Integrating ODB-II, Android, and Google App Engine to Decrease Emissions and Improve Driving Habits

Chris Furmanczyk
UW CSE
1222 Lakeview Blvd E #3
Seattle, WA 98102
1.425.232.0416
furmac@cs.washington.edu

David Nufer
UW CSE
8515 SW Halter Terrace
Beaverton, OR 97008
1.503.530.6381
dnufer@cs.washington.edu

Brent Sandona
UW CSE
185 Stevens Way
Seattle, WA 98195
1.206.543.1695
sandona1@cs.washington.edu

Benjamin Ullom
UW CSE
4122 Whitman Ave N
Seattle, WA 98103
1.206.228.8058
ullom@cs.washington.edu

ABSTRACT

Every time a car is driven, sensors within the car generate information on a number of crucial systems that are typically only accessed when something goes wrong and the car is taken to a mechanic. The diagnostic (OBD-II) port on a vehicle provides a way to access this information so that it can be used to monitor the state of the vehicle at any time. Saving and tracking this information allows long-term driving trends and engine health to be monitored. When provided with this information, it is possible for drivers to significantly decrease their fuel consumption and CO_2 emissions, improve driving safety, and make better maintenance decisions. Existing products are either expensive all-in-one solutions for mechanics or smaller devices designed for hobbyists to use in tuning, typically by connecting them to a PC. Our solution is targeted at average consumers and must be easy to use, easy to install, and require little maintenance. Our projects consists of three parts: 1) a hardware device that physically plugs into the car to gather information and transfer it to 2) an application on a smart phone running Android that saves the information, tags it with GPS and time stamps, and uploads it to 3) a Google App that will take the information and display it (route and sensor data) in an easy-to-read format.

Keywords

BD-II, Car, Mileage, Android, Google App, ELM327, ISO, CAN, PWM, VPW, GPS, Safe driving, Efficiency

1. INTRODUCTION

Every time a car is driven, sensors within the car gather information on a number of crucial systems that are typically only accessed when something goes wrong and the car is taken to a mechanic. The diagnostic (OBD-II) port on a vehicle provides a way to access this information so that it can be used to monitor the state of the vehicle at any time. By saving and tracking this information, long term driving trends and engine health can be monitored. When provided with this information, it is possible for drivers to significantly improve their driving habits. Our goal is to design a device to interface with a car's OBD-II port and generate data that can be viewed later in an easily understood format on a web site. Our device will collect vehicle speed, mass air flow, oxygen sensor values, RPM, and 12 other values (which can be used to calculate mileage), accelerometer values, (which can be used to signal unsafe driving events), and provide access to trouble codes (which will give the driver information about what is wrong with their car). By gathering this information, saving it, and allowing the driver to access it after they are done driving, we think that we can help the driver: a) decrease fuel consumption and CO_2 emissions, b) improve driving habits and safety, and c) become empowered when it comes to maintaining and repairing their vehicle.

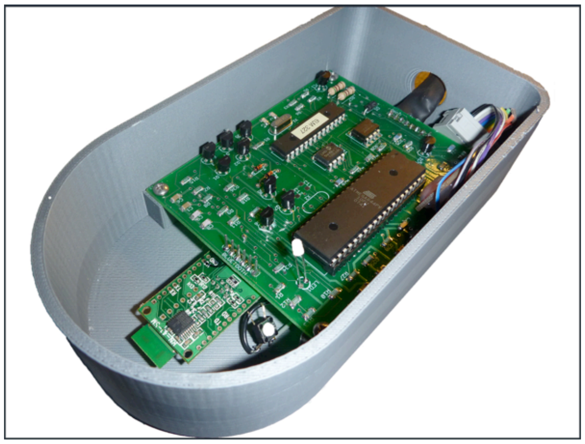


Figure 1 Vayta Hardware Device

To meet our goals, we decided to divide the project into three parts: 1) a hardware device that physically plugs into the car to gather information and transfer it to 2) a smart phone running Android that saves the information, tags it with GPS and time stamps, and uploads it to 3) a Google App that will take the information and display it (route and sensor data) in an easy-to-read format. This modular approach allowed us to work on the different parts independently and will also allow for greater flexibility in the future. Any individual part can change significantly without affecting the performance of any other part, allowing for multiple smart phone platforms and different hardware devices to be supported in the future.

