On Board Droid

Firoz Umran, Josh Estes, Matthew Huereca and Alexander Powell

School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — The On Board Diagnostics (OBD) port on a vehicle is used to communicate data with a vehicles ECU. The data read from the OBD can be used for various purposes. The most common is to read faults in the vehicle indicated when the vehicles check engine light (CEL) turns on. The purpose of this project is to read this data from the OBD and display it in a meaningful manner to the user through their Android powered device, which will connect to the OBD wirelessly via Bluetooth. For added functionality the vehicle will also be wired so that the windows will go up and down, doors will lock and unlock and the vehicle will start all via commands from the users Android device.**

I. Introduction

OBD-II became a standard in 1996. All cars produced after this point adhere to this standard. This standard sets the pinouts on the connector to be the same in all cars. The table below shows the standard OBD pinouts followed by a figure of how a typical OBD connector looks.

TABLE I
OBDII Port pinouts

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

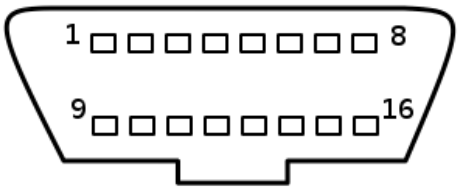


Fig. 1 Standard OBD Connector

For the purposes of this project the only pins we will be concerned about are:

(1) Pin 5 – signal ground
(2) Pin 7 – K line ISO 9141-2
(3) Pin 15 – L line of ISO 9141-2
(4) Pin 16 – Battery Voltage (12v)

The reason for this is that we will be using a 1998 Honda Accord for this project and the protocol associated with this vehicle is ISO 9141-2 so those are the only pins necessary along with power and ground.

Our project will wirelessly connect to the car via Bluetooth from an android device. The project will be able to do two specific functions, OBD functions and physical functions.

A. OBD Functions

These functions are all read from the ECU of the car for various data such as Air Intake Temperature, RPM, Speed and of course reading the Check Engine Light. This data will be displayed on the Android device in various forms such as gauges, text and graphs.

B. Physical Functions

These functions concern with doing things on the car itself. These functions consist of locking, unlocking, opening trunk, winding windows down, winding windows up and starting the car. The purpose of this is to mimic a remote entry system on most vehicles using a Bluetooth enabled Android device.

II HIGH LEVEL DESIGN

Figure 1 shows the high level design diagram for our project. It shows the flow of data: The android application send data to the MCU. The MCU checks the format of the data and determines whether to send it on to the ELM or to perform a physical function. If passed to the ELM, the data is translated to code that is readable by the ECU (Engine Control Unit). The OBD-II takes the ECU's response and sends it back to the ELM, which is then sent

to the MCU and finally shows up on the Android phone in a user friendly format.

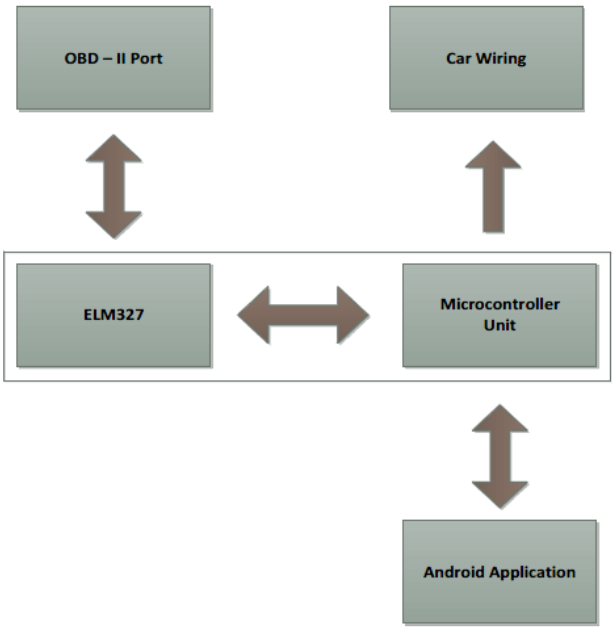


Fig. 2 High Level Design diagram

III. HARDWARE

This section will describe the hardware used in the project outside of the vehicle and smartphone. Specifically, the main hardware components that make up the PCB will be covered. The main components of the PCB are: the Bluetooth module, two microcontrollers and the integration of specific functions from the Arduino test board.

