2011

Team Cool Beans

Poor Man's LoJack

Our solution to stolen cars and short range vehicle security systems

Bryon Wheeler	konkop@gmail.com
Philip Krebs	u0321860@gmail.com
Gregory Beck	saigon.delta@gmail.com
Simon Chulin	simon.chulin@gmail.com

Project Website:
www.GodImGood.com

Table of Contents

Team Cool Beans Poor Man's LoJack

1.1 Introduction

Our senior project has been the design and implementation of a low cost, LoJack type car communication system. Our goal has been to put together a two part system that will implement the vehicle tracking ability of the original LoJack system. As additional features it also adds remote start, remote door lock/unlock, and remote vehicle monitoring. All of these features are controlled or displayed remotely from the vehicle on an Android phone. An Arduino based microcontroller provides the vehicle based computing.

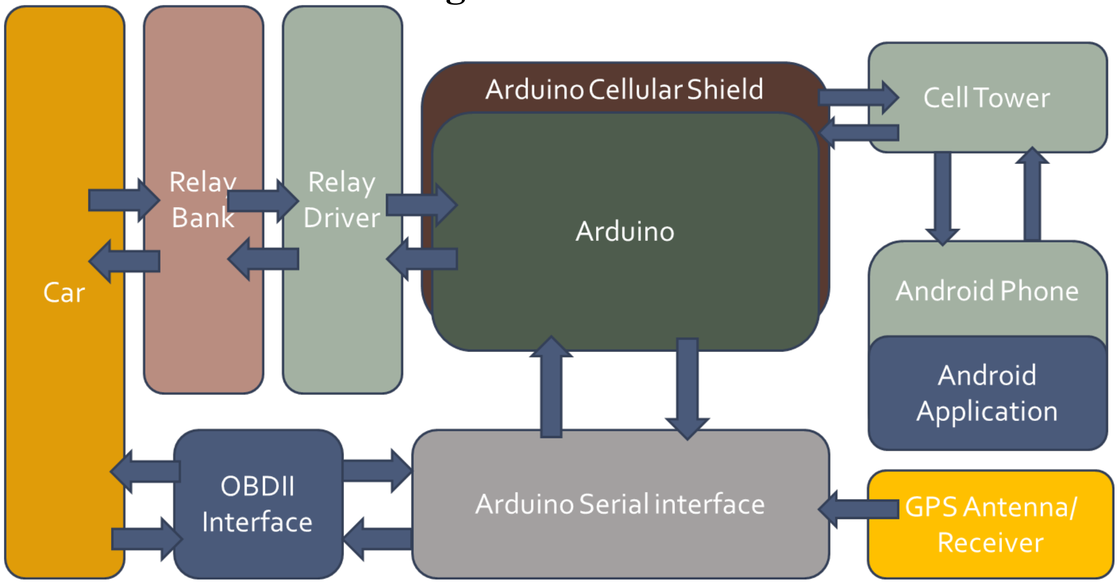
1.2 Motivation

The motivation for re-designing a LoJack type system is three-fold. While Lo Jack requires its monthly service fee, our implementation will allow the use of any GSM based wireless provider. In the United States this means using one of T-Mobile's or AT&T's prepaid or contract plans. As well, Android phones are becoming increasingly popular and while most new cars have remote features they have a limited range usually 100 feet or less. Because the Poor Man's LoJack System is cellular based range is only limited by the coverage of the provider on both the vehicle and Android phone. Additionally we have implemented extra features that LoJack does not have.

1.3 Technical Description

We have built an embedded system for the car using an Arduino (an open source micro controller with an easy to use programmer interface found at Arduino.cc) Mega board and an Arduino shield that provides a cellular interface. This allows one to simply purchase a SIM card to enable cellular communication between the Android (Google's mobile device operating system http://www.android.com) and the Arduino uses NMEA 2.0 GPS location strings from an rs232 GPS receiver to determine the car's location. The Arduino packages the GPS data, OBDI (On Board Diagnostic system version 2) data, and car status data and sends it to the Android device. The Android device receives the data and displays it on a GUI (Graphical User Interface). From the GUI a user can send commands to the car. The Android device packages the command and sends it to the car platform. The car platform parses the command and triggers a pin on the Arduino board to change logical state. The pins attached to these command lines are connected to a circuit card for current handling purposes. The circuit card has solid state relays and solid state relays that are used to lock or unlock the doors and start the car. To power our Lo Jack system we used a pre-built ATX (Advanced Technology eXtended – mid grade computer form factor made by Intel) car computer power supply. This makes sure there is stable power at all times especially during engine start as well as good surge protection.