In this paper we will discuss related work, the technical details of our project, how our device performed, future work, and what we would change if we were to do this project over.

2. RELATED WORK

There are a few products on the market that gather and display information from the OBD-II port, but none of them provide all of the functionality provided by our project. All of the devices currently on the market have one or more of the following drawbacks:

- Require a PC to be connected to view/analyze the data.
- Show only live data and do not offer long-term data tracking.
- Require a cable connection directly to a computer.
- Do not provide route tracking and data correlation with GPS location.
- Do not provide a website to view data

One such product is OBDLink, by ScanTool, connects to a PC using either USB or Bluetooth. It allows long-term data collection and analysis using provided Windows software [1]. Since this device doesn't have access to GPS data, it cannot correlate the data collected with route information. It also uses Windows only software, so the data is not as easily accessible as it would be on a web site.

Another product is Rev, by DevToaster, is software for an iPhone or iPod Touch that can do live data monitoring, GPS tracking, and limited long-term data gathering [2]. It is software only, so you must purchase a scan tool that connects using an Ad-hoc WiFi network. Their website suggests that there are at least three such devices available. This software has the same drawbacks as our Android App (platform specific) and is designed more for instant or short-term data viewing rather than long-term driving trends.

There are also several devices [3,4] available that plug into the OBD-II port and save information to a SD card. The card can then be connected to a PC to view/analyze the data. These devices have the same drawbacks as the OBDLink above, but don't require a PC to be in the car to gather data. However, this does mean that the device has to be removed from the vehicle and the data manually uploaded to a PC in order to be of use.

3. TECHNICAL DETAILS

3.1 Overview

Our hardware device consists of an ELM327 to interface with the OBD port, an ATmega644 microcontroller to facilitate communication, a 1MB flash memory module to store data, an accelerometer, and a Bluetooth module to interface with the smart phone.

The Android app connects to the hardware device via Bluetooth, stores the data in an internal database (along with a GPS location), shows a live snapshot of the gathered data, and uploads the data (in XML format) to a central server using FTP.

The Google App processes the data and displays the route on a map. When you click on the GPS points, you can see the data that was gathered.

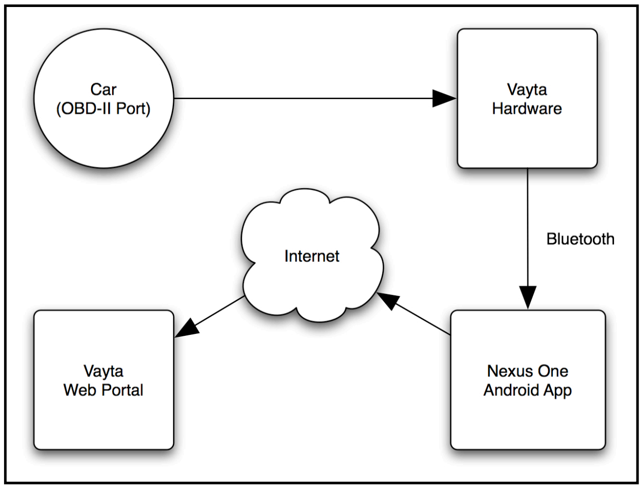


Figure 2 Data Flow Overview

3.2 Implementation Details

3.2.1 Hardware

The Vayta hardware device is designed to interface with the car using the OBD-II port. Since there are several different communications in use, depending on manufacturer and model year, our goal was to develop a device that could use any of the protocols that are commonly in use. Once connected to the vehicle, our device requests the values of 16 different measures once per second. At the same time, it reads the accelerometer values. Once it has read all of the data, it stores it in a flash memory module until the Android app requests it. Once the data is requested, our device reads it from the memory and transmits the raw values to the smart phone over a Bluetooth wireless connection.

