Table of Contents

1.0 Executive Summary

One can communicate with a vehicles' Electronic Control Unit through the vehicles' OBDII port. Using this method one can view information on a car such as mileage, mpg, fuel consumption and error codes. One can also view the air/fuel ratio, timing and many other parameters to observe the performance of a vehicle whether it is for fuel economy or for speed. This can all be done using a Scantool device that connects to the OBDII port and provides readouts to the user. Normally this method is mainly used when the vehicle has a check engine light (CEL) lit. The Scantool can read the data and show the user the error code that the OBDII port is throwing. However, it is up to the user to now take the code and research what it means and how to fix it on their specific vehicle. Now, there are ways to read these check engine lights and other data using your android powered device and connecting through a Bluetooth unit. This method is very effective and allows the user to connect to the internet and find out how to solve their check engine light error. Also, there are applications that will use the android phone as a remote keyless entry system so that one may start, unlock, and lock their vehicle which is especially useful if the vehicle did not come with that feature pre-installed. The drawback to these applications is that there is not one application that performs both functionalities described above. Also both applications need a separate device to perform those specific functions, therefore if one wanted to have both applications, the vehicle may become cluttered with devices. Finally, the main drawback is that they are expensive. Unlocking and starting your car from an android device can cost upwards in the 300's.

We propose that we use an Android phone to do both functions in one application as well as one device that will connect to the OBDII port and be able to unlock, start and lock the vehicle in question. And we also propose that this be done wirelessly and information may be sent wirelessly via Bluetooth connection so that the user will not need to have a wire running from their phone to the OBDII port on their vehicle. This way one can have full control over their vehicle through their phone and look up trouble codes and how to fix them using the android phone connected to the internet. Also the user will be able to save the data that has been read from the OBDII port away into log files on the android device to be viewed later. Logging data in this way will be a much cheaper and efficient way of keeping track of information on the user's car and providing remote access to ones vehicle. The system will be able to read a multitude of data and erase error codes. The system may also be able to start the car, wind down the windows and open the trunk depending on the vehicle. A complete list of the specifications and requirements of this project will be discussed in the following section as well as a discussion about the motivations and goals for this project.

2.0 Project Description

The following section will contain information with regards to our motivation and goals of this project, in addition to our objectives and our requirements and specifications. The requirements section will be broken down into subcategories, software and hardware.

2.1 Motivation and Goals

The motivation for this project is simple; we wanted to engineer a cheaper and easier way for automobile drivers to monitor critical vehicle data from a car's on board computer. As well as remotely start a vehicle, control windows, locks, and car alarms. We wanted drivers to be able to diagnose warning codes such as the dreaded "check engine" light, without having to rush their vehicles to the repair shop. Some versions of this project already exist in one form or another, these projects, and how they differ from ours, will be discussed in a later section. Our goal is to build a version that is cheaper, more inclusive, and easier to install than the products that are available today.

Overall, we believe that this project can potentially save users time and money in many different ways. For one, users of this app could potentially save hundreds of dollars on mechanic diagnostic fees. Also, by using this application to read critical vehicle data, such as tire or brake information, the user could save their own life and the lives of others.

2.2 Software Requirements

The software requirement section will be broken down into two sub sections, the first will pertain to the requesting and reading of vehicle data and the second will pertain to the graphical user interface (GUI).

2.2.1 Requesting and Receiving Data

The Android application software will be written in Java and will be the engine that powers the entire project. A main aspect of this project will be sending request data. The application requirements as far as sending requests are as follows:

- Must make a connection to microcontroller powered by Bluetooth
- Must interpret what user wants to do and send request to proper location
- Must include proper headers on all request data so packet ends up at intended destination
- Must be able to send and receive data on the following functions:
  - Timing Advance
  - EngineRPM

- Coolant Temperature
- Throttle Position
- Fuel Level
- Time Since Engine Start
- Air Intake Temperature
- Speed/ Average Speed
- Mass Air Flow
- Intake Manifold Pressure
- Fuel Pressure
- Engine Load
- Fuel Economy/ Average Fuel Economy/ Miles to Empty
- Battery Voltage
- Error Codes

In terms of receiving data, after the return bytes arrive at the phone software, the application must be able to take those bytes and convert them into legible data that the user can read. Data will be sent to the phone in a specific format that is discussed in much further detail below, this format will need to be converted so that the application knows what the data is, and then it needs to convert the actual data accordingly.

2.2.2 GUI Requirements

Like all applications, a user interface is one of the most important features. If the user interface is not intuitive and inviting, the entire project would fail. The GUI should be simple while still looking attractive. It should be intuitive and allow users to take advantage of everything the project has to offer. This can be a challenge because as a developer, you cannot always display everything you would like on such a small screen. However, it is necessary to come up with a way to show the user what they expect in an attractive and readable form. The requirements for this aspect of the project are simple; we must allow the user to take advantage of all the features the project offers and present information in a readable fashion.

2.3 Hardware Requirements

This section detailing the requirements of the hardware will be broken down into two subsections. The first will be the requirements of the OBD-II reader chip; the second will be the requirements of the microcontroller.

2.3.1 OBD-II Reader

The ELM-327 was selected as the OBD-II reading chip for this project. The ELM must conform to the following requirements:

- Make a connection to the automobiles engine control unit (ECU) through the OBD-II port in order to retrieve information.
- Interface with a microprocessor in order to receive commands and provide responses from the ECU.
- Support all five OBD-II protocols.

2.3.2 Microcontroller

The ATmega328 was selected as the microcontroller for the project. The MCU is one of the most critical components of the project. It is necessary that all of the requirements are met. The requirements for our microcontroller will be as follows:

- Receive messages from the Android device over a Bluetooth connection
- Check headers of incoming requests from the android device and determine where the message should be sent
- Interface with the ELM-327 and pass messages to the ELM
- Receive response messages from the ELM and send them back to the Android device
- Provide necessary voltage to start car, roll down windows, unlock doors, sound alarm, and pop trunk

2.4 Project Specifications

The chart in **figure 2.4.1** shows the main functions of our project on the left and the maximum amount of time it should take to execute those functions. We believe starting the car and unlocking the doors will take slightly longer than the rest of the functions because in addition to supplying voltage to the appropriate wires, the microcontroller will also need to disable the car's alarm.

Function	**Spec**
Starting Car	10 Seconds
Window Control	3 Seconds
Lock Control	5 Seconds
Alarm Control	3 Seconds
Trunk Control	3 Seconds
OBD-II Reading	3 Seconds
Range	25 Feet

Figure 2.4.1 – Specification Table

3.0 Research related to Project Definition

3.1 Related Projects

The products discussed below each exude similar functionality and purpose as our project however neither of these encompass the broad range of features that our project does. In a sense, our project is a combination of the projects discussed below. We were not able to find any other projects with the same repertoire built into one piece of hardware and one application like ours will be.

3.1.1 Viper SmartStart System™

The Viper SmartStart is an application for BlackBerry, iPhone and Android mobile devices and its purpose is to allow its users to "start your car from virtually anywhere."
The software's other main features:

- Lock/Arm
- Unlock/disarm
- Trunk release
- Panic or car finder

This product is similar to ours in that we will also implement all of this core functionality; however we hope to do so at a fraction of the cost.

The Viper SmartStart application is free, however the hardware module can cost anywhere from $400 to $600, depending on the type of vehicle and type of system being installed, in addition to the price of installation at retailers such as Best Buy.

Some differences between our project and the Viper System are:

- Connectivity – our project utilizes Bluetooth whereas the Viper system connects over a network. The advantages to using Bluetooth are that it is more cost effective. The Viper users have to pay a monthly charge and sign 1 or 3 year contracts to use the system whereas Bluetooth is free. Another advantage is that a user cannot accidentally start his or her car from hundreds of miles away.
- Features – In addition to the features listed above, our project will also include the OBD-II reader which will allow users to check and clear error codes, and read vehicle diagnostic information. The Viper SmartStart does not include this functionality.
- Cost – As stated above, we plan to implement the core functionality of the Viper system for only a fraction of the cost and no monthly charges or long term commitments for users.

Despite the existence of this product, we believe that our system includes enough extra features, and for such a small fraction of the cost, that our project has motivation and significance.

3.1.2 Torque Android Application

Torque is an engine diagnostics application that allows users to monitor their car's ECU and retrieves information in a similar manner to our project. The hardware that this application uses is a Bluetooth reader to connect the android device to the car's OBD II port. We plan to base our hardware off of the ELM327 Bluetooth OBD-II reader interfaced with a microprocessor to allow for extra features. The reading capabilities of our project should meet or exceed those of this product. Some of the advertised features listed for the Torque software:

- View live engine data on your Android phone - Connect to your vehicle ECU
- Fully customizable dashboard screens - Design your own layouts and custom dials, use your own themes
- Retrieve Fault Codes (DTCs) and clear Check Engine lights - View fault descriptions using the built-in databases
- Upload live data to your webserver or the torque web viewer in real-time
- Check the performance of your vehicle with BHP / Torque / 0-60 & Quarter Mile widgets

We expect our software to deliver most if not all of these features with the addition of the extra features such as starting the car, locking and unlocking the doors, arming and disarming the alarm and more.

Even though this application exists for a relatively cheap cost of 5.99, we believe our project still has purpose because of the added functionality mentioned above.

3.1.3 OnStar Application

Recently GM has released an app for iPhone and Android that allows users to start their car from anywhere in the world, lock and unlock the doors, honk the horn and read data from their cars such as fuel level, oil life and tire pressure.

Some of the differences between this OnStar App and our project are, like the SmartStart tool, our project will read more information than this OnStar app. Our project would be geared more towards a savvy car enthusiast who knows what he or she wants to read from the OBD-II data. The OnStar app seems to be geared more towards the everyday driver.

The **figures 3.1.3a** and **3.1.3b** below show some of the different interfaces of the OnStar application. These can be compared to our screen shots given in section 15 of the User Interface heading.

The first image below shows the screen that allows the user to lock or unlock the doors, start the car, or sound the horn. The second screen shows the fuel tank level, and how many miles can be driven before the vehicle runs out of gas

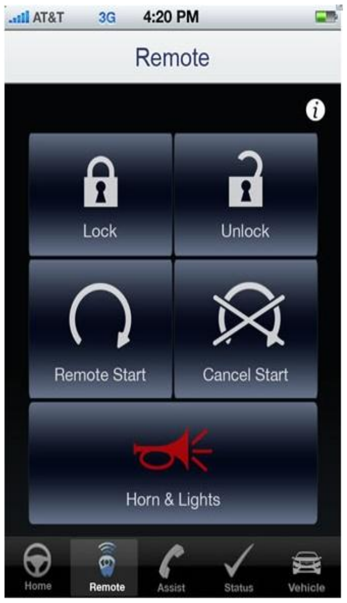


Figure 3.1.3a – Car function Screen

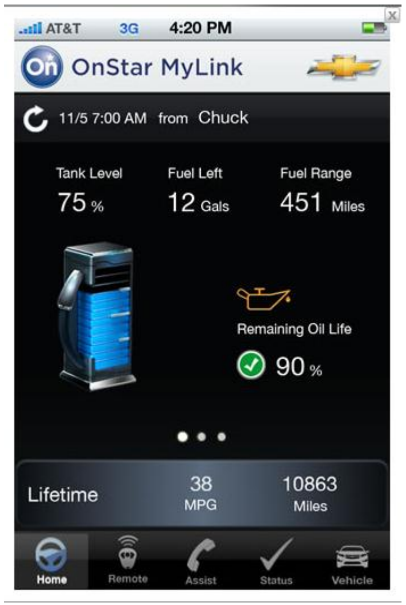


Figure 3.1.3b – Car data screen

3.2 High Level Design Options

There were two design options being contemplated. For our project, the main factors we considered when deciding on a design were price and user experience. We wanted the cheapest and most simple design while still achieving the best possible user experience.

The two options we considered differ only in connectivity. The first design had only one connection between the android app and the hardware module, the second design featured a dual Bluetooth connection. High-level diagrams can be seen in the subsections below.

3.2.1 Single Connection Design

This design is the one that we decided to move forward with. We believed it to yield the better performance of the two in terms of speed and user experience. The **figure 3.1.1** below shows a high level overview.

In the design shown above, the flow of data is as follows:

- Android application sends data to MCU
- MCU then checks format of data. If data is intended for the OBD-II port, it is passed on to the ELM and the MCU would await the response from the ELM; else if the data request is to start the car, control windows, unlock doors etc. The appropriate voltage is applied to the appropriate wires.
- Once the ELM receives data in a readable format, it converts the data into a something the engine control unit (ECU) understands and passes it on to the OBD-II port.
- The OBD-II takes the ECU's response and gives it back to the ELM.
- The ELM then sends the response back to the MCU, which then relays it back to the Android phone. The application could convert the response bytes into a user friendly format. Such can be seen in the Functions section 7.

We believed this design would yield better speed than our second option because the application will not have to switch between Bluetooth connections depending on what the user wants to do. It is always sending the data to the MCU as opposed to sending data to two different destinations. (MCU and ELM)

The diagram below shows two different arrows, the bi-directional arrows represent data flowing in both directions, for instance, from the android application to the microcontroller, then from the microcontroller to the ELM and from the ELM to the OBD-II port, data will need to travel both ways along that path. However, data will only need to travel one way from the MCU to the car wires. We don't believe it necessary to send an acknowledgment back because we are assuming the user will see if the request failed to go through just by observing the vehicle. If you send a request to start the vehicle, and somehow the request is lost or corrupted, the user would see that the car failed to start, and they would simply, resend the request.

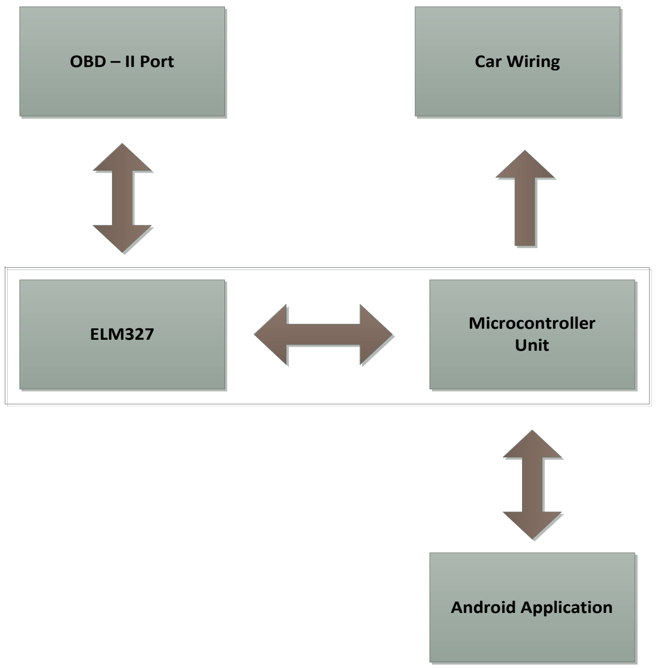


Figure 3.1.1 – Block Diagram for single connection design option

3.2.2 Double Connection Design

The second design features two separate Bluetooth connections. We considered this design because it would appeared to be the simplest to implement; however, we feared that the speed will suffer since the mobile phone can only connect to one Bluetooth connection at a time, there would be constant switching involved.

If this design were to be implemented, it would reduce the workload of the Microcontroller. In our previous design, the microcontroller would have to check the header of the data it receives to figure out its final destination. In this design, that responsibility lies with the application, the data flow is as follows:

- The application user interface would be split into two options. One would be the OBD-II features involving the retrieval of data from the vehicle, and the other would be starting the car, unlocking doors and other commands involving wiring.

- If the user selects the first option. A Bluetooth connection with the ELM is automatically made and the request is sent.
- If the user selects the option involving the car wiring, a Bluetooth connection to the microcontroller is automatically made and the request is sent.
- From here, the design is similar to the first design with some minor variations.
- If the ELM receives a request, it converts it and sends it through the OBD-II port to the ECU, which sends its response back to the ELM and then the ELM relays it directly back to the phone application.
- If the microcontroller receives a request, it simply applies voltage to appropriate wire.

The reason this design is simpler is because the data isn't being relayed from component to component as much. Also, the programming of the microcontroller would be simpler. For instance, instead of all the data coming and going through the microcontroller like in design one. In this design, the MCU would only have to take in data from one source (the phone) and send data to one source (the car). This would greatly simplify the microcontroller programming.

The **figure 3.2.1** shows the high level block diagram of this design. Notice how the first design has three bi-directional connections, but this design has only two. The data path is more specific, interacting only with the components it needs to.

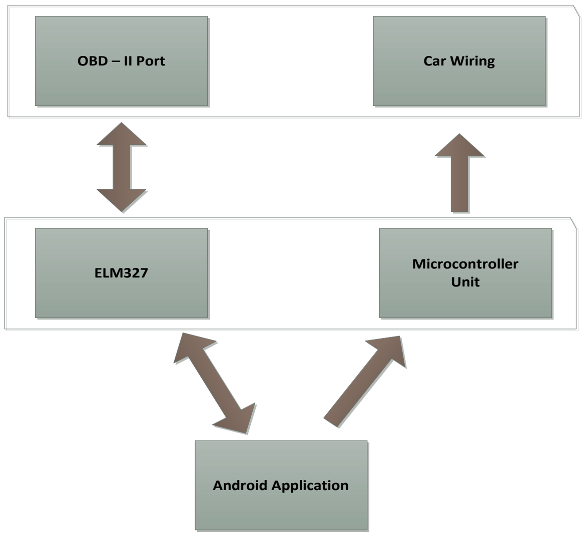


Figure 3.2.1 - Block diagram two connection design

In this design, data flowing in both directions when a request is sent to the ELM, since the ELM will have to send the response back to the phone. However, the microcontroller will only have to have one input (the phone) and one output (the car wires) and the data only needs to flow in one direction for the same reasons as mentioned above in the previous design.

3.3 Bluetooth vs. Wi-Fi

Our project required a wireless communication between a smart-phone and the hardware device connected to the 1998 Honda Accord. The two standard methods of wireless communication are Bluetooth Technology and Wi-Fi. We needed to examine the benefits of each of our primary wireless communication options. Only after close examination, we will choose the best wireless technology for the project.

3.3.1 Bluetooth Technology

Bluetooth technology is a short-range wireless communications technology. It uses radio waves to communicate, similar to the ones used for television or your standard AM/FM radio. Although Bluetooth technology uses radio waves, it's only designed to be used in the "Personal Area Network", more commonly known as PAN. Bluetooth's range is dependent upon the application. There are three common classes of radios used within Bluetooth technology. The class of the radio used in the Bluetooth chip, indicates the range it's capable of. Class 1 radios have a range of 300 feet and are typically used for industrial purposes. Class 2 radios have a range of 33 feet and are common among mobile devices such as phones and computers. Class 3 radios have a range of 3 feet and are used in very specific applications.

Bluetooth technology operates at an adaptive frequency between 2.4 GHz and 2.485 GHz. This allows for minimal interference. The Bluetooth device becomes aware of the operating signals in the area and avoids them to prevent interference.

Bluetooth technology has a unique design that allows for minimal power consumption. The most widely used class 2 radio, which is used in this project, uses 2.5 mW of power in most cases.

3.3.2 Wi-Fi Technology

Wi-Fi enabled devices allow for connectivity to the internet through a wireless network. These wireless networks can range from as small as a few rooms to as large as a few miles. Typically the smaller Wi-Fi ranges are used for personal use within a house or office. The larger Wi-Fi networks may span a large corporate office or university campus. The range of Wi-Fi is nice, but may be outside the scope of our project since our wireless devices will be communicating within feet of one another.

Wi-Fi is efficient when it comes to fast speeds and large ranges, but has it's downfall in power consumption. Over the years, the main concern with Wi-Fi enabled devices, is battery life. The amount of battery life in a device can be dependent upon the use of Wi-Fi. Since this project will be operating on a 12 volt car battery and the lithium ion battery in the mobile device, we need minimal power consumption. Wi-Fi may consume more power than necessary for our application.

3.3.3 Bluetooth Vs. Wi-Fi Conclusion

Bluetooth technology allows for minimal power consumption but has a small network range while Wi-Fi consumes a significant amount of power allowing for large network ranges. The project allows for either short or long ranges but would prefer minimal power consumption. After the analysis of both Bluetooth Technology and Wi-Fi networks, we've decided to use Bluetooth Technology for our devices to communicate wirelessly.

4.0 Hardware

4.1 OBDII

The vast majority of this project is centered around the Onboard Diagnostics standard in our vehicles, more commonly known as the OBD. We will be dealing with the most recent version, the OBDII setup. The OBD is able to communicate with the engine control unit, more commonly known as the ECU. The ECU is onboard intelligence that helps manage the vehicle.

4.1.1 OBDII Background

In 1970, the United States government made an effort to clean up air pollution by passing the Clean Air Act. As a result of the Clean Air Act, the OBD standard was created and introduced the Society of Automotive Engineers(SAE). In the early stages, manufacturers had specialized monitoring devices and tools to aide in corporate matters. At this time, these devices and tools were not typically found in the hands of the consumer. This is considered the first OBD standard and the SAE hoped it would encourage manufacturers to develop more efficient vehicles in the emissions and fuel economy departments.

The first OBD standard had multiple issues and would need some revising to be an efficient regulation. The information retained on a vehicle's ECU would change from vehicle manufacturer to vehicle manufacturer. Each vehicle manufacturer had a unique set of diagnostic error codes, making it hard to create universal diagnostic tools. The data link connector was not universal among vehicle manufacturers. This means diagnostic devices could not interface with the ECU of all makes and models. The first OBD standard needed some serious standardizing to make it an efficient regulation. After this realization, the SAE came out with a new standard, the OBDII.

In 1996, the OBDII standard was developed bye the SAE. The objective was to correct the issues in the first OBD standard. A physical data link connector became a mandatory part of the new OBDII standard. The common connector, J1962, is found in all makes and models. This new standardized connector allowed for manufacturers to make universal diagnostic tools that could hook up to any vehicle. General diagnostic trouble codes were also made universal among vehicle manufacturers. The standard allowed for vehicle manufacturers to have “extra” diagnostic trouble codes as well to fit their specific needs. For example: BMW may need more diagnostic trouble codes than most manufacturers to accommodate for their vehicle’s excessive features. SAE specified four regions of the vehicle that represented a set of diagnostic trouble codes. Although the OBDII standard is a vast improvement over the first OBD standard, it’s not perfect. There are multiple different protocols that correspond to the different makes and models of vehicles. The different protocols operate at different speeds of data transfer. Typically, the faster the protocol, the better it is considered. A protocol that can get twenty-five readings per second is more useful than a protocol that can only read ten readings per second. In 2008, the SAE corrected the standard by making ISO 15765-4 the standardized protocol for all vehicles after 2008. The ISO 15765-4 protocol allows for faster read speeds than before.

OBDII ports were not required to be in cars until the beginning of 1996. All cars and small trucks built in 1996, or later, should have OBDII capabilities. Some vehicles that were built towards the end of 1996 may also be equipped with the OBDII port. **Figure 4.1.1a** shows what an OBDII port may look like. The OBD standard was developed in an attempt for vehicle manufacturers to produce more efficient cars along the lines of fuel economy and emissions. If a vehicle has OBDII capabilities, it will have an OBDII port and a written indication under the hood of the vehicle. The documentation located on the bottom side of the hood should read: “OBDII compliant” if the vehicle is OBDII equipped.

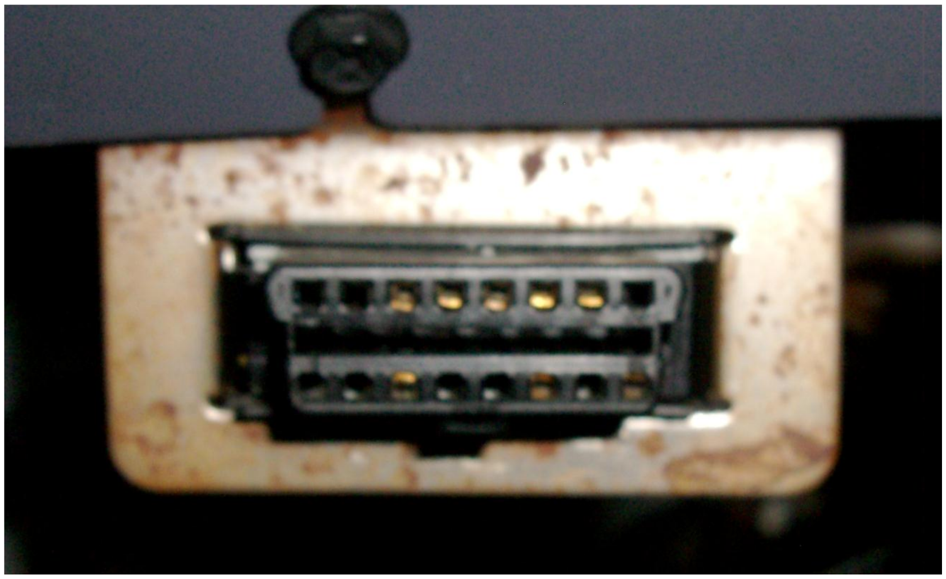


Figure 4.1.1a – OBDII port of 2000 Dodge Durango

The OBDII standard allows for all parties to benefit. Vehicle owners and mechanics use OBDII technology to access useful information from the ECU to diagnose vehicle trouble or get other useful information such as fuel economy. The OBDII port has the capabilities to allow the user to read error codes and target in on specific areas of the vehicle for close monitoring.

4.1.2 OBDII ELM327

The OBDII reader used in this project was equipped with Bluetooth technology. Specifically the ELM-327 Bluetooth chip was used.

The J1962 connector used as the ODBII interface for test tools does not directly connect with standard computers. Another issue is the wide range of different ODBII protocols used. The protocols differ on multiple levels including formatting and signaling. Additional hardware or wireless communications are necessary to help decipher protocols. An integrated circuit named the ELM327 was created to connect between RS232 ports and OBD ports. The ELM327 can handle all of the OBD protocols including the newest protocols in high tech vehicles.

The ELM327 uses ASCII to communicate with the OBDII port. The ELM327 also includes enough onboard memory to be able to keep track of any necessary changes. Some of these changes may include setting the timeout interval when it's receiving messages from the ECU. In the case that the ELM327 doesn't receive an AT type command, it assumes that the command is intended for the ECU. Before the ELM327 passes the command to the ECU, it makes sure that the command meets the standards of the OBDII set by the SAE. In the case that the ELM327 doesn't understand the request, it responds with a question mark.

The ELM327 plays the part of a command line interface, commonly known as a CLI. This means that the ELM327 will always respond with a '>' character to the serial port. The ELM327 will not execute any commands until it reads in a line break or carriage return.

4.1.2.1 ELM327 Circuit

Although the ELM327 is a great integrated circuit, it's not enough to complete the desired functions for this project. To be able to utilize the ELM327, a complete circuit needs to be created for interfacing capabilities. **Figure 4.1.2.1a** is a block diagram of the circuit needed to complete the task of becoming an OBDII reader. It is necessary to have a clock to power the ELM327 integrated circuit.

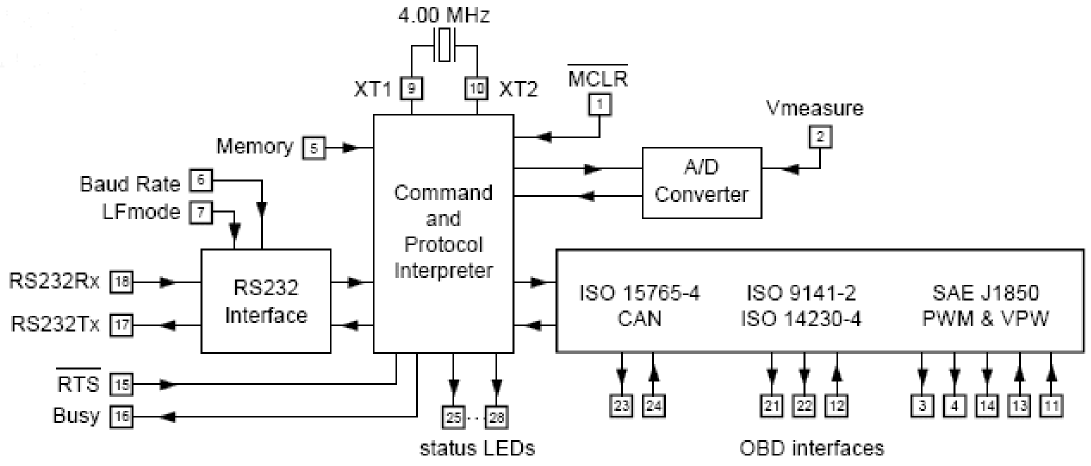


Figure 4.1.2.1a - OBDII reader block diagram – Pending Permission

ELM electronics have come up with a recommended circuit for the ELM327 IC. The schematic for the recommended circuit is seen in **figure 4.1.2.1b**.

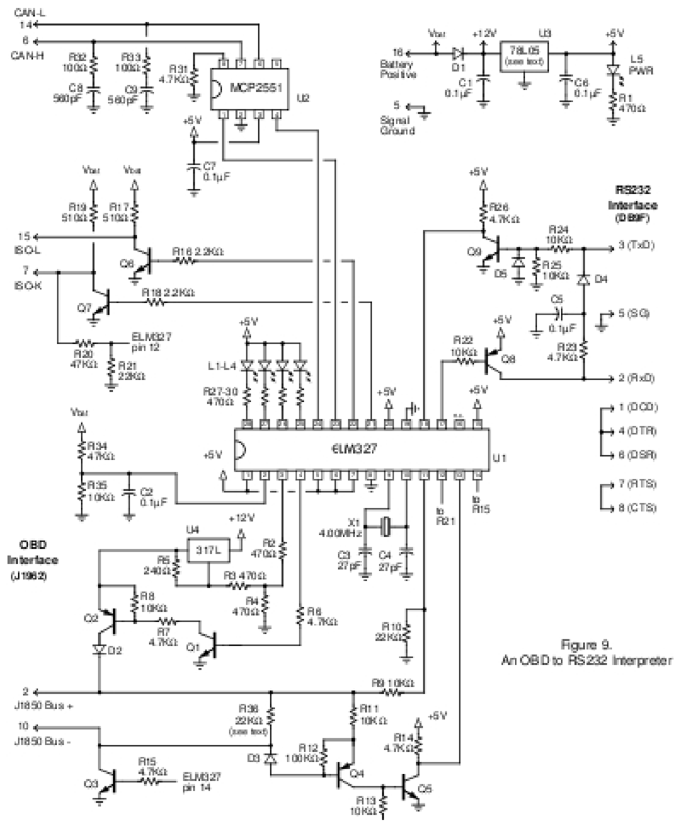


Figure 4.1.2.1b – Typical circuit schematic for ELM327- Printed with permission

4.1.3 OBDII Specifications

Where is my OBDII Port located? Regulations specify that the OBDII port must be within three feet of the driver and must not require tools for access. The typical location is under the dash on the driver side. Some manufacturers will actually “hide” the OBDII port elsewhere, such as behind an ashtray.

4.1.3.1 OBDII Port and Pins

The OBDII reader has sixteen contacts that plug into the OBDII port to interface with the vehicle. Each pin-out has a specific function. Pint number one is not a standard pin for all makes and models. The functionality of pin one is left up to the discretion of the manufacturer. General Motors typically uses pin one as "J2411 GMLAN/SWC/Single-Wire CAN.” Pin number two is designated for the positive BUS line of the SAE-J1850 PWM and SAE-1850 VPW. Ford and Chrysler use pin three of the OBDII port. Ford uses this pin as DCL(+) while Chrysler uses it as CCD BUS(+). Pin number four has a universal functionality among all makes and models. Pin four is always designated as the “chassis ground.” Pin number five also has universal functionality among vehicle manufacturers as "Signal ground." Pin number six is designated as "CAN High" among vehicle manufacturers. Pin seven is the "K line" of ISO 9141-2 and ISO 14230-4 in all vehicle makes and models. Pins eight, nine, twelve and thirteen are left to the manufacturer's discretion. Pin number ten is designated as the "Negative BUS Line of SAE-J1850 PWM" among all vehicle manufacturers. Pin eleven is commonly used by Ford and Chrysler in their onboard diagnostics unit system. Ford typically uses pin eleven as "DCL(-)" while Chrysler uses it as "CCD BUS(-)." Pin number fourteen is designated by the SAE to be the "CAN Low." Pin fifteen is designated by the SAE to be "L line" of ISO 9141-2 and ISO 14230-4. The last pin, pin sixteen, is designated to be the battery voltage in all vehicle makes and models. The figure below, **figure 4.1.3.1a**, summarizes the descriptions of each of the numbered contacts in the OBDII port.