A. BlueSmirf Gold Bluetooth Module

We decided to use the BlueSmirf Gold Bluetooth module because it has a class 1 radio as the form of wireless communication. It also has low power consumption since the average current used is about 25 milliamps. The frequency hopping scheme allows for the Bluetooth to operate in harsh environments like WiFi, 802.11g and Zigbee. The Bluetooth connection is also encrypted when using the BlueSmirf Gold Bluetooth module. The frequency operation is between 2.4 and 2.524 GHz. The operating voltage conveniently varies from 3.3 volts to 6 volts. The built-in antenna eliminates another external part.

B. ELM327 Microcontroller

The ELM327 is an OBDII interpreter integrated circuit. This IC supports all of the protocols used by vehicle manufacturers to communicate with the OBDII port. The ELM327 allows for customized protocols so special OBDII functions can be implemented. Among the great features of the ELM327, it's also appealing from a specifications standpoint. Low power consumption while in stand-by mode is critical when installed within a vehicle. The IC features a non-volatile memory location for storing user data. The baud rate is adjustable for easy interface with other hardware parts.

C. ATMega328 Microcontroller

The ATMega328 is an 8-bit processor with a 28 pin-out design. The ATMega328 is equipped with 32K of flash memory and 23 I/O lines. With an external crystal, the MCU can run at a rate up to 20 MHz. The ATMega328 can operate between the 1.8 volts and 5 volts.

D. Arduino UNO Integration

The Arduino UNO is the test board used in the project. The main attraction of this test board is how well it works with the ATMeaga328 MCU. During the construction of the project, we decided to integrate some of the features of the Arduino UNO into the final PCB. The input voltage range from 6 volts to 20 volts allows for an easy power access directly from the OBDII port of the vehicle. The voltage regulator design allows creates a 3.3 volt pin which perfectly suits the BlueSmirf Gold Bluetooth module.

E. Overall Setup

The schematic below shows the overall schematic and design of the project.

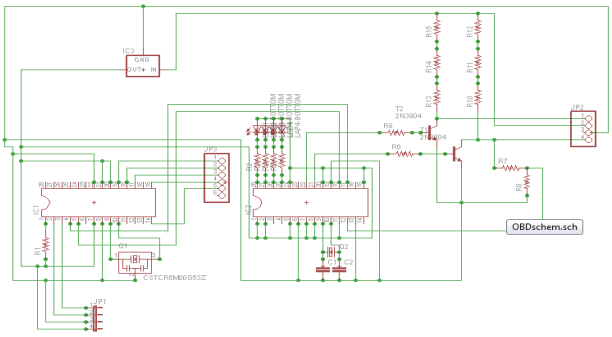


Fig. 3 Overall hardware schematic

IV. PROTOCOL

This section will describe in detail the protocol being used by the Android device to communicate with the microcontrollers to perform the specific functions. This section will be split into a discussion of the protocol for the OBD Functions, the protocol for our ELM 327 microcontroller and the protocol for the physical functions.

A. OBD Function Protocol

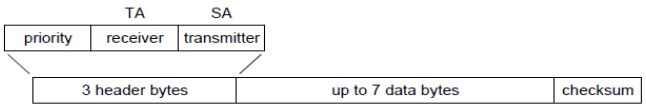
There are five separate OBDII protocols:
(1) SAE J1850 PWM (Standard of the Ford Motor Company)- Pin 2: Bus+, Pin 10: Bus-, High voltage is +5 V, Message length is restricted to 12 bytes
(2) SAE J1850 VPW (Standard of GM)- Pin 2: Bus+, Bus idles low, High voltage is +7 V, Decision point is +3.5 V, Message 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

For this project we will only be dealing with protocol 3. In order to receive data from the OBD a request must first be made to the OBDII port in this form:

	Mode	PID
Request	XX	YY

XX indicates the mode of the request in which there are 9. And YY indicates the PID of the request. For example if one wants to view the air intake temperature of a vehicle. The request is a mode 1 and its PID is 0F so one must send 010F to the OBDII port. The different modes are:
(1) 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.
(2) 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.
(3) 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.
(4) Mode 4 – Simply instructs the vehicle to clear all error codes.
(5) Mode 5 – An optional mode used for requesting results of an oxygen sensor test. Some vehicles report this under mode 6.
(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.
(7) 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.
(8) Mode 8 – Used to request control of an on board system. This mode is manufacturer defined.
(9) Mode 9 – Optional mode used to report vehicle information such as the VIN and information stored in the ECU.