ELM327

The ELM communicates with the OBD-II port by sending PIDs (Parameter IDs) and receiving data from the car. The enumeration of PIDs is standardized, meaning that 0x010D (mode 01, PID 0D) is always a request for a single byte value containing the car's speed. However, it is not required that every car function with every PID.

The ELM is a device originally for terminal interaction with a car's OBD-II port. It accepts commands in ASCII to change its settings or establish communication with the OBD-II port. Its data responses are returned as ASCII encoded hex values with a '>' character to represent the end of a command. Therefore, all responses from the ELM must be reverted into raw byte data by equating 0x30 ('0') with byte value 0, 0x46 ('F') with byte value 15, and so forth for all hex values 0-F.

Interpreting responses from the ELM also includes separating data from request confirmations and other values. The first step in this process is to filter out any non-hex characters. This is accomplished by only analyzing incoming data from the ELM that is ASCII '0'-'9' or 'A'-'F'. Secondly, the ELM returns a request confirmation for each line of data. Normally this is of the form '4&##' where & is a confirmation of the mode and ## is a confirmation of the PID. Therefore, ignoring the first four characters (that pass through the hex filter) will bypass the confirmation. The PID for the VIN (vehicle identification number) is the exception. The PID for the VIN is a multiline response with each line having both a confirmation and two characters for a line number. Therefore, ignoring the first six characters will bypass the confirmation in this case. The final step is to only accept as much data as is needed. Some PIDs used, including 0x0114, return two bytes of data when only the first is used. Then when the '>' character is seen, it is known that the response is complete and the ASCII can be converted into raw hex values.

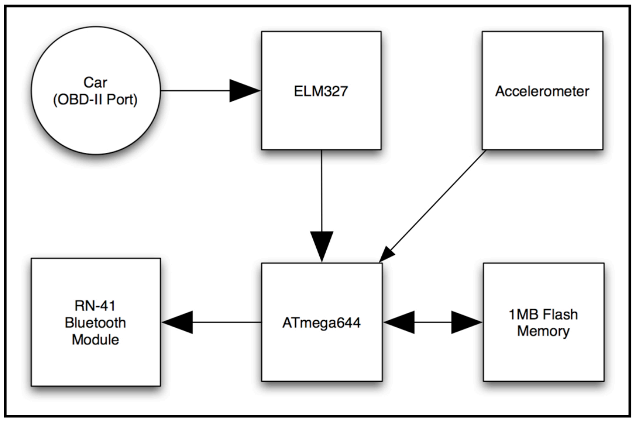


Figure 3 Hardware Data Flow

Flash Memory

The flash memory allows for long term buffering of gathered data as well as not subjecting the user to constantly having Bluetooth communications open. Its storage is structured to make efficient use of space while being able to use the device across multiple vehicles. In addition, the lack of a page erase influenced decisions about long-term storage. The memory in use features 1 MB of storage divided into 4 Kbyte sectors and 256 byte pages.

Pointers for the current memory locations (current reading and writing locations) are stored within the ATmega644's EEPROM. To prevent the EEPROM from dying quickly (it only has ~100,000 writes) the decision was made to only update these pointers in EEPROM when a sector transition was made. Sector transitions were chosen because locations in memory can only be written to after a sector erase has been performed on that location. If a sector boundary wasn't reached but the pointer was saved before a reset, the new data would not be recordable. The consequence of this method is that data is only saved through a reset when an entire sector has been filled.

Additionally, to standardize how the Android application would receive the data, a page worth of data was collected before being written to memory. Each page begins with 25 bytes for the VIN in order to identify which vehicle the data is associated with. The following 231 bytes are divided into 21 byte data-slices. Each data-slice represents a complete collection of data (all the PID results and the accelerometer values). Therefore, each page contains 11 of these data-slices.

ATmega644

The ATmega644 connects all the hardware components together. This specific microcontroller was chosen for its numerous communication ports, including two UART ports and one SPI port. The four main functions of the microcontroller are to: 1) communicate with the ELM to receive OBD-II port information, 2) take accelerometer data, 3) store data to flash memory, and 4) retrieve data from flash memory and send it through Bluetooth.