OBDII Port Contact Specifications	
1. Manufacturer Discretion. GM: J2411 GMLAN/SWC/Single-Wire CAN	9. -
2. Positive BUS Line of SAE-J1850 PWM and SAE-1850 VPW	10. Negative BUS Line of SAE-J1850 PWM
3. Ford DCL(+) Argentina, Brazil (pre OBD-II) 1997-2000, USA, Europe, etc. Chrysler CCD Bus(+)	11. Ford DCL(-) Argentina, Brazil (pre OBD-II) 1997-2000, USA, Europe, etc. Chrysler CCD Bus(-)
4. Chassis ground	12. -
5. Signal ground	13. -
6. CAN High (ISO 15765-4 and SAE-J2284)	14. CAN low (ISO 15765-4 SAE-J2284)
7. K line of ISO 9141-2 and ISO 14230-4	15. L line of ISO 9141-2 and ISO 14230-4
8. -	16. Battery voltage

Figure 4.1.3.1a – OBD-II pins

4.1.4 OBDII Uses

The OBDII port has many different uses and functions. Some, but not all, will be used in this project. The OBDII port is capable of reading the following information from a vehicle:

- Turbo Boost Pressure (PSI) *
- Fuel Economy (Real-Time/Avg/Trip)
- Timing Position
- Speed (MPH, KPH)
- Engine RPM
- Coolant Temperature
- Injection Pulse width (IPW)
- Throttle Position (as a percentage)
- Air Intake Temperature
- Mass Air Flow (g/sec)
- Throttle Position
- Fuel Level *
- Barometer
- Battery Voltage
- Engine Oil Temperature
- Injection Control Pressure (ICP)
- Transmission Temperature
- Load

The OBDII port is capable of altering minor specifications within the vehicle. The OBDII port is can modify the following:

- Change Speed Limiter
- Adjust Timing + or - 2 degrees
- Seat Belt Reminder Chime *
- Auto Door Lock *
- High Rev Function
- Change Rev Limiter
- Calibrate Speedo
- Tune transmission

The '*' indicates that the specified function may be specific to certain vehicles, not all car makes. For example: It's not possible to read the Turbo Boost Pressure of a vehicle that doesn't have a turbo-powered engine.

4.2 Window Mobility

It is possible to control the windows from your Android-based smart-phone. To implement this feature, we'll need to splice or tap into the wiring harness of the vehicle using wire taps. A 12-volt signal needs to be sent to the motor controlling the windows in order to roll the windows up or down. A wire tap is pictured below in **figure 4.2a**.

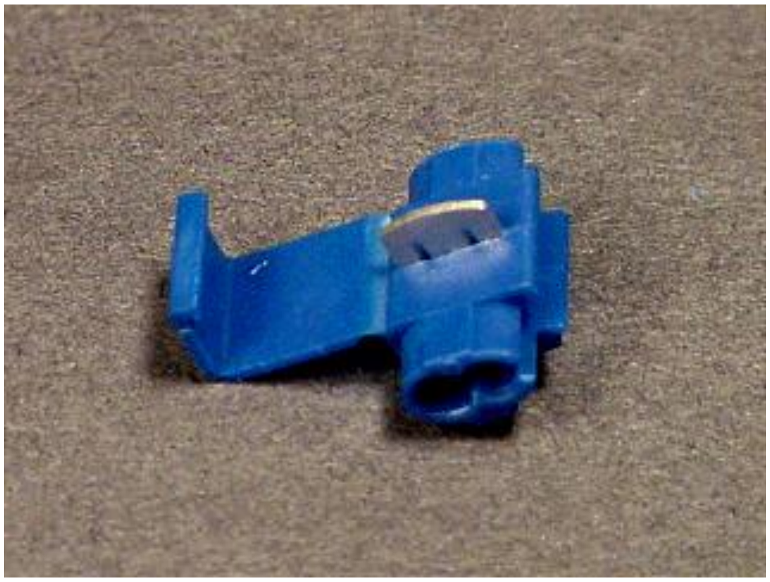


Figure 4.2a - The figure above is an example of a wiretap. It taps in and makes a connection with the wire without having to completely cut the wire. – Permission granted from http://www.midterminc.com (See e-mail in appendix)

4.3 Hardware Selection

This section will detail the different hardware components to be used in this project and why they were selected.

4.3.1 ELM-327

We debated whether to create our own integrated circuit to read from the OBD-II or to use a pre-designed chip. We decided to use the ELM327 which is a widely used IC when it comes to automotive applications. The ELM327 is popular because of its versatility when it comes to the different OBD-II protocols. It supports all five of them. If we had designed our own chip, it would have been time consuming to provide support for five different OBD-II protocols, and we wanted our project to support as many different vehicles as possible. Since the ELM chip was already fairly low-cost, we did not see the need in "re-inventing the wheel" and we decided to just go with the ELM.

4.3.2 MCU Selection

The ATmega328 was selected as the microcontroller for this project. We selected this chip because it has the speed and power to support our purposes. In addition, it is very well documented and widely used so we should have no problem programming it. It has a max of 23 I/O pins which will be enough for our project.

4.3.3 Test Board

When selecting our test board, we needed to make sure it was user friendly, and that it was powerful enough to meet our needs. We chose Arduino because our research showed that Arduino boards are quick and easy when it comes to learning how to program on them. Also, with the Arduino board we had the option to purchase a board with Bluetooth built in or we had the option to build a board with PCB software. The boards are very well documented and we liked that they could be programmed in C or C++.

4.3.4 Android Phone

Android was selected as the platform of choice for this project for many reasons. For one, it is one of the most widely used mobile phone platforms. Since it is so widely used, it gives our application a big potential user base. Another reason we chose Android is because it is very developer friendly. It utilizes the java programming language which we all have experience with and we are comfortable programming with it. In addition, two of our team members already had Android devices, so we could use those for testing and developing and did not have to buy a phone.

5.0 Software Selection

This section details the software components involved in building the project and why they were selected. The microcontroller programming software, the android programming software and the PCB software used to design the integrated circuit.

5.1 MCU Programming Software

Since we are using an Arduino board, the Arduino language will be used for programming the microcontroller. There are other options such as AVR Studio; however, from our research, we have gathered that Arduino’s programming IDE is very user friendly and a fairly fast learning process.

5.2 Android Programming Software

Eclipse IDE will be used to program the Android application. We chose Eclipse because it has a plug-in to easily program for Android using Java. Most of the team is familiar with Eclipse when programming in java so it should reduce the learning curve.

5.3 PCB Design Software

In order to put our MCU and ELM327 on a single board, we needed to design a PCB. We have decided to use EAGLE PCB Editor to design this board. Since the Arduino boards are open source, the EAGLE files are given on the website along with permission to edit and design new boards. The ELM327 Schematics are also available. So EAGLE will be used to interface these two circuits and print them on the same board, we will use the Arduino circuit for testing and we will design our final PCB based on the Arduino, using only what we need.

6.0 OBD-II Protocol

6.1 Background

Despite the fact that OBD-II is standard on all vehicles made after 1996, there exists five different signaling protocols. Most vehicles utilize just one of these protocols. The five protocols and some properties of each can be seen below:

1. SAE J1850 PWM (Standard of the Ford Motor Company)
    - Pin 2: Bus+
    - Pin 10: Bus-
    - High voltage is +5 V
    - Message length is restricted to 12 bytes
2. SAE J1850 VPW (Standard of GM)
    - Pin 2: Bus+
    - Bus idles low
    - High voltage is +7 V
    - Decision point is +3.5 V
    - Message length is restricted to 12 bytes
3. ISO 9141-2 (Primarily used in Chrysler, European and Asian vehicles)
    - Pin 7: K-line
    - Pin 15: L-line
    - UART signaling
    - Message restricted to 12 bytes
4. ISO 14230
    - Mostly the same as ISO 9141-2
    - Message can contain up to 255 bytes
5. ISO 15765 CAN
    - Pin 6: CAN high
    - Pin 14: CAN low

All OBD-II pin-outs use the same connector but different pins are used except for pin 4 (battery ground) and pin 16 (battery positive).

For the purposes of this project, our hardware will support all protocols and our software will support the ISO 9141 protocol due to the fact that the testing will be done on a 1998 Honda Accord which uses the ISO 9141 protocol; however, if time permits we will extend our software to support the other four protocols.

6.2 Requesting Data

Data requests are sent in a standard format from the diagnostics tool to the OBD-II port. The first 3 bytes sent are the header. Then, 1 to 7 data bytes follow. Lastly, there is an error check byte. A high level view of a request message can be seen in figure 6.2.1 below.

Header Bytes

0	1	2	3	4	…	8	9	10
Header	Header	Header	Data	Data	Data	Data	Data	CRC

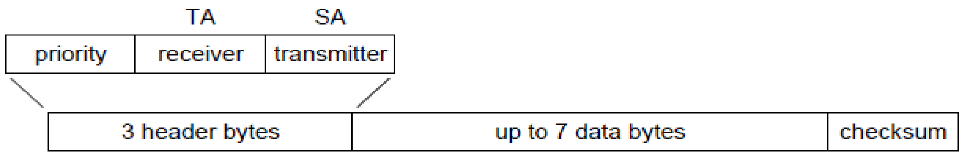


Figure 6.2.1 – High Level Request Message – Printed with permission
The header bytes can vary depending upon the protocol, the below figures show a breakdown of the header bytes **figure 6.2.2** shows the header for a vehicle using the ISO 9141 protocol.

	Byte 0 (Priority/Type)	Byte 1 (Target Addr.)	Byte 2 (Source Addr.)
Request	104 (0x68)	106 (0x6A)	241 (0xF1)

Figure 6.2.2 – Header Bytes for ISO 9141 Request

The **figure 6.2.3** below gives the header bytes for a request message sent to an OBD-II port using the SAE J1850 PWM protocol.

	Byte 0 (Priority/Type)	Byte 1 (Target Addr.)	Byte 2 (Source Addr.)
Request	97 (0x61)	106 (0x6A)	241 (0xF1)

Figure 6.2.3 – Header Bytes for PWM Request

The **figure 6.2.4** below gives the header bytes for a request message sent to an OBD-II port using the ISO 14230 protocol. The bits LLLLLL represent the length of the data byte section of the message.

	Byte 0 (Priority/Type)	Byte 1 (Target Addr.)	Byte 2 (Source Addr)
Request	11LL LLLL (binary)	51 (0x33)	241 (0xF1)

Figure 6.2.4 – Header Bytes for ISO 14230 protocol

Data Bytes
The first data byte indicates the mode. There are 9 possible modes for diagnostic requests; therefore the range of the first byte is from 1 to 9. Below is a description of each mode.

- Mode 1 – Used to obtain current diagnostic data: Number of trouble codes set, status of onboard tests, vehicle data such as engine RPM, temperatures, ignition advance, speed, air flow rates, information on fuel system.
- Mode 2 – Similar to mode 1 except instead of current data, it pertains to data that was stored at a moment in time, such as when an error code was turned on.
- Mode 3 – Requests all diagnostic trouble codes from vehicle. It is possible that there will be more than one response message if the number of error codes exceeds the available data bytes.
- Mode 4 – Simply instructs the vehicle to clear all error codes.
- Mode 5 – An optional mode used for requesting results of an oxygen sensor test. Some vehicles report this under mode 6.
- Mode 6 – Used for obtaining test results for non-continuously monitored systems. This is optional and is defined by the vehicle manufacturer if used at all. For this reason, it probably won't be included in our project.
- Mode 7 – Optional mode similar to mode 3. This mode returns trouble codes which may be set after a single drive cycle. This is useful for checking the results after a repair has been done.
- Mode 8 – Used to request control of an on board system. This mode is manufacturer defined.
- Mode 9 – Optional mode used to report vehicle information such as the VIN and information stored in the ECU.

The second data byte is the Parameter Identification (PID) bytes. This byte is used to specify what data is being requested. The remaining data bytes in the request message are for further specification of what data is being requested.

The final data byte is the CRC or checksum byte, depending on what protocol is being used. This byte is used to check for any errors that might have occurred during data transfer.

The **figure 6.2.5** below takes the information discussed in this section and details each byte that makes up a request message, in addition, the chart explains the purpose and function for each byte

Byte	Header	The first header byte details the priority and type of the following message.
0	Header	The second header byte details the target address of the message, i.e. which part of the ECU should receive this request.
1	Header	The third and final header byte describes the source address; in this case it would be the address of the OBD-II hardware device so the response message knows where to go.
2	Data	The first data byte specifies the mode; the mode can be anywhere from 1 to 9.
3	Data	The second data byte is the Parameter Identification (PID). This is a 2 digit hex value that indicates the data that is being requested.
4 - 9	Data	Further specification of data
10	Error check	CRC or checksum byte

6.2.5 – Request Message

The section below will detail the PIDs for the most important modes (1 and 2) and the values that each PID returns. Some of the PIDs discussed are used only for mode 1 or mode 2 exclusively. Specifically, mode 1 does not use PID 02, and mode 2 uses only PID 00 and PID 02 – 0D.

6.3 Return Values

The response sent from the vehicle's ECU back to the OBD-II port has a similar structure as the request message; there are 3 header bytes, up to 7 data bytes and an error correction byte. The high-level view is the same as the request message and can be seen in **figure 6.2.1** above.

The following charts display the three header values for the ECU's response message. In the same manner as the request messages, these responses differ depending on the protocol used. **Figure 6.3.1** below describes the header for the VPW or ISO 9141 protocol.

Header Bytes

	Byte 0	Byte 1	Byte 2
Response	72 (0x48)	107 (0x6B)	ECU Address byte

Figure 6.3.1 – ISO 9141 ECU response Header

The next chart, **figure 6.3.2** shows the three header bytes for a response message sent from the ECU using the SAE J1850 PWM protocol.

	Byte 0	Byte 1	Byte 2
Response	65 (0x41)	107 (0x6B)	ECU Address byte

Figure 6.3.2 – PWM ECU response Header

The next chart, **figure 6.3.3** shows the three header bytes for a response message sent from the ECU using the ISO 14230 protocol. The bits LLLLLL make up a 6 bit binary value that represents the length of the data byte section.

	Byte 0	Byte 1	Byte 2
Response	10LL LLLL	241(0xF1)	ECU Address byte

Figure 6.3.3 – PWM ECU response Header

Data Bytes

The data bytes are organized in the same way for a response as they are for a request. The first response data byte is the mode, just as in the request message, except the response mode has the number 64 added to it. For instance, if the first data byte in the request is a 1, for mode 1, the first data byte in the response would be a 65, since 1 + 64 = 65.
The second data byte in the response message is the PID just as in the request. The PID indicates which values were requested and the remaining data bytes make up the actual response data.
The final byte is the error check byte to determine whether or not an error had occurred during transmission.

The number of data bytes that occur in a response message typically depend of the PID. For instance PID 00 will always return 4 data bytes, A – D, but PID 4 only returns one data byte, A. The data bytes will be referred to as A, B, C, D etc. This also corresponds to the order of transmission on the bus and the order of significance.

- PID 00 – This PID determines which PIDs are supported for the vehicle. The bits of the 4 data bytes A, B, C and D correspond to PIDs 1 – 32.

The most significant bit of A would be PID 01, and the least significant bit of D would be PID 32.

- PID 01 – This PID returns four data bytes. Data byte A describes how many error codes there are using bits 0 – 6. Bit 7 of data byte A is set to 1 if the MIL lamp is on or 0 if it is off. The chart in **figure 6.3.4/5** describes how the remaining bits are used.

Test	Bit = 1 if supported	Bit = 1 if incomplete
Misfire	B0	B4
Fuel system	B1	B5
Components	B2	B6
Reserved	B3	B7

Figure 6.3.4 – Data byte B error codes

Test	Bit = 1 if supported	Bit = 1 if incomplete
Catalyst	C0	D0
Heated catalyst	C1	D1
Evaporative System	C2	D2
Secondary Air System	C3	D3
A/C Refrigerant	C4	D4
Oxygen Sensor	C5	D5
Oxygen Sensor Heater	C6	D6
EGR System	C7	D7

6.3.5 – Data bytes C and D error codes

PID 02 represents the freeze frame trouble codes and returns 2 bytes of data.

PID 03 – Fuel System Status. This PID returns two bytes A and B. Data byte A corresponds to fuel system 1 and data byte B corresponds to fuel system 2. Only one bit per each of these bytes can be set to a 1. The bits are laid out as follows.

0 – Open loop operation	3 – Open loop due to system fault
1 – Closed Loop	4 – Closed loop with a fault
2 – Open loop due to driving conditions	5 to 7 – Padding, should be 0

PID 04 is the load value percentage and it returns just 1 data byte. There is a calculation involved in obtaining the final result.
PID 05 is the coolant temperature in degrees Celsius and it returns 1 data byte. PID 06 is the short term fuel percentage which returns 1 data byte. PIDs 07 – 09 are similar to PID 06 with regards to Data bytes returned. PID 0A represents Fuel pressure and in kilo Pascals and returns 1 data byte. PID 0B represents intake manifold pressure and returns 1 data byte.

The PIDs expressed above as well as the remaining are found in Figure 6.3.6 and are also explained in detail in section 11 entitles OBD Reading Functions. We made the table below to use as a quick reference when programming the application. The table is also useful for the succeeding sections.

The following table in figure 6.3.6 shows what the remaining PIDs return and the equations to calculate the actual values.

PID	Description	# Bytes	Calculation
02	Freeze frame trouble code.	2, A and B	N/A
04	Load value percent.	1, A	A*100/255 = engine load%
05	Coolant temperature in degrees C	1, A	Deg = A – 40
06	Short term fuel percent	1, A	.7812 * (Byte A – 128)
07 - 09	Similar to 6		
0A	Fuel pressure in kPa	1, A	Pressure = A * 3
0B	Intake manifold pressure kPa	1, A	Pressure = A
0C	Engine RPM	2, A and B	RPM = .25 * (A*256 + B)
0D	Vehicle speed, in km/h	1, A	Speed = A
0E	Timing advance in degrees	1, A	Advance = (.5 * A) – 64
0F	Intake air temperature in degrees Celsius	1, A	Degrees = A - 40
10	MAF air flow	2, A and B	Air flow = .01*(256*A+B)
11	Throttle position	1, A	Position % = .3922 * A
12	Sec. air status	1, A	Bit 0 – Upstream of catalytic Bit 1 – Downstream Bit 2 – Atmosphere Remaining – Reserved, 0
13	Oxygen sensor locations/bank	1, A	Bit 0 – Bank 1 – Sensor 1 Bit 1 – Bank 1 – Sensor 2 Bit 2 – Bank 1 – Sensor 3

			Bit 3 – Bank 1 – Sensor 4 Bit 4,7 – Repeat for Bank 2
14	Oxygen sensor voltage/bank 1 sensor 1	2, A and B	Oxy sensor voltage = .005*A Short term fuel% = .7812 * (B-128)
15 – 1B	Same as 14, but for remaining Banks		
1C	OBD design requirements	1, A	01 – OBD II 02 – OBD 03 – OBD and OBD2 04 – OBD I 05 – none 06 – EOBD
1D	Alternate Oxy sensor locations	1, A	Similar to 13
1E	Auxiliary input status	1, A	Bit 0 defines status,
1F	Padded		
20	Same as PID 00, for 21 - 40		

Figure 6.3.6 – PID Chart

7.0 Communicating with ELM-327

The ELM-327 was designed to communicate with a computer through an RS232 connection. However, for the purposes of this project we will be connecting to our MCU through the Tx and Rx pins.
The Android software will be communicating with the ELM327 through the microprocessor. We need to ensure that we are communicating with the ELM through the right com port. In addition, the settings of the ELM need to be adjusted to make sure data is being sent and received at the proper speed. Otherwise messages will be received in jumbled order and the ELM will not work properly. The connection also needs to be set to 8 data bits, no parity bits and 1 stop bit. Once the ELM-327 is properly connected, it will send the following message:

```
ELM327 1.4b.
>
```

When talking to the ELM, commands can be sent intended for the ELM's internal use or they can be messages to be passed on the ECU. If they are intended for the ELM the messages start with the letters ‘AT’ and whether it’s intended for the ELM or the OBD-II, the message must end with a carriage return or hex 0D. If an incomplete string is sent without a carriage return, an internal timer is started and after about 20 seconds, the ELM will print a question mark character ‘?’.

When processing commands, the ELM is constantly listening for new commands. If a new command is sent while the ELM is processing a previous command, the previous command is stopped and control is returned to the user.

7.1 AT Commands

This section will detail some of the more useful AT commands that may be used in this project. The user of the application probably will not be given the ability to most change parameters within the ELM because it would require specialized knowledge of the chip; however, we may be hardcoding the following AT commands if necessary.

- Allow Long (AL)
  Extends the number of data bytes that the ELM can accept from 7 to 8.
- Buffer Dump (BD)
  All messages sent and received by the ELM are stored in a buffer. This is used to check where messages failed or to resend previous messages. When this command is sent, the buffer is printed.
- There are various other commands to set the baud rate for each OBD-II protocol. We will be using these to sync the baud rate with our microcontroller so that data is received and sent at the correct speeds.

8.0 Interfacing MCU

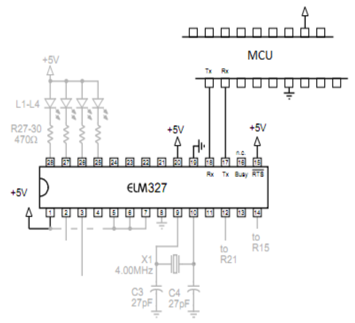
In order to communicate with the EML-327 device, we will need to send data over Bluetooth to the microcontroller. The microcontroller will then check the header of the data being sent in order to determine what to do with the data. If it determines that the data should go to the ELM, then it will be passed through. Otherwise, if the command is to start the ignition, lock or unlock the doors, roll down the windows or pop the trunk. The MCU will supply power to the necessary car wires.

8.1 Interfacing with ELM-327

When interfacing an ATmega328 with the ELM-327, we must note that the ELM-327 is, in itself, a microprocessor. The ELM chip utilizes a standard UART interface which is connected to the RS232 Tx and Rx pins. The microcontroller will be connected to the ELM using these Tx and Rx pins and they will be powered by the same 5V source. The ELM provides a hand shaking feature which helps to simplify the flow of data. This hand shaking feature consists of two pins, the input pin is ‘request to send’ (RTS) and the output pin is ‘busy’ which tells the system that the ELM is processing. The hand shaking feature will be implemented as follows:

- One of the port pins on the microcontroller will be connected to the RTS (pin 15) on the ELM
- Another port pin will be connected to the busy (pin 16) pin.
- When the MCU determines that a command needs to be sent to the ELM, the busy pin will be checked. If it is a high logic level, then the microcontroller needs to a) wait for the busy pin to go low, or b) set the RTS pin to low in order to request to send data.
- Once the busy pin goes low, the ELM waits indefinitely for a command from the microcontroller.

The **figure 8.1.1** shows an image of how the microcontroller will be interfaced with the ELM.



As seen in the figure (left), the pins 17 and 18 of the ELM will be connected to Tx and Rx pins on the microprocessor and the two will share the same input voltage.
Section 9 below will detail the Arduino board and the pins attributed to it.

Figure 8.1.1 - ELM and MCU interface Printed with permission

9.0 Programming the ATmega328

Since we are using an Arduino board, we will be using Arduino's C-based programming language and the Arduino IDE for windows. Some of the main functions of the microcontroller will be to apply voltages to specific output pins and to send and receive requests and responses to the ELM-327. This section will detail how that will be done using the Arduino programming language.

9.1 Arduino Programming

The Arduino board can be seen in **figure 9.1.1** the different components on the board are color coordinated and will be explained below.

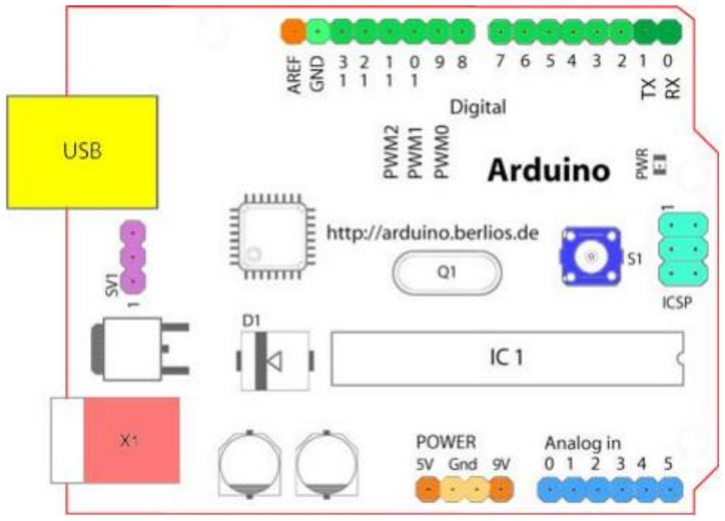


Figure 9.1.1 Arduino Board – Printed with permission
The components as listed on the Arduino data sheet are as follows
Starting clockwise from the top center:

- Analog Reference pin (orange)
- Digital Ground (light green)
- Digital Pins 2-13 (green)
- Digital Pins 0-1/Serial In/Out - TX/RX (dark green) - These pins cannot be used for digital I/O (digitalRead and digitalWrite) if you are also using serial communication (e.g. Serial.begin).
- Reset Button - S1 (dark blue)
- In-circuit Serial Programmer (blue-green)
- Analog In Pins 0-5 (light blue)
- Power and Ground Pins (power: orange, grounds: light orange)
- External Power Supply In (9-12VDC) - X1 (pink)
- Toggles External Power and USB Power (place jumper on two pins closest to desired supply) - SV1 (purple)
- USB (used for uploading sketches to the board and for serial communication between the board and the computer; can be used to power the board) (yellow)

9.2 Digital Pins

The digital pins on Arduino board have many functions that will be discussed. They can also be used for general purpose input and output. pinMode(), digitalRead() and digitalWrite() are commands that are used when dealing with these pins. The pins, when used as an input, can be given the value HIGH or LOW, the max current that can be used, per pin, is 40 mA. **Figure 9.2.1** describes each digital pin and their uses.

Pins	**Uses**
Serial : • 0 (Rx) • 1 (Tx)	Pin 0 is the Rx pin; it is used to receive serial data. Pin 1 is the Tx pin; is used to transmit serial data.
External Interrupts: • 2 • 3	The external interrupt pins have the ability to trigger an interrupt on the low value, rising/falling edge, or change in value using the attachInterrupt() function.
PWM: • 3 • 5 • 6 • 9 • 10 • 11	These pins supply an 8 bit pulse width modulation (PWM) output. The analogWrite() function is used.
BT Reset: • 7	Connected to the Bluetooth line of the Arduino BT board.
SPI: • 10 (SS) • 11 (MOSI) • 12 (MISO) • 13 (SCK)	These pins support SPI communication; however, they are not currently supported in the Arduino language.

Figure 9.2.1 – Digital Pins Table

9.3 Analog Pins

The analog pins have many built in functions that will be discussed later, the analog input pins can do a 10-bit analog to digital conversion (ADC) using the analogRead() function. The analog pins 0 through 5 can also be used as digital pins 14 through 19.

Pin 4 (SDA) and pin 5 (SCL) both support the I^{2}C (TWI) communication. This can be achieved by using the wire library.

9.4 Power Pins

The table in **figure 9.4.1** describes the 3 power pins and their uses.

Pins	Uses
VIN	This pin is the input voltage for the board when it is using an external power source. Meaning something besides the USB connection or other regulated power source.
5V	This is the regulated power supply to the board and its components. This power can come from the USB or the VIN using an on board regulator.
GND	These are the ground pins.

Figure 9.4.1 – Power Pins Table

The figure above discusses the input voltage pins and the ground pins, these will be used to supply power to the board. For our project, the VIN will probably be used to power the board from the hot wire in the car.

9.5 Arduino Functions

This section will detail some of the built in functions that will need to be used when programming our chip.

- Setup()
  The setup program is called when the program starts running. It is used to initialize variables, set pin modes and start using libraries. This function will run only one time, after startups and after resets.
- Loop()
  After the setup function runs, the program then moves on the loop function. As the name suggests the program loops through this function over and over allowing the program to change and respond. It is used to actively control the Arduino board.
- pinMode(pin, mode)
  This function is used to configure a pin to be either an input or an output. The parameters are pin, which is the number of the pin whose mode we are setting (int) and mode, which is either INPUT or OUTPUT.

So far there are three methods that have been discussed; they will appear in code as follows.

```
void setup()
```

```
        initialize variables here
        call pinMode( int, INPUT/OUTPUT) here
end setup

void loop()
        call other methods and
        check if statements here
end loop
```

Within the loop function other Arduino functions will be used such as:

Digital I/O

- digitalWrite(pin, value)
  This function can be called on any pin that has been configured as an output. The value can either be HIGH or LOW. This function will also be used to pull up resistors when a pin is set as INPUT.

  We will use this in our microcontroller when we want to start the engine, disable alarm, lock or unlock doors, pop trunk, roll down windows etc. All of these actions will require voltage being applied to certain wires in the car.

- digitalRead(pin)
  This function is meant to read the value from the specified pin. The return value will either be HIGH or LOW. If the specified pin isn't connected to anything, the return value can either be HIGH or LOW and can change randomly.

  We will use this method while interfacing with the ELM. We will need to read pins in the hand shaking mechanism provided to us by the ELM and discussed in section 8.1.

Analog I/O

- analogRead(pin)
  This function reads the value from the specified analog pin. The board contains a 6 channel 10-bit analog to digital converter this means that it maps input voltages between 0 and 5 to integer values between 0 and 1023.

- analogWrite(pin, value)
  This method will write an analog value to a pin (PWM wave). After a call to this function, the pin will generate a steady wave until the next call.

Advanced I/O

- shiftOut(dataPin, clockPin, bitOrder, value)

This function shifts out a byte of data one bit at a time. Each bit is written to the data pin, at which point the clockPin is toggled to indicate that the bit is available. This is known as a synchronous serial protocol and is a common way that micro controllers communicate with one another.

The dataPin is an int value that represents the pin to output each bit. The clockPin is an int value that is toggled once the dataPin has been set. The bitOrder can be either MSBFIRST or LSBFIRST and signifies the order in which to set the dataPin. The value is a byte and is the data to shift out.

We will likely use this method when communicating OBD-II requests between our microprocessor and the ELM-327. In addition to this function there are several other functions that are used for interfacing the Arduino with other devices.

- Serial.begin(speed)
- Serial.available()
- Serial.read()
- Serial.flush()
- Serial.print()
- Serial.println(data)

The functions above communicate happens via the Arduino board's serial or USB connection and on the digital Tx and Rx pins.

In addition to these functions listed and explained above, there are many other simple Arduino functions available that may be used when programming the board that will not be discussed here in this document.

9.6 Data Types and Other Syntax

Because the Arduin language is based off of C, most of the variable types that are available in any C language program are available when programming an Arduino board. The following is a list of the available data types:

- boolean
- char
- byte
- int
- unsigned int
- long
- unsigned long
- float
- double
- string
- array

Also, control statements such as if statements, for loops, while loops, and switch statements are all available in the Arduino programming language. Syntax for writing and calling functions is similar to C and does not need to be documented in this paper.

10.0 PCB Design

Since one of the goals of this project was to combine both the ELM327 and the MCU that controls the car both on the same chip, it is necessary to use PCB software. As mentioned in a prior section, we chose Cadsoft's EAGLE PCB Layout Editor. We have the schematics for both the Arduino board we are using and the ELM327 chip. The schematics for each component will be discussed in the following sections, followed by the final schematic.

10.1 Arduino Board Schematic

The **figure 10.1.1** shows the schematic of the board we will be using to program the MCU as well as what we are basing our final PCB off of. The left half of the schematic shows where the Bluetooth chip will ultimately go. The right half shows the MCU.