1.4 Hardware Block Diagram



1.5 Required Functionality for Demo Day

- Remote GPS vehicle tracking using Google Maps
- Remote engine start
- Remote lock and unlock
- OBDII remote data monitoring (rpm, speed, coolant temp)

2 Hardware Design

This section describes the hardware implementation of our LoJack system.

2.1 Hardware Description

The bulk of our analog hardware design has been connecting the pieces into a system as well as the relay driver board. Our digital hardware will mostly be COTS items.

2.2 Hardware Pieces

Our hardware implementation contains the following pieces where work has been done by the group Specifically Phil Krebs designed and setup all interfacing and working with the vehicle. Simon and Greg Developed the Android interface. And Bryon Wheeler coded the Arduino that sends communication between the car and Android.

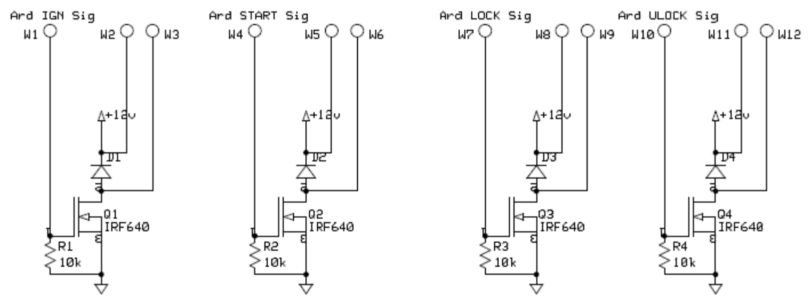
2.2.1 Relay Driver

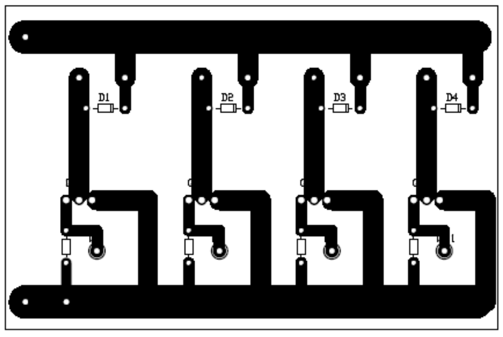
The purpose of this circuit is to amplify the signals from the Arduino output's (5V 500ma) to drive relays. As such the driver board is diode protected so that back EMF does not ruin the driver circuitry. Attached to the driver board is a bank of 3 SPDT 30A automotive grade relays. The relays are used to start the car, lock the car doors, and unlock the car doors.

The relay driver circuit was designed using a free design tool called Express PCB and Express SCH (http://www.expresspcb.com/). The program has the option to have circuit cards built from the design for around $100. We opted to etch and create our own PCB. The following design/build process was used.

1. Determine current and power requirements so we have a viable trace thickness.
2. Design circuit using Express SCH software.
3. Place and route traces in Express PCB software.
4. Print final design on overhead transparency using laser printer on darkest setting.
5. Iron printed overhead transparency on to copper base layer.
6. Remove overhead plastic effectively transferring printed circuit design onto copper.
7. Touch up circuit traces that didn't transfer well with a Sharpie magic marker.
8. Etch circuit card 15 minutes using PCB etchant (Ferric Chloride obtainable at any electronics parts house).
9. Dispose of waste Ferric Chloride at a hazardous waste collection center.
10. Drill holes for components.
11. Place and solder components.
12. Test functionality.

The schematic and PCB layout below are the final design and were used to create the PCB.





Although 2 PCBs were built for redundancy the circuit has proved to be reliable and remains in our finished design untouched.

2.2.2 Arduino Mega

The Arduino Mega was purchased online for $70.00. Using a USB cable, Arduino software downloaded from http://www.arduino.com/ and some sample code, called sketches, included with the software a person could easily get some results. Some sample examples would include blinking lights and simple music being played. With some effort using basic serial communication to the mega and being careful to a lot the right time to the right parts we were able to get Cell phone, OBDII and GPS (Global Positioning System – using satellites and a local antenna) to communicate without any problems.