The ELM is connected to the Atmega644 through a UART port. Communication with the ELM is initialized by resetting the ELM's current settings and choosing a compatible protocol. From then on, PIDs are sent and the responses are analyzed as described in the section above. An interrupt-based timer sets a data collection flag once a second. When an entire data-slice has been collected, this flag is turned off. This prevents a data-slice from being collected more than once per second.

Accelerometer data is collected using the built-in analog to digital converter. When all of the PIDs have been collected, interrupts for collecting accelerometer data are triggered. The X-, Y-, and Z-axes are each connected to a different single-input analog to digital converter, selected by a MUX register. Data for each axis is collected during each data-slice.

The flash memory is connected to the ATmega644 through the SPI port. Storing data in the flash memory starts with placing the data in a buffer. When a page worth of data has been placed in the buffer, a check is made to see if the memory is busy. Once it is not, the buffer is written to memory. Page structure and memory management are described in the section above.

The Bluetooth module is connected to the ATmega644 through a UART port. Reading from flash memory occurs when a request for data has been received over Bluetooth from the Android application in the form of the byte 0xCD. This is detected using a UART received interrupt. If there is not an entire page worth of new data in memory at the time then the end signal, 0xEE, is sent. If new pages do exist in memory, a single page is placed within a buffer. This buffer is then sent through Bluetooth, using the UART transmit interrupt, to the Android application. When the buffer has finished being sent, the ATmega644 fetches another page from memory. If there are no more pages, it instead sends the end signal.

3.2.2 Android App

The Android App's main purpose is to act as a bridge between the hardware and the Google App, but can also be used for other purposes like displaying live data. There are three main components of the Android App: 1) Communication with the device, 2) Data Processing and Storage, and 3) Data transmission to the Google App. The application also allows the user to customize some settings like username and password for the Google App, and watch live data as it is collected from the device.

Communication with the Device

Communication with the device is done over Bluetooth. Before the phone can started downloading data from the device, the phone and the device must pair with each other over Bluetooth. The pairing process is carried out through the phone's operating system. The pairing process only needs to be carried out once for a device because the phone will remember which devices it is paired to. Once the phone and the device are paired, the Android App will be able to connect and send requests for data. Data collected from the device is divided into trips. A single trip can be started and stopped by a user in the application by pressing the "Start Trip" button. This action causes the application to start a background service on a phone, which runs for the duration of the trip. The background service controls most of the functions of the application.

When the background service starts, it determines a unique ID number to assign the current trip. The service then checks to make sure Bluetooth is both available and enabled on the phone. If Bluetooth is not enabled, the application will prompt the user to turn on Bluetooth. Once Bluetooth is enabled, the background service will try to connect with the device. If the connection is ever broken, or somehow interrupted, the background service will automatically try to reconnect with the device.

Once a connection with the device has been established, the phone will start sending requests for data to the device. The frequency of these requests can be set by the user. A request for data is initiated by the background service by sending a byte with value 0xCD to the device. The background service will then continue to read data from the device until it reads an end sentinel with value 0xEE. The data is then processed, and entered into a database on the phone as described in the next section. The background service will now wait for certain interval before making another request.

Data Processing and Storage

Data is stored in two tables of a SQLite database on the phone. The first table holds data for all the measurements that have been collected, along with which trip each data point belongs to. The second table associates trips with VIN numbers. The data collected from the device is raw, which means that is must be converted to proper units. Unit conversions for the OBD metrics are calculated according to the Wikipedia page of OBD-II PIDs [CITATION NEEDED]. Raw Accelerometer values are converted into units of g's (gravity), using this formula, where 'value' is the acceleration in g's and 'A' is the raw byte value:

value = (A-77) * 3.6/77

This results in a value between -3.6g and 3.6g. After all the raw values for a sample have been converted into the correct units, they are entered into the database. If GPS is available on the phone, latitude, longitude, and altitude are also entered into the database.