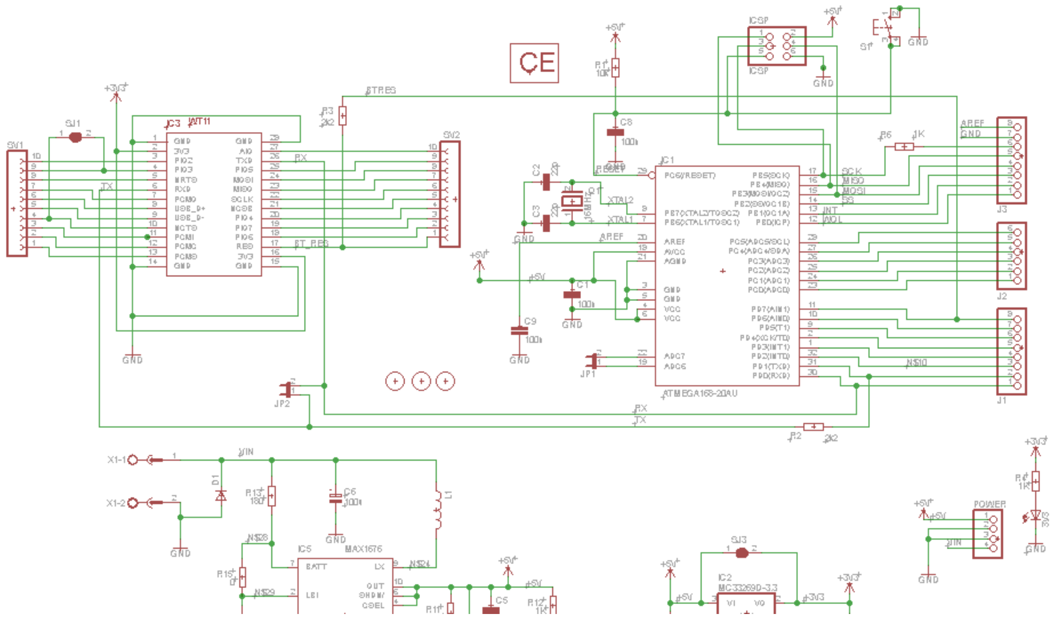


Figure 10.1.1 – Arduino Board Schematic- Printed with permission

The board shows all the wiring needed to put it on a PCB. As discussed in the "Interfacing the MCU" section in section 8, we will make a connection between the Tx and Rx pins on the board and the Tx and Rx pins on the ELM chip. These

Tx and Rx pins are pins 0 and 1 on the Arduino board and are the first two pins starting from the bottom.

The image of the board (.brd) file can be seen in **figure 10.1.2**. The Tx and Rx pins on the actual board are visible as the first two pins from the top right. On the final board file, we will see a connection made between these pins and the corresponding pins on the ELM chip.

The board file below can be compared to **figure 9.1.1**. The only difference is the Bluetooth module instead of the USB port; however, this board file shows where all the wirings and interconnections lead.

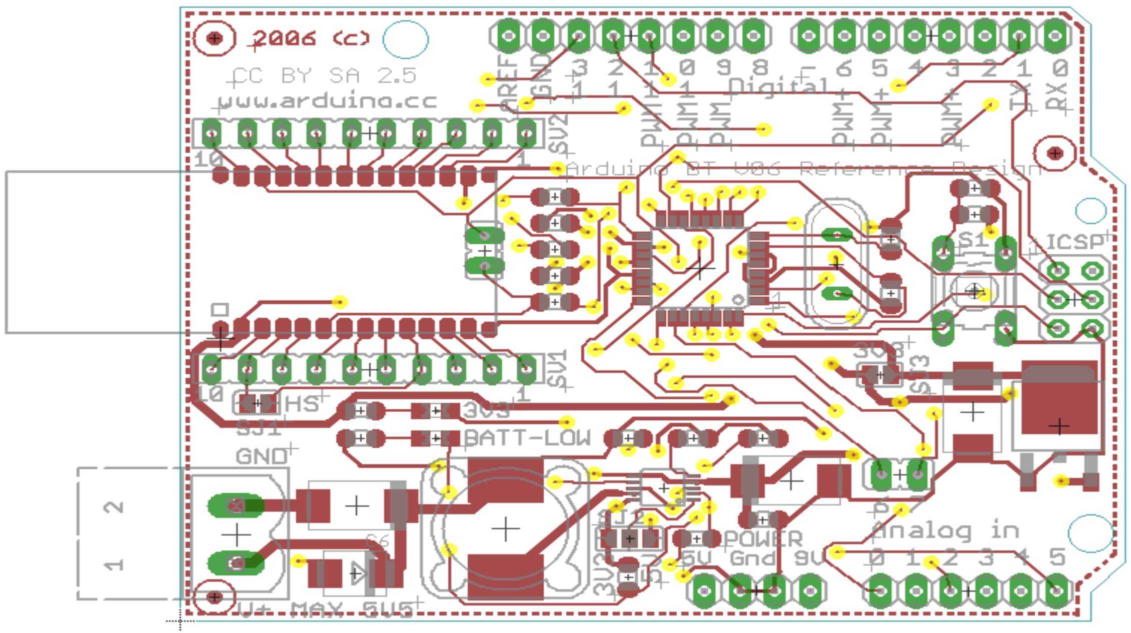


Figure 10.1.2 – Board File for Arduino Board – Printed with permission

10.2 ELM327 Schematic

The **figure 10.2.1** shows the ELM327 schematic and the different connections between the different pins on the ELM chip. The Tx and Rx pins, which is what we will be concerned with are labeled pins 1 and 4 on the top left schematic. We will make a connection between these pins and the Tx and Rx pins from on our MCU as mentioned above.

The actual ELM327 will be purchased preprogrammed; the schematic below will be used when designing the PCB in order to make proper connections with the MCU and Bluetooth.

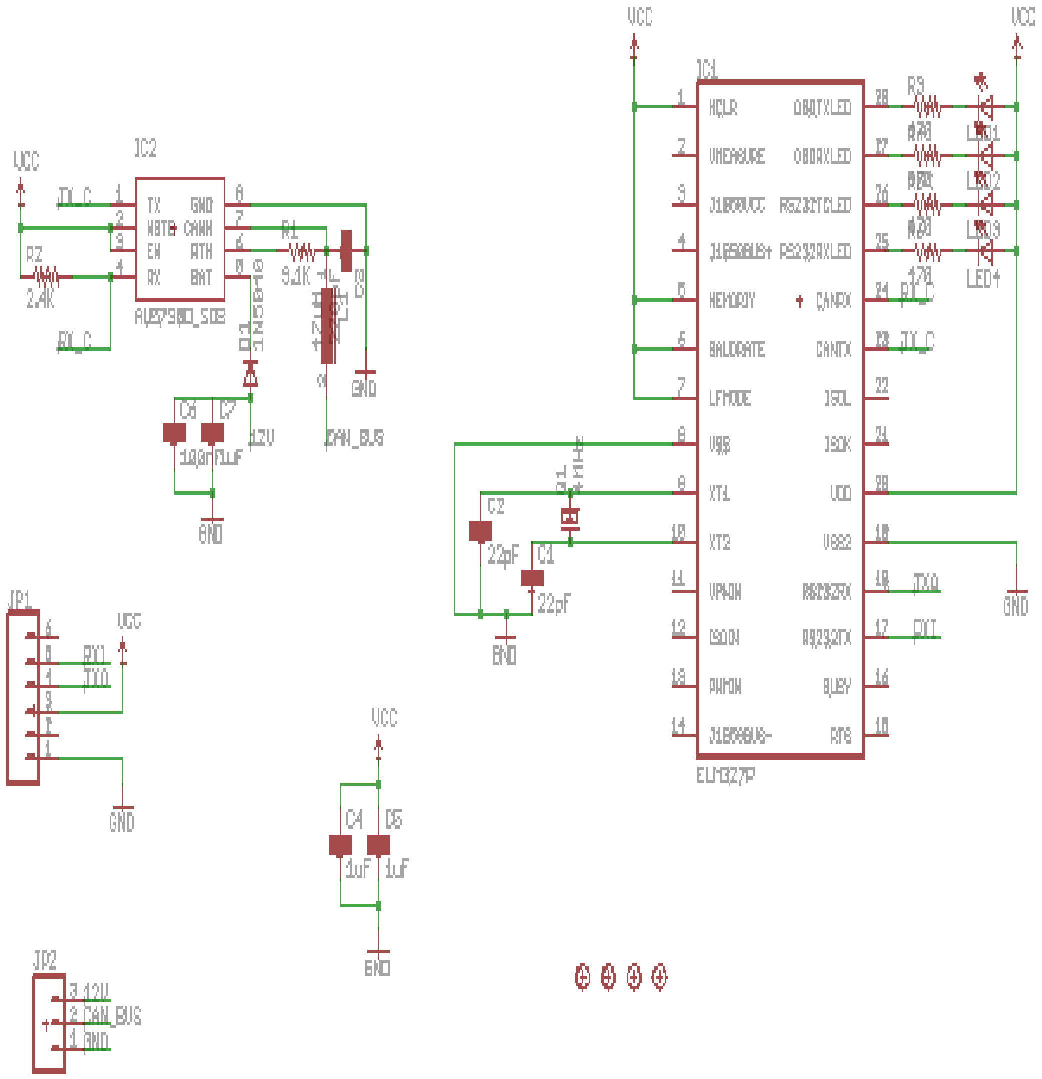


Figure 10.2.1 – ELM327 Schematic

The image in **figure 10.2.2** is the board file for the ELM327, this image shows the connections between the different components of the ELM chip. The actual ELM has 28 output pins and the board file shows where each of these pins are tied to.

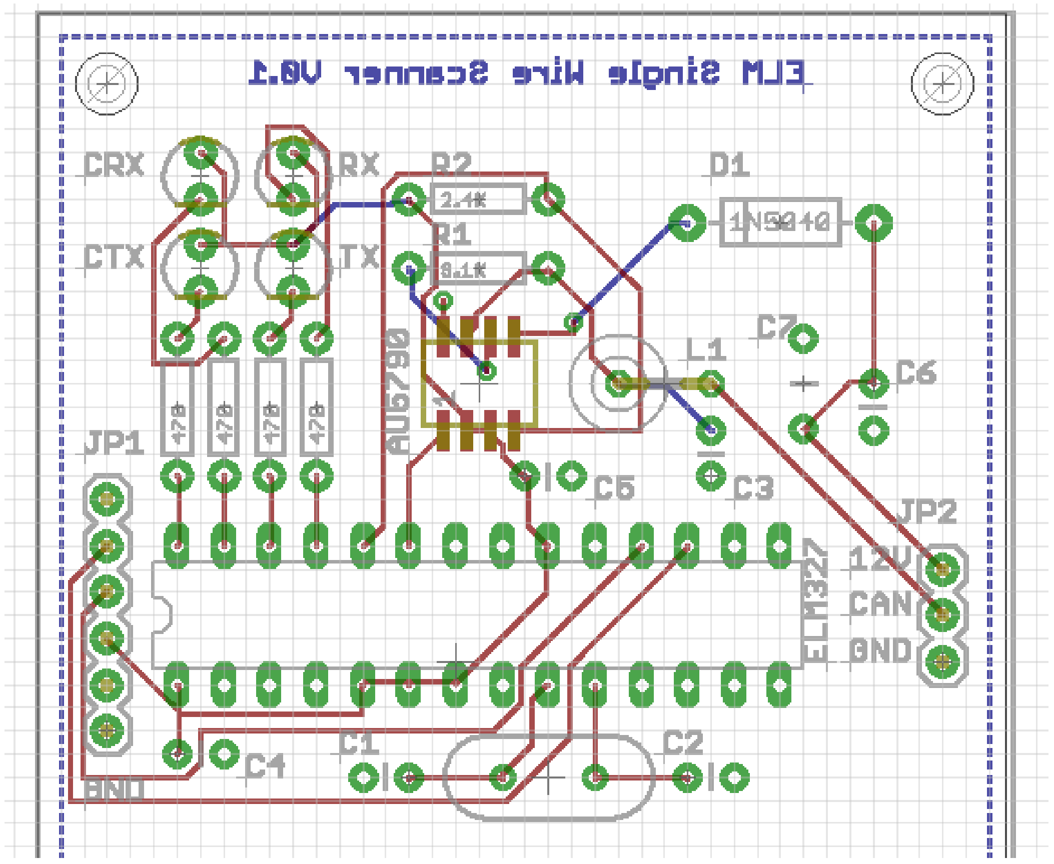


Figure 10.2.2 – ELM board File

10.3 Final Design

This section will detail the final design by describing the parts that were used and why they were selected. The schematic for the final design is pictured in **figure 10.3.1** below.

In the figure, we show the MCU which is on the right side of the schematic. We too the pins and connected them to outputs from the EAGLE library called “pinheads.” We made sure to connect the Tx and Rx pins on from the MCU to the Bluetooth chip. We also had to connect the Tx and Rx to the ELM327 so that it can transmit and receive data to and from the MCU.

The ELM327 chip from the schematic below was taken directly from the ELM data sheet schematic. The Bluetooth chip is connected the same way as the Arduino BT board which is what we will be using for testing.

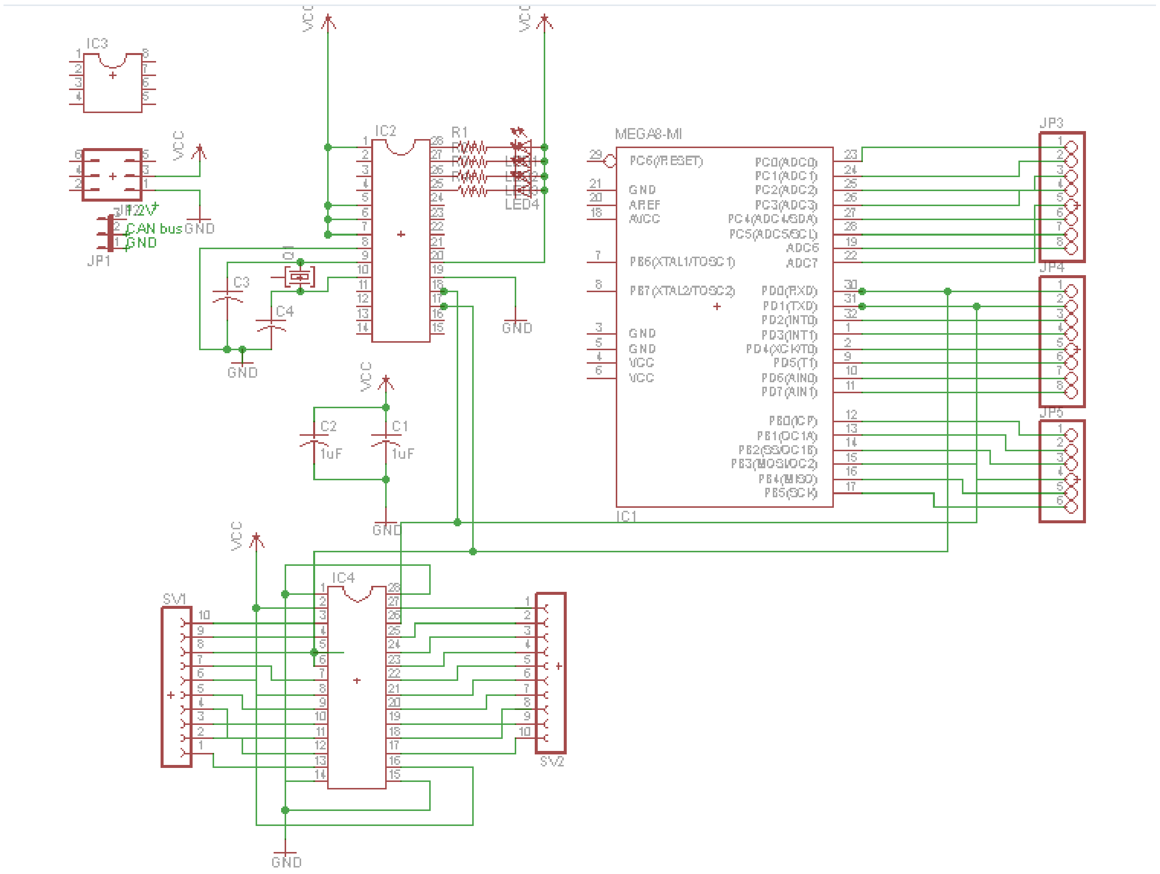


Figure 10.3.1 – Final PCB Design

11.0 OBDII Reader Functions

11.1 Reading

In order to read any data from the OBD a request must first be made to the OBD so that it may respond. However, because our system has other functions integrated into it such as starting the car, unlocking doors and rolling down windows, the request will actually first be sent from the phone to the microcontroller and from the microcontroller to the ELM327 which will translate that information into the proper string following the protocol that the OBDII port can read. Finally, the ELM327 will send the translated string the OBD and then the response from the OBD will then traverse back up its sent path. The request will be a string which will usually be 2 bytes. The first byte will indicate the mode 01 – 09 and the second will be the PID depending on what mode the request sent is as some requests do not require PIDs. From there the OBD will then send a response. The first byte of the response will indicate the mode. However, 40 must be subtracted to obtain the actual value of the mode. The next byte may indicate the PID depending on the mode or it may begin the data. The response data will contain up to 7 data bytes that are important to deciphering what data the OBD Most of the data received must be converted in order to be shown in the proper syntax. The request string must fit the form as described in **figure 11.1.1a**:

	Mode	PID
Request	XX	YY

Figure 11.1.1a – Request to ELM 327

In the figure above XX and YY each describe a 2 digit hexadecimal number. The second hexadecimal number labeled as the PID is not needed in certain mode requests such as mode 03. The data received is also of a specific format shown in **figure 11.1.1b:**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	XX	YY	ZZ	AA	BB	CC	DD

Figure 11.1.1b – Request to ELM 327

For most responses only byte A will contain a value. The other bytes will be padded with 0's however for some responses 2 bytes are taken up and for error codes all bytes may be used including extra lines depending on how many error codes are found in the vehicle. The string will be read as "XXYYZZAABBCCDD" in which case the data must be split by every 2 characters to be deciphered. The data also contains header bytes that are to be stripped out because they are not necessary to send and receive data from phone to ELM327 to OBD and back.

The OBDII port on a vehicle can show three kinds of data: Diagnostic Trouble Codes (DTCs), real-time data, and freeze frame data. Freeze frame data relates to a sort of "snapshot" of all the real-time data fields during a DTC error condition. Usually mechanics will use this to help diagnose exactly what is the error that caused the check engine light to display. For the following section however we will be discussing real-time data and Diagnostic Trouble Codes. Real-time data will relate to the data that comes from the various sensors on a vehicle. This includes just about all the functions displayed in all the sections labeled 11.1.x.where "x" is any of the corresponding subsection numbers. Diagnostic Trouble codes are error codes that occur when something in a vehicle malfunctions and can be used to help repair a vehicle or prevent a vehicle from needing a costly repair. Therefore, the ability to diagnose and repair these problems is greatly needed. This section will go into great detail in discussing the different values that the OBD reader will grab from the OBDII port and how this will be implemented.

11.1.1 Preface

All of the OBD functions will be coded in one package called function. Specifically it will be called *org.obddroid.function* because of how android packages are declared. The function package will consist of all the function objects. The functions objects purpose are to hold the protocol of what string needs to be sent to the OBD to determine the value of the data the user wishes to obtain and to convert the data received by the OBD to the proper value that the user needs. The following sections detail how the functions must be translated and how they need to be implemented. The sections will also contain information on what methods will need to be overloaded and what methods will not. Also they will discuss how the constructors for each of the function objects should look and they relation between the functions and the super classes to them.

Before we begin discussing about the specific functions, the hierarchy of the classes must first be described. The main class at the top of the food chain will the *ObdFuntion( )* class. Some function objects will directly inherit from this class although others will inherit from yet another class that inherits from this one. Therefore, the next level in the food chain will consist of *ObdTempOBDFunction( ) ObdPressureFunction( )*, and *ObdNumFunction( )*, which will all extend *OBDFunction( )*.The functions that give integer results will be extending *ObdNumFunction( )* while the functions that provide data about temperature will extend *ObdTempFunction( )*.Finally, the functions that supply information about pressure in the system will extend *PressureOBDFunction( )*. Still some classes will extend directly to *ObdFuntion( )*. The three super classes will contain generic methods that may be overridden depending on the function, if not it is assumed that the function will utilize the generic method. The class diagram in **figure 11.1.1a** visually displays this hierarchy and shows the variables and methods of each class.

In **figure 11.1.1a** the number above the classes represents the section in which that specific class will be discussed. All the inheritance relations will be depicted in the class diagram. Within each section the definition of what the function is displaying and what that value that is coming out of the OBD means in relation to the automotive world. Then the next topic to be discussed will be the string that needs to be sent to request for data from the OBD followed by how the string that the OBD will respond with should be displayed. Next will be the discussion on how the data value will be translated to a numerical digit. Also any formulas on how to translate this value to the proper value and the proper units based on the users decision. Finally a discussion on how the function relates to the figure shown along with how certain methods may be overloaded and how all the other methods will be used.

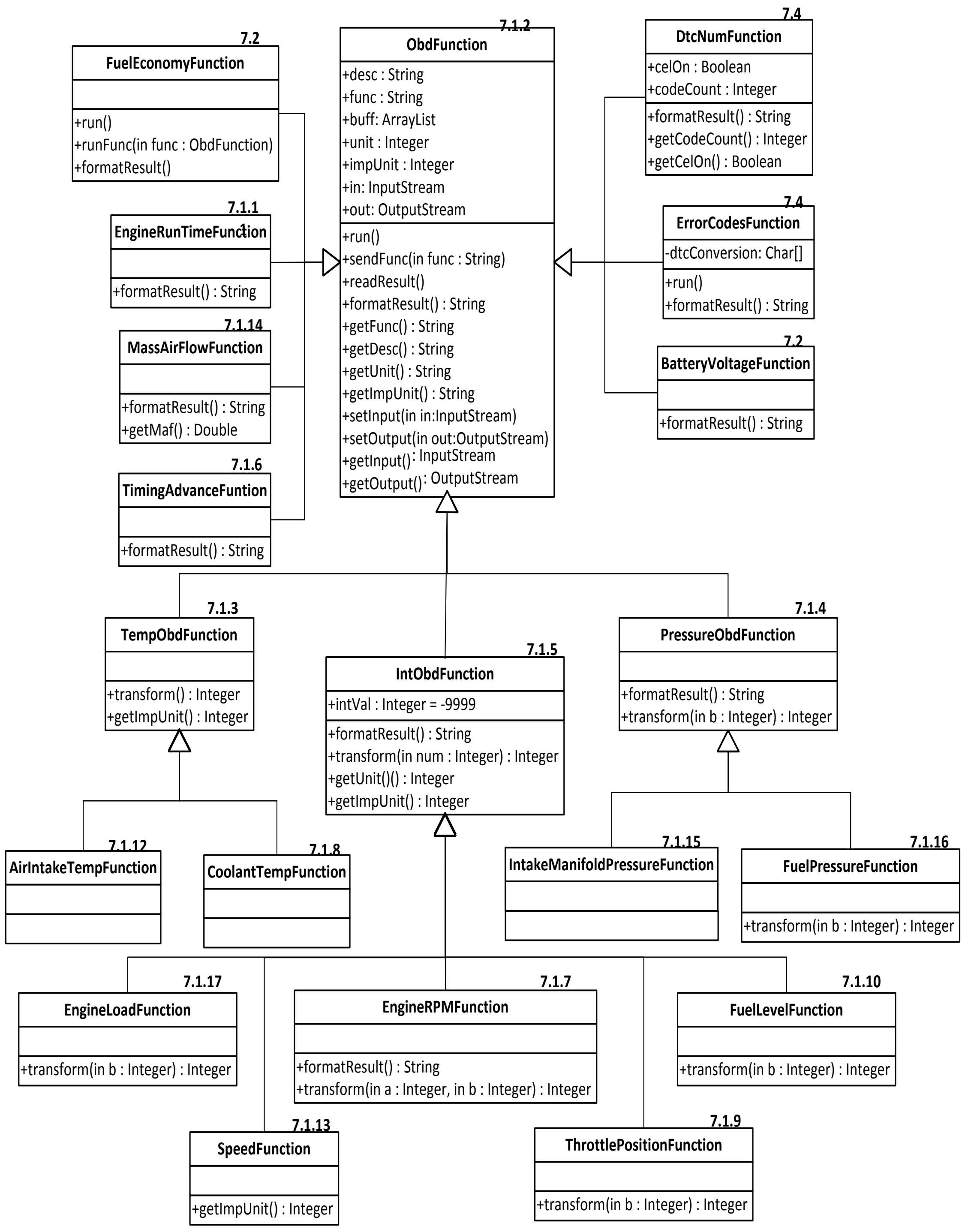


Figure 11.1.1a – Class Diagram of org.obd.function package

11.1.2 OBD Function

This class will extend Thread and overwrite the *run( )* method. The run method here will simply send the function to the OBD and read the result and store the values accordingly. Some classes may override the *OBDFunction( )*'s run method, although most will simply use this. The reason for it being a thread with a run method is that most of the data that these functions require are live data and must continually be updated. The constructor for the *OBDFunction( )* class will have a header similar to this *OBDFunction( )*(String func, String desc, String unit, String impUnit), where func will represent the code for the function that must be sent to the OBD, desc is the description or name of the function, unit will be the units that we use in the United States for the function and impUnit would be the Imperial units that they would use in places like the UK. An array list called buff must also be included that will act as a buffer when the bytes are read from the OBD. It will translate the values read in from a string to byte form.

A few of the other methods in *OBDFunction( )* are *sendFunction( )*, *readResult( )* and *formatResult( )*. *FormatResult( )* is a method that some functions will override. If a function does not override this method then the function will just remove the space that is in the beginning of the result string. *SendFunction( )* simply sends the functions code to the OBD and *readResult( )* will pull in the data the OBD sends in response to *sendFunction( )*. Some of the other methods involved in this class will be to simply return the value of the variables. When data is read and written from the OBD we must implement an input stream and output stream. In java an input stream reads the bytes written in from a source, while an output stream is a way of writing data out to a source in byte sequence. Therefore we will need a variable of InputStream type named in and one of OutputStream type named out. We will also need methods to set and get the input stream and output stream hence the methods: *getInput( ), getOutput( ), setInput( ),* and *setOutput( )*. These methods will stay apart of this super class and are not to be overloaded by any child classes.

11.1.3 Temperature OBD Function

This class will be used for those classes that have to deal with temperature. Currently, there are two classes that inherit from *TempOBDFunction( )*. Those classes are *AirIntakeTempOBDFunction( )* and *CoolantTempOBDFunction( )( )*. Some vehicles may have the ability to send information on more temperatures, such as oil and transmission temperature. Since this is the case, programming the system in this way allows for easy addition of those simple methods. *TempOBDFunction( )* uses the methods from *OBDFunction( )* except for *getImpUnit( )*, which this class overwrites. Within this classes *getImpUnit( )* method we have the formula for converting from degrees Celsius (°C) to the unit we use in the United States, Fahrenheit (°F), which is temperature in Celsius times 9, then divided by 5, finally you add 32 to that value, or in other words $F =$

$(C*9)/5 + 32$. *TempOBDFunction( )* also contains the method *transform( )*, whose job is to convert the byte striped from the OBD's response to a temperature value by offsetting the number by 40. In other words $temperature = byte - 40$. Classes that inherit from this class should not need to overload any methods and should only need to show what its command string that must be sent to the ELM327 is.

11.1.4 Number OBD Functions

IntOBDFunction( ) as described in **figure 11.1.1a** is the class in which many of the other classes inherit from. This class contains methods that the other classes below may inherit from. This class also contains the *formatResult( )* method, which it overload from its parent class *OBDFunction( )*. The *formatResult( )* method in this class which will grab the second byte from the result string and convert it from hex to decimal then converts that value to a string and returns it with the units appended to the end of it depending on whether the user has chosen an imperial or metric unit. The method *transform( )* is a method that other classes may override to use formulas to convert the string received from the OBD to a proper value. The next methods are *getUnit( )* and *getImpUnit( )* are to be overwritten if the class needs to perform any conversions to change from metric to imperial units. If not these methods simply return the same value based on what *transform( )* returns.

11.1.5 Pressure OBD Functions

Figure 11.1.1a shows that Pressure*OBDFunction( )* currently only has two subclasses. Those two classes are IntakeManifoldPressureFunction( ) and FuelPressureFunction( ). Fuel pressure has an equation to translate its data therefore it will overwrite the *transform( )* method. Otherwise the classes simply use the methods supplied in this class. The *transform( )* method in this class doesn't do any calculation and will just return the value that is stripped from the second byte of the OBD response string. The *formatResult(*) method is the one that needs discussion. This method will either return the value that was grabbed from the OBD response converted from hexadecimal to decimal or if the user has chosen imperial units will divide this value by 101.3 to convert it from kilopascals to atmosphere. There may be other values that can be obtained from the OBD about pressure in the system as time goes on therefore, the hierarchy described here allows for easy introduction of a new function that obtains data from the OBD about the pressure on the system.

11.1.6 Timing Advance

This refers to the Ignition timing advance on cylinder #1 of a vehicle which is when a spark will occur in relation to where the piston is positioned. This spark is usually delayed to give the air-fuel mixture time to burn. Timing Advance is typically measured in degrees before top dead center or °BTDC. Proper timing is necessary to how long the engine lasts along with fuel economy and performance. Getting this value just right is essential to proper vehicle operation. Many hours are spent trying to figure out the proper timing of a specific vehicle, however this should be left to experts as improper timing can be fatal to an engine.

Timing Advance is PID 0E in mode 01.Therefore the string sent to the microcontroller needs to be “010E”. **Figure 11.1.6a** depicts how this string will look:

	Mode	PID
Request	01	0E

Figure 11.1.6a – Timing Advance Request

And the OBD will send a response that the ELM327 will send to the microcontroller and then to the phone will look like **figure 11.1.6b** with the XX replaced by a hexadecimal value:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0E	XX	00	00	00	00

Figure 11.1.6b – Response to Timing Advance Request

Where 41 indicates that this is a response to a mode 01 request, the request being of PID 0E and XX representing the hexadecimal value of the timing advance that must be divided by 2. The value is also offset by 64 therefore this value must be subtracted to get the correct timing. So, the formula for timing advance becomes:

$$Advance\ (Degrees) = (.5 * XX) - 64$$

The *Timingadvance( )* function drawn in **figure 11.1.1a** depicts that this function inherits directly from the *OBDFunction( )* class. The reason for this is that it has no other units besides degree, so it does not need to be converted to different units for imperial and metric measurements. *Timingadvance( )* does however need to overload two methods from *OBDFunction( )*. Those methods would be *formatResult( )* and *transform( )*. The *transform( )* method will take in an integer as its parameter, which will be the data value achieved from the response to the OBD function, and *transform( )* that value using the formula above. The method will then return that calculated value. Finally, *formatResult( )* will handle the task of taking that value and converting it into a readable string that will be displayed in the GUI. The value will be formatted as a number with one decimal value following and then converted to a string to be displayed.

11.1.7 Engine RPM

RPM refers to the revolutions per minute that the crankshaft on a vehicle is rotating. It is normally measured in thousands of revolutions and displayed on a gauge on a vehicle. However, some vehicles do not include this although it is very necessary to ensure you do not over-heat the engine and to aid in economical and performance driving. In essence the higher the RPM the more heat will be produced along with more speed, less RPM will produce less fuel consumption and less acceleration.

RPM is PID 0C in mode 01 which represents the string “0C01”.Therefore, the request string to the OBD should be of the form described in **figure 11.1.7a**

	Mode	PID
Request	01	0C

Figure 11.1.7a – Engine RPM request

The request above will only get a response from the OBD if the engine is running because if they engine is not running the Engine RPM cannot be displayed as the engine is not turned on. If the engine is running the OBD will respond with a string that can be decoded using **figure 11.1.4b:**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0C	XX	YY	00	00	00

Figure 11.1.7b – Response to Engine RPM

Where 41 indicates mode 01 (41 – 40 = 01), 0C representing PID 0C and XX YY begin a two byte hex number which must be converted. In order to convert this number you must first transfer the number from hex to an actual decimal. After that you must divide this number by 4 because RPM sent from the OBD is sent in increments of ¼ RPM, therefore the formula will be as follows:

$$RPM = .25 * (XX * 256 + YY)$$

The *EngineRPMFunction( )* shown in **figure 11.1.1a** is a class that inherits from *IntOBDFunction( )* as it is a numerical function that must be transformed. This function however only has one unit value and need not overload the *getImpUnit( )* method shown in *OBDFunction( )*. This class will however need to overwrite the *transform( )* and *formatResult( )* functions. In its *transform( )* function the class will take in two parameters, the first byte of data and the second byte of data from the OBD, and use the formula found above to translate those two bytes into the proper numerical value for the engines RPM. Objects of this class will also overwrite the method *formatResult( )* by using this method to return a string of the engine RPM value retrieved by *transform( )* as a 4 digit value with no decimals following. This will be used to display as a number (in revs per minute) and on a gauge similar to the tachometer on many cars today inside our GUI if the user selects to view this value.

