pi display

http://www.instructables.com/id/Raspberry-Pi-Arduino-Serial-Communication/ - Interfacing of Arduino and Raspberry Pi via USB

Conclusion

Our main takeaway from this project was the confirmation that the Arduino is very capable of multitasking and some boards, such as the Mega, have a very large amount of inputs and outputs. If we were to do this project again, it would be completely feasible to use one Arduino Mega in place of the three Arduino boards we used. In retrospect, the use of multiple Arduinos only complicated the sending and parsing of data when one Arduino would have been quite capable of taking in each sensor's data, parsing all the values, and building the data string for sending to the Pi.

If given more time, we would have worked on perfecting the data sent from the Arduino to the Pi. The data received on the Pi would sometimes include erroneous values or would be in an incorrect order, for reasons we were unable to determine in time. If we were to do this project again, we would develop a more robust protocol for sending data to the Pi that would include error checking and data validation on the Pi's end. One possible cause for these data errors was the number of serial connections the data passed trough on it's way to the Pi. A serial connection can sometimes send garbage or transmit/receive too quickly, resulting in strange values being sent. We tried to combat this with short delays when reading the serial buffer and had some success, but it was not a perfect solution.