2.2.3 Cellular Shield

We bought the Arduino specific cellular shield which is the sm5100 (see http://SparkFun.com/ for sm5100b cellular shield for the Arduino). We used some antennas and cables we already had but these parts can easily be purchased from http://SparkFun.com/ (you can buy the whole kit with the shield, antenna and cables or buy them separately). If they are purchased separately make sure that the antenna covers the range of your cellular provider. In the US the general band is either 800MHz or 1900MHz band. The cell card does require fairly high amperage of around 2 Amps at 5V at full load. If there is a power shortage the cell will go into a bad state and will need to be reset.

2.2.4 OBDII interface

The vehicle OBDII port operates at +12VDC logic levels. We used the ODBII diagnostics box that handled the standard translation between the car and Arduino. This device can be purchased from OBDDiagnostics.com. This device was designed to work with their designed software and hooked up to a personal computer. In order for it to work on the computer the OBDII interpreter provided a level conversion and a UART interface. In our case it needs to talk to an Arduino which is a 5V TTL (Transistor to Transistor Logic – a protocol similar to computer serial communication but communication is held at 5V). To convert this to the Arduino 5V TTL we use a cheap max232 converter bought from e-bay.com from china for $3.99 that changes the logic levels and inverts the voltages between the OBDII interpreter and the Arduino.

2.2.5 GPS Receiver

All GPS receivers work the same way. They use serial as a communication protocol and a very low baud rate of 4800 or less. For our unit we needed a second max232 to convert from serial to the Arduino TTL. All GPS units will dump all of their data onto serial out as ASCII (standard PC textual format) data at a consistent rate. More advanced versions will send out a multiple types of GPS standards. For the Arduino to gather the information it only needs to check that the first 6 characters for the GPS standard code to make sure that it matches the format we need. In our case we used the NMEA 2.0 location strings. The GPS we are using is the Garmin 16xHVS which can be purchased from Garmin.com. Our unit is a 12V unit and is powered by the car power supply. Lower cost versions exist that are designed specifically for the Arduino. These units are powered by 5V and already talk in the TTL format and therefore don't need the max232. But make sure that the correct antenna is purchased with it.

2.2.6 Hardware Power Requirements

Hardware Element	Current Draw	Native Voltage	Power
Relay Bank	N/A(only when fired)	12V	-
Relay Driver	200mA	12V	2.4W
Arduino/Cellular Shield	600mA	5V	3W
OBDII interface	N/A(OBDII powered)	N/A(OBDII powered)	-
GPS Antenna	40mA	12V	0.48W
Totals	-	-	~6W

3 Software Design

This section describes the software implementation of our LoJack system.

3.1 Description

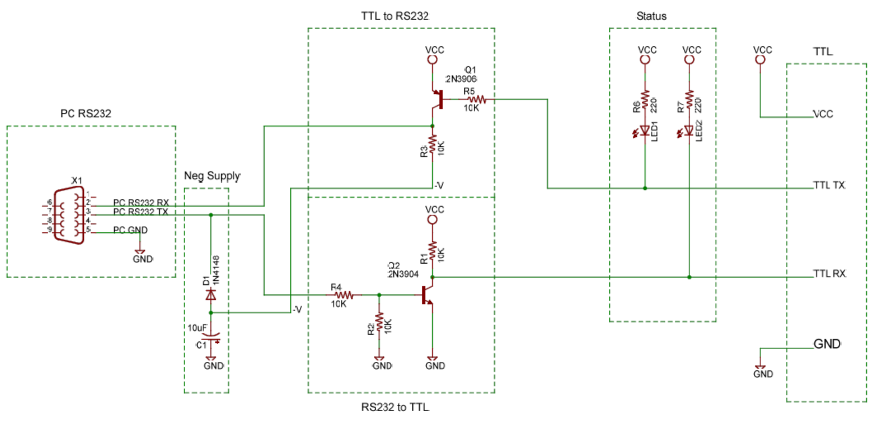
We decided to go with the simplest and most robust message interchange format between the Arduino and Android platforms and that is text messaging. The Arduino is collecting the information from the OBDII port and GPS and forwarding it to the text message.

3.1.1 Arduino LoJack MSG Based Controller

After receiving the Arduino we had to download the software located at http://arduino.cc/en/Main/Software. The first step for a new user to the android it to upload to the Arduino one of the test sketches called blinking light. It is suggested that you go to Arduino.cc and run the start tutorial to make sure a user is up to par on their unit. Next was serial communication in order to understand what was going on with the program I had it send information through the USB port using the serial mode available with the Arduino.