11.1.8 Coolant Temperature

Engine Coolant Temperature or (ETC) will display the temperature of the coolant that runs through the engine. If this gets to hot the engine will overheat and become inoperable therefore this is an important component to be monitored. Our system will have this displayed as a number that will show blue when the coolant is cold, white when it is at normal operating temperature and it will display red when the coolant becomes too hot and may cause the engine to malfunction. The white level will be anywhere in between 180℉ and 210℉ anything lower will be blue and any higher will be red. We may also introduce a warning signal that will display if the coolant hits anything above 250℉. Also, running an engine at improper temperatures will hurt performance and fuel consumption.

In order to request data from the OBD about the coolant temperature, which is PID 05, one will need to send the request in the form of a string representative of the two columns displayed in **figure 11.1.8a** below:

	Mode	PID
Request	01	05

Figure 11.1.8a – Coolant Temperature Request

This indicates the string “0105” will be the string sent to the OBD. Following the request, the response from the OBD will need to be as shown in **figure 11.1.8b**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	05	XX	00	00	00	00

Figure 11.1.8b – Response to Coolant Temperature

Where 41 indicates that it is mode 1(41 – 40 = 1), 05 shows that it is PID 05 and XX represents a number in hex that will represent the actual temperature of the coolant in degrees Celsius. This number however is offset by 40 from the actual value to allow for temperatures below zero. For example, if XX was say 7B then that would equal 123 decimal but the actual coolant temperature would be 123 minus 40, which equates to 83 degrees Celsius. Thus the formula should be:

$$Coolant\ Temperature\ (°C) = XX - 40$$

CoolantTempOBDFunction( ) will extend *TempOBDFunction( )*. It will use the convert( ) and getImpNum( ) methods from *TempOBDFunction( )* to convert it to the proper temperature value and proper unit. The *CoolantTempOBDFunction( )* class will contain its constructor that calls the constructor of the class in which it inherits. The constructor will need to show the function string as “0105” and the description string as “Coolant Temperature” therefore its constructor will be super(“0105”, “Coolant Temperature”, “C”, “F”);.

11.1.9 Throttle Position

When the accelerator in a vehicle is pressed the throttle must move open and allow air to pass through. Throttle position refers to the exact location of the butterfly valve in the throttle which either lets more or less air into the engine. This is typically measured by a potentiometer attached to the butterfly spindle on the throttle body. More air will result in more combustion and greater acceleration. WOT refers to wide open throttle meaning the throttle position is maxed out and the vehicle will accelerate hard. They value outputted by the OBD will actually be represented as a percent (%) of WOT where 100% would mean you are “putting the petal to the metal” or in other words have the acceleration pedal depressed completely.

The request needed to be sent to obtain the Throttle position from the OBD must be of mode 01 and PID 11, therefore the request to be sent to the device should look like the following **figure 11.1.9a**:

	Mode	PID
Request	01	11

Figure 11.1.9a – Throttle Position Request

Then the OBD will respond to the request for the Throttle Position with a bit string formatted to fit the form of **figure 11.1.9b** below:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	11	XX	00	00	00	00

Figure 11.1.9b – Response to Throttle Position

Where 41 dictates mode 01, 11 is the PID and XX represents the hexadecimal value for the Throttle position that must be formatted by multiplying .3922. This value will come in as a percent (%). This implies the formula to find Throttle Position:

$$Throttle\ position\ (\%) = XX * .3922$$

ThrottlePositionFunction( ) according to **figure 11.1.1a** will inherit from *IntOBDFunction( )*, which in turn inherits from the main class *OBDFunction( )*. *ThrottlePositionFunction( )* will overload the *transform( )* method found in *IntOBDFunction( )*. It will take in an integer as a parameter and use the formula above to return the percent value of the throttle position. Since percent is the only unit used, the methods *getUnit( )* and *getImpUnit( )* are going to remain the same and will return only the transformed value. Also the *formatResult( )* method doesn't need to be overwritten either because it will use the format as defined in *IntOBDFunction( )* which is to grab the one byte value out of the return header and store that in an integer variable b and then call *transform( )* on b storing it into a variable then returning a formatted string that contains the value with no decimal places followed by the unit of the value in either imperial or metric depending on the users selection. In this case though, either selection will provide the unit of percent or %.

11.1.10 Fuel Level

The Fuel Level will designate how much fuel is left in the vehicle. This will be displayed as a graphic gauge that will show green when over 75%, yellow when between 75 and 50, orange between 50 and 25 and red when below 25. We can use the fuel level value to help determine other parameters. One such parameter will be the cruising range. Depending on the average miles per gallon you can multiply that by the amount of gallons left in the car, which can be found by multiplying fuel level by the vehicles fuel capacity in gallons, to get an estimated amount of miles the car to drive before it needs a fill up.

To get this value from the OBD the signal that needs to be sent must mode 01 and PID 2F.Therefore it will be in the form depicted in the following **figure 11.1.10a:**

	Mode	PID
Request	01	2F

Figure 11.1.10a – Fuel Level Request

The OBD will respond to the request for the Fuel Level with a bit string that must be fit to the **figure 11.1.10b** below:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	2F	XX	00	00	00	00

Figure 11.1.10b – Response to Throttle Position

Where 41 indicates that this is a response to a mode 01 on PID 2F and XX will represent the Fuel level in a percentage of nominal fuel tank. However this value must be formatted by using the following formula:

$$Fuel\ Level\ (\%) = 100 * (\frac{XX}{255})$$

Using this formula the *FuelLevelFunction( )* class will overwrite the *transform( )* method found in *IntOBDFunction( )*. This indicates that *FuelLevelFunction( )* will directly inherit from *IntOBDFunction( )* and indirectly inherit from *OBDFunction( )* through *IntOBDFunction( )* as described in **figure 11.1.1a** above. *FuelLevelFunction( )* is similar to *ThrottlePositionFunction( )* because it has only one unit in either imperial or metric units, therefore it will not overload the *getUnit( )* or *getImpUnit( )* methods. So, it will simply return the transformed value after *formatResult( )* is called. And since *formatResult( )* need not be overloaded either, the fuel level value will be a string that contains the value of the fuel level, without a decimal, followed by its unit which is percent (%) regardless if the user has chosen to view imperial units or metric units.

11.1.11 Time since Engine Start

This indicates how many seconds it has been since the engine has been started. This value will be a two bit number displayed as seconds since start. The display will be a string formatted to display in the form “hh mm ss”. Where hh is hours, mm is minutes and ss is seconds. The conversion of this will be discussed below when the *formatResult( )* method is explained. The request for this data to be sent to the microcontroller to the ELM327 from the android device needs to be of Mode 01 and on PID 1F so the request will be described in **figure 11.1.11a**:

	Mode	PID
Request	01	1F

Figure 11.1.11a – Engine Runtime Request

And the response to the request for the engine runtime from the OBD will be returned as described in the following **figure 11.1.11b**:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	1F	XX	YY	00	00	00

Figure 11.1.11b – Response to Engine Runtime

Where 41 indicates mode 01 and 1F is the PID and XX is the first byte of the time and YY is the second byte. Since engine runtime is a value based on time the minimum value will be 0 and the maximum value for this will be 65,535. Since the value comes in as 2 bytes the value will need to be formatted so it will display them properly. The formula should be as follows:

$$Run\ Time\ (seconds) = (XX * 256) + YY$$

Looking at **figure 11.1.1a** one can see that *EngineRunTimeFunction( )* inherits directly from *OBDFunction( )*. This is because the value calculated is a time value that must be formatted differently than any other function. Since the formula above only returns the runtime as seconds one must use formatting techniques to show this in hours minutes and seconds. This will be achieved by using a mix of integer division and modulo division. To get hours you will take the value in seconds (sec) and integer divide it by 3600, in other words hours = sec/3600. Then, to get minutes you must take the remainder of that and divide that value by 60, so minutes = (sec%3600)/60. Finally, to find how many seconds you only need to see how many seconds are left over when dividing the entire value received by 60, or seconds = sec%60. Then we put these values together separated by a space. This will all be coded into the *formatResult( )* method. All the other methods will remain the same, and since this does not inherit from *IntOBDFunction( )* or Temp*OBDFunction( )*, there is no *getUnit( )*or *getImpUnit( )* method to worry about.

11.1.12 Air Intake Temperature

This value comes from the air intake sensor and displays the exact temperature of the air in the air intake that goes to the engine. The hotter the air going into the engine the less dense the air will be and cause the vehicle to burn less gas, however this can result in poor performance and potential harm in the engine overall. Colder air is denser and causes the car to burn fuel more steadily (richer). This will hike up performance and save the engine which is the reason cold air intakes for vehicles are so popular. The value of this data is very dependent on the temperature of the air outside the engine so finding a nominal value for this temperature will be difficult as the temperature of the environment fluctuates.

To request a response for the Intake Air Temperature from the OBD one will need to send data that displays mode 01 and PID 0F, therefore the string sent will be “010F”.So the string will need to be in the form displayed in **figure 11.1.12a** below:

	Mode	PID
Request	01	0F

Figure 11.1.12a – Air Intake Temperature Request

And the OBD will respond with to the request for the air intake temperature with its information string shown in **figure 11.1.12b**.

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0F	XX	00	00	00	00

Figure 11.1.12b – Response to Engine Runtime

Where 41 is mode 01 and 0F dictates PID 0f. Finally, XX is the temperature of the air in the Air Intake in degrees Celsius. However this temperature is offset by 40 and must be formatted to display properly by using this simple formula:

$$Air\ Intake\ Temperature\ (°C) = XX - 40$$

This function will extend *TempOBDFunction( )* because it is based on temperature. It is formatted the same as other temperature functions and can be read as either Celsius or Fahrenheit depending on the user's choice. Therefore, the class for this object function need only contain its constructor that has "010F" as its function String and "Air Intake Temperature" as the description string. In other words the constructor will only contain this line: "*super*("010F", "Air Intake Temperature", "C", "F");.

11.1.13 Speed(MPH, KPH)

Speed is the most obvious function of the OBD reader. It will show the speed the car is currently going in both gauge form and a digital number similar to what is currently on the speedometer. It will be able to be displayed in miles per hour (MPH) and Kilometers per hour (KPH). This data will come from the OBD and not from the accelerometer or the GPS on the phone such as other apps for the android.

To find the Vehicle Speed the request sent to the OBD needs to be a mode 01 request in PID 0D. Therefore the request string must look like the following **figure 11.1.13a**:

	Mode	PID
Request	01	0D

Figure 11.1.13a – Speed Request

Then the OBD will respond to the request for speed by responding to the data with a request string that resembles the following **figure 11.1.13b**:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0D	XX	00	00	00	00

Figure 11.1.13b – Response to Speed Request

Where XX represents the hexadecimal speed of the vehicle in Kilometers per hour (KPH). However if we want the value of this data as Mile per hour we must multiply this number by .625 therefore the formulas for speed are:

$$Speed\ (KPH) = XX$$
$$Speed\ (MPH) = XX * .625$$

SpeedFunction( ) extends the *IntOBDFunction( )* method since it is a numerical value that can be expressed in either an imperial or a metric unit. The imperial unit for speed would be miles per hour while the metric unit can be read as kilometers per hour. The method *getImpUnit( )* will respond with the imperial value if the user has chosen that as the units they wish to display in the options menu. *SpeedFunction( )* does not overload any other methods therefore the *transform( )* method will simply return the value that it is suppose to be and it will be formatted in a generic way using the *formatResult( )* method.

11.1.14 Mass Air Flow (MAF)

Mass Air Flow indicates the rate at which the air is flowing into the engine. This value is import to calculate the engine load and determine how much fuel needs to be injected, when to ignite the cylinder and when to shift gears. This value is also useful to help calculate miles per gallon (MPG). The MAF will be displayed as a double value with the units in grams per second (g/s).

To get this value from the OBD one must indicate a mode 01 request in PID 10 therefore the resultant string will be in the form similar to **figure 11.1.14a** below:

	Mode	PID
Request	01	10

Figure 11.1.14a – Mass Air Flow Request

Then the OBD will respond to the request for the Mass Air Flow rate with a String in that is formatted like in **figure 11.1.14b** below:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	10	XX	YY	00	00	00

Figure 11.1.14b – Response to Mass Air Flow Request

41 indicating that this is mode 01 and 10 showing what PID the response is for. XX and YY will represent the 2 byte data value as a response that must be converted and then divided by 100.Therefore the formula must be:

$$Mass\ Air\ Flow\ \left(\frac{g}{sec}\right) = \frac{(256 * YY) + XX}{100}$$

Figure 11.1.1a depicts the *MassAirFlowFunction( )* inheriting directly from *OBDFunction( )*. This class will utilize the methods of *OBDFunction( )*, however the class does not use the *formatResult( )* method. Instead, it overloads this method and returns a string formatted to its own specification. Because, the result achieved is 2 bytes we must first grab these two bytes from the returned string that came from the OBD. After retrieving these bytes and storing them in the proper integers we must then use the formula above to change it to the proper value. This value will be calculated as a double then returned as the string representation of that double. The units for Mass Air Flow are g/sec or grams per second. This is the only units for Mass Air Flow so whether the user chooses imperial or metric units that is the unit they will be viewing.

11.1.15 Intake Manifold Pressure (MAP)

Intake Manifold Pressure or MAP refers to the absolute pressure inside the intake manifold. This value is defined as the measure of the restriction to the airflow through the engine. As this pressure builds up it will be harder for the air to be forced into the engine. MAP can also be used to help obtain information on a vehicles instantaneous fuel economy. Since the MAP data is real time data it's continually changing values will be more useful for real time fuel economy rather than average fuel economy. This value is normally measure in kilopascals or kPa (Absolute). MAP may also be measured in atmospheric units (atm) if the user is viewing data in imperial units.

To retrieve this value one must send a request for mode 01 and PID 0B from the OBD. For example see **figure 11.1.15a**:

	Mode	PID
Request	01	10

Figure 11.1.15a – Intake Manifold Pressure (MAP) Request

The OBD will then respond to the request for intake manifold pressure (MAP) with this String of values to be deciphered shown in **figure 11.1.15b:**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0B	XX	00	00	00	00

Figure 11.1.15b – Response to Intake Manifold Pressure (MAP) Request

Where 41 and 0B indicate a response to a mode 01 PID 0B request and XX refers to the actual hexadecimal value of the pressure in the intake manifold. This is a straight up value and needs no conversion unless one would like the value displayed as atmospheric units instead of kilopascals. In which case you must take the value found and divide by 101.3 to achieve atm. Therefore, to convert from kilopascals to atmosphere the formula is: atmosphere = kilopascals/101.3.

The formula above is used in the *formatResult( )* function for Pressure*OBDFunction( )* shown in **figure 11.1.1a.** Also according to the figure one can see that IntakeManifoldPressureFunction( ) inherits from

Pressure*OBDFunction( )*. This class does not overload any methods of Pressure*OBDFunction( )*. It only has its own constructor that shows what units it has and what the string sent to the microcontroller should be. Since it is a pressure function its units will be the same as all other pressure functions so the conversion for this will be the same for other pressure functions as one will see with fuel pressure below.

11.1.16 Fuel Pressure

Fuel Pressure is defined as the pressure in which fuel is given to the fuel injectors by the vehicles fuel pump. A loss in fuel pressure can be very bad for performance. Low fuel pressure means that the fuel will not be put into the engine as quickly or efficiently as need be. This can result from a bad fuel pump a leak in the lines or a few other malfunctions in the fuel system. The nominal value for fuel pressure should be around 10 – 15 psi. This value will be displayed as a gauge and possibly as a numerical value as well.

#change this and other PSI values to atm

Fuel pressure is mode 01 PID 0A. So to get this value one must send this request string to the OBD on the vehicle shown in **figure 11.1.16a**:

	Mode	PID
Request	01	0A

Figure 11.1.16a – Fuel Pressure Request

And the OBD will respond to the request for the Fuel Pressure with its string that must then be formatted as shown in **figure 11.1.16b**:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	0A	XX	00	00	00	00

Figure 11.1.16b – Response to Fuel Pressure

With 41 meaning that it is mode 1 and 0A showing what PID it is. Also XX represents the 2 digit hexadecimal representation of the fuel pressure. However, to obtain the actual absolute value one must multiply this value by 3 to receive the true value in kPa. This then gives us the formula to calculate fuel pressure as:

$$Fuel\ Pressure(kPa) = 3 * XX$$

As with IntakeManifoldPressureFunction( ), FuelPressureFunction( ) also inherits from Pressure*OBDFunction( )* and it uses the *formatResult( )* method from Pressure*OBDFunction( )*. This class has the same units as well, which are kilopascals (kPa) and atmosphere (atm). The difference in this class from

IntakeManifoldPressureFunction( ), other than the string that must be sent to the OBD, is that in order to obtain the correct value a *transform( )* must be done. Therefore, the *transform( )* method of *PressureOBDFunction( )* must be overloaded to include the equation above. This will mean that the *transform( )* function takes in the value that is stripped from the OBD's response and converts it using the formula and then returns the converted value. Of course, this value may then be converted again in *PressureOBDFunction( )*'s *formatResult( )* method if the user has chosen to go with imperial units.

11.1.17 Engine Load

Engine Load refers to how much of the engine is being used. Specifically for this application it refers to how much percent of the total engine capacity is being used. This value is the ratio of the current airflow of the vehicle divided by the peak airflow. The current airflow is related to how much throttle you are applying on the vehicle at this current time while the peak airflow is the maximum amount of airflow the engine will ever be able to obtain. The value that is received will be a percent (%) that may be displayed as a numerical value and as a gauge depending on the users' selection.

To obtain this value from the OBD the request that must be made must be of mode 01 and PID 04. Therefore the request will be the string shown in **figure 11.1.17a**:

	Mode	PID
Request	01	04

Figure 11.1.17a – Engine Load Request

The OBD will then respond to the request for the Engine Load with the string in the form shown in **figure 11.1.17b**:

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	04	XX	00	00	00	00

Figure 11.1.17b – Response to Engine Load Request

The 41 represents mode 01, 04 means it is PID 04 and XX represents the data that was requested. The data will be a hexadecimal value that must be formatted to its proper value by multiplying by 100 and dividing by 255 so the formula will be:

$$Engine\ Load = XX * \frac{100}{255}$$

EngineLoadFunction( ), shown in **figure 11.1.1a** shows that it will inherit from *IntOBDFunction( )* inferring that it is a numerical value that must be converted. However, since the units for Engine Load are percent (%), there is no need to overload the methods *getUnit( )* and *getImpUnit( )* found in the *IntOBDFunction( )* class. But, the formula above does need to be implemented, so it will be put into the *transform( )* method which this class will overload from its inherited class. The number will be formatted as described in *IntOBDFunction( )*: a string representing the integer value. Therefore, it will not contain any values after the decimal point.

11.2 Fuel Economy

Fuel economy refers to the calculated fuel consumption of the vehicle. This value is calculated using data from the mass air flow (MAF) sensor and the speed of the vehicle. This value will be a double that represents how many miles per gallon the vehicle is getting at that instantaneous moment. This can be used to help a driver operate his/her vehicle more efficiently. The system will also save this data in order to create an average of the instantaneous values to get an average mile per gallon for the vehicle. Fuel Economy doesn't come straight from the OBD. Rather it's a value that will be calculated using the values that come out of the OBD. In order to calculate the fuel economy one must use the following formula:

$$Fuel\ Economy(MPG) = \frac{(14.7 * 6.17 * 4.54 * speed * 0.621371)}{(3600 * 100 * maf)},$$

- 14.7 = grams of air to 1 gram of gasoline, which is the ideal air/fuel ratio of most vehicles
- 6.17 = density of gasoline in pounds per gallon (lb/gal)
- 4.54 = convert pounds per gallon (lb/gal) to grams per pound (g/lb)
- .621361 = conversion of kilometers per hour (KPH) to miles per hour (mph)
- 3600 = seconds per hour
- 100 = grams per second (g/sec) for mass air flow (Maf)
- Speed = speed of vehicle, from OBD
- Maf = mass air flow of vehicle from OBD

The formula above will give u fuel economy in miles per gallon (MPG), however if one wants this value in metric units which is kilometers per liter (kml) one simply needs to multiply the value found in the formula above by .354013 or in other words: $Fuel\ Economy\ (MPG) = Fuel\ Economy(KML)\ * .354013$. Now, in order to obtain the average fuel economy value, every time the fuel economy value is sampled one must log this value and then find the average of all the values in the log. More on logging can be found in the **section 8** that discusses how logging of data will be implemented.

As we can see from **figure 11.1.1a** *FuelEconomyFunction( )* extends *OBDFunction( )* directly and overloads the *run( )* and *formatResult( )* methods. It also contains a *runFunc( )* method whose job is to run a function that is specified in its parameter in order to obtain values from the objects created. In the case of this class there will be two objects, speed and maf, which will be then used to calculate the fuel economy value. In the overloaded *run( )* method, the class will first create the two objects of *SpeedFunction( )* and *MassAirFlowFunction( )*. The class will then call the *sendFunc( )* method described earlier, then set variables for the speed and mass air flow by using each objects specific *formatResult( )* method. Finally, the class will use the formula above to get the proper value. The last method, *formatResult( )* overloads the *formatResult( )* method in *OBDFunction( )* and it will either return the value achieved in *run( )* or if the user has picked metric units will multiply that value by .354013 to achieve the fuel economy in kilometers per liter.

11.3 Battery Voltage

If a vehicles voltage becomes too low it means that the battery is not operating properly and may need to be replaced. Our system will measure the voltage and report this back as a double in which yellow will indicate the battery operating out of normal voltage range and white being within normal range.

**find out if high voltage is bad aswell*

Battery voltage is different the previous functions. The voltage from the battery can be determined by querying the ELM327 chip rather than by requesting this data from the vehicles OBD. The ELM327 has a long list of commands that it accepts and can change parameters for different operating conditions. All of these commands begin with the header “AT”, as does the command to retrieve the voltage from the car. When the command begins with AT the ELM327 knows that it must handle this request. To get the input voltage which is the voltage that the car battery contains one must enter this command to the ELM327 device shown in **figure 11.3a** below:

	Mode	PID
Request	AT	RV

Figure 11.3a – Engine Load Request

In which “RV” stands for read voltage. The ELM327 will then respond with a 3 figure decimal value that represents the voltage. This value will be accurate down to about 2%. The value that should be displayed should be something similar to about 12.5V or so.

Since *BatteryVoltageFunction*( ) is not the same as the other functions in that it gets its data from the ELM327 rather than from the OBD, it is not technically an

OBDFunction( ). However the method for it is still the same: send a request and translate the response. Therefore, we may have this function inherit from the *OBDFunction( )* class and utilize the methods found in that class. Because, of this we will only need to overwrite one method instead of having to rewrite all the methods for doing the same procedures. The method *formatResult( )*( ) is the only one that need be changed. This method will now be changed to return the string representation of the double value that was returned by the ELM3211. The value is returned as a straight number therefore we do not need to grab bytes or convert it in anyway, the value simply needs to be converted to a string formatted to display the number with one digit following the decimal place.

11.4 Error Codes/ Clearing

Diagnostic Trouble Codes or DTCs are codes that occur when something in a vehicle becomes faulty such as a sensor or misfire. When an error is triggered the check engine light (CEL) on the vehicle shall turn on. To figure out this error normally one would plug in a SCANTOOL to read this data from the OBDII port. This data comes through as an error code that must be deciphered based on make and model of the car. After these codes are fixed the check engine light may still remain on until it is either reset through a lengthy complicated method of switching the vehicle on and off or by simply sending a signal from the reader to the OBDII port to clear it. DTC is one of the most appealing features of any OBDII scanner. The ability to then connect to the internet to find a solution almost instantaneously will also add to the value of this device.

The first thing one would want to do when trying to read error codes is first find out how many faults are present. To do this we must do a mode 01 request on PID 01. In this case one must send the string represented by **figure 11.4a** below:

	Mode	PID
Request	01	01

Figure 11.4a – DTC Code Count Request

Then the OBD will respond to the request with a string. This string will need to be converted because if the Malfunction Indicator Light (MIL) is on the most significant bit (MSB) in the third bit will be set to 1. The string sent by the OBD is represented by **figure 11.4b:**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	41	01	XX	YY	ZZ	AA	BB

Figure 11.4b – Response to DTC Count Request

41 indicates mode 01, 01 shows PID 01 and XX is the actual number of error codes. However one must subtract this number by 80 hex to find the actual true

number of error codes. This will only work of course if the check engine light is on because it will set the most significant bit or MSB to 1.The main use of this is to check whether or not the malfunction indicator light (MIL) also known as CEL is on or not. A better method however to find the number of error codes is to and(&) the hex value with 7f hex and that will return the number of error codes, in other words number of error codes = XX & 7f. Also, if you and that value with 80 and get 1 then you will know if the check engine light was on or not. The following bits after YY, ZZ AA and BB are bit mapped and are use to describe which tests were and were not supported and completed.

After one determines the number of errors the next step is to find out specifically what those error codes are. To do this one simply sends a mode 03 request. Mode 03 requires no PID therefore, the string sent will simply be “03”. The response to this request from the OBD will be a bit string that is similar to what is represented in **figure 11.4c**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	43	XX	YY	ZZ	AA	BB	CC

Figure 11.4c – Response to DTC Request

43 indicates that it is a mode 03 request (43 – 40 = 3) and XX and YY is the actual trouble code read as XXYY. There may be more data to this in which case it is every 2 bytes that must be read in pairs to obtain what the codes are. A “0000” indicates padding and the end of the actual transmission. These pairs of values must be decoded first by replacing the first hex digit received using this chart:

1ST Digit	**Replace w/**	**Description**
0	P0	Power Train Code – SAE defined
1	P1	“ “ – Manufacturer Defined
2	P2	“ “ – SAE Defined
3	P3	“ “ – Jointly Defined
4	C0	Chassis Code – SAE defined
5	C1	“ “ – Manufacturer Defined
6	C2	“ “ – Manufacturer Defined
7	C3	“ “ – Reserved for Future
8	B0	Body Code – SAE defined
9	B1	“ “ – Manufacturer Defined

A	B2	" " – Manufacturer Defined
B	B3	" " – Reserved for Future
C	U0	Network Code – SAE defined
D	U1	" " – Manufacturer Defined
E	U2	" " – Manufacturer Defined
F	U3	" " – Reserved for Future

Figure 11.4d – Decoding of Error Codes

For example, let us assume that the string sent by the OBD in a request for the trouble codes was "43013300000000". Then we would separate the string out to fit the form of **figure 11.3c.** The resultant of this is depicted in **figure 11.4e:**

	Mode	PID	Byte A	Byte B	Byte C	Byte D	Byte E
Response	43	01	33	00	00	00	00

Figure 11.4e - Example Response to DTC Request

Then you would notice that it is a mode 03 response and that the next two bytes combined are "0133". The first digit of this value is 0 which corresponds to P0 on the chart. Next, we must concatenate this value to the rest of the string to obtain P0133 as the error code. When one researches this error code one will see that this is the error code for "oxygen sensor circuit slow response". Knowing what the error is one may now set in motion the appropriate methods to fix this fault.

Finally, once the Check engine light is fixed one will want that light to go away. The only way to do this is to clear it through the OBD. This is achieved by simply sending a mode 04 request with no PID to the OBD. Once that request is received by the OBD the OBD will respond with the byte string "44" to indicate that it has received this and has completed the operation. However, one must wary of sending this command as issuing this command will perform all these operations:

- Reset the number of trouble codes
- Erase any diagnostic trouble codes
- Erase stored freeze frame data
- Erase DTC that initiated freeze frame
- Erase oxygen sensor test data
- Erase mode 06 and 07 information
- However, it will not erase permanent (Mode 0A) trouble codes, which can only be reset by the ECU

The issue that may occur by doing this however is that the vehicle may not perform correctly as it recalibrates or “relearns” information that was necessary to run properly. After a short period however the vehicle should begin operating normally or at the very least begin to operate as it did before the reset request was issued. Also if what caused the check engine light was not fixed eventually it will turn on again as the car cycles itself. To avoid erasing all this data by accident the device will need to verify that the user really wants to erase the data by displaying a prompt that will explain what the repercussions of issuing this request may be and asking the user if they are certain they want to perform this task.

DtcNumFunction( ) is the class that will take care of the first function discussed in this section. *DtcNumFunction( )* will be the function that finds out exactly how many error codes are in the system and if the check engine light (CEL) has been activated. This function, according to **figure 11.1.1a**, will directly inherit from *OBDFunction( )*. This class will overload the *formatResult( )* method found in *OBDFunction( )* by returning a string that will say whether or not the check engine light is on followed by how many codes are in the system, using the formulas supplied earlier in this section. This method will not only set the amount of error codes in the system but it will also set whether or not the check engine light (CEL) is on or not. This class also contains two of its own functions *getCodeCount( )* and *getCelOn( )*, which will return how many error codes there are and whether or not the malfunction indicator light is on respectively. After that object has been created and the code count is achieved the next step is to look to the *ErrorCodesFunction( )* which will display the actual error codes themselves. Again this method also inherits from the *OBDFunction( )* class but it creates a *DtcNumFunction( )* object and uses this to get the amount of codes so that it may use it in its *run( )* method, which this class overloads from *OBDFunction( )*. The run method here creates a *DtcNumFunction( )* object, obtains the code count adds 2 to it and divides by 3, code count +2 /3, in order to obtain how many loops are needed to get all the error codes. We then check the first value of the error code change it to the proper letter and finally append the rest of code to it to obtain what the error code is and return each one using the overloaded *formatResult( )* method.

12.0 Logging

The On Board Diagnostic system in a vehicle reports data in real time to the user based on certain requests as shown in all of **section 11.** However, the OBD has no way of saving this data to be viewed at a later time to see if there are any important changes when certain things have been done to a vehicle. For instance, if one has put an aftermarket part on their vehicle they can see how their average fuel economy was affected if this data was saved. Same goes for using different grades of gasoline. In order to achieve this, our system will need to implement a logging feature that will save away the data that was received from the On Board Diagnostics on the vehicle. This will allow the user to view the date and time that the data was recording along with the actual value of that data. Also, the user will be able to calculate an average value and then clear that log to get a new data set that the user may use to put against their old log and see what changes have taken effect. This feature may also be used in the future to help further diagnose malfunctions in the vehicle. This can be useful in repairs and provide useful assistance to mechanics in the field.

To implement this we will need to add a new log object that will be created every time data is read from the OBD. We will also need to create a new activity within the activity package to display the logs to the user depending on which log the user wants to see. **Figure 12.0.a** below shows a class diagram of how this system will be added to the current system. This is not the overall diagram though; it is merely enough of the class diagram to show what is necessary to this section of the entire system. It shows how the main activity class will interact with the log class and how the new log activity class will interact with the system as well.