When the user ends the trip, the background service stops, and the data for the trip is written to an XML file according to the following format:

```
<trip id="123456789">
  <vin>1234567890123456</vin>
  <username>johndoe</username>
  <password>secret</password>
  <data>
    <timepoint id="123456789">
      <pid_0104>value</pid_0104>
      <pid_0105>value</pid_0105>
      ...
      <accel>accel_x,accel_y,accel_z</accel>
      <GPS>lat,lon,alt</GPS>
    </timepoint>
    <timepoint id="123456790"> ... </timepoint>
    ...
  </data>
</trip>
```

Inside each timepoint tag, there is a tag for each of the PIDs that we are measuring, and the data tag can have an arbitrary number of timepoints inside of it. The username, password and VIN are loaded from the phone's memory after the user has set them from within the app.

XML files are GZIP compressed when being stored on the phone and transmitted to the Google App to save memory and upload time. After some testing, it was determined that the XML files reduce in size by between 80% and 90%, which saves a huge amount of memory and upload time.

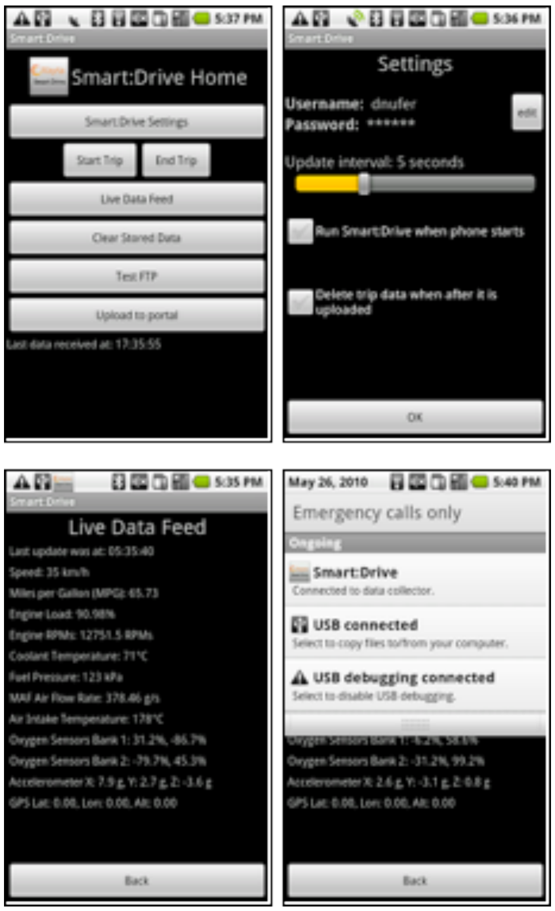
Currently, data remains on the phone until the user manually clears it. This is accomplished by pressing a button in the application. All data on the phone is cleared by through the following process: dropping both tables from the database, and deleting all the XML files that have been saved on the phone.

Data Transmission to the Google App

Originally, we planned to upload XML data files using the FTP protocol. File uploading via ftp was completed successfully using another Android application called AndFTP [6]. AndFTP has the ability to let other applications easily upload files to any FTP server easily. While this was successful, we later found out that Google App Engine does not support upload files via FTP. Instead, we tried using posting the XML files over HTTP to the Google App. However, at the point we discovered this problem, there was not enough time to implement HTTP post requests successfully.

User Interface

There are three major screens in the application: the Home screen, the Settings page, and the Live Data Feed. A notification also appears in the notification area of the phone when a trip is in progress.



Last data received at: 17:35:55

Username: dnufer
Password: ******
Update interval: 5 seconds
Run Smart:Drive when phone starts
Delete trip data when after it is uploaded

Last update was at: 05:35:40
Speed: 35 km/h
Miles per Gallon (MPG): 65.73
Engine Load: 90.98%
Engine RPMs: 12751.5 RPMs
Coolant Temperature: 71°C
Fuel Pressure: 123 kPa
MAF Air Flow Rate: 378.46 g/s
Air Intake Temperature: 178°C
Oxygen Sensors Bank 1: 31.2%, -86.7%
Oxygen Sensors Bank 2: -79.7%, 45.3%
Accelerometer X: 7.9 g, Y: 2.7 g, Z: -3.6 g
GPS Lat: 0.00, Lon: 0.00, Alt: 0.00