- **Serial communication to the GPS**

  The GPS serial communication should have been very simple. There are tutorials on arduino.cc that have the code necessary to parse through the message. A common problem that is encountered when working with the Arduino in understanding that the Arduino is TTL which is not serial communication (rs232). A simple circuit (see image below) can be built using some transistors and resistors that will allow for the receiving of serial data and converting it to TTL. This little circuit will work but would have inherent problems if we were to use the transmit line at the same time as the receive line. It is suggested that the cheap max232 chips from ebay.com from china are used for both the ODBII and GPS. With a functioning serial port the GPS code from Arduino.cc should now return on the serial port the data in an easy to read format.



- **Cell Phone Shield Sending Messages**

The cell shield is the most important part of the Arduino side. If it wasn't working well the whole project wouldn't work well. The device we received did not come with pin headers so soldering some wires to the cell shield in order for it to plug into the Arduino should be easy. Take extra care to make sure that you are soldering to the right pins and note that the cell transmits it's serial data through the tx and rx pins. Using the logic given in the tronixstuff.wordpress.com anyone can get the basic messaging going on the Arduino. If you hook up the cell and you get garbage back from it, try hooking it up to the Arduino's main tx and rx ports. Then try different baud rates such as 115,000 until you get ASCII data back from the cell card. Also there is a bug in the tutorial code, make sure that when a text message is sent when the CTRL+Z character is sent that that line is a Print() and not a Println(). If you send it using a Println() your cell will enter a bad state and you will not be able to send any more messages on the cell until it is rebooted. Lastly make sure that the power supply for the cell phone is sufficient to support the 2A it might need. If it is powered by the USB or other lower power source the shield can enter a bad state. The cell shield can take from 15-30 seconds to boot up. In order for the Arduino to work properly every time we made sure to wait for the right message from the cell before setting it up to talk to the Arduino. The messages that the Arduino waits for is the "Connected to Network" and "Cell Ready."

- **Relays**

Driving the relays was fairly simple. After the relay driver circuit was built a single line of code in the Arduino could turn on a port. Simple code had to be written to handle the features for demo day. Depending on the case (doors, starting the car, etc.) the timing of each pin had to be set. Specifically to lock the doors and Arduino pin was held high for 1000ms and to unlock a second relay was held high for 1000ms. To start the car we must lock the doors first. Next pull the ignition high wait 200ms then pull the starter high for about 1500 seconds then take it low. Leave the ignition high and the car will keep running. If you want to kill the car simply pull the ignition low.

- **ODBII**

The car actually was a little challenging. The problem was that the protocol for the device we used had no documentation. So I had started out by building an old computer that still had a serial port, running extension cords and putting the computer out in my car spying on the serial port to try to figure out the protocol. After not getting very far another team mate that was more skilled in this area hooked it up to his ford and the protocol it used was a simple PWM protocol. At that point we were able to do everything we needed with little effort. When communicating with the car we simply had to make sure that we sent a couple of startup parameters to the ODBII diagnostics chip first. After that we could send a one line command and wait a short while for the response. I also implemented a timeout here because sometimes the car doesn't respond and we don't want the Arduino to be in the waiting state for too long.

- **Receiving SMS**

The cell shield was setup to forward all incoming messages straight to the serial port. Next the shield sends two lines for each message. The first line received hold message information such as timing, sender, etc. In order to respond to the sender we will pull the sender from this line so the response message will return to them. This also would make it easy to implement security. We did not but to do this the numbers could be written to the Arduino EEPROM memory (this memory can hold data even when the device is turned off.) and when a response is received validate the numbers against this list. The next line is the message and in our case the message will be up to 160 ASCII characters. Each single character will do one specific action. All lower case letters are actions that go to the relay or GPS. All numbers and upper case letters go to the ODBII computer. I will parse through each letter and do the command that the user has requested. The Arduino will place all responses in a return message with a comma between each response. If the message length reaches 160 then it will send the message and start the next queue. There is also an auto mode were when the user sends a certain character I will send updates as quickly as I get them.

3.1.2 Android LoJack MSG Based Controller

Before being able to design for the Android, we had to set up the Android Integrated Development Environment and SDK. To do this we followed the tutorials included in the Android developer's forums (http://developer.android.com/sdk/installing.html). After installing and configuring Eclipse to work for the Android, we were presented with a plethora of different operating systems, we chose Android 2.3.1, which was the latest version running on our Android powered mobile device. Before we started programming, we had to do some research about how the Android OS works, and found that unlike the iOS, the Android uses basic Java for the back end programming, and simple XML for creating the graphical layouts such as buttons and other interfaces seen by the end user. The full programming experience with all of the hang-ups and solutions is listed below.