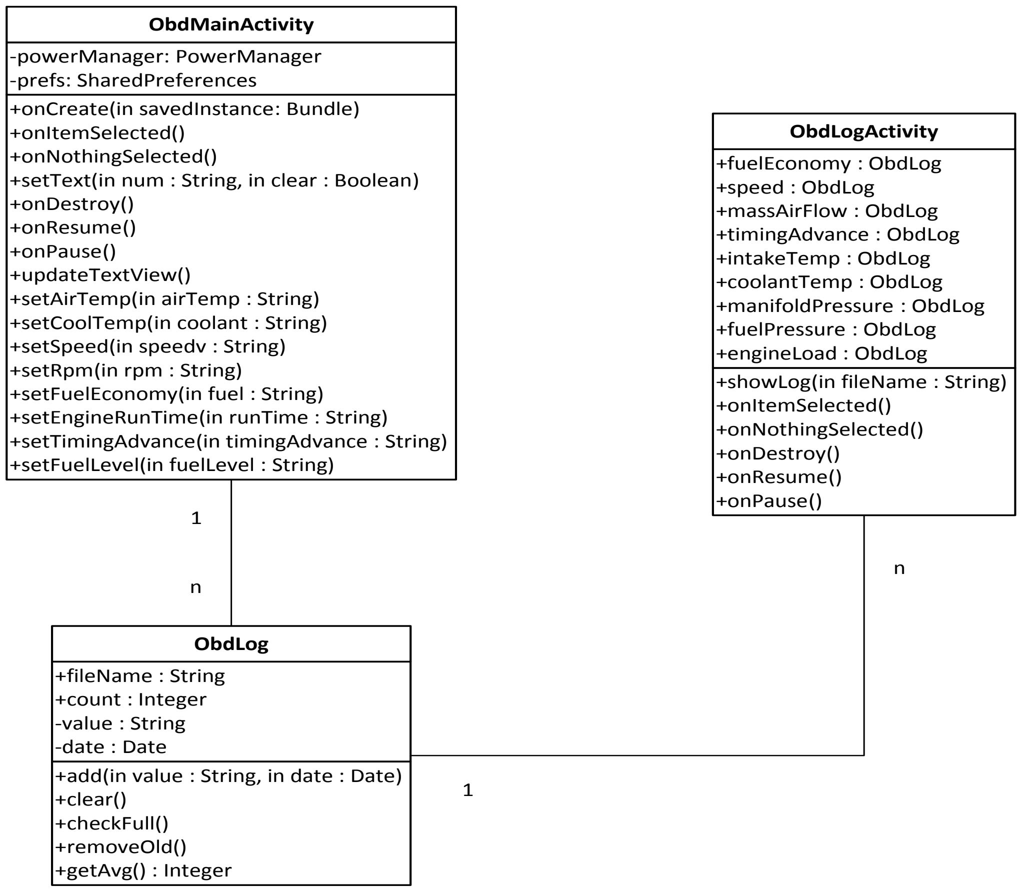


Figure 12.0.a - Logging Class Diagram

From the diagram above we can see what methods and how each class relates to one another to create and view the logs. When a log is created it is added to the log file with the name of the value that the log contains. The log file will be a simple text file in which the first line will be the number of log entries and every line following will consist of the date and time the log was taken along with the value of the log. The first class that will be discussed is *OBDLog( )*. An *OBDLog( )* will consist of a string that denotes what the name of the log file will be and a count which will be the amount of logs inside the specific log file. The *add( )* method in this class has the job of taking the specific logs date, time and value and adding it to the log file specified by file name. The *add( )* method must also increase the count variable. Before *add( )* does these operations it must first call *checkFull( )*, which will check if the log file has too much data in it. Since the data is all text a value of 10,000 or even more should not harm the performance of the system too much. Therefore, if the log has more than 10,000 values in it the system will then call *removeOld( )* which will remove the first 5,000 values from the system so that more values may be added. *RemoveOld( )* will then update count to reflect this change. The final method of this class is *getAvg( )*. This methods purpose is to sum up all the values in the log and then divide that by count to return the average of all the values. This average will then be displayed to the user in the *OBDLog( )*Activity( ) class.

The *OBDLogActivity( )* class consists of many *OBDLog( )* objects. Not all the OBD functions will have logs tied to them. For instance, a log of the RPM values would not be informative as the RPM of a vehicle varies too rapidly. *OBDLogActivity( )* is the class that deals with displaying the all the logs to the user along with the average value. Therefore, this class contains methods inherited from the Activity class such as; *onDestroy( ), onResume( ), onNothingSelected( )* and *onItemSelected( )*. It will also contain *onCreate( )* which is called as soon as the user selects the log item from the main menu. When *onCreate( )* runs it will create a drop box of the different logs that the user may view. When the user selects a log the *onItemSelected( )* method comes in and from there *showLog( )* will be called*. ShowLog( )* will display the log that its fileName parameter dictates. It will achieve this by simply opening the log file given by fileName and showing it to the user. The display will contain the last 20 values entered in to the log file with a button to display the older values. The purpose of this is to improve performance as showing the entire log will take too much time and is usually not needed. The system will then call the *getAvg( )* method in the *OBDLog( )* class to obtain the average and display that to the user as well. All the data is tied to text files that must be located in the android system. Therefore, if this file is not found the program should create a blank log file that must be filled. We will code the program to perform this check every time the program is run.

In order to add logs to the log file the OBD reader must be run and values must be read from the OBDII port. To do this the user must select the OBD Reader from the main menu in which case the *ObdMainActivity( )* would be called. When *ObdMainActivity( )* is called the *onCreate( )* method is run which will create a view for the user based on the functions the user has chosen to run. Then the proper set methods will be called. Next the system will send and receive data to the OBDII port. When this happens the data received will be formatted and then displayed to the user within the set method. Each time the data is displayed an *OBDLog( )* object will be created. That objects data will then be added to the specific log by using the *add( )* method in *OBDLog( )*. This is how the log files will be filled up. It is within the specific set method that this is done in order to ensure that the data goes into the proper log file. As the log files are filled up the user will then be able to switch to the log section and view the values entered. Also, if the user chooses to view the values on a bigger screen the user only need to open the log file on a computer by either sending it to a computer or mounting the phone as a disk and viewing the log file on the disk.

13.0 Other Functions

Alongside the functions for reading data from the OBDII port discussed in **section 7** this application will also be able to perform other functions thanks to the output ports on the microcontroller board. The application will also be able to roll up/down the windows, unlock/lock the car, start the car and possibly pop the trunk depending on if the vehicle has that feature. In order to do this we must first add this ability to the user interface of the android powered device. We then must add hardware with this capability to the chip that the android talks to and finally we must find the correct wires that deal with the functions discussed so that we may tap into those wires to implement these functions. The following sections will discuss how the software on the phone and how the hardware of the chip must be modified to allow these functions to be implemented for this application.

13.1 Software Implementation

Implementing these functions will require us to make a new protocol for these functions. What will happen is that when the user clicks for a function to be performed a message will be sent from the android device to the microcontroller and the microcontroller will then read the header and know what to do from there. Normally it will send the entire string to the ELM327, but if it receives the header of the protocol we have created the microcontroller will know that it must now handle the request by loading the proper voltages in the right ports. The chart in **figure 13.1a** below shows what the new protocol will be:

Function	Header	Data
Unlock	FC	01
Lock	FC	02
Pop Trunk	FC	03
Panic	FC	04
Windows Down	FC	05
Windows Up	FC	06
Start	FC	07

Figure 13.1.a - Protocol for Other Functions

We can see that the header for this protocol is “FC” for all functions, therefore once the microcontroller reads the header is FC it will know it is to perform a function that is not on the OBD. It will then read the data and follow the instructions discussed in **section 13.2** below. To perform these functions on the software side we must first create a menu that the user can see and use. The user interface is discussed in the GUI section above. To create the interface for the user we must add a class to the activity package in the android application. This activity labeled *remoteStartActivity( )* will contain methods to create and handle the selections of the user. This class will create 7 objects from the *org.obdDroid.otherFunction* package. These 7 objects are: *unlockFunction( ), lockFunction( ), trunkFunction( ), panicFunction( ), windowDownFunction( ), windowUpFunction( )* and *startFunction( )*. These functions will inherit from a super class *otherFunction( )*. The class diagram is shown in **figure 13.1b** below followed by a walkthrough of each class and how they are implemented together which includes a description of the methods and variables in each class and their purpose.

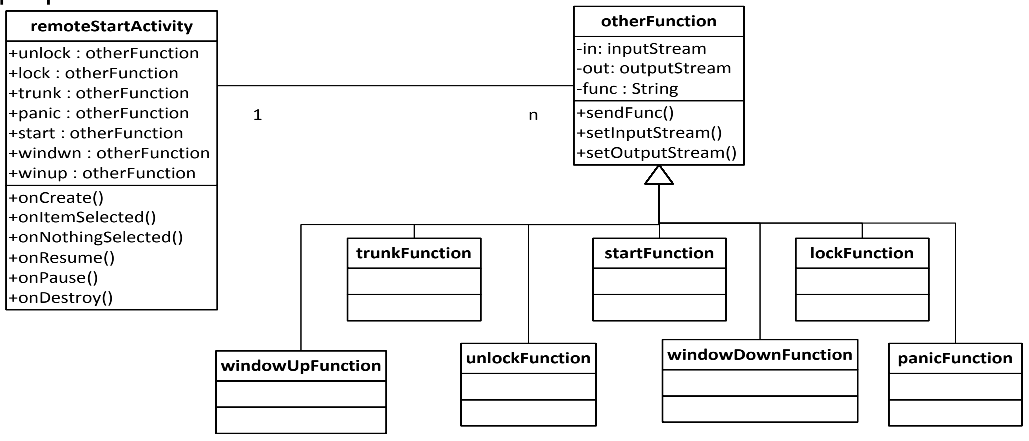


Figure 13.1b Other Functions Class Diagram

The first class from the figure above is the *remoteStartActivity( )* class. When this class is run from the android device it will create the seven objects that inherit from the *otherFunction( )* class. Next, *remoteStartActivity( )*'s *onCreate( )* method is called which will create the user interface. The *onSelectedItem( )* method is the method that will be invoked when a user selects a function on the screen. The way this works is that if the user selects an item *onSelectedItem( )* will then call the *sendFunc( )* method with the string specified in the protocol of the specific function that the user has selected as the parameter. *SendFunc( )* will send that to the microcontroller to be evaluated and initiated. The *sendFunc( )* method comes from the *otherFunction( )* class in which all of the other functions will inherit from. This class will also contain the *setInputStream( )* and *setOutputStream( )* method that will send data to the microcontroller. Each class beneath the *otherFunction( )* method really only needs its constructor which will contain the string of what its protocol should be as a parameter for instance, the string for the *unlock( )* class is "FC01", similar to the form of the *OBDFunction( )* classes.

13.2 Hardware Implementation

In order to perform the functions discussed in this section we must tap the trigger wires located in various positions in the vehicle. The tap must come from the i/o port on the microcontroller. Depending on how the vehicle is wired the voltage that comes out of the I/O port must be either a positive trigger or a negative trigger. Positive trigger means that the voltage must be +12 where as negative trigger indicates that a -12 volt signal must come out of the specific port. For this project we will be using a 1998 Honda Accord LX. This vehicle is a negative trigger vehicle which means all the voltages that come out of the microcontroller need to be negative. The microcontroller will be programmed to hand both negative and positive triggering. **Figure 13.2.a** below depicts a chart that describes the location and color of the wire that needs to be tapped in order to perform the function indicated under the column called device. This chart is of course specific only to the Honda LX but the implementation will be similar for any vehicle. In order to tap the wires we must use a vampire tap as described in **<u>section x.</u>** This way we will not need to completely splice the wire on the vehicle and disturb the wiring of the factory system since this is just an add-on to the vehicle and not a replacement for any system currently on the vehicle.

PART	COLOR	LOCATION
12 VOLT CONSTANT	WHITE (+)	IGNITION SWITCH HARNESS
STARTER	BLACK/WHITE (+) See NOTE *1	IGNITION SWITCH HARNESS
IGNITION 1	BLACK/YELLOW (+)	IGNITION SWITCH HARNESS

PARKING LIGHTS (-)	RED/YELLOW (-)	@ STEERING COLUMN HARNESS
PARKING LIGHTS (+)	RED/BLACK (+)	IN DRIVER SIDE FUSEBOX
POWER LOCK	BLACK/BLUE (Negative (-)) See NOTE *2	IN PASSENGER SIDE FUSEBOX
POWER UNLOCK	ORANGE (Negative (-)) See NOTE *2	IN PASSENGER SIDE FUSEBOX
DOOR TRIGGER	BLACK/WHITE (-)	IN PASSENGER SIDE FUSEBOX
DOMELIGHT SUPERVISION	USE DOOR TRIGGER, Requires Part #R30-H Relay	
TRUNK RELEASE	WHITE/RED (+), Requires Part #R30-H Relay	IN DRIVERS KICK PANEL
HORN	LIGHT GREEN/BLUE (-)	@ STEERING COLUMN HARNESS
BRAKE	WHITE/BLACK (+)	@ SWITCH ABOVE BRAKE PEDAL
FACTORY ALARM DISARM	BLUE (-) See NOTE *4	
ANTI-THEFT	TRANSPONDER ANTI-THEFT SYSTEM, REQUIRES 791 BYPASS MODULE AND EXTRA IGNITION KEY	TRANSPONDER @ IGNITION SWITCH TUMBLER

Figure 13.2a - Honda Accord Wiring Chart

Using the chart above we can now create a diagram of how the wiring from the chip to the vehicle should be. The diagram in **figure 13.2b** below will display where the wires will come from the chip and where they will go to the vehicle to perform these specific functions. Also we will place a clip on the wires from the chip to the car so that we can plug and unplug the wires to remove the chip so that we do not have wires hanging everywhere which will be shown in the diagram which we will refer to below:

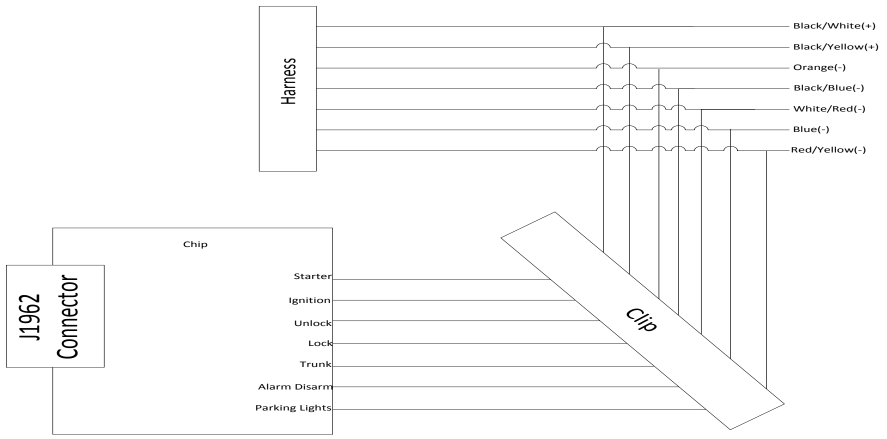


Figure 13.2c - Chip to Harness Diagram

The diagram above shows how the wiring of the system will be done. It shows which wires need to go to which port from the harness on the vehicle and it also includes the clip that we will use so that the wires may be disconnected for the chip to be easily removed. Although the chip doesn't exactly show how the microcontroller needs to be programmed. For instance if one wants to unlock and lock the car the microcontroller will need to send the -12 volt signal to the port that either unlocks or locks the car however, for starting the car first a signal must be sent to disarm the alarm, then the ignition and finally the starter which will then trigger the car to start. Also when the car locks we may also send the port for the parking lights with a -12 volt signal twice to initiate a blinking of the parking lights along with the signal going to the horn to initiate honk of a horn to indicate the car has been locked and one blink of the parking lights to indicate the car has been unlocked. Also we may use the horn to sound off when the panic button is hit, in which the port for the horn will be loaded with -12 volts in sequential succession, meaning that it will have -12 volts then 0 then -12 volts again. It will continually perform this pattern until another button is hit on the android device or the microcontroller is unplugged. We must remember that this sequence is specific to the Honda Accord, as some vehicles have an extra ignition that must be triggered before the car can start. Also some vehicles do not have a factory alarm while some vehicles may have an alarm system so complex that starting the car with this method simply cannot be done. However, this will work for many cars and will work for almost all cars with OBDII in the market with slight modifications. To allow for these modifications we will ensure that the chip has extra I/O ports in-case the vehicle would need them.

One may notice however that the window up and window down function is not discussed in the chart or the diagram. To perform this we still need to load the port on the chip with a -12v output but it will need to tap into the vehicle in a different way. Also because winding down the window is not just a single trigger the system will be programmed to load the port with -12 volts while the button is pressed rather than if the button is pressed. That way when the button is released the window will stop rolling down.

Normally the way that the window works is that it is tied to the battery and when the switch is closed the circuit is closed because all the parts are tied to the same ground. While the circuit is closed the motor will run to wind the windows down. This wiring will be the same for the other windows as well as for the winding up of the windows. So, in order to roll the windows up/down we simply need to mimic a close in the circuit to the motor. Therefore, we simply tap into the wire after the switch from the chip and send a -12 volt signal on that wire which will turn the motor on to perform its function and once the signal is stopped so will the motor. Now one may wonder, "what if the window is all the way down and the roll down button is pressed again?" This is actually not an issue as these windows are tied to relays that cut off once the window has reached its maximum or minimum distance. This is why when you press the switch if the window is already all the way down it does nothing. Since we are simply tapping the wire this feature will still hold and work the same way. The only drawback is if the user wishes the window to go all the way down or up they must hold the button until it goes all the way up, there will be no auto up feature.

14.0 User Interface Design and Implementation

The user interface is the primary point of interaction for the user. A good interface can make or break a program. With a program as complicated and feature-filled as ours, it’s important to have an interface that clearly and simply shows all of the options and features available. When designing the interface, we tried to keep things similar to what a typical driver would have seen before. For example, the physical function controls are organized like a normal electronic key fob that you’d have on a keychain. The various gauges are arranged to look reminiscent of a normal car dashboard, and everything is large and easy to see so it’s not as dangerous to glance at while driving.

It’s not only important to keep the interface similar to what one would expect from a normal car, but also to stick with standard Android user interface conventions. To anyone familiar with the Android operating system, when you hit the physical “Menu” key you expect a menu to pop up in the lower part of the screen with various settings and options. Likewise, single pressing a gauge in the OBD Reader screen to view more in-depth information, or long-pressing it to customize it are natural extensions of similar behavior on the Android home screen. Since the program is on a touchscreen, the interface also can’t be cluttered or small. Everything is large and easy to press and navigate around. Our goal in designing this interface was to make a clean, natural, and easy to use user interface, and we feel we have achieved that goal.

14.1 Research

When we started designing the Android user interface for our project, we began by looking at other similar programs for Android and PC and seeing how we could effectively portray the data the OBD would give us, and how to create an intuitive input interface for the physical car functions.

Ideally we would have as much screen resolution as possible and be able to fit several dials and gauges and graphs onscreen at once like in the popular laptop OBD scanner program ScanXL seen in **Figure 14.1a**. Since we’re working on a phone interface however, we can’t show as much data all at once. To get around this, we split these views into a few different screens so that it’s still intuitive to use, but the screen doesn’t get clogged with information. We did this by having separate gauge and graph screens, and only showing one graphed variable at a time instead of four or more.

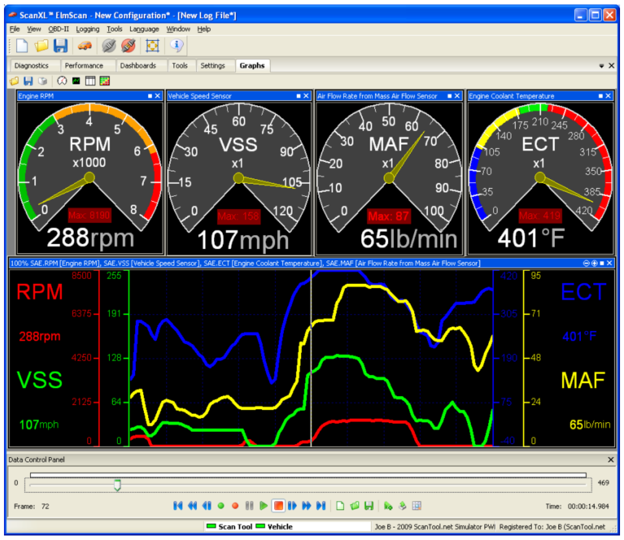
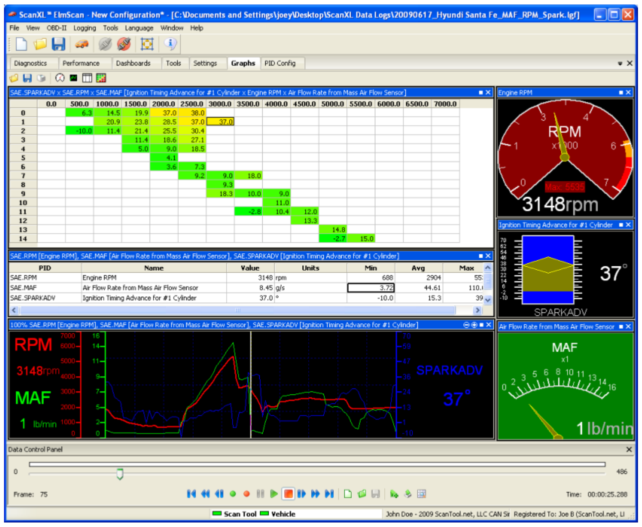

ScanXL™ ElmScan - New Configuration* - [New Log File*]
File View OBD-II Logging Tools Language Window Help
Diagnostics Performance Dashboards Tools Settings Graphs
RPM x1000
288rpm
VSS
107mph
MAF
65lb/min
ECT
401°F
Data Control Panel
Frame: 72
Time: 00:00:14.984

Figure 14.1a – ScanXL Professional graph screen

Similarly, we are much more limited with showing hard data as seen in **Figure 14.1b**. Though we have logs and graphs of data, they have to be separated. There's simply no way to cram all of this data onto a single screen in a legible way.


ScanXL™ ElmScan - New Configuration* - [C:\Documents and Settings\joey\Desktop\ScanXL Data Logs\20090617_Hyundi Santa Fe_MAF_RPM_Spark.lgf]
File View OBD-II Logging Tools Language Window Help
Diagnostics Performance Dashboards Tools Settings Graphs PID Config
RPM
3148rpm
SPARKADV
37°
MAF
1 lb/min
Data Control Panel
Frame: 75
Time: 00:00:25.288

SAE.SPARKADV x SAE.RPM x SAE.MAF [Ignition Timing Advance for #1 Cylinder x Engine RPM x Air Flow Rate from Mass Air Flow Sensor]

	0.0	500.0	1000.0	1500.0	2000.0	2500.0	3000.0	3500.0	4000.0	4500.0	5000.0	5500.0	6000.0	6500.0	7000.0
0		6.3	14.5	19.9	37.0	38.0									
1			20.9	23.8	28.5	37.0	37.0								
2		-10.0	11.4	21.4	25.5	30.4									
3				11.4	18.6	27.1									
4				5.0	9.0	18.5									
5					4.1										
6					3.6	7.3									
7						9.2	9.0	18.0							
8							9.3								
9							18.3	10.0	9.0						
10									11.0						
11								-2.8	10.4	12.0					
12										13.3					
13											14.8				
14											-2.7	15.0			

SAE.RPM [Engine RPM], SAE.MAF [Air Flow Rate from Mass Air Flow Sensor], SAE.SPARKADV [Ignition Timing Advance for #1 Cylinder]

PID	Name	Value	Units	Min	Avg	Max
SAE.RPM	Engine RPM	3148	rpm	688	2904	55
SAE.MAF	Air Flow Rate from Mass Air Flow Sensor	8.45	g/s	3.72	44.61	110.
SAE.SPARKADV	Ignition Timing Advance for #1 Cylinder	37.0	°	-10.0	15.3	39

Figure 14.1b – ScanXL Professional data view

Looking more at Android-specific OBD reading programs, we found that we really liked some of the concepts found in the Android app Torque, shown below in **Figure 14.1c**. There are several swipe-able customizable screens. All of the gauges can be configured by the user to read whatever information from the OBD reader that they want. We incorporated this into our design by having a main screen with several gauges, and allowing the user to long-press any of the gauges to customize it to their liking.

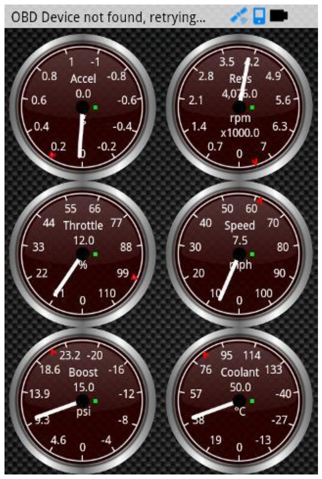


Figure 14.1c – Torque main interface

One homebrew program that specifically helped with creating/designing the gauges was "Vintage Thermometer, seen in **Figure 14.1d**. The actual code used in the program was available for this program, and it was very helpful in learning how to draw on the Canvas in Android, and importantly, how to accept data input to move a dial in a circle. It was also useful for learning how to efficiently refresh the gauge without taxing the phone's CPU.

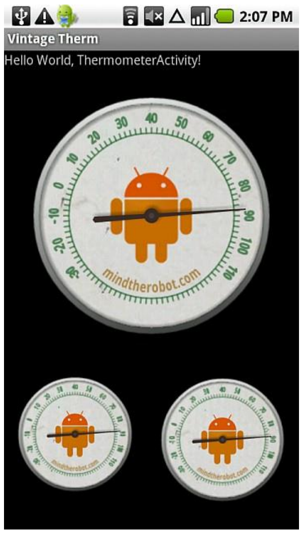


Figure 14.1d – Vintage Thermometer example

Another program that we used for inspiration for our design was alOBD Scanner., shown in **Figure 14.1e**. Though the main menu screen is a bit simplistic for our purposes, there is one really nice feature: in the program, there is an option to view a real-time graph of the incoming data while seeing the data itself above. Our design is similar, but replaces the simple number in the top portion with a large graphical gauge showing the data. This graph view is an intuitive way to view and graph data from the OBD. You could, for example, use this view as a graphical speedometer, and also see your speed over time.

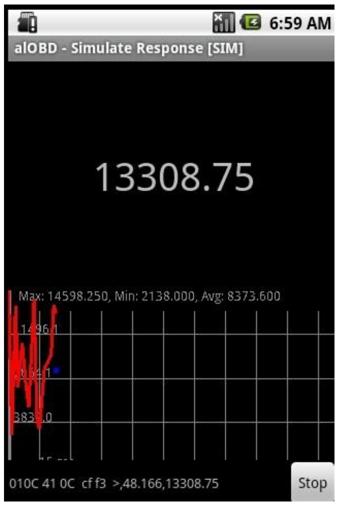

6:59 AM
alOBD - Simulate Response [SIM]
13308.75
Max: 14598.250, Min: 2138.000, Avg: 8373.600
010C 41 0C cf f3 >,48.166,13308.75
Stop

Figure 14.1e – alOBD Scanner graph view

For the electronic key fob part of our project, we looked to existing programs that performed a similar function. The Viper SmartStart user interface shown in **Figure 14.1f** is an elegant way to portray the functions we needed, with a large "Start" button in the center, surrounded by the four hardware functions. For our keypad however, we also wanted to include the functionality to roll the windows up or down, so some modifications were necessary. These will be discussed in the section 14.2.6.

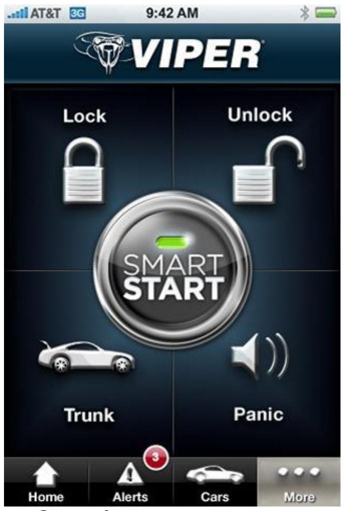


Figure 14.1f – Viper SmartStart main interface

14.2 User Interface Walkthrough

The next few pages will be dedicated to showing each screen of the UI and explaining some of the functions and reasoning behind each one.

14.2.1 Splash Screen

Figure 14.2.1a shows a simple loading screen that tells the user what the program does, who designed it, and masks any load time the program may have. It was eventually scrapped when we found that the program loaded fast enough by itself.

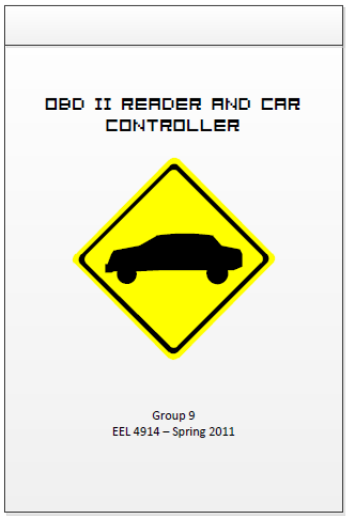


Figure 14.2.1a – Splash/Loading screen

14.2.2 Main Menu

Figure 14.2.2a is the main menu of the program, done in the familiar list view format seen in the Android Settings menu. The user can choose Start Connection to manually enable Bluetooth and send a connection request to our device to let them interact with the car. The connection is normally started automatically, but this option is available in case there was an error in getting connected – whether our device was out of range, there was an error pairing, or any other bug – this lets the user conveniently retry the connection. OBD II Reader lets the user access the OBD interface and read data from gauges or graphs. The Keypad button gives to access the physical functions of the car (starting, unlocking, rolling the window up and down, etc.). Logs lets the user view log files previously taken from the OBD reader and average or graph them. Error Codes lets the user read and clear error codes from the car via the OBD port. Finally, settings will let the user configure various options about the program.

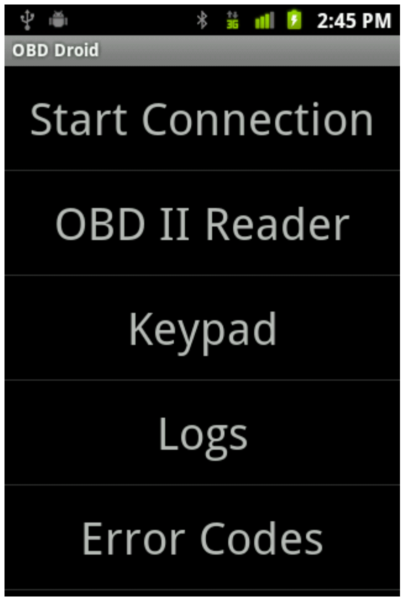


Figure 14.2.2a – Main menu

14.2.3 OBD Reader

Figure 14.2.3a shows the original mockup of our main view of the OBD reader. It was scrapped and replaced with just the obd gauge view, described below. It shows four different gauges which display various data streaming live from the physical OBD reader. Four gauges were chosen to be on the screen at once due to size and complexity. Putting more gauges on the screen would look too cluttered and would not be large enough to be readable at a glance. This layout also helps with performance because the CPU only has to redraw four gauges for every refresh instead of 6 or more. By default, the four functions shown on the screen will be Speed, Throttle, Boost and Acceleration. However, these functions are completely customizable, as seen and discussed further below in section 14.2.4. There is also a way to view a larger gauge display and real-time graph of the incoming data by single pressing a gauge, discussed more in section 14.2.5. The range of values displayed on the face of each gauge will be hardcoded in and set to an appropriate range. The numbers and ticks on each gauge will then be scaled appropriately for each function. Because there is no standard view that looks like this, a custom view will have to be implemented. This is discussed in detail in section 14.3.2.

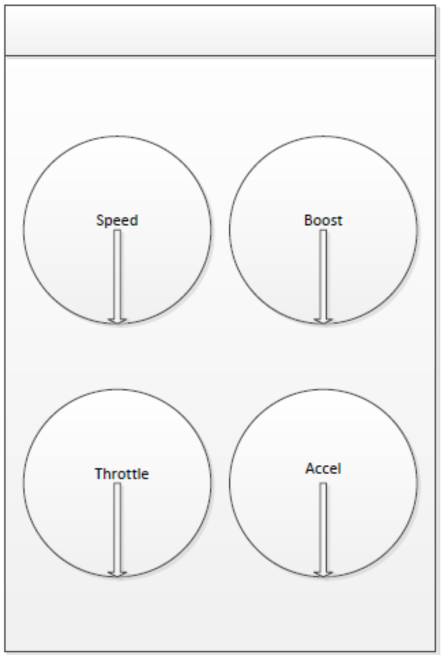


Figure 14.2.3a – OBD Reader view

14.2.4 Customize Gauge Screen

All of the gauges can be customized to read and display almost any value obtainable from the OBD reader. The user accomplishes this by long-pressing on the gauge they wish to customize and selecting a function, pulling up the menu shown in **Figure 14.2.4a**. The "long press to customize" function is a well-known function in Android to signify an ancillary function (similar to right-clicking in Windows). From here, the user can choose the function that they desire from the list of possible functions that can be read from the OBD reader. When an item is pressed, this screen recedes and the previously selected gauge is updated to read the new value. The name on the gauge and the scale used are also adjusted accordingly. This functionality was inspired by similar functionality in Torque, though in that program the user is allowed to not only customize the value of each gauge, but move them around, resize them and put them on multiple pages, similar to the Android home screen. We considered doing this, but decided it would not be feasible to add all of this extra functionality in the amount of time allotted for the senior design project. We decided to stick simply to being able to customize the readout of the four standard gauges on the main screen.