Most of the requests will be of Mode 1 except when dealing with reading the check engine light on a vehicle which will be dealt with in Mode 3. Once a request is sent the port will respond with a message in the following form:



However, the 3 header bytes are for the user. The bytes of interest are the 7 data bytes. The first two bytes will simply be an echo of the request that was sent to it while the remaining four bytes will contain the data. Depending on what the request is the data that comes in must be properly formatted in order for the user to view the responses in a meaningful manner. The table below shows the different Mode 1 requests to be used and the formulas to convert the data.

TABLE 2
PID CONVERSION CHART

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	RPM = .25 *

		and B	(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

B. ELM 327 Protocol

Before we can even talk to the OBD we must be sure that our microcontroller that is preprogrammed with the protocol is setup properly. In order to do this the ELM327 comes with its own set of commands that can be used to configure the MCU. These commands all begin with "AT" and will hereby be referred to as AT commands. The commands of importance are as follows:

(1) ATDP – describe the current protocol
(2) ATRV – get voltage
(3) ATZ – serial reset
(4) AtE0 – Echo off (used so the request does not appear in the response)
(5) ATIB – set the baud rate for different devices

C. Physical Functions Protocol

When requests are sent to our ATmega 328 microcontroller the ATmega will decipher the message and decide whether it should handle the function or send it to the ELM chip. It will handle the function if the request begins with FC. For the physical functions we have created our own protocol each beginning with FC as shown in the table below.

TABLE 3
PHYSICAL FUNCTION PROTOCOL

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

V. SOFTWARE

The software on the Android device is programmed in the JAVA programming language. This language is a high-level object oriented language. There are several sections of interest to discuss about the programming of the software. Those sections include the OBD real-time OBD functions themselves, the keypad for the physical functions, the error codes and the logging feature.

A.OBD Real-Time Functions

Each specific OBD function has its own object associated with it. Within that object are methods to send, receive and convert data to, from and for the user. Each specific object uses the formulas found in TABLE 2 to convert the data it has received. These objects are built in a hierarchy which makes it easier to add more objects in the system without having to rewrite a massive amount of code and since they are "real-time" functions each object is also a threaded object that has a run method to be called when the user wants to constantly read this data. There is one function that was programmed that is not mentioned in the table because it does not come straight out of the OBD but it is a very useful function. That function is the ability to get the instantaneous fuel economy from the car. To do this we must first get the values for the Mass Air Flow and the Speed of the vehicle then we use the formula to calculate the value.

$$\text{Fuel Economy(MPG)} = \frac{(14.7 * 6.17 * 4.54 * \text{speed} * 0.621371)}{(3600 * 100 * \text{maf})} \quad (1)$$

B.Keypad

The keypad is where all the physical functions are located. These functions are similar to the OBD functions except that there are no formulas because no data is sent back after the request is made. Rather something physical should happen such as the car unlocking. Below is a class diagram of all the objects used for the keypad.

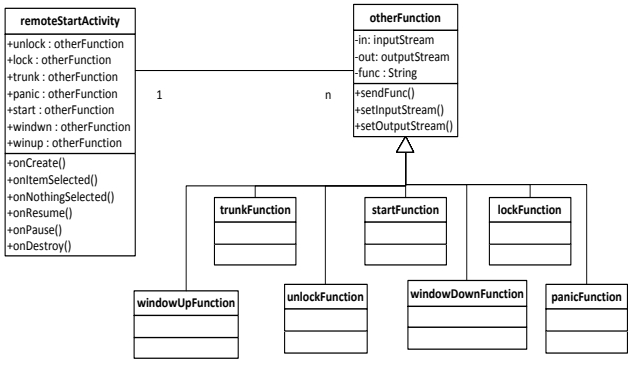


Fig. 4 Class diagram of the physical functions. Shows the relationship between the functions and the activity in which they are present in. All the functions inherit from the main class called otherFunction which contains all the necessary methods for sending data.