Smart:Drive
Connected to data collector.
USB connected
Select to copy files to/from your computer.
USB debugging connected
Select to disable USB debugging.
Oxygen Sensors Bank 1: -6.2%, 58.8%
Oxygen Sensors Bank 2: -31.2%, 99.2%
Accelerometer X: 2.6 g, Y: -3.1 g, Z: 0.8 g
GPS Lat: 0.00, Lon: 0.00, Alt: 0.00

Figure 4 Android App User Interface

The Home screen contains all of the controls of the application. The user can navigate to the Settings page or the Live Data Feed, can start and end trips, can clear stored data, and can upload data to the Google App. The Settings page allows the user to set their username and password for the Google App, as well as set the time interval for data requests to the device. The live data feed shows different metrics for the single newest data sample that the phone has received from the device. The notification for the application displays the status of the connection to the device.

3.2.3 Google App

The Vayta web portal is the user-friendly front end for the users of the Vayta system. Data should be periodically uploaded as an XML file from the Android application via an HTTP post. This operation can occur automatically without user intervention using a combination of when the phone approaches a memory threshold and when a certain number of trips have been recorded. The web application has a SAX parser to convert the XML file into the application's data model representation.

The web portal requires unique usernames and has password-protected accounts. Signing in to the web portal directs users to their front page where they can view updates, smart driving tips, community stats, and past driving trips. The past driving trips link to the maps display page. The maps display page loads a Google Maps image that shows the path the user took during that trip. The map displays markers at the various GPS coordinates at which the Vayta system recorded data. The user can investigate their instantaneous driving data at these various markers, including statistics such as engine temperature, RPMs, and vehicle speed.

The Vayta web portal was hosted on Google App Engine. We chose this hosting service because it was free and provided us with all the bandwidth we anticipated needing. Google App Engine supports two different server-side scripting languages. Originally it supported the Python Django framework, but recently it supported Java Server Pages. We wrote the portal in Java Server Pages because everyone in our group was familiar with Java, but not with Python. We used Google's Java datastore API to save permanent data on the server. This was built on top of the Java Data Object Query Language (JDOQL). Other APIs the web portal used were SAX for parsing XML data from the Android app and the Javascript library from the Google Maps API.

The Vayta web portal's data model representation is a 3-layer hierarchy. The top of the hierarchy is VaytaUser objects (users). These objects encapsulate usernames and passwords. Each user is associated with multiple Trip objects (trips). These objects encapsulate a Date time when the trip took place. Trips are associated with users by storing the user's unique key. This backwards associativity increases the search space in the database, but provides optimal data object size. The lowest layer in the data model hierarchy is the VaytaPoint object (point), which encapsulates instantaneous driving data. Each trip is associated with multiple points, and these points are also backwards associative with their trip.

Security was not a primary concern in this web application. When users sign in, if the username and password combination is recognized in the database, then their identity is saved in a cookie. This cookie serves as validation in the application. For instance, visiting the user's front page with a recognized username set in the cookie will load that user's data.

For the user's front page, the application loads all trips from the database with a user key that matches the username set in the cookie. It displays these trips as a string and wraps them in an anchor that links to the maps display page. The maps display page takes an HTTP get parameter that is the key of the selected trip.

The maps display page loads all the points with a trip key that matches the key passed as a get parameter. It loads a Google Map using Javascript in road view mode centered on the first GPS coordinate of the trip. Then for each of the points from the trip, the script places a marker on the map at the points GPS coordinates and connects these markers with a path. The script also adds an event listener to each of the markers such that a Javascript handler will retrieve the points data on click and display them in a tabular format on the side of the map.

To upload data from the Android app, there is a servlet that accepts XML files as HTTP post requests. The XML is in a pre-determined format that represents trip data. The servlet reads the file and parses it using the SAX protocol into the application's data model representation of users, trips, and points. At the time of this writing, there is a bug that prevents uploading this data, but this accurately describes the intent and what was attempted to implement the data transfer.