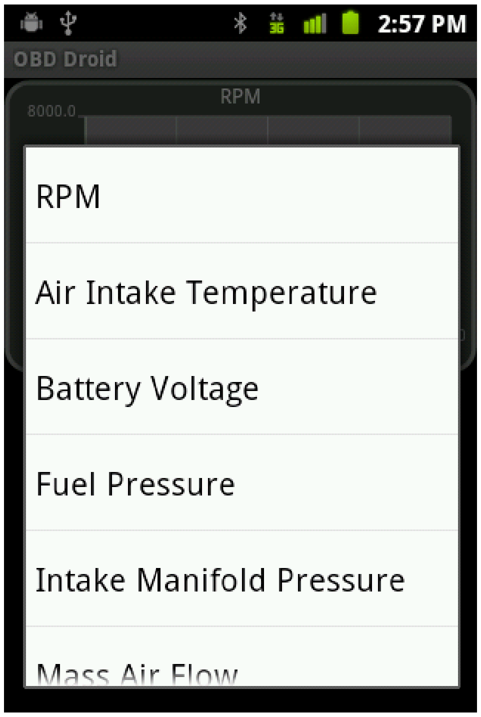


Figure 14.2.4a – Customize gauge screen

14.2.5 OBD Graph Screen

If the user single-presses on the gauge from the main OBD reader screen, they are brought to the screen shown in Figure **14.2.5a**. This is based somewhat on the design of the alOBD reader program, specifically the screen shown in **Figure 14.1e**, but instead of simply showing a changing number above the graph, a large gauge is displayed. The gauge is enlarged to show more detail, and a graph underneath it graphs the data over time with an appropriate range of values on the y-axis. The gauge uses the same custom gauge view used in the OBD reader screen, discussed in detail in section 14.3.2. The graph below is also a custom view, which is discussed in section 14.3.3. This is likely to be the most computationally intensive screen, since it has a real-time graph and a gauge running at once. If the user interface becomes choppy or unresponsive we may have to lower the refresh rate on one or both views. We feel this is one of the most important screens, since it gives the user a very clear view of a specific function. The user could use it as a digital speedometer for example, and be able to see (and log) their speed over time. Pressing the back button on the Android phone will return the user back to the main OBD reader view.

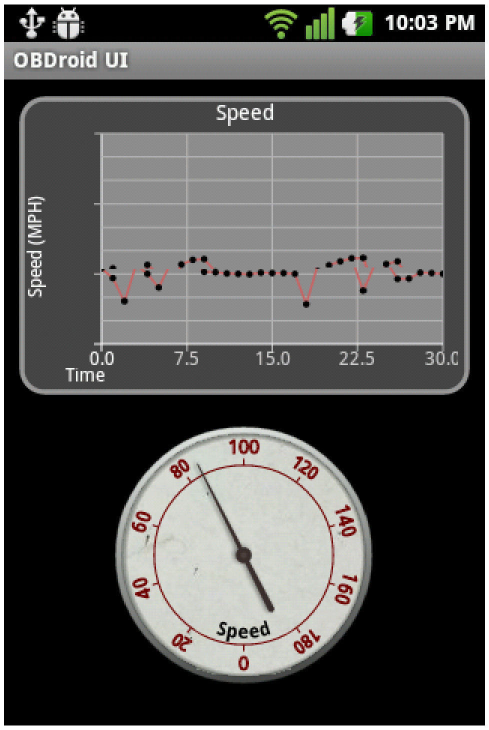


Figure 14.2.5a – Graph View

14.2.6 Keypad

Mentioned previously, our virtual electronic key fob shown in **Figure 14.2.6a** is based roughly on the Viper Smart Start main menu screen seen in **Figure 14.1f**. It has a similar layout to a standard key fob, with some modifications for our program. It has the standard functions that most users are used to: remotely locking and unlocking the car, sounding the horn and opening the trunk. We have the large start button in the center, mimicking the Viper UI and fashioned to look like a modern keyless start button. We made the Lock, Unlock and Start the largest buttons since they will likely be the functions used the most. Since the interface is obviously on a touchscreen instead of a more tactile medium some slight adjustments have to be made. There are no ridges or rubber buttons to let the user know where their fingers are. To avoid accidental button presses, we will make it so that once a user initiates a command, no other button can be pressed for a second or two. This way, if the user quickly hits the “unlock” button and slides their finger away, they won’t accidentally then set off the car alarm. It will be important that when the user accesses the Keypad function he is connected to our device over Bluetooth first. If the phone is not connected to our device, then a warning message will pop up and inform them of the situation and go back to the menu instead of letting the user press buttons and frustratingly think that nothing is happening.

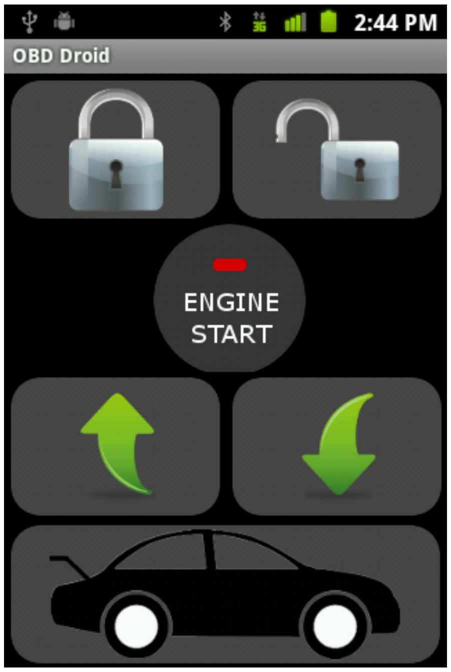


Figure 14.2.6a – Keypad view

14.2.7 Log Selection Screen

One of the features of our program is the log recorder, accessible via the main menu, which can be seen in **Figure 14.2.7a**. While the OBD reader is connected, timestamped logs are recorded of each variable and stored in a text file on the user’s SD card. The user can access this text file from the log selection screen by scrolling through the list of saved logs and pressing on the name of the appropriate function. Not every function of the OBD is logged however; for some functions, it just doesn’t make sense (DTC codes) and for others the numbers would change too rapidly to really have any significance - for example, the throttle jumps and changes wildly depending on what you’re doing, and a log of that would be mostly meaningless. The logging of data is automatically stopped and saved if the OBD Reader is turned off or if the phone loses connection to the device for any reason. If there is no existing log data and the user attempts to access the logs from that function, a blank file will be presented with a message stating the absence of any log data.

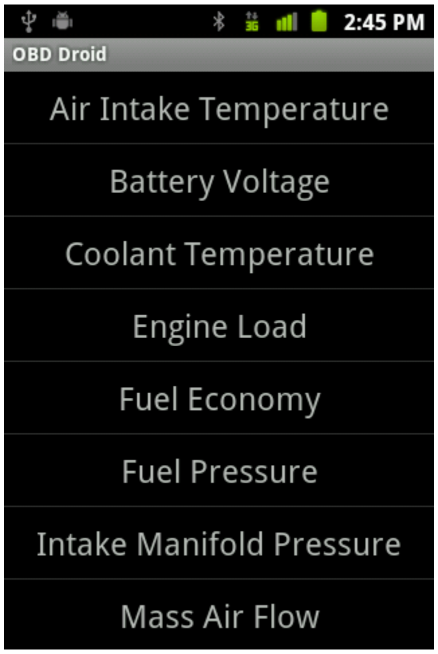
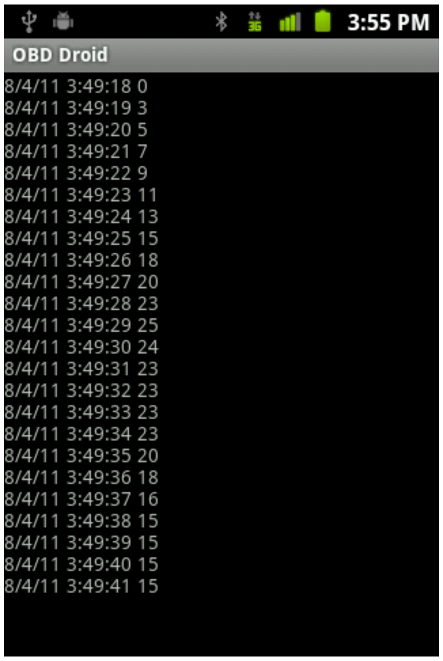


Figure 14.2.7a – Log Selection Screen

14.2.8 Log Viewing Screen

Figure 14.2.8a shows the screen view when viewing the log for one of the functions. By default, twenty entries are shown for performance/legibility purposes. At the bottom of the screen is a “Show more…” button that loads the next twenty entries in the log file (if available). Showing more still lets the user view the original sets entries, and scroll through everything that has been loaded up to that point. Though if the user really wants to view the entire log file all at once, it would be much easier and convenient to simply connect the phone to a computer and manually open the log file on the computer. It’s important that the log data is in an easily parse-able format, so when the data is extracted into an array for graphing, the process to separate the actual data and the time is easy and fast. Up to 10,000 entries can be stored in a single log file before being culled by removing the first 5,000 entries. This keeps the file size down (since we have to have log files for every function) and keeps down the computation cost of opening and writing to a large file. If the user presses the physical menu button on their Android phone, a menu is pulled up that gives the user the option to average all of the values in the log – so they could, for example, find their average speed during a trip, or their average gas mileage. Also in the menu screen is the option to view a graph of the data, discussed further below with **Figure 14.2.9a**.



```
8/4/11 3:49:18 0
8/4/11 3:49:19 3
8/4/11 3:49:20 5
8/4/11 3:49:21 7
8/4/11 3:49:22 9
8/4/11 3:49:23 11
8/4/11 3:49:24 13
8/4/11 3:49:25 15
8/4/11 3:49:26 18
8/4/11 3:49:27 20
8/4/11 3:49:28 23
8/4/11 3:49:29 25
8/4/11 3:49:30 24
8/4/11 3:49:31 23
8/4/11 3:49:32 23
8/4/11 3:49:33 23
8/4/11 3:49:34 23
8/4/11 3:49:35 20
8/4/11 3:49:36 18
8/4/11 3:49:37 16
8/4/11 3:49:38 15
8/4/11 3:49:39 15
8/4/11 3:49:40 15
8/4/11 3:49:41 15
```

Figure 14.2.8a – Log viewing screen

14.2.9 Log Graph Screen

The graph view seen in **Figure 14.2.9a** is a plot of the data contained in the log file the user is currently viewing. It creates large, readable graphs using all of the data in the log file. The graph requires a custom view to generate, which is explained in further detail in section 14.3.3. Because it's a non-moving static graph it's easier to generate than the dynamic graphs used in the OBD reader graphs. The top contains text stating the function being graphed and the time period in which it was recorded. The Y-axis uses the same range used in the OBD reader gauge view. There are options below the graph to return back to the text view of the log and to select a different log function. The log can contain a very large amount of data (mentioned previously, up to 10,000 log entries before being culled), so it can quickly become unwieldy to graph all of the data at once. This is especially true with data that is separated by long periods of time. The graphing function will only graph from the most recent set of data so that strange or broken looking graphs don't result from a long period of time passing between log data.

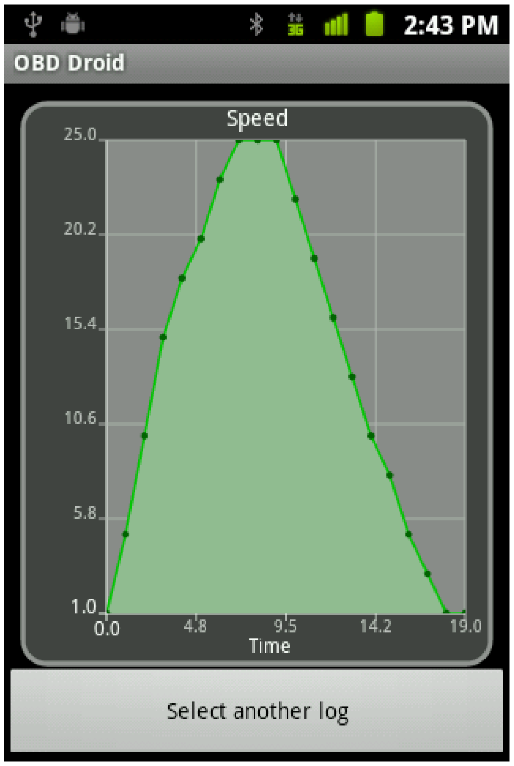
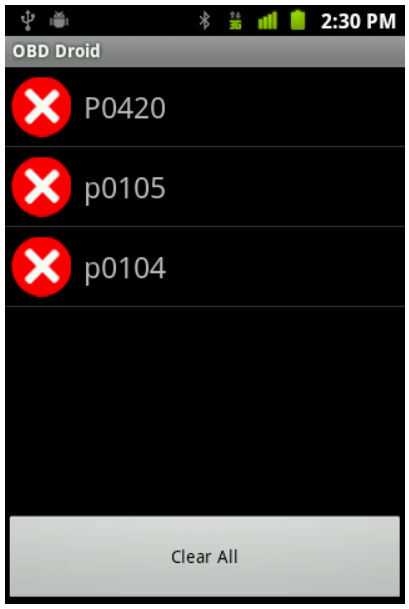


Figure 14.2.9a – Log Graph View

14.2.10 Error Code Screen

The error code screen is shown in **Figure 14.2.10a**. If an error light shows up on the user's car dashboard, they can check the specific fault code, known as a DTC (Diagnostic Trouble Code). They can also clear it if they so desire from this screen. In order for this screen to be useful, the user must first be connected to the device over Bluetooth. If there is no connection, an error message will appear and the user will be sent back to the main menu to try to manually connect to the device. Once connected, this screen displays a scrollable list of the names of all of the errors retrieved. If the user presses on any of the errors, a menu will pop up stating the specific error code (P0128 for "Coolant Thermostat Malfunction," for example), an option to send a message back to the OBD to clear that error – not recommended if you haven't actually fixed the problem – and a link to a website that brings up more specific information and research about that specific DTC code. Alternatively, the user can press the "Clear All" button on the bottom of the screen to clear every error from the OBD.



P0420

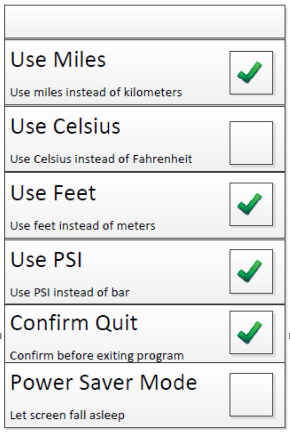
p0105

p0104

Figure 14.2.10a – Error code screen

14.2.11 Settings Screen

The settings screen, accessed from the Main Menu, is seen in **Figure 14.2.11a**. It keeps things familiar to regular Android users by again using the familiar list view and checkbox configuration of the stock Android OS settings screen. Our Settings screen contains several program-wide options. The user can choose metric or English units for several different types of measurements and readings taken from the OBD, including miles vs. kilometers, Celsius vs. Fahrenheit, feet vs. meters, and PSI vs. Bar. The user also has the choice of whether the program pops up a notification when quitting to make sure that they want to – enabled by default so the user doesn't accidentally quit the program. By default, the program forces the screen to always be turned on so the user can monitor whatever they want while they're driving. This behavior can be turned off by checking the "Power Saver Mode" checkbox. There is also an option for connecting automatically. Enabled by default, this will have the phone attempt to connect to the Bluetooth device whenever the program is first launched. If disabled, the user will have to manually Start Connection each time they want to connect.



Use Miles Use miles instead of kilometers	✓
Use Celsius Use Celsius instead of Fahrenheit	
Use Feet Use feet instead of meters	✓
Use PSI Use PSI instead of bar	✓
Confirm Quit Confirm before exiting program	✓
Power Saver Mode Let screen fall asleep	

Figure 14.2.11a – Settings screen

14.2.12 UI Block Diagram

Figure 14.2.12a is the overall block diagram of how the different UI screens are arranged and flow into each other. The program flow is designed to be relatively simple without having to delve deep into menus to get to the important functions. Most functions branch off of the main menu screen, with some ancillary functions available to the OBD II reader and Log screen. Each one of these boxes represents a separate Android "activity" comprised of multiple *Views* and *ViewGroups* (discussed more in Section 14.4).

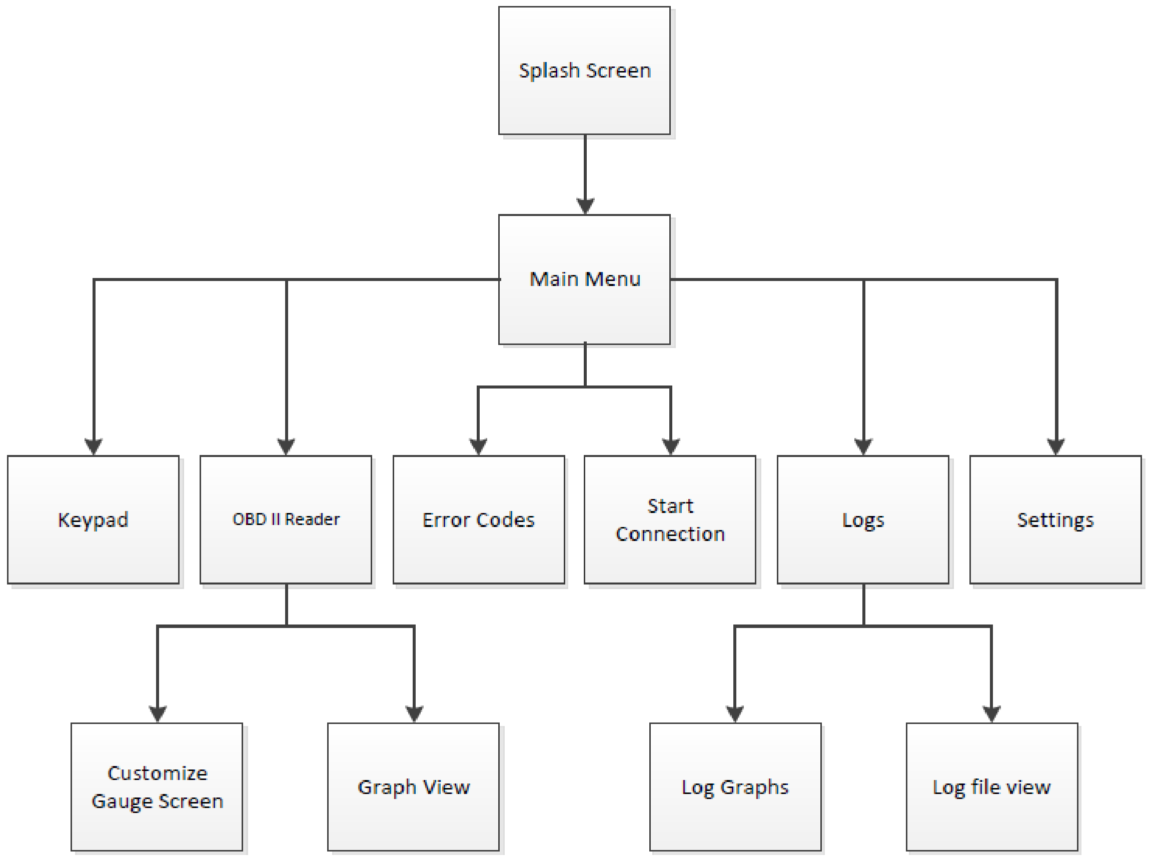


Figure 14.2.12a – UI Block Diagram

14.3 GUI Software Implementation

In the following sections we will describe how to implement the various Android GUI processes in the program. First, however, it's important to understand how GUIs are created in Android, and the choice that we have when designing them.

There are two major ways to declare a layout. You can declare the layout for a view as an XML file, which uses mostly standard XML vocabulary and syntax for creating classes and subclasses. The other method is to put the UI structure inside the program itself and instantiate the elements of your layout at runtime. While either choice is valid, XML is the preferred method to use when designing a UI in Android. This is because it allows the developer to keep the program code separate from the presentation, similar to how CSS controls the look and feel of a webpage while the HTML contains the actual content. Using XML lets the developer modify the UI without having to worry about creating new bugs or problems in the code. The XML file is created separately, and then it is simply called inside the activity with the *setContentView()* command. It also makes it easier and simpler to create new layout for different screen resolutions, vertical or horizontal screen orientations, and support for different languages. For the most part we will be using XML to create our GUI.

An Activity represents a single screen in Android. Our program has several interconnected activities that make up the core of the program. The activity's UI is itself is built from a hierarchy of *View* and *ViewGroup* objects. Views are the basic building blocks of UI creation, and the class includes subclasses like *widgets* that let the developer place GUI items such as text boxes and buttons. ViewGroups are more overarching classes and control the overall layout of an Activity. Views and ViewGroups form a tree – each ViewGroup is a branch that can contain either more ViewGroups or terminate with a View which acts as the leaf. *setContentView()* is called by the Activity and is passed the root of the tree, and each child draws itself, traversing down the hierarchy until everything is drawn in order. This helps to determine which views overlap and which order they go in.

Once the UI is built, it's important to be able to handle UI events, which are basically just some form of user input, whether it's a click, touch, trackball movement, etc. We first have to create an event listener and pair it with a View. Our program only accepts touch events, so we use *View.OnTouchListener()* TO detect touch events and interpret them appropriately. Also, because some of our activities will be custom classes that don't use the default views, we will have to override some of the callback methods to *View*, such as *onTouchEvent()*. This is only necessary when building custom components like gauges, and is not required for more standard *Views*.

Helpfully, menus – that is, the popup menus that appear when the physical menu button on the phone is pressed - are a special type of View and are handled separately. We simply call the *onCreateOptionsMenu()* method for the activities that use them – in our case, the OBD Reader screen and the Log viewing screen, and insert the menu items that we want. Android automatically handles their position in the View hierarchy and places them appropriately. There's also no need to create unique event handlers for menus – it's taken care of by *onOptionsItemSelected()* to easily control what happens when one of the menu items is pressed (switching to a new activity, deleting a log file, etc.).

14.3.1 Bluetooth Implementation

The Android phone's Bluetooth connection is the primary communication method for talking with our device, so it is very important that our program is able to quickly and efficiently connect. All of the APIs that we need are available in the *android.bluetooth* package. There are a few main classes required for creating Bluetooth connections and connecting to a device. The *BluetoothAdapter* class is required for any Bluetooth activity. It represents the adapter located on the phone itself, lets it instantiate new *BluetoothDevice* classes, and can create a *BluetoothServerSocket* to let it listen for other devices. The *BluetoothDevice* class is used for remote devices and can create a BluetoothSocket or request information from the remote device. *BluetoothSocket* lets two Bluetooth devices connect with a socket and exchange data. In order to create a connection, at least one device must open the *BluetoothServerSocket* class, which listens for incoming requests and returns a *BluetoothSocket* in order to connect. Finally, *BluetoothClass* is a read-only list of a device's properties, though it is not an exhaustive list.

The first thing you have to do before setting up a Bluetooth connection is setting the right permissions in the application manifest file: BLUETOOTH and BLUETOOTH_ADMIN. BLUETOOTH lets you request/accept connections and communicate over Bluetooth, but BLUETOOTH_ADMIN is needed to start device discovery and change Bluetooth settings. Our device will require the BLUETOOTH_ADMIN permission (which in turn also needs the lower BLUETOOTH permission) since our program will be modifying the Bluetooth settings. Next, before we can do anything the program first has to ensure that Bluetooth is turned on in the phone. Using the *BluetoothAdapter* class, the *isEnabled()* function is called to check its status. If it's disabled, we can call *startActivityForResult()* to pop up a dialog box (without leaving the program) asking to enable Bluetooth. RESULT_OK will be returned if Bluetooth was successfully enabled, and RESULT_CANCELED if not. If the user canceled the request he can enable it again from the main menu screen.

Once enabled, the Android phone must now be able to find our device, either using device discovery or by finding it in the list of previously paired devices. Our device will always be discoverable, so we don't have to worry about making the phone itself discoverable. This whole process is done (again) through the *BluetoothAdapter* class. Before doing the more intensive device discovery process, we should first check if our device is already in the list of paired devices. To do this we call *getBondedDevices()* which returns a list of paired devices. If our device is on there, we're good to go. If not, we have to search for it. Calling *startDiscovery()* begins this process. Our application must register a BroadcastReceiver for the ACTION_FOUND Intent so we can get information on each device found. All that's needed in order to start the connection is the MAC address of the device. Note that in Android the remote device *must* be paired before the two devices are allowed to connect.

In order for two devices to connect, both must hold a connected *BluetoothSocket* on the same RFCOMM (Radio Frequency COMMunication) channel. This can be done a few different ways, but in our program, we will be connecting as a client to the device. First we have to get a *BluetoothSocket* by calling *createRfcommSocketToServiceRecord(UUID)*. The UUID (Universally Unique Identifier) is a 128-bit string used to uniquely identify the program's Bluetooth service. Once the socket is created, we initiate the connection with *connect()*, where the remote device ensures the UUID is correct and accepts the connection. Because *connect()* is a blocking call, it should be run on a separate thread outside of the main Activity UI thread so it doesn't prevent any other interaction with the program. The program also needs to make sure that device discovery is not happening while trying to connect, since this slows down the connection and could cause it to fail.

Finally, once the two devices are paired and connected, the program has to manage the data connection and read/write over the Bluetooth connection. Using the *BluetoothSocket*, we handle Input and Output streams via *getInputStream()* and *getOutputStream()*. We read and write data to the streams using *read(byte[])* and *write(byte[])*. It is important to create a separate thread for stream reading and writing, since they are blocking methods. It is also important to close the connection once the program is closed, using the *cancel()* command on a *BluetoothSocket*.

14.3.2 Gauge Implementation

In order to create our main OBD reader screen, we need gauges that can show the status of the desired OBD value. In order to do this, we first have to create a custom view since there isn't any kind of "gauge view" by default built into Android, like how *TextView* or *ListView* are. To start, we first have to create a new class derived from the built-in class *View*. In order to make our gauges circles (instead of ovals), we must make the view square by overriding the *onMeasure()* method and forcing the width and height to be the same. This also lets us scale the gauges to an appropriate size. One convenient thing to do before delving into drawing gauges is to create a toolbox of sorts – basically just methods that bring together a few of the basic drawing tools into more useful ones. The methods are contained in a class called *initDrawingTools()*. These include methods such as *drawRim()* which draws a light circle inside of a dark circle to create an offset look, with a *LinearGradient* used to make it look more metallic. The method *drawFace()* uses a bitmap texture from the *BitmapShader* class to draw the background picture of the gauge for us. It's important to set *setFilterBitmap()* in the *Paint* instance to true so that it scales smoothly regardless of the resolution on the phone. Our custom method *drawScale()* is one of the most important. With the range of values hard-coded in for each function, the program determines an appropriate scale and number of notches for each OBD reader function. *Canvas.rotate()* is used to draw each notch and number on the gauge, and call *Canvas.restore()* afterward to set the canvas upright again. *DrawTitle()* is used with *Canvas.drawTextOnPath()* to draw the name of the function on the gauge. This gives us some leeway about how we draw the text, so it can be curved around the bottom if we want it to. This method also lets us change the text when the function that a gauge is displaying gets changed – everything (name, scale, etc.) should change appropriate with it and be redrawn. Finally, *drawHand()* is used to create the moving dial on our gauge. It's drawn using a solid color *Path* and using *Canvas.rotate()* to make it move in a circle the way a real dial does. To make the the dial move to match the data being received from the OBD reader, we set up a sensor event for the Bluetooth input stream and whenever it changes, we update the rotational position of the dial so that it matches the current value. Once this view is created, we can use it within the activities for the OBD Reader and OBD Reader Graph screens.

Optimization is important as well, especially since the hardware is relatively limited compared to a PC environment. We noticed while using the Android program Torque that the display could be choppy and laggy at times. We want our program to be as fast and efficient as possible, and able to be used on Android phones that aren't super high-end. One simple way to optimize UI drawing is to separate moving parts from non-moving parts. Moving parts would just be the dial itself – we don't plan to implement any other changing lights or colors into the gauges. Non-moving parts would include the rim, the notches and numbers around the edges, and the background of the gauge. Parts of the gauge that do not move can be drawn all together onto a single bitmap using *Canvas.drawBitmap()* in *onDraw()*. This makes it so only the moving parts are updated with each refresh, while the rest of it just sits in the background and doesn't have to be redrawn every single time. Instead of drawing to the "real" canvas we draw to the bitmap background. Then only after we've done that do we call the *draw()* command for the entire bitmap, and then the *draw()* command for the dial.. This trades some phone memory for an increase in performance. Limiting the number of gauges drawn to four, besides being good from a design/visibility standpoint, is also good for performance since it doesn't have to draw the updates to too many gauges for every refresh. We hope to keep the OBD reader interface feeling snappy and responsive while still being fully functional.

14.3.3 Graph Implementation

Support for graphing 2D plots is surprisingly lackluster on Android – there is no built-in support to create dynamic line graphs (to be used in the OBD Reader graph screen), or even static, unmoving line graphs such as those we are using to view graphs from the log screen. After researching the methods used to get around this, we found that there are basically two things usually done about this. One rather clever solution is to just bypass all of the native Android code and use a webview. Using a jQuery-based javascript library called “Flot”, a developer can basically create a local HTML webpage that contains the graph which is written in javascript. This webpage is then displayed in a webview inside the program. While clever and easy to implement, this wasn’t the solution for our program. For one thing, this method doesn’t support dynamic charts that update the data in real time, which would make it useless for us for the OBD graphs. Another issue with this method is performance – creating graphs on a webpage and then displaying it is inherently less efficient than directly creating the graph in code. The other commonly used method to get around Android’s graphing limitations is the use a third party graphing library. There are several available, but most of them either were unable to do dynamic charts (which are necessary for our project) or cost money. We finally found a third party API called AndroidPlot that was both free and supported dynamic chart creation, so we could use it for log screen graphing *and* for graphing the OBD values in real-time. We simply import the AndroidPlot libraries and use them like any other to create a custom GraphView that we can reuse multiple times throughout our program to fit our requirements.

Creating the static graphs for the logs isn’t too difficult. First, we have to edit /res/layout/main.xml to include an entry for the *XYPlot* view. Next we create the activity LogActivity.java and import the AndroidPlot libraries as well as the other standard Android libraries. We create the *onCreate()* method and *setContentView()* to main.xml inside it to set the view to be the plot we want. The plot is initialized using *findViewById()* using the name we gave it in the xml file as the parameter. This allows us to distinguish between graph definitions in the XML file, so if we were to have multiple graphs onscreen at once we would be able to keep them straight by their names. To actually retrieve the data used to populate the graph, we have to open the log text file and loop through it, collecting the data in an array for the y-value and collecting the timestamp data into a separate array for the x-value data. Once we have our two arrays, we use the *SimpleXYSeries* constructor to turn them into a graph-able list of numbers. *LineAndPointFormatter()* is used to format the colors of the lines and points on the graph – important, otherwise it would just be black on black and unreadable. We plan to use an easily distinguishable red-on-black color scheme for our graphs. The number lists are added to the plot with the *addSeries()* command. Finally, since the AndroidPlot library displays a developer debugging guide by default, we use the *disableAllMarkup* command to remove it so our graph is nice and clean. By the end of it all, we will have a screen that neatly displays the log data in graph form. Graphing real-time data from the OBD however, is a little more complicated.

We are reading real-time data from the OBD reader and displaying it in graph form whenever a user presses on a gauge in the OBD reader view. The procedure begins the same way as a static graph – adding an entry with an id for the plot in main.xml. Also similar to static plots, we have to set up and initialize the plots using the *addSeries* command, set the range (hard coded for each specific OBD function) with *setRangeBoundaries()* and the domain (30 points of data shown onscreen at once) with *setDomainBoundaries()*. For the data itself, we will create a custom event listener that parses data from the Bluetooth input stream and turns it into OBD codes. Every time a new sensor data is read, an event is created. A separate method, *onSensorChanged()* will handle what happens when a new event occurs. An array will read in the data and update the values currently held in the array. We call *removeFirst()* to get rid of the oldest sample in the history, then update the list with *setModel()*. Finally, the plot is redrawn with the aptly named *redraw()* command to include the new data. *onSensorChanged* is looped through and whenever it receives new data will update the array and redraw the graph. This process effectively creates a real-time dynamic graph of the incoming OBD data.