C. Error Codes

To read the error codes thrown by a cars ECU a mode 3 request is sent to the OBD. However, the formatting of this message is different than the others. All the error codes are sent at once, 3 per line, and a vehicle may have multiple errors. So what must first be done is send a mode 1 PID 1 request to obtain how many error codes are in the system. Then use that to determine how many lines will be returned and split the error codes from there. Every two bytes consist of one error code. Finally, once you have the error code you must take the first digit and use the chart below to convert the code to something that can be searched for in an online database.

TABLE 4
ERROR CODE CONVERSION

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

D. Logging Feature

The logging feature is the ability to store away the real-time data that has previously been read so that it may be viewed again at a later time. Adding to that the user will also be able to calculate average values of all data read. For example, the user can take all the instantaneous fuel economy values and use that to calculate the vehicles average fuel economy, which is extremely useful in these times of high gas prices. The way this works is that a log file is created for every OBDII real-time function. Every time that data is read in for that specific function it gets put in that functions log file. Then when the user selects to view that log the program will simply open that text file and display it to the user.

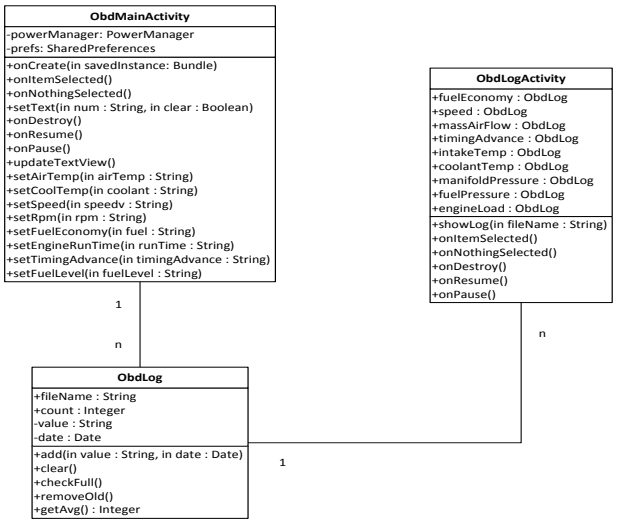


Fig. 5 This is the class diagram of the logging feature. It shows the main screen and its methods that relate to the Log object being created and entered into the logging screen to be called upon and viewed by the user at a later time.

VI. USER INTERFACE

The user interface of our Android program allows the user to interact with the OBD data as well as control the car's functions. It was designed to mimic features already familiar to drivers (gauges and keypads) while still fitting in with our project's design goals. Figure 6 shows the main menu of the program. Start Connection allows user to manually connect to the device, though the program will try to connect automatically when first launched. OBD II Reader opens the OBD gauge and graph screen that lets the user see data being read from the OBD. Keypad shows a virtual car keypad that mimics the design of physical keypads used to unlock, lock, etc. Logs lets the user view logs saved to the user's SD card. They can view

the raw data in text form, or see it graphed out. Error Codes displays any errors read from the OBD and gives their associated DTC (Diagnostic Trouble Code). There is an option to Clear All errors from this screen as well. Settings contains user controlled settings such as setting the units, the option to keep the screen on at all times while using the program, and whether to confirm when quitting.

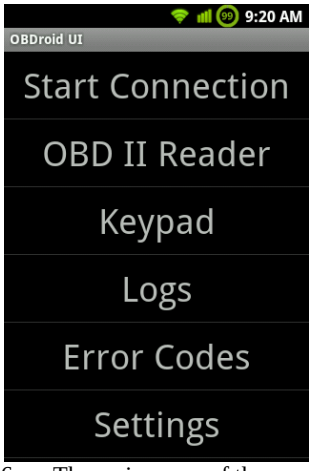


Fig. 6 The main menu of the program

Figure 7 shows the main OBD reader screen. From here the user can press and hold on the gauge to choose which function to read. The user can also press the menu key on their android phone and have the options to run custom functions by sending commands directly to the OBD.