- **Creating the tabbed interface**

Creating the tabbed interface wasn't too difficult thanks to the great tutorials on the android development forums. We started by going to http://developer.android.com/ and finding the tutorial for creating the tabbed interface. A new project was set up to work with our Android 2.3.1 phone, and a file for each tab was created, along with a main project file that would ultimately be used to configure and launch the whole program. Each tab consists of two header pictures placed in the tab part of the page (one used when the tab is selected, and one used when the tab is active), a title, and a body, the code for which is attached After creating our three tabs (Remote, Map, and Stats), they had to be populated to be useful for our purposes.

- **Remote Tab**

The remote tab of the Android interface had to perform one main function, send messages to the Arduino telling it to perform different operations depending on the selected command. In order to create a user friendly way to do this, we had to create a four-button interface: Lock, Unlock, Start, and Stop by following the online Android Development tutorial[1] .While the functionality of these buttons is fairly self-explanatory; the creation process was a whole different story. Since the program theoretically had to run on any Android phone, some of which could potentially not contain touch screens, every button had to be made of three different graphical icons. Each button had a separate graphic for each of its three states, “up”, “over”, and “down”, these graphics were created in Fireworks[2] and exported to Eclipse[3] in the .png format. An XML instance of each button was then created in the butons.xml file, and the view was added to the CarTabActivity.java file which functioned as the brains of the “Remote” tab.

[1] http://developer.android.com/resources/tutorials/views/hello-formstuff.html#CustomButton
[2] Adobe Fireworks, CS4, 2011, [Computer Software]
[3] Eclipse, Eclipse Classic, 2011, [Computer Software]

- **Maps Tab**

Creating the Maps tab consisted largely of following online tutorials as well [4] [5]. The maps tab of the interface consisted of a map with a marker to indicate the location of the car filling the whole body of the tab. In order to make this possible, the Google Maps View was used with a single overlay to hold the marker displaying the location of the car. Getting the Google Map View to display properly was a little tricky, we had to obtain an API Key from the Google website (http://code.google.com/android/add-ons/google-apis/mapkey.html), which was custom made for our IDE based on some hash values which were auto generated within eclipse. After filling out all the necessary Google forms and getting our keys, all that was left the creation of the XML based map view and the tab was practically done.

- **Stats Tab**

The last and perhaps simplest tab was the Stats tab. The whole purpose of the tab was to simply display several text views containing the different information sent back from the car. To make the stats tab function properly, all that was necessary was to take our pre-made tabs interface and add several static text views to it that could then be populated on the fly.

- **Sending SMS**

Adding ability to use the Short Message Service (SMS) wasn't too difficult either. The main glitch that may be a little confusing on this one is figuring out that in order for the app to be able to access the cellular network, the right permissions first have to be set in the Android Manifest file. In order to send a text message, the built in SMS Manager's "Send Text Message" function was used, and as is shown in the SMSReceiver.java of the appendix, it simply sends the message, and waits for a status response to make sure that it was delivered[6].

- **Receiving SMS**

In order to receive information from the car, the program needed the ability to receive the incoming text messages and interpret them properly. This consisted of adding the right permissions to the android manifest file and essentially creating a listener thread. Breaking things down further, the process of receiving a message consists of telling the android manifest which file to call when a text message is received, making the phone handle all of the extra threading to wait for texts and then subsequently translate the message to String objects.

- **Parsing**

After receiving a text message it is sent to the parser, which interprets which tab it was sent for and sends it on its way. The parser simply scans the message for special key characters to figure out where to break up the string, then reads the first character to figure out which class the message is meant for and sends the parsed information to that class. Since Java is made to be a secure language, communication between classes is a little tricky and requires a public static class to be created and then used as a go-between for all the other classes. The parser.java method from the appendix describes more in depth details of how communication between the different classes was made possible. .

[4] http://developer.android.com/resources/tutorials/views/hello-mapview.html
[5] http://vkroz.wordpress.com/2009/07/03/programming-android-%E2%80%93-map-view-within-tab-view/
[6] http://mobiforge.com/developing/story/sms-messaging-android