4. EVALUATION AND RESULTS

After building the hardware device and completing the Android App and Google App, we were able to successfully record data from several trips in a car. Our hardware device connected without issues using the ISO protocol, saved the data, and uploaded it to the Android phone. The Android App saved the data, displayed a live view, tagged the data with GPS coordinates, and uploaded to a server using FTP. We were then able to open the XML files and successfully viewed the data collected from the vehicle. We were hoping to collect a full set of measurements every second, but using the ISO 14230 protocol we were only able to collect a full set every 3-4 seconds. Unfortunately, we did not have enough time to complete the integration with the Google App, so the data could not be uploaded directly to the Web portal.

5. DISCUSSION

5.1 Our Solution

Our solution accomplishes our basic goals. The hardware interfaces with a car's diagnostic port and gathers information. This information is transmitted through an Android phone to the web portal. The web portal can then analyze the data and, when GPS coordinates are included, plot that data onto a map. These goals have been realized with the process of collecting and uploading real-world data.

However, there are a couple of points that didn't turn out as well as we had hoped. On the hardware front, we originally had planned to make use of every protocol. In the end, we only had the ISO 9141 and ISO 14230 communication protocols working. While this does limit the usefulness somewhat, many vehicles on the road today use one of these protocols and we had no problem finding one to test with. The major disadvantage of being forced to use ISO was the speed, as a data-slice takes approximately 3.5 seconds rather than less than a second when using PWM or VPW.

Additionally, our current data collection and storage system is static. It does not provide the ability for a live feed of data. The inconsistency of PIDs supported between vehicles is another problem we did not address. A dynamic request system could check for which PIDs are available and only make those requests in order to save time. Having a page erase memory module would also allow for more flexible memory storage, as would updating the EEPROM pointers only when going into sleep mode or detecting a power down.

The main goals of the Android application were to: 1) Collect data from the hardware, 2) Store the data on the device, and 3) Transmit that data the Google App. The first two goals were completed successfully. We were able to collect data from the hardware over Bluetooth, convert the data to correct units and display it to the user, and store the data on the phone for future use. The final goal was partially successful. We were able to upload XML data to a server over FTP, but we ran out of time before we were able to send the data to the Google App.

For the web portal we had meant to perform automatic analysis and present only the most relevant information to the user. However, the web portal currently only presents a subset of the data collected from the car in raw format. Also the web portal is not the complete application that we had discussed. It only supports the primary feature of presenting users' driving data. Some of the secondary features include analyzing when a car is due for a check-up, calculating an eco-score, and sharing information between users in a community-driven web space. These secondary features were not implemented.

5.2 Future Work

While we were able to develop a working prototype that accomplished many of our goals, we simply did not have enough time to provide all of the functionality that we would have liked. If work on the device continues, the following areas should be addressed:

- **Support for multiple OBD protocols.** Currently, the device only works with the ISO 9141 and ISO 14230 protocols. The hardware should support 8 protocols total, but we were not able to get it to work reliably with any of the others.

- **Make the hardware device smaller.** The hardware device ended up being larger than we wanted in order to fit all of the components. Ideally, the device would be a small box connected directly to the OBD connector, rather than a large box connected by a cable. Our hardware device is about 4"x7" and must sit on the floor, which could present a safety hazard.

- **Dynamic Memory Structure.** The flash memory module that we used was somewhat limiting in the way it could store data. The data structure we had to use was very rigid and does not allow for varying PID support.

- **Detect car ignition state. M**ake changes to allow the hardware device to stay plugged in and detect when the car is turned on (and go into low power mode when the car is off).

- C**ollect data every second.** We would like to collect data on regular intervals, preferably once per second, regardless of the protocol that is being used. The firmware would need to be optimized for each protocol in order to accomplish this.

- **Keep time on the hardware.** In order to properly perform calculations with the data, the exact time that the measurement was taken is needed. Currently, the data is time stamped when it is uploaded to the phone, which could have very little to do with when it was taken. The hardware device does not have a real-time clock, but we should be able to approximate by periodically synching the time from the Android phone.