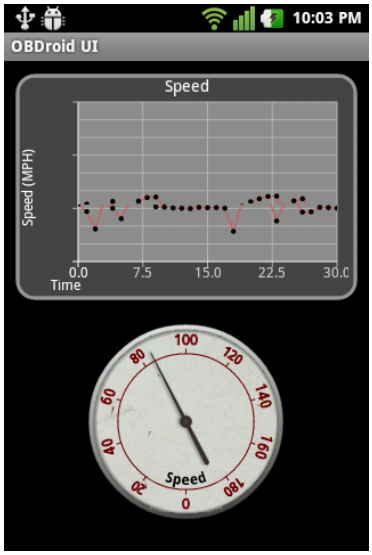


Fig. 7 The OBD reader screen

The Android app was programmed using Java, and most of the interface was created with XML. For many of the screens the java file just calls the XML file that contains the formatting of the screen, similar to how HTML and CSS work together by keeping the "content" and the styling separated. Each screen is considered an "activity" and each activity is a separate class. Most of the interface was created using standard views found in the Android libraries, such as text boxes and buttons. However, some interface objects were too complicated to be created from the built in libraries alone. Custom views had to be created for the gauges and the graphs. The gauge is drawn using the canvas commands found in Java. In particular, the rotate command used on a canvas object was very useful in drawing the notches and numbers in a circle around the edges of the gauge. Whenever a gauge is needed, we just create a gauge object and manipulate it from there. For the graphs, we used an Android graphing library called AndroidPlot. We chose AndroidPlot specifically because it provided the ability to create both static and dynamic graphs. Static graphs are used for the log functionality – taking text log data and converting it into a graph. The dynamic graphing ability is used for the OBD reader screen to display the data in real time in conjunction with the gauge. In our implementation we use a linked list to hold the most recently acquired data from the OBD, and draw that data to the graph. When a new data point is received, it's added to the linked list, the oldest data point is removed and the graph is redrawn again.

VII WIRING THE CAR

Since we are using a 1998 Honda Accord for the purpose of this project we will need to wire the car based on its factory specs. For the OBD port we simply plug the connector in and it's ready to send and receive data. However, to unlock the car we need to take the pin that goes high when an unlock signal is sent and wire it to a relay that connects to the unlock trigger on the car. Since the unlock trigger is negative logic it must be tied to ground so when the relay is activated the unlock trigger will get a ground signal and unlock the car. The table below shows where all the trigger wires are located and whether they are positive (+) or negative (-) triggered. Following the table is a figure on how the system will be wired to the relay harness.

TABLE 5
ACCORD WIRING CHART

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

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

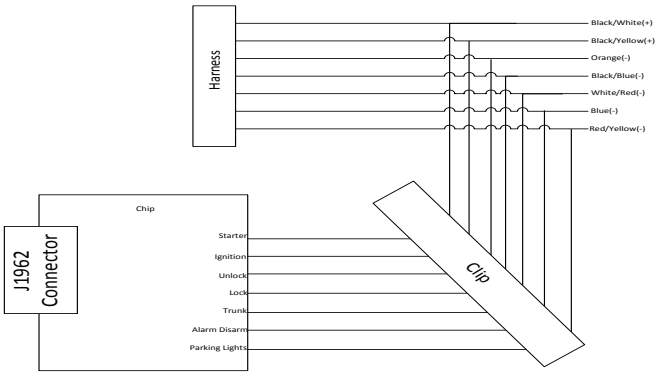


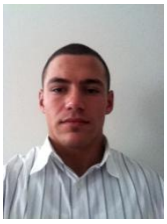
Fig. 7 Shows the wiring from the chip to the relay harness and then to the rest of the car.

VIII 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. Our project provides an intuitive and unique way to interact with the car, and we have learned much about designing and testing hardware and getting software and hardware to work together. Creating this project has been an excellent learning experience.

IX BIOGRAPHY

Matthew Huereca is a senior student of the Computer Engineering department at the University of Central Florida. He will graduate in the summer of 2011 and plans to pursue a career in the computer engineering industry.



Alexander Powell is an Electrical Engineering student with a Computer Science minor and will graduate The University of Central Florida in summer of 2011. He plans to enter the engineering industry with hopes of returning to graduate school in 2012.

Firoz Umran is a Computer Engineering student at the University of Central Florida. He will graduate in the summer of 2011 and plans to obtain a career in Engineering.

Josh Estes is a Computer Engineering student and will graduate The University of Central Florida in summer of 2011. He plans to enter the engineering industry with hopes of returning to graduate school in to pursue a Masters in Business Administration.

VII References

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>.