14.4 Activities

Discussed briefly in section 14.3, activities are the main building blocks of an Android program. Each activity represents a “screen” of our program, and the activities are linked to create a cohesive program. While running, an activity can be *Resumed* (in the foreground with user focus), *Paused* (in the background but still running/partially visible) or *Stopped* (completely obscured by another activity, can be killed to free memory of necessary). Managing the activity lifecycle is important. There are callback methods such as *onCreate()* for when the activity is just being started, *onResume()* for when an activity becomes visible again, and *onDestroy()* to control what to do when an activity is about to be destroyed (for example, save some data before the activity is destroyed). In our program, the user navigates back and forth frequently through the menus, so it’s important to specify what happens when an activity is paused or resumed or destroyed.

For the UI, each activity contains a view hierarchy. View objects are individual widgets that can be placed on the screen. Views include things like text boxes, images, buttons, checkboxes, etc. ViewGroups are overall layouts for a set of view objects. In the linear layout, every view object follows the next sequentially, either horizontally or vertically (specified in the main.xml file). Relative layout is useful when we need to place things spatially relative to one another. The list view layout provides a scrollable list of view objects – text in our case. There can be nested ViewGroups, which is often the case when using linear layout, though if the nesting goes too deep it's usually better to go with a relative layout instead. **Figure 14.4a** shows an activity diagram showing each of the activities or screens in the program, and the ViewGroups and Views that make up each one. The tree is traversed from top to bottom, left to right, and the order is important.

The SplashScreen activity uses the linear layout, arranging vertically a text view with the name of the program, an image view with a logo of our device, and another text view for our group number and other information. The main menu, though very important as a portal for the rest of the program, is relatively simple view-wise. It simply uses a list view with a set of TextViews representing each menu item. The Settings activity acts the same way: just a basic list view with text items. The checkboxes to turn on or off the options displayed are actually included as part of TextView. Keypad is an interesting design challenge. Getting the boxes displayed is relatively simple, but putting the big start button in the center of them all will be a challenge. The screen is essentially just several pictures with actions tied to them – the square sections are placed relative to each other – one below or under the other one, with the square representing to big Start button (stylized to look like a circle) must be the last view object to be placed so that it overlaps the other buttons and images.

The LogSelect activity is another simple list view – just a textual list of the logs that are available to be read. Branching off of this however are two more activities: LogView and LogGraph. LogView is pretty much just viewing a text file, with a "read more" button on the bottom. We use a vertical linear layout, with a text view first of the text read from the log, followed by a Button to activate the read more functionality and access twenty more logs. LogGraph is relatively tricky UI-wise. There is a base vertical linear layout. First in line is our custom-created view to show graphs from the log data, GraphView. The details of GraphView are discussed in section 14.3.3. Below our custom GraphView is yet another linear layout, this time horizontal. This ViewGroup contains two buttons, one for returning to the LogView activity, and one for returning to the LogSelect activity.

The Error Code activity, though it looks relatively simple, is also one of the more complex UI designs. The overall ViewGroup is a linear layout. Starting at the top is a TextView for the "Error Codes" title. Below that is another ViewGroup, a list view with a list of all of the error codes that are available to be displayed. This is accomplished, as usual, with text view. Back into the first linear layout, there's a Button object used for the "clear all" button at the bottom of the activity.

Finally, the all-important OBDReader activity uses a relative layout to easily place the four gauges symmetrically on the screen. The gauges themselves are created from another custom class, GaugeView, the implementation of which is discussed in detail in section 14.3.2. The customize gauge screen accessed when long-pressing on a gauge is actually not its own activity. It is a context menu that is “registered” to the gauge view with *registerForContextMenu()* and then “inflated” to populate the menu with items. The OBDGraph screen however *is* it’s own activity. It’s the only one that uses both of our custom-created views, GaugeView and GraphView. It uses a vertical linear layout with a single gauge on top and a dynamic graph below.

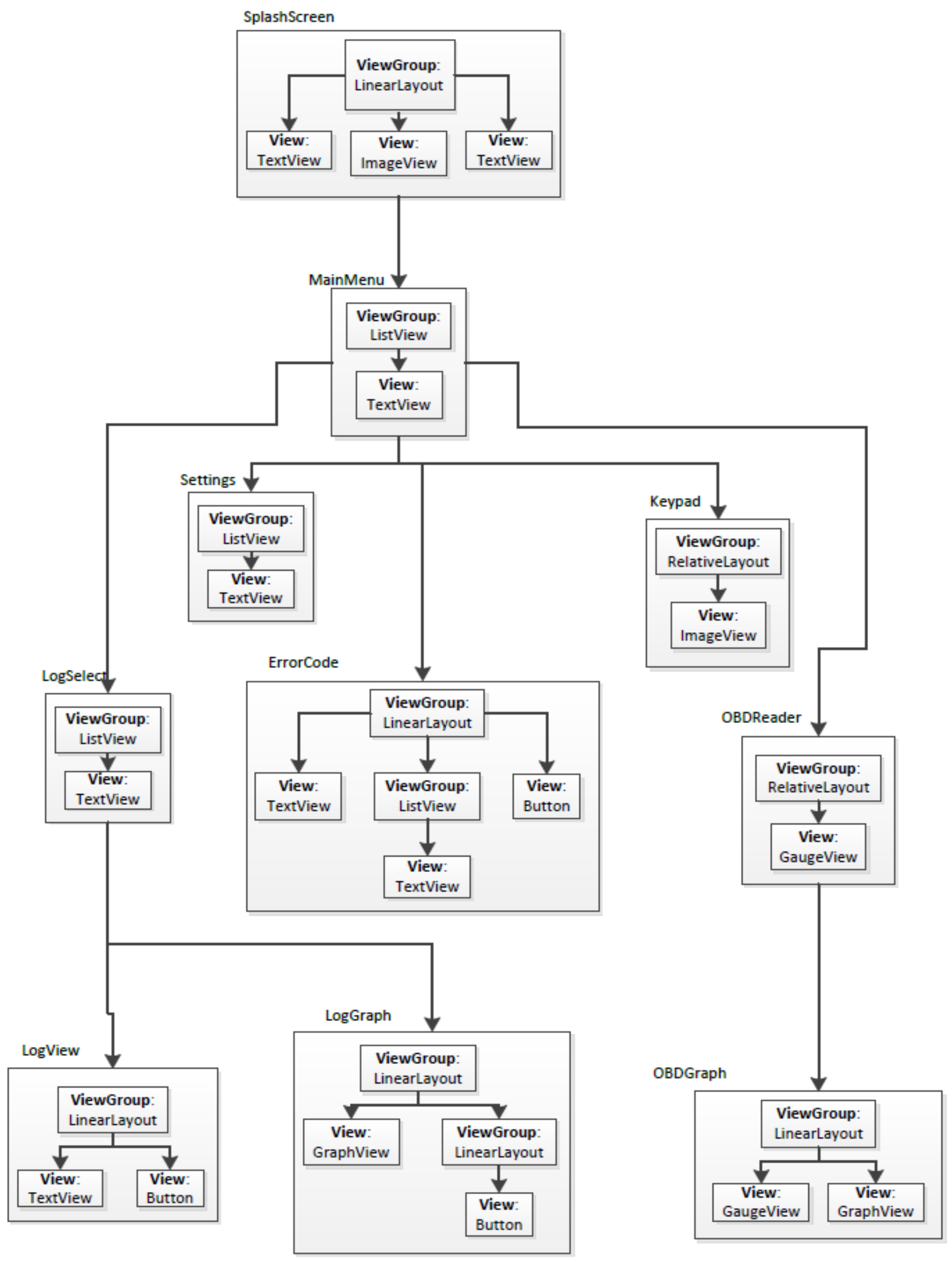


Figure 14.4a – Activity Diagram

15.0 Prototype Testing

Testing is a crucial part of any project, big or small. It's not uncommon that testing take the longest of any of the phases from start to finish. The testing phase will uncover problems of all sorts. The problems may range anywhere from design, logical or your common mistake. In a project like this, testing needs to be thought-out and efficient. We will test the hardware and software as individual components before interfacing or connecting them with another project component. As soon as two components are connected, we will ensure that their functionality is still valid as a unit. This "build-and-test" strategy will be used from the point that two components are connected, until the last component is added to the prototype. This form of testing may seem tedious and more time consuming in the beginning, but it will allow us to narrow the scope of problems when they occur. The smaller the scope of the problem is, the easier the problem is to find and fix. Different strategies will be used for both the hardware and software aspects, as well as the technology that interfaces the different systems together. The key strategy to effective testing is the "build-and-test" strategy we've come up with.

15.1 Hardware Test Environment

The ultimate test environment for the prototype will be in a 1998 Honda Accord. The testing location will take place in a parking lot or parking garage. Individual hardware parts of the prototype will be tested indoors in the electrical engineering lab at ideal temperatures and environments.

15.2 Hardware Specific Testing

Each piece of hardware needs to be tested for correct functionality before it can be used in the prototype. All hardware parts need to be tested, including the smart-phone and test vehicle. We cannot assume that the vehicle and smart-phone are functional.

15.2.1 Vehicle OBDII Port Testing

To ensure that the OBDII port of the 1998 Honda Accord is functional, we will test the port by inspection and electrical functionality. Once the OBDII port is located, we will inspect the port visually to make sure the sockets are not clogged with debris. We will also visually inspect the back of J1850 connector to make sure that all of the wires running from the contacts to their respective destinations are intact. Once the OBDII port passes a visual inspection, we can do additional testing with an OBDII reader. For multiple testing purposes we will purchase a commercial ELM327 OBDII reader. We will use the commercial OBDII reader to test the functionality of the ODBII port in the 1998 Honda Accord. Since we don't know for sure that the Android application written for this project works, we'll test the OBDII port with a free Android application that has similar functionality. By downloading the free version of "Torque" from the Android Market, we can test the OBDII port of the 1998 Honda Accord. A smart-phone and the ELM327 OBDII reader will connect via Bluetooth Technology. Once a connection has been made, the Torque application should be able to receive data from the car. We will perform every function that the free version of Torque will allow us to do. Once we can confirm that the OBDII port in the 1998 Honda Accord is functional, we can move forward with our testing.

15.2.2 Smart-Phone Testing

The smart-phone will need to be heavily tested for Bluetooth functionality. This first test will be simple on and off testing. We'll go to the settings of the phone and turn the Bluetooth on and off, while checking to see if the icon appears in the taskbar located at the top of the screen. Once the taskbar indicates that the Bluetooth is being turned on and off correctly, we can move to the test procedure.

The second test will require a laptop. The smart-phone will have to be in a mode named "discoverable." This mode allows for other Bluetooth enabled devices to search for the smart-phone for Bluetooth connection. While the smart-phone and laptop have their Bluetooth enabled and the laptop is discoverable by other devices, we will search for other Bluetooth enabled devices in the area from the smart-phone. If everything is working correctly, the laptop will appear in the list of Bluetooth enabled devices within the range of the smart-phone. At this point we can request to make a connection with the laptop. The smart-phone user will then have to enter an arbitrary four-digit password to start the connection. The laptop user will be prompted with a request to connect to the smart-phone via Bluetooth and asked to enter the arbitrary password that the smart-phone user entered to start the connection. The laptop user will enter the arbitrary password and attempt to send a random picture to the smart-phone. If the smart-phone successfully receives the picture, then we are certain that the smart-phone is capable of receiving data from another device via Bluetooth.

The last test that the smart-phone will endure, will be with a similar application to the one created in this project and a commercial Bluetooth enabled OBDII reader. We will download the free version of the application "Torque" and connect, via Bluetooth, to the ELM327 OBDII reader while it's plugged into the 1998 Honda Accord. To narrow the scope of the possible failures, the OBDII port on the 1998 Honda Accord will have to pass it's testing prior to attempting the last test for the smart-phone. If we did not test the OBDII port on the vehicle prior, a failed test could be a result of a nonfunctional OBDII port. The smart-phone user will open the Torque application and search for devices via Bluetooth. The commercial OBDII reader will then appear in the list of Bluetooth devices in the area. When the smart-phone user attempts to connect to the ELM327 OBDII reader, they will be prompted to enter a pass-code. The ELM327 OBDII reader has a built-in pass-code that the smart-phone user will have to enter to make a successful connection. The standard pass-code for the ELM327 OBDII reader is "1234." Once the Bluetooth connection is made, the smart-phone user will go through all of the possible functions on the free version of Torque to confirm the functionality of the smart-phone's Bluetooth capability. Pressing the accelerator on the vehicle should cause the RPM gauge in the free version of Torque to display the same measurement as the RPM gauge in the vehicle. We must note that there will be some time delay for the reading being displayed on the smart-phone due to the time it takes the Torque application to receive the necessary data. Once we can confirm that the free Torque application is displaying accurate data from the car, we can assure that the smart-phone's Bluetooth capabilities are functional.

15.3 Software Test Environment

Software will be tested several different ways and in different environments. The common syntax examining and testing will be done in the same environment that it will be written in, Eclipse. As individual functions are completed and added to the Android application, they'll be tested on the smart-phone. For consistent testing and accurate results, we will test each function on the 1998 Honda Accord with components that have already passed their respective tests. This testing strategy allows us to eliminate the possibility of hardware deficiencies and focus in on the software aspect when troubleshooting. A small amount of software testing, for the microcontroller, will take place in an Electrical Engineering lab at ideal temperatures and conditions.

15.3.1 Testbed

All testing will be run in both software (Eclipse) and hardware (phone) environments. Coding and testing will be done in Eclipse 2.6.2, with ADT (Android Development Tools) version 15.01. The test computer runs Windows 7 64-bit, and two separate AVDs (Android Virtual Device) running Android version 2.1 and 2.2 will be used to test software compatibility when hardware-specific things (Bluetooth, GPS, etc.) are not required. Hardware testing will be done on an LG Optimus T running Android version 2.2. We're targetting the application to require at least Android 2.1, since it is widely available on most Android devices, and included many changes to the underlying Android system since Android 1.6.

While we don't know of any specific function or feature available in 2.1 that's not available in previous versions, it makes testing and debugging easier since we don't have to worry about supporting deprecated features or using something that's not supported by the older version.

15.4 Software Specific Testing

Software testing is a critical aspect to finishing this project in a timely manner. A commonly and simplistic strategy is to test as you go. In a project this large, we actually have to apply this strategy that our professors have been pounding in our heads for the last four years.

15.4.1 Android Application Testing

The Android application will be examined and debugged at the code level by each of the group members. After a function or chunk of code has passed the syntax check done by Eclipse's compiler we can move onto the next step of testing. Since a compiler like the one in Eclipse cannot check code for anything besides correct syntax, we must examine the code as a group to check for logical errors. In typical homework programs from programming classes, we can bypass logical examinations because we can base the program's functionality upon a correct output response from the user's input. In our Android application, this testing shortcut isn't an option because the user environment is not the same place that the source code is written. Every time that a function or chunk of code needs to be tested for functionality, the application needs to be loaded to the smart-phone and then tested using the vehicle commercial OBDII reader. Testing the application for correct functionality just got much more time consuming with respect to typical homework-assigned programs. Another issue is that the input for our application is being sent from the vehicle and not from a user. This implies that there's only one test case coming from the user, while typical programs have many different test cases. Once our group examines the code for simplistic logical errors, we can agree that it's worth the time to setup the vehicle to test the application for correct functionality. The code will be compiled an additional time in Android's platform, Android Virtual Device (AVD). We do not suspect that the compiler in AVD would find any errors since the code was already compiled in Eclipse. AVD's purpose is to load the code, in application form, onto the smart-phone. After AVD indicated that the application was successfully loaded to the smart-phone, we can disconnect the smart-phone and attempt to use the application. The smart-phone user can quickly inspect the application's GUI to make sure that the prior content was not effected by the update and that the recently added content is the desired appearance. Not only do we need to make sure that the function recently added works, but we need to make sure that the prior existing functions still perform properly. The amount of time needed to thoroughly test each function as it's added to the Android application, makes the premature, logical inspection a critical step in completing the project in a timely manner. After inspecting the GUI of the entire application, we can move forward with the actual functionality testing on the 1998 Honda Accord with the commercially purchased OBDII reader.

15.4.1.1 Bluetooth Connectivity Testing

Once the basic GUI and Bluetooth functionality is done at the coding level, we can load the application to the smart-phone for it's first functional test. At this point, all that we want to test is that the application will successfully connect to another wireless device via Bluetooth. Since the OBDII reader doesn't have a screen to easily indicate a Bluetooth connection has been made, we will test the application's Bluetooth functionality using a laptop and another smart-phone. First we will test the application by entering an exiting all of the GUI screens multiple times. This test is to check for bugs that would cause the application to crash while doing simple navigation within the application itself. Once the application is stable we can try to make a Bluetooth connection with another device. After both devices have their Bluetooth capabilities enabled, the person testing the application will search for discoverable Bluetooth enabled devices in range by selecting the Bluetooth search function. When a list of Bluetooth enabled devices appear on the screen, we can proceed to the next step of making a successful Bluetooth connection. The user would then select the desired device for connectivity and enter a password if necessary. We can then check the Bluetooth connection of the laptop that the application is attempting to connect with to see if the connection was successful. This check can be made by going into the Bluetooth properties of any typical laptop. After we ca confirm one successful Bluetooth connection, we need to test the Bluetooth capability of the application on at least two other devices, while connecting to each device a minimum of five times. The excessive connecting and disconnecting to multiple devices should uncover any issues with our applications Bluetooth capabilities.

15.4.1.2 RPM Function Testing

The RPM function will be the first function implemented into the application for a few different reasons, including testing purposes. First of all, the function at the code level isn't too complex and it doesn't rely on other functions to operate. Since the RPM function doesn't rely on other functions to operate, the testing is simple. We will load the application onto the smart-phone using AVD. Then we can connect the application with the commercial OBDII reader via Bluetooth. When a successful connection is established, the smart-phone user will then navigate to the RPM function. Upon arrival of the RPM function, the smart-phone user will examine the reading of the RPM's on the RPM gauge in the GUI and compare that value to the value being display on the RPM gauge of the test vehicle. The user will have to use their best judgment to decide if the RPM readings are accurate. The user will need to understand that there’s a time delay in the readings that are displayed on the application because of the time it takes to get the reading from the ECU and ultimately send it to the smart-phone. Although we cannot guarantee the readings on the application are ideal, we can get a great deal of confidence through specific testing tactics. One group member will vary the throttle position between high and low while another group member examines the application RPM gauge and RPM gauge of the vehicle. The application RPM gauge should behave in the same manner as the RPM gauge on the vehicle, but with a small time delay. The same low and high RPM’s should be achieved while keeping a similar motion. Although this test is purely by inspection, it deems to be accurate.

15.4.1.3 Logging Function Testing

Every function in the application will be tested for accuracy, but may require different forms of testing. The second function we plan to implement is what we call the “Logging Function.” This function records the data that is sent to the smart-phone from the OBDII reader in a data log. The data is necessary for both the car enthusiast that would use this application and to help the developers test other functions within the application. First we need to make sure that the logging function works correctly. Each piece of data that is recorded will have a time stamp attached to it. The time stamp will play a key factor in testing the logging function for accuracy. Once the logging function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the RPM function we implemented earlier, to help us test the logging function. When testing the logging function, we will do multiple trials to see how accurate the logging really is. Once we have the smart-phone successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader, we can start the series of testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Document the time the testing begins.
3. Select the RPM function on the Android application.
4. Have the vehicle operator press the accelerator while analyzing the RPM reading on the vehicle.
5. Vehicle operator shall document the maximum RPM reading reached to the greatest degree of accuracy.
6. The smart-phone operator shall analyze and document the maximum RPM reading reached on the android application.

7. The smart-phone operator shall then select the logging function and record the highest RPM reading that was logged along with the associated time stamp in the logging function.
8. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there should be five pieces of test data to help decide if the logging function is functioning properly. The figure below, figure 15.4.1.3a, will be filled out to help analyze the data with more accuracy.

Logging Function Testing					
Trial Number	Actual Time	Logging Time	Vehicle RPM Reading	RPM Application Reading	Logging Function Reading
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Figure 15.4.1.3a – Table used to test Logging Function

After all ten trials are recorded we can analyze the data to see how accurate the time stamp is. The far left column represents what trial number the data is associated with. The second column, “Actual Time”, represents the time manually recorded by the tester. The third column, “Logging Time”, represents the time stamp in the logging function’s log, when the highest RPM reading was recorded. The fourth column, “Vehicle RPM Reading”, represents the highest

RPM reading read from the vehicle's in-dash odometer by the tester. The fifth column, "RPM Application Reading", is the highest reading read from the GUI of the application's RPM function by the tester. Since multiple readings have to be read at the same time, this test will take a minimum of two people. A third tester to try to collect both, the vehicle RPM reading and the RPM application reading, is preferred, but not mandatory. The third tester would allow for debate among the collected readings and ultimately help collect better test data. The last column, "Logging Function Reading", is the highest RPM that was logged in the logging function during that specific trial. Since the testing will take a few seconds alone, we will not be looking for the time stamp to be incredibly accurate. We expect the times logged and actual times to be within a minute of each other. If our data indicates a difference of a minute or greater, we will do further testing with emphasis on the logging time stamp. The main concern with the logging function is that it is logging accurate readings from the car. All three of the RPM data recordings need to be within 300 RPM's of each other. If the RPM readings deviate much further than 300 RPM's of each other, we will need to analyze and re-test the function.

15.4.1.4 Fuel Level Function Testing

The third function that will be implemented is the fuel level function. This function will read the amount of fuel that is left in the tank of the vehicle. After completing the fuel level function code, we will implement and test the completed code for the application at the code level. We will ensure that all of the code compiles without any errors or warnings. Once the code successfully compiles, we will analyze the code for logical mistakes. When the code has passed visual inspection, it's time to upload the partially completed application to the smart-phone for field-testing. For testing purposes, Firoz Umran can confirm that the fuel level odometer in his 1998 Honda Accord is an accurate representation of the amount of fuel left in the gas tank. We will keep a standard field test setup for testing each of the functions. The standard procedure will include using the commercially bought ELM327 OBDII reader, the smart-phone and the 1998 Honda Accord. Once a successful connection between the smart-phone and the OBDII reader has been established, the fuel level function will be ready for field-testing. To help ensure the degree of accuracy of our application, we will test the fuel level function ten different times. At the beginning of each trial, the tester will clear the data log in the logging function and record the time the testing is taking place. The smart-phone user will select the fuel level function in the GUI menu and record the reading of the fuel level according to the GUI on the application screen. The tester will then record the fuel level that is displayed on the odometer in the 1998 Honda Accord. The smart-phone user will then exit the fuel level function and select the logging function. The tester shall analyze the logged data within the logging function and record the time stamp and fuel level indicated. The steps below shall be followed for each trial, for ten trials.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Analyze and record the time testing begins.
3. Select the Fuel Level Function on the Android application menu.
4. The smart-phone user shall record the fuel level measurement indicated on the GUI of the smart-phone to the best accuracy possible.
5. The vehicle operator shall record the fuel level measurement displayed on the in-dash gauge to the greatest degree of accuracy.
6. The smart-phone operator shall then select the logging function and record the fuel level measurement that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

The field-testing of the fuel level function will take more time than most of the other functions. For each of the trials' data to have value, the fuel level needs to vary from trial to trial. If time allows, we will do one trial per day for ten days. The time between trials allows for the 1998 Honda Accord to be driven, which consumes fuel and ultimately changes the fuel level for the following trial. A backup plan for quicker testing has been created in the event that time is scarce and the testing needs to be done promptly.
The faster testing method that allows for the trials to be done consecutively will require the vehicle's fuel level for the first trial to be between empty and an eighth of a tank. If the fuel level is any higher than an eighth of a tank, we will need to siphon the fuel into five-gallon gasoline containers. Once the fuel level is an eighth of a tank or less, the first trial is ready to begin. If the fuel level is already in the desired range, we will need about ten gallons of fuel in external gasoline containers. After each trial, about a gallon of gasoline will be added to the vehicle's fuel tank for the following trial. The addition of fuel after each trial should alter the fuel level enough to notice a difference in the data from trial to trial.

After each test trial, there should be five pieces of test data to help decide if the fuel level function is functioning properly. The figure below, figure 15.4.1.4a, will be filled out to help analyze the data with more accuracy.

Fuel Level Function Testing					
Trial Number	Actual Time	Logging Time	Vehicle Fuel Reading	Application Fuel Reading	Logging Fuel Reading

1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Figure 15.4.1.4a – Table used to test Fuel Level Function

After all ten trials have been recorded we must analyze the data to understand the accuracy of the fuel level function. Although, the time stamp from the logging function was specifically tested and analyzed during the Logging Function testing, it is possible that the addition of the fuel level function altered the functionality of the logging function, therefore we cannot take the time stamp reading for granted. The far left column represents what trial number the data is associated with. The second column, “Actual Time”, represents the time manually recorded by the tester. The third column, “Logging Time”, represents the time stamp in the logging function’s log, when the fuel level reading was recorded. The fourth column, “Vehicle Fuel Reading”, represents the fuel level reading read by the tester, from the vehicle’s in-dash gauge. The fifth column, “Fuel Level Application Reading”, is the fuel level reading read from the GUI of the application’s Fuel Level function, by the tester. The last column, “Logging Function Reading”, is the fuel level that was logged in the logging function during that specific trial.

As a group we will analyze the data gathered over all ten trials. Since the time stamp in the logging function was previously tested, we will look for consistency among all time stamps from all test trials, but not emphasize it as our main concern. We expect the times logged and actual times to be within a minute of each other. If our data indicates a difference of a minute or greater, we will do further testing with emphasis on the logging time stamp. The greatest concern is that the fuel level, for all three readings, are within a reasonable margin of error.

We would like the fuel level readings to be within a sixteenth of a tank from one another. If the fuel level readings deviate much further than a sixteenth of a tank from one another, we will have to decide if the readings are inaccurate or if the margin of error is larger than anticipated. In vehicles with relatively small fuel tanks, such as the 1998 Honda Accord, it's more difficult to get an accurate reading from the fuel gauge.

15.4.1.5 Time Since Engine Start Function Testing

The fourth function that will be implemented is the Time Since Engine Start (TSES) function. This function will read the last time that the engine in the 1998 Honda Accord was started. After completing the TSES function code, we will implement and test the completed code for the application at the code level. We will ensure that all of the code compiles without any errors or warnings. Once the code successfully compiles, we will analyze the code for logical mistakes. When the code has passed visual inspection, it's time to upload the partially completed application to the smart-phone for field-testing. The standard field test procedure will include using the commercially bought ELM327 OBDII reader, the smart-phone and the 1998 Honda Accord. Once a successful connection between the smart-phone and the OBDII reader has been established, the TSES function will be ready for field-testing. To help ensure the degree of accuracy of our application, we will test the TSES function ten different times. At the beginning of each trial, the tester will clear the data log in the logging function and record the time the testing is taking place. The tester will begin the test by starting the vehicle and recording the time that the engine started. After a successful start and recorded time, the engine will be shut off. The smart-phone user will select the TSES function in the GUI menu and record the time of that the engine was started last, according to the GUI on the application screen. The smart-phone user must also record the time that the TSES was recorded to ensure that the logging function's time stamp is still accurate. The smart-phone user will then exit the TSES function and select the logging function. The tester shall analyze the logged data within the logging function and record the time that the engine was last started according to the log along with it's time stamp. It's important to understand that the difference between the time stamp and the time that the engine was last started. The time stamp indicates when the data was sent from the car to the smart-phone, while the time next to it is the actual data that was requested, the TSES. The steps below shall be followed for each trial, for ten trials.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Start the vehicle while recording start time, then turn vehicle off.
3. Select the Time Since Engine Start Function on the Android application menu.

4. The smart-phone user shall record the TSES indicated on the GUI of the smart-phone and the time the reading occurred.
5. The smart-phone operator shall then select the logging function and record the TSES that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

The tester shall allow anywhere from a few minutes to a few hours between trials to vary the test data. Random test data will allow for a higher degree of confidence in the results. After each test trial, there will be five pieces of test data to help decide if the TSES function is functioning properly. The figure below, figure 15.4.1.5a, will be filled out to help analyze the data with more accuracy.

Time Since Engine Start Function Testing					
Trial Number	Actual Reading Time	Logging Time Stamp	Actual TSES	Application TSES Reading	Logging TSES Reading
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Figure 15.4.1.5a – Table used to test Time Since Engine Start Function

After all ten trials have been recorded we must analyze the data to understand the accuracy of the TSES function. Although, the time stamp from the logging function was specifically tested and analyzed during the Logging Function testing, it is possible that the addition of the TSES function altered the functionality of the logging function, therefore we cannot take the time stamp reading for granted. The far left column represents what trial number the data is associated with. The second column, “Actual Reading Time”, represents the time that the tester read the reading for the TSES from the application’s TSES function itself. The third column, “Logging Time Stamp”, represents the time stamp in the logging function’s log, when the TSES reading was sent to the smart-phone. The fourth column, “Actual TSES Reading”, represents the TSES calculation. This calculation is simply the difference in time between the time the tester started the engine to the time the tester uses the TSES function on the smart-phone. This reading is accurate and will be used as the correct amount of time that the engine was last started. The fifth column, “Application TSES Reading”, is the TSES reading read from the GUI of the application’s TSES function, by the tester. The last column, “Logging TSES Reading”, is the TSES that was logged in the logging function during that specific trial.

As a group we shall analyze the data gathered over all ten trials. Since the time stamp in the logging function was previously tested, we will look for consistency among all time stamps from all test trials, but not emphasize it as our main concern. We expect the times logged and actual times to be within a minute of each other. If our data indicates a difference of a minute or greater, we will do further testing with emphasis on the logging time stamp. The greatest concern is that the TSES, for all three readings, are accurate and within a reasonable margin of error. As testers, we understand that the calculated TSES should be the most accurate reading, therefore our goal is achieve readings that are within 45 seconds of the calculated TSES. If the test results deviate significantly from our expectations, we shall analyze the data further to determine the outcome.

15.4.1.6 Fuel Economy Function Testing

The fifth function that will be added to the application is the Fuel Economy function. This function tells the smart-phone user their fuel economy, average fuel economy and how many miles until the fuel tank is empty. Once the fuel economy function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the fuel economy function. When testing the fuel economy function, we will do multiple trials to understand the accuracy of the user readings. The fuel economy function will be tested under the assumption that the logging function is accurate. Only one test trial per day shall be performed to help vary test results. For each trial the smart-phone will have to be successfully connected to the vehicle via a

Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Fuel Economy Function in the application menu and read and record the three fuel economy readings supplied. The time of the readings shall also be recorded.
3. Smart-phone user shall exit the Fuel Economy Function and navigate to the logging function.
4. The smart-phone operator shall then select the logging function and record the fuel economy data that was logged along with the associated time stamp in the logging function.
5. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be eight pieces of test data to help decide if the fuel economy function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the fuel economy function is. The purpose of recording the time that the tester performed the fuel economy reading from the GUI, is to ensure that the implementation of the new function to the already existing android application, did not alter the functionality of the logging function. The fuel economy readings read from the GUI will be checked against the data that was logged in the logging function. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application. The fuel economy readings will be inspected by Firoz Umran because of his familiarity with the vehicle.