- F**ix bugs in data upload to website.** We did not have time to properly integrate the communication of data between the Android app and the Google app. This severely limits the usefulness of the device and should be a priority.

- **Conserve smart phone battery life.** Make changes to the Android app that takes battery life into consideration (e.g. controlling when GPS in on).

- **Make data upload an automatic process for the Android app.** The less the end user needs to do, the more likely they are to use the device. If we can let the Android app run in the background and automatically start/end trips and upload data it could greatly increase usage of the device.

- **View/clear trouble codes.** Most similar products on the market allow the user to view and clear trouble codes. We did not have time to implement this function, but it should be fairly simple and would open up another market for the device.

5.3 Hindsight

The ELM was responsible for saving a considerable amount of time, already designed and programmed for communicating with the OBD-II port. However, it also happens to be a large limitation within our design. It takes a considerable amount of space on our board and was originally designed for human interaction, translating everything into ASCII and including non-data characters. A future approach, with more time available, would exclude the ELM from its design. Instead, we would program all of the functions of the ELM onto the microcontroller. This would also allow for a smaller microcontroller, as one less UART port would be required.

There are several changes to make in the Android application. First, data upload would be automatic. Currently, the user must manually upload data for a whole trip after the trip is completed. In the future, the application would automatically stream data up to the Google App during a trip if an Internet connection was available. If no Internet connection was available during a trip, then the application would automatically batch the data up the next time it gets an internet connection. Second, battery life concerns would be considered. The GPS receiver can drain battery quickly if it is used too much. Also, since the main functions of the application run in the background, it is possible that the background service drains a lot of battery. Requiring the Bluetooth module to be on so much also reduces the battery life of the phone. Third, the user interface could be made more user friendly since most of the development was focused on other parts of the application.

There are only two noteworthy changes that could be made to the web portal. The first is to rigorously analyze and optimize the database. We implemented the database to minimize the size of data objects, but this came at the cost of additional time complexity in queries. This was not a problem in our alpha, because there was very limited data. The second change is that we should analyze security risks. We reasoned that in alpha, we can trust our users, but before releasing the product, we should perform a security audit.

6. CONCLUSION

With rising fuel prices and concerns about global warming, people are more concerned than ever with decreasing fuel usage. When drivers are provided with data about their driving habits (such as average mileage), they can easily decrease their fuel consumption, sometimes significantly. When combined with route information, drivers would be able to compare and modify routes based on mileage information. The successful development of a working prototype indicates that this type of product could be introduced into the market. If the device could be made to support most smart phone platforms, made easy enough to use, and the price brought low enough, our device could help to significantly reduce fuel usage, which would have the added benefit of reducing CO_2 emissions. Also, with a large enough user base it would be possible use competition to encourage users to reduce fuel usage even further. It may also be possible to start harvesting for specific models of vehicles and possibly predict vehicle breakdowns and suggest vehicle-specific preventative maintenance. While there is still a lot of work that could be done to refine the device and make it more robust, our project has met most of its goals and shown that it is possible to develop this type of product.

7. REFERENCES

[1] OBDLink Scan Tool, ScanTool.net LLC. Retrieved 6/8/10.
DOI= http://www.scantool.net/obdlink.html?gclid=CJbk6IOZjaICFQdkgwodxlXdTw.

[2] Rev, DevToaster LLC. Retrieved 6/8/10. DOI=http://www.devtoaster.com/products/rev/.

[3] PC Based Scan Tools, ScanTool.net LLC. Retrieved 6/8/10. DOI=http://www.scantool.net/scan-tools/pc-based/

[4] ScanGuageII, Linear Logic LLC. Retrieved 6/8/10. DOI=http://www.scangauge.com/products/

[5] OBD-II PIDs, Wikipedia. Retrieved 6/8/10. DOI=http://en.wikipedia.org/wiki/OBD-II_PIDs

[6] Lysesoft: AndFTP. Retrieved 6/8/10. DOI=http://www.lysesoft.com/products/andftp/