15.4.1.7 Speed Function Testing

The sixth function that will be added to the application is the Speed function. This function tells the smart-phone user their speed or average speed or a length of time. Once the speed function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the speed function. When testing the speed function, we will do multiple trials to understand the accuracy of the user readings. The speed function will be tested under the assumption that the logging function is accurate. Each testing trial will require at least three testers for best results. One tester is needed to operate the vehicle, one to operate the smart-phone and another to observe the vehicle odometers. For each trial the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Speed Function in the application menu.
3. The vehicle driver will set the trip on the dashboard then proceed to drive for about three minutes (just enough time to allow for average speed calculations).
4. While the vehicle is in motion: The smart-phone user will compare the speed indicated on the GUI and the speed indicated on the odometer in the dashboard of the vehicle. Note: The speed on the android application will have a time delay, with respect to the vehicle's odometer, due to the time taken to send data to the smart-phone. The observer shall time the length of the trip with a stopwatch.
5. After about three minutes of driving the test is over. The smart-phone user shall exit the Speed Function and navigate to the logging function.
6. The smart-phone operator shall then select the logging function and record the Speed data that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be eight pieces of test data to help decide if the speed function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the speed function is. The purpose of recording the time that the tester performed the speed and average speed reading from the GUI, is to ensure that the implementation of the new function to the already existing android application, did not alter the functionality of the logging function. The speed data read from the GUI will be checked against the data that was logged in the logging function and the observed speed from the odometers in the vehicle. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.8 Coolant Temperature Function Testing

The seventh function that will be added to the application is the Coolant Temperature function. This function tells the smart-phone user the temperature of their coolant in real time. Once the coolant temperature function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the coolant temperature function. When testing the coolant temperature function, we will do multiple trials to understand the accuracy of the user readings. The coolant temperature function will be tested under the assumption that the logging function is accurate. For each trial the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Coolant Temperature Function in the application menu.
3. The vehicle operator will start the car and leave it in park while observing the temperature gauge.
4. A third tester shall record the time and temperature from the in-dash gauge and the application’s GUI every 30 seconds.
5. After four minutes or eight readings have been recorded, enough test data will have been gathered for one trial. The smart-phone user shall exit the Coolant Temperature Function and navigate to the logging function. The vehicle can also be shut off.
6. The smart-phone operator shall then select the logging function and record the Coolant Temperature data that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be several pieces of test data to help decide if the Coolant Temperature function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the Coolant Temperature function is. The purpose of recording the time that the tester recorded the coolant temperature from the GUI and the in-dash odometer is so the logged data in the logging function will be associated with the correct set of data. The Coolant Temperature data read from the GUI will be checked against the data that was logged in the logging function and the observed coolant temperature from the odometer in the vehicle. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.9 Air Intake Temperature Function Testing

The eighth function that will be added to the application is the Air Intake Temperature function. This function tells the smart-phone user the temperature of their air intake in real time. Once the air intake temperature function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the air intake temperature function. When testing the air intake temperature function, we will do multiple trials to understand the accuracy of the user readings. The air intake temperature function will be tested under the assumption that the logging function is accurate. For each trial the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Air Intake Temperature Function in the application menu.
3. The vehicle operator will start the car drive around for four minutes.
4. A third tester shall record the time and temperature from the application's GUI every 30 seconds.
5. After four minutes or eight readings have been recorded, enough test data will have been gathered for one trial. The smart-phone user shall exit the Air Intake Temperature Function and navigate to the logging function. The vehicle can also be shut off after all the data has been recorded.
6. The smart-phone operator shall then select the logging function and record the Air Intake Temperature data that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be several pieces of test data to help decide if the Air Intake Temperature function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the Air Intake Temperature function is. The purpose of recording the time that the tester recorded the Air Intake temperature from the GUI is so the logged data in the logging function will be associated with the correct set of data. The Air Intake Temperature data read from the GUI will be checked against the data that was logged in the logging function. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.10 Timing Advance Function Testing

The ninth function that will be added to the application is the Timing Advance function. This function tells the smart-phone user the timing advance of the engine. Once the Timing Advance function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Timing Advance function. The Timing Advance function will be tested under the assumption that the logging function is accurate. Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Timing Advance Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.
2. Smart-phone tester shall select the Timing Advance Function in the application menu.
3. The displayed data on the application's GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Timing Advance Function.
5. The smart-phone operator shall then select the logging function and record the Timing Advance data that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the data read from the GUI of the application's Timing Advance Function, we will use another application such as Torque to generate a third Timing Advance reading. The Timing Advance data read from the GUI will be checked against the data that was logged in the logging function and the data recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.11 Mass Air Flow Function Testing

The tenth function that will be added to the application is the Mass Air Flow function. This function tells the smart-phone user the Mass Air Flow of the engine. Once the Mass Air Flow function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Mass Air Flow function. The Mass Air Flow function will be tested under the assumption that the logging function is accurate.

Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Mass Air Flow Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.
2. Smart-phone tester shall select the Mass Air Flow Function in the application menu.
3. The displayed data on the application’s GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Mass Air Flow Function.
5. The smart-phone operator shall then select the logging function and record the Mass Air Flow data that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the data read from the GUI of the application’s Mass Air Flow Function, we will use another application such as Torque to generate a third Mass Air Flow reading. The Mass Air Flow data read from the GUI will be checked against the data that was logged in the logging function and the data recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.12 Intake Manifold Pressure Function Testing

The eleventh function that will be added to the application is the Intake Manifold Pressure function. This function tells the smart-phone user the pressure on their intake manifold in real time. Once the Intake Manifold Pressure function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Intake Manifold Pressure function. When testing the Intake Manifold Pressure function, we will do multiple trials to understand the accuracy of the user readings. The Intake Manifold Pressure function will be tested under the assumption that the logging function is accurate. For each trial the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Intake Manifold Pressure Function in the application menu.
3. The vehicle operator will start the car drive around for four minutes.
4. A third tester shall record the time and pressure from the application's GUI every 30 seconds.
5. After four minutes or eight readings have been recorded, enough test data will have been gathered for one trial. The smart-phone user shall exit the Intake Manifold Pressure Function and navigate to the logging function. The vehicle can also be shut off after all the data has been recorded.
6. The smart-phone operator shall then select the logging function and record the Intake Manifold Pressure data that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be several pieces of test data to help decide if the Intake Manifold Pressure function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the Intake Manifold Pressure function is. The purpose of recording the time that the tester recorded the Intake Manifold Pressure from the GUI is so the logged data in the logging function will be associated with the correct set of data. The Intake Manifold Pressure data read from the GUI will be checked against the data that was logged in the logging function. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.13 Fuel Pressure Function Testing

The twelfth function that will be added to the application is the Fuel Pressure function. This function tells the smart-phone user their fuel pressure in real time. Once the Fuel Pressure function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Fuel Pressure function. When testing the Fuel Pressure function, we will do multiple trials to understand the accuracy of the user readings. The Fuel Pressure function will be tested under the assumption that the logging function is accurate. For each trial the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The testing will follow the following steps and be repeated ten times.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to each testing trial.
2. Smart-phone tester shall select the Fuel Pressure Function in the application menu.
3. The vehicle operator will start the car drive around for four minutes.
4. A third tester shall record the time and pressure from the application's GUI every 30 seconds.
5. After four minutes or eight readings have been recorded, enough test data will have been gathered for one trial. The smart-phone user shall exit the Fuel Pressure Function and navigate to the logging function. The vehicle can also be shut off after all the data has been recorded.
6. The smart-phone operator shall then select the logging function and record the Fuel Pressure data that was logged along with the associated time stamp in the logging function.
7. The smart-phone user shall clear the contents of the data log in the logging function and repeat all the steps above, nine more times.

After each test trial, there will be several pieces of test data to help decide if the Fuel Pressure function is functioning properly. After all ten trials are recorded we can analyze the data to see how accurate the Fuel Pressure function is. The purpose of recording the time that the tester recorded the Fuel Pressure from the GUI is so the logged data in the logging function will be associated with the correct set of data. The Fuel Pressure data read from the GUI will be checked against the data that was logged in the logging function. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.14 Engine Load Function Testing

The thirteenth function that will be added to the application is the Engine Load function. This function tells the smart-phone user the Engine Load on the engine. Once the Engine Load function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Engine Load function. The Engine Load function will be tested under the assumption that the logging function is accurate. Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Engine Load Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.

2. Smart-phone tester shall select the Engine Load Function in the application menu.
3. The displayed data on the application's GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Engine Load Function.
5. The smart-phone operator shall then select the logging function and record the Engine Load data that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the data read from the GUI of the application's Engine Load Function, we will use another application such as Torque to generate a third Engine Load reading. The Engine Load data read from the GUI will be checked against the data that was logged in the logging function and the data recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.15 Battery Voltage Function Testing

The fourteenth function that will be added to the application is the Battery Voltage function. This function tells the smart-phone user the voltage of the battery. Once the Battery Voltage function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Battery Voltage function. The Battery Voltage function will be tested under the assumption that the logging function is accurate. Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Battery Voltage Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.
2. Smart-phone tester shall select the Battery Voltage Function in the application menu.
3. The displayed data on the application's GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Battery Voltage Function.
5. The smart-phone operator shall then select the logging function and record the Battery Voltage data that was logged along with the associated time stamp in the logging function.

6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the data read from the GUI of the application's Battery Voltage Function, we will use another application such as Torque to generate a third Battery Voltage reading. The Battery Voltage data read from the GUI will be checked against the data that was logged in the logging function and the data recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.16 Error Code Function Testing

The fifteenth function that will be added to the application is the Error Code function. This function tells the smart-phone user the error code associated with a check engine light. Once the Error Code function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Error Code function. The Error Code function will be tested under the assumption that the logging function is accurate. Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Error Code Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.
2. Smart-phone tester shall select the Error Code Function in the application menu.
3. The displayed error code on the application's GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Error Code Function.
5. The smart-phone operator shall then select the logging function and record the error code that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the error code read from the GUI of the application's Error Code Function, we will use another application such as Torque to generate a third Error Code reading. The Error Code read from the GUI will be checked against the error codes that were logged in the logging function and the error codes recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.1.17 Throttle Position Function Testing

The fourteenth function that will be added to the application is the Throttle Position function. This function tells the smart-phone user the position of the throttle. Once the Throttle Position function compiles without any errors or warnings, we can inspect the code for logic mistakes. After the code has been inspected for logic mistakes, we can load the application onto the smart-phone for the final stage of testing. We will use the logging function that was implemented earlier, to help test the Throttle Position function. The Throttle Position function will be tested under the assumption that the logging function is accurate. Prior to testing, the smart-phone will have to be successfully connected to the vehicle via a Bluetooth connection with the commercial OBDII reader before we can start testing. The following steps will be performed to test the functionality of the Throttle Position Function.

1. The smart-phone user shall ensure that the data log in the logging function is cleared prior to testing.
2. Smart-phone tester shall select the Throttle Position Function in the application menu.
3. The displayed data on the application's GUI shall be recorded along with the time the data was read.
4. The smart-phone user shall exit the Throttle Position Function.
5. The smart-phone operator shall then select the logging function and record the Throttle Position data that was logged along with the associated time stamp in the logging function.
6. The smart-phone user shall clear the contents of the data log in the logging function.

To test the accuracy of the data read from the GUI of the application's Throttle Position Function, we will use another application such as Torque to generate a third Throttle Position reading. The Throttle Position data read from the GUI will be checked against the data that was logged in the logging function and the data recorded from another application such as Torque. By this point in the testing phase, we anticipate that the logging function will be completely accurate since it was one of the first functions added to the application.

15.4.2 Microcontroller Testing

After the microcontroller has been programmed on the test board, we will build a circuit on a breadboard to test the functionality. LED lights will be used to indicate a successful signal transmission from the microcontroller. The tester shall go through all possible signal configurations using the LED lights. Once the microcontroller has been successfully tested, we can implement it with the rest of the hardware into the PCB board. After the PCB board has been built, it can take the place of the commercial ODBII reader. The microcontroller shall then be tested using the application itself.

15.4.3 GUI Testing

OBD gauge testing: connect the device and access the OBD reader screen. By default it should contain four gauges for reading speed, acceleration, boost and throttle. Ensure that each gauge can be long-pressed to pull up the customization dialog. Test each gauge with every function to ensure that not only all of the functions work, but that there are no positional bugs depending on which gauge is used. There should be no conflict with setting all of the gauges to read the same data. The gauges should also be checked to make sure the markings and numbers on the scale are correct for each function, and don't run into any overlapping or drawing conflicts when the function changes.

Graph testing: connect the device, access the OBD reader screen and single press each gauge. It should launch the graph view for the appropriate function. To make sure it works properly, open the graph view, go back to the OBD reader view, change the function, and then go back to the graph view. It should be updated to the new function. The graph should start at the left side, slowly make its way over to the right edge, and once it hits it should begin scrolling the whole graph over to the left, ensuring that the parts of the graph scrolling offscreen are cleared from the view. The range of values will be the same as those on the gauge. The large gauge above the graph should work like those on the main OBD screen and conform to the same testing.

Bluetooth connection testing: when the program is launched, the program should check whether or not Bluetooth is enabled. If Bluetooth is disabled, a popup message should appear asking if the user would like to turn it on. If the user already has Bluetooth enabled, there should be no popup. From the main menu, press "Start Connection" to attempt to connect to the device. If the phone has not been paired with the device previously, it should bring up the Bluetooth options screen and search for available Bluetooth devices. Once paired, the user will return to the program where it will establish the connection to it. If the phone was already paired with the device, it will connect automatically when "Start Connection" is pressed. The program should be able to gracefully handle interruptions to the Bluetooth connection. If the user is on any of the views in the program that rely on the input stream from the Bluetooth device, an error message should pop up informing the user of the situation and letting them open the Bluetooth options screen to see if they can reconnect. If the user is on the gauge or graph view the numbers should return to their default (unconnected) states – and if recording a log, it will stop reading data and close any associated log files.

Keypad view testing: connect to the device and navigate to the Keypad screen. Press the Unlock button, and ensure the car unlocks. Press the Lock button and ensure that the car locks. Press the Start button and ensure the car starts. Press the Up and Down buttons and make sure the windows roll up and down respectively. Finally, press the panic and trunk buttons and observe that they perform their respective functions. These functions should be tested from a variety of distances to make sure any interference doesn't produce errant results (i.e., it should either perform the function or not, nothing in between). Since the car doesn't communicate back to the phone whether it has completed the desired task, the car will have to be physically observed to make sure each button performs its function properly.

Android hardware button testing: pressing the physical buttons on the Android device should behave in a predictable and standard way. The back button in particular can be troublesome in some applications, where it will close the application without warning and return to the Android home screen instead of going back by a screen. We would like to avoid this annoying behavior in our program. For our purposes, the back button should always return to the previous screen, or if a dialog menu is displayed, should dismiss the dialog. If the user is on the settings, keypad, OBD reader, error code or log screens, it should return back to the main menu. If, on the OBD reader view, the user has pulled up the customization menu for one of the gauges, pressing back should dismiss the menu but remain on the OBD reader view. If the graph view is accessed pressing back should return to the OBD reader view. Similar to other OBD readers, the only time the back button should be able to be used to close the program is when it's at the main menu. If this occurs, the program should prompt the user with popup box asking if the user is sure he wants to quit the program. The Android "menu" key should also behave in a predictable and standard way. It usually pops up a small menu on the lower portion of the screen, with options applicable to the screen at hand. We will need to test that the menu button works and pulls up the appropriate menu on the views that need it, and that nothing occurs when the button is pressed on screens that don't utilize it. Also in keeping with Android convention, pressing anywhere on the screen that is not the menu while the menu is up should dismiss it and not act as a press. The physical search key is not used in our program, and it should be tested on each screen to make sure there is no aberrant behavior with the button.

Log view testing: launch the program and select Log View from the main menu without connecting to the device or running the OBD reader view. Since there is no log data, the program should create blank files for each of the functions we're logging if it detects that no such log file exists. An error message should be displayed if the user attempts to graph or average values from this blank file. Next, to test that it is reading values properly, connect to the device via Bluetooth, and run the OBD reader for a while to let the logs collect data. Go back to the main menu and select Log View again. From the list of available logs, select each one and ensure not only that data has been written to it, but that it's the correct data (timestamp and value). Only 20 logs are shown by default, with a "Show more…" button; we need to make sure that the Show More button works as intended, increasing the number of log entries displayed while also letting the user scroll back up to see the first set of logs. Press the menu button while looking at a log to pull up the log settings menu and select "Find Average." The screen should pop up a message stating the average. To make sure that it's doing the calculation right, find the average by hand and compare it with the number displayed in the Android program to make sure they match up. Next test the graph view with each log set. Make sure that it uses an appropriate time scale on the X-axis, since the log data might be separated by significant lengths of time, and sticking those two data sets right next to each other would be incorrect. We also need to test that the Y-axis values are set up correctly, based on the range of values chosen for each function from the other gauge and graph screens. We also need to test to make sure the Clear log functionality works properly. When pressed it should delete any logs contained in the file for that particular function, and replace it with a blank log file that behaves the way it's supposed to (i.e., not allowing graphs or averaging). Finally, we must ensure that the log files are being saved properly. Browse on the phone or connect to a computer and see that the log files are stored in the pre-established folder on the user's SD card. There should also be a backup storage location on the phone storage in case an SD card is not installed or corrupted/write protected.

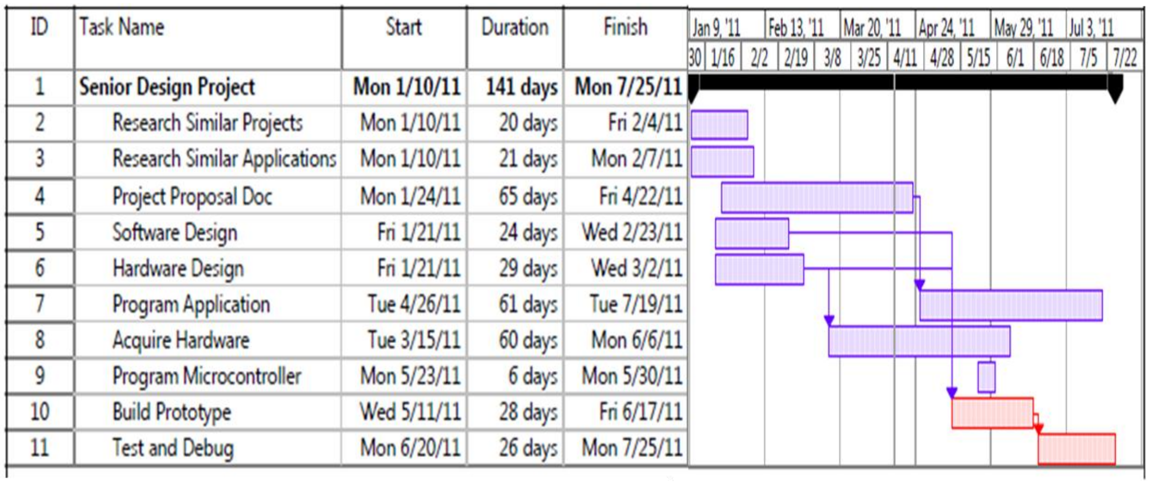
16.0 Administrative Content

Administrative content covers the project's planning and financial management. In a four-member group project, planning and financing are key aspects to the project's success. The project needs to be planned out evenly over the time allotted for completion. Planning and setting milestone dates are crucial to the success and quality of the project. As project administrators, we need to explore all options to help keep the cost of the project under budget. The following two sections, 16.1 and 16.2, explain how the project's time and budget is managed.

16.1 Milestone Timeline

In general, senior design projects are intended to take two, sixteen-week long semesters to complete. However, this project will need to be completed in a single sixteen-week semester and a twelve-week semester. Since we have four weeks less to complete the project than the ideal allotted amount of time, we need to plan and work efficiently.

The early stages of the project dealt with research. Researching similar projects and similar applications allowed the group members to find out what has and has not been done. Researching similar projects and applications took roughly three weeks to complete. The research helped determine exact specifications of the functionality of the project. Shortly after the start of research, we we're able to start the documentation process. The ideas and knowledge obtained from research were documented accordingly. The documentation is projected to be the longest process in the project at 65 days in length. While still researching and documenting simultaneously, we started the design phase. We designed the software and hardware aspects of the project in 24 and 29 days, respectively. Once the design was complete, we started purchasing the parts and test equipment necessary for the project. We project having all of the part and equipment necessary to the project by the first week of June. If the projection is accurate, we will have acquired the parts and hardware over a 60-day span. After all of the documentation is complete, we will begin to write the application. The goal is to complete the application programming in 61 days. While some group members are writing the application, others will start building the hardware prototype. The hardware prototype should take about four weeks to complete. The hardware prototype should be completed about two and a half weeks into June, so there's plenty of time to correct the unexpected problems. While the prototype is being constructed, we will program the microcontroller so it's ready to be tested with the prototype. The microcontroller should take a little less than a week to program. We anticipate that extensive testing and debugging will be necessary. The testing and debugging process will begin as soon as it is possible. Although, we don't foresee extensive testing and debugging will take place until after the hardware prototype is complete. Thorough testing and debugging shall take place during the last three and a half weeks of the project to ensure prompt delivery by the due date. Figure 16.1a shown below, is a visual representation of the milestones we have and wish to achieve throughout the course of the project.



ID	Task Name	Start	Duration	Finish
1	**Senior Design Project**	**Mon 1/10/11**	**141 days**	**Mon 7/25/11**
2	Research Similar Projects	Mon 1/10/11	20 days	Fri 2/4/11
3	Research Similar Applications	Mon 1/10/11	21 days	Mon 2/7/11
4	Project Proposal Doc	Mon 1/24/11	65 days	Fri 4/22/11
5	Software Design	Fri 1/21/11	24 days	Wed 2/23/11
6	Hardware Design	Fri 1/21/11	29 days	Wed 3/2/11
7	Program Application	Tue 4/26/11	61 days	Tue 7/19/11
8	Acquire Hardware	Tue 3/15/11	60 days	Mon 6/6/11
9	Program Microcontroller	Mon 5/23/11	6 days	Mon 5/30/11
10	Build Prototype	Wed 5/11/11	28 days	Fri 6/17/11
11	Test and Debug	Mon 6/20/11	26 days	Mon 7/25/11

Figure 16.1a – Estimated timeline of project from start to finish

16.2 Budget and Finance

The financing for this project came from the pocket of each of the group members: Alexander Powell, Firoz Umran, Josh Estes and Matthew Huereca. In some cases, the part of software was free or already owned by one of the group members. The ELM327 OBDII reader was purchased for $39.99 and used testing purposes. Each group member contributed $15.00 towards this device. The testing subject is a 1998 Honda Accord and was supplied by Firoz Umran for free. We estimate our final PCB board to cost about $150.00, which will be paid for by each of the group members. If the estimation is accurate, each group member will contribute $37.50. The smart-phone used will be contributed by one of the group members and is valued at $200.00. The platform used to program the Android application is a free download, courtesy of Eclipse. The computers used during the project are the possessions of each of the group members and will not consume any of the project fund. The microprocessor used cost $15.00 and will be paid for by each of the group members. Each group member will contribute $2.50 for the microcontroller. Five wire taps are used in the hardware design and cost $0.50 each. Each group member shall contribute $0.63 for the wire taps. If all of our assumptions are accurate and the hardware design is sufficient, we anticipate the total expenditure of the project to be $202.49. This implies that each group member will have to contribute $50.63 towards the project. The table below**, figure 16.2.a**, shows a complete breakdown of the project spending.

Project Cost Analysis			
Part	Quantity	Cost	Financing

ELM327 OBDII Reader	1	$39.99	Project Fund
1998 Honda Accord	1	$3,625.00	Loan from Firoz Umran
PCB Board	1	$150.00	Project Fund
Smart-Phone	1	$200.00	Loan from Group Members
Eclipse Platform	4	$0.00	Free Download
Computer	4	$4000.00	Loan from Group Members
Microprocessor	1	$15.00	Project Fund
Wire Tap	5	$0.50 Each	Project Fund
Total Expenditure		$202.49	Project Fund

Figure 16.2a – Project Cost Analysis Table

17.0 Conclusion

We believe that our project will be a useful tool for reading critical vehicle data as well as a handy way to lock and unlock car doors on the go and start the car before even entering the vehicle.

We have demonstrated our high level design which will include a connection from the android Bluetooth to the Bluetooth PCB. The ATMega MCU on the PCB receives the data and then either passes it on to the ELM 327 to read car data, or the MCU performs a function such as car start or door lock.

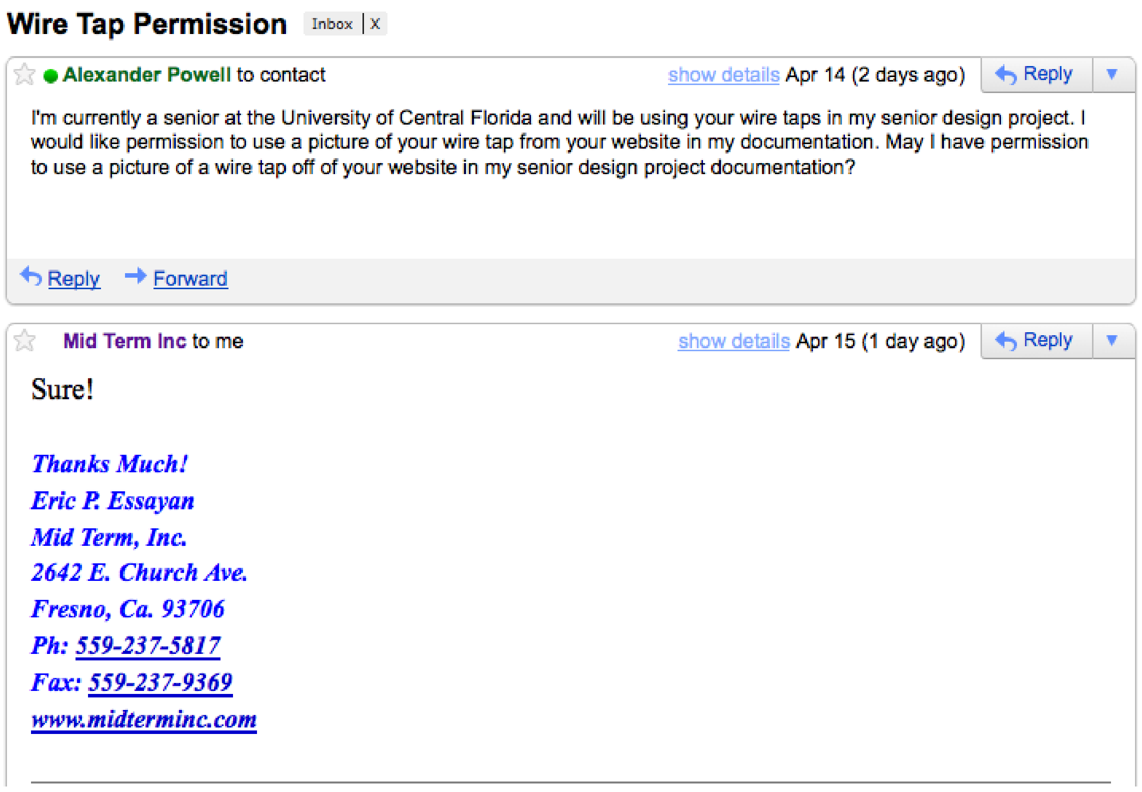
Our hardware design features our final schematic for the PCB. It shows the connections made between the Bluetooth chip, the MCU and the ELM327. The schematic was designed using EAGLE software.

The programming of the functions was done using specific request messages that the OBD-II port and car ECU reads. These messages contain headers, and data requests. The response messages were also discussed and class diagrams were created.

We believe the project has a need and motivation despite the existence of similar products. Our project will have more functionality and will be done made at a cheaper price than any of the other projects mentioned in the report.

Appendices

A. Copy write permission



Wire Tap Permission Inbox | X

Alexander Powell to contact — show details Apr 14 (2 days ago) — Reply

I'm currently a senior at the University of Central Florida and will be using your wire taps in my senior design project. I would like permission to use a picture of your wire tap from your website in my documentation. May I have permission to use a picture of a wire tap off of your website in my senior design project documentation?

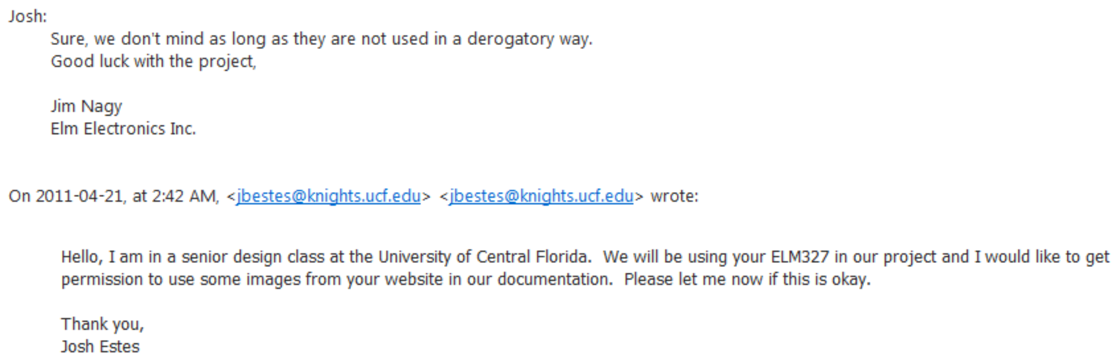
Reply Forward

Mid Term Inc to me — show details Apr 15 (1 day ago) — Reply

Sure!

Thanks Much!
Eric P. Essayan
Mid Term, Inc.
2642 E. Church Ave.
Fresno, Ca. 93706
Ph: 559-237-5817
Fax: 559-237-9369
www.midterminc.com

ELM Permission



Josh:

Sure, we don't mind as long as they are not used in a derogatory way.
Good luck with the project,

Jim Nagy
Elm Electronics Inc.

On 2011-04-21, at 2:42 AM, <jbestes@knights.ucf.edu> <jbestes@knights.ucf.edu> wrote:

Hello, I am in a senior design class at the University of Central Florida. We will be using your ELM327 in our project and I would like to get permission to use some images from your website in our documentation. Please let me now if this is okay.

Thank you,
Josh Estes

B. Works Cited

Beloussov, Alexandre. “alOBD Scanner.” *Android Market*. N.p., 27 Jan. 2011. Web. 28 Feb. 2011. <https://market.android.com/details?id=com.obd2>.

“Build One.” *blueOBD*. N.p., 2010. Web. 3 Feb. 2011. <http://www.blueobd.com/build_one.html>.

Hawkins, Ian. “Torque (Free/Basic).” *Android Market*. N.p., 9 Mar. 2011. Web. 28 Feb. 2011. <https://market.android.com/details?id=org.prowl.torquefree>.

“HOWTO Read Your Car’s Mind.” *ThinkyThings*. N.p., 10 May 2007. Web. 25 Feb. 2011. <http://www.thinkythings.org/obdii/#references>.

Memruk, Ivan. “Android Custom UI: Making a Vintage Themometer.” *Mind The Robot*. N.p., 7 June 2010. Web. 17 Apr. 2011. <http://mindtherobot.com/blog/272/android-custom-ui-making-a-vintage-thermometer/>.

“Mode 1 and Mode 2 Parameter IDs.” *OBDII Diagnostics*. N.p., n.d. Web. 23 Feb. 2011. <http://www.obddiagnostics.com/obdinfo/pids1-2.html>.

Noxon, Jeff. “Opendiag OBD-II Schematics & PCB Layout.” *Planetfall*. N.p., 13 Jan. 2009. Web. 21 Feb. 2011. <http://www.planetfall.com/cms/content/opendiag-obd-ii-schematics-pcb-layout>.

“OBD2 Diagnostic Operational Modes.” *CanOBD2*. Innova, 2011. Web. 18 Feb. 2011.

“OBD FAQ: OBD-II Communication Protocols.” *OBD-Codes*. N.p., n.d. Web. 21 Mar. 2011. <http://www.obd-codes.com/faq/obd-ii-protocols.php>.

“OBD-II Background.” *The OBD II Home Page*. N.p., 2011. Web. 23 Feb. 2011. <http://www.obdii.com/background.html>.

“OBDII Message Structure.” *OBD Diagnostics*. N.p., n.d. Web. 21 Feb. 2011. <http://www.obddiagnostics.com/obdinfo/msg_struct.html>.

“OBD to RS232 Interpreter.” *ELM Electronics*. N.p., n.d. Web. 5 Mar. 2011. <http://www.elmelectronics.com/DSheets/ELM327DS.pdf>.

“ScanXL Professional.” *ScanTool*. N.p., n.d. Web. 25 Feb. 2011. <http://www.scantool.net/scanxl-pro.html>.

“Scothlock Connectors.” *Mid Term Terminal and Connectors Company*. N.p., 2005. Web. 9 Apr. 2011. <http://www.midterminc.com/en-us/dept_52.html>.

“Viper SmartStart.” *Android Market*. N.p., 5 Jan. 2011. Web. 28 Feb. 2011. <https://market.android.com/details?id=com.directed.android.viper>.

"Viper SmartStart for Android." *Viper*. N.p., n.d. Web. 21 Mar. 2011. <http://www.viper.com/smartstart/android/Features.aspx>.

"What Is Your Car Trying To Tell You." *The Wire Up*. N.p., 16 Oct. 2008. Web. 16 Oct. 208. <http://www.thewireup.com/2008/10/what-is-your-car-trying-to-tell-you.html>.