University of Central Florida

Senior Design Project

Gasoline Economy Management G.E.M.

Group 8:
Pedro Betancourt
Alexander Patino
Jacob Pulliam

Supervisor:
Dr. Samual Richie

April 2015

College of Engineering and Computer Science

Contents

Contents i

List of Figures vi

List of Tables viii

1 Executive Summary 1

2 Project Description 3
- 2.1 Motivation . . . 3
- 2.2 Objectives . . . 4
- 2.3 Hardware Requirements and Specifications . . . 5
  - 2.3.1 Power . . . 5
  - 2.3.2 OBD-II to Serial Interface . . . 6
  - 2.3.3 On Board Processing . . . 6
  - 2.3.4 Data Storage and Transmission . . . 6
- 2.4 Software Requirements and Specifications . . . 7
  - 2.4.1 User Application . . . 7
    - 2.4.1.1 Bluetooth Connectivity . . . 7
    - 2.4.1.2 Data Storage . . . 7
    - 2.4.1.3 Interface and Display . . . 8
  - 2.4.2 MCU . . . 8

3 Research Related to Project Description 9
- 3.1 Similar Devices . . . 9
- 3.2 Fuel Economy . . . 10
- 3.3 Hardware Research . . . 12
  - 3.3.1 OBD-II Interface . . . 12
    - 3.3.1.1 SAE J1850 PWM/VPW . . . 13

3.3.1.2 ISO 9141-2 . . . 13
3.3.1.3 ISO 14230 (KWP2000) . . . 14
3.3.1.4 ISO 15765 (CAN) . . . 14
3.3.1.5 OBD-II PID's . . . 14
3.3.2 Communication . . . 15
3.3.2.1 I^2C . . . 17
3.3.2.2 SPI . . . 17
3.3.3 MCU . . . 18
3.3.3.1 MSP430 Family . . . 18
3.3.3.2 PIC24F16KA102 Family . . . 19
3.3.3.3 ATmega Family . . . 19
3.3.3.4 CC2541 . . . 20
3.3.3.5 MCU Summary . . . 20
3.3.4 Bluetooth . . . 21
3.3.4.1 Pairing . . . 21
3.3.4.2 Profiles . . . 22
3.3.4.3 Implementation . . . 22
3.3.5 Antennas . . . 23
3.3.6 Storage . . . 23
3.3.7 Power . . . 24
3.4 Software Research . . . 25
3.4.1 Android . . . 25
3.4.1.1 Development Environment . . . 25
3.4.1.2 User Interface . . . 25
3.4.1.3 Fragments . . . 29
3.4.1.4 Bluetooth Connectivity . . . 29
3.4.1.5 Bluetooth Low Energy . . . 31
3.4.2 MCU Firmware . . . 31
3.4.2.1 ECU Data . . . 31
3.4.2.2 Implementing OBD Requests . . . 32
3.4.2.3 Interfacing an SD Card . . . 33
3.4.2.4 Bluetooth Stack . . . 33
3.4.2.5 Bluetooth Low Energy Protocol . . . 34
3.4.2.6 BTstack . . . 34
3.4.2.7 RN4020 . . . 35
3.4.2.8 RN-42 . . . 36

4 Design Constraints and Standards 43

4.1 Design Constraints . . . 43
4.1.1 Economic . . . 43
4.1.2 Environmental . . . 44
4.1.3 Social . . . 44
4.1.4 Political . . . 44
4.1.5 Ethical . . . 44
4.1.6 Health and Safety . . . 44
4.1.7 Manufacturability . . . 45
4.1.8 Sustainability . . . 45
4.1.9 Legal . . . 45
4.2 Related Standards . . . 45
4.2.1 SAE J 2561-2001 . . . 46
4.2.2 ANSI/CEA 2040-2011 . . . 46
4.2.3 RS-232 . . . 46
4.2.4 IEEE 802.15 . . . 46
4.2.5 Software Design Standards . . . 46
4.2.6 Software Documentation Standards . . . 46
4.2.7 Software Coding Standards . . . 47

5 Hardware Design Details 48
5.1 Enclosure . . . 48
5.2 Power Supply . . . 48
5.3 OBD-II to Serial Interface . . . 52
5.3.1 STN1110 . . . 52
5.3.2 Transceivers . . . 55
5.4 Microcontroller Hardware Design . . . 59
5.4.1 Microcontroller Choice . . . 59
5.4.2 JTAG Interface . . . 61
5.4.3 RN4020 . . . 62
5.4.4 MicroSD Card . . . 63

6 Software Design Details 65
6.1 Introduction . . . 65
6.2 System Overview . . . 65
6.2.1 Software Development Tools . . . 66
6.3 Design Considerations . . . 66
6.3.1 Reusability . . . 66
6.3.2 Maintainability . . . 66

6.3.3 Testability . . . 66
6.3.4 Performance . . . 67
6.3.5 Portability . . . 67
6.3.6 Safety . . . 67
6.3.7 Assumptions and Dependencies . . . 67
6.4 System Architecture . . . 68
6.4.1 User Interface . . . 68
6.4.2 Decomposition Description . . . 71
6.5 Detailed System Design . . . 75
6.5.1 Component Descriptions . . . 75
6.6 Fuel Optimization Algorithm . . . 83

7 Project Prototype Details 85
7.1 BOM . . . 85
7.2 PCB Vendor and Assembly . . . 85
7.2.1 Printed Circuit Board . . . 85
7.2.2 Assembly . . . 86
7.3 Software . . . 87
7.3.1 Initial Firmware Design . . . 87
7.3.1.1 Main System . . . 87
7.3.1.2 Vehicle Profile Subsystem . . . 88
7.3.1.3 Bluetooth Connection Subsystem . . . 88
7.3.1.4 Onboard Storage Subsystem . . . 88
7.3.1.5 OBD-II Data Subsystem . . . 89
7.3.1.6 Power Control Subsystem . . . 89
7.3.2 Final Firmware Design . . . 89

8 Project Prototype Testing 99
8.1 Hardware . . . 99
8.1.1 Test Equipment . . . 99
8.1.1.1 OBD-II Test Bench . . . 99
8.1.1.2 Bus Pirate . . . 101
8.1.1.3 MSP430 Flash Emulation Tool . . . 101
8.2 Hardware Testing Procedure . . . 101
8.2.1 Power Test . . . 101
8.2.2 STN1110 Verification . . . 102
8.2.3 Bluetooth Verification . . . 103
8.2.4 Testing Results . . . 104

8.3 Software 104
8.3.1 Verification and Validation 104
8.3.2 Activity Testing 105
8.3.3 Service Testing 105
8.3.4 Content Provider Testing 105
8.3.5 Accessibility Testing 105
8.3.6 UI Testing 106

9 Administrative Content **107**
9.1 Budget and Finances 107
9.2 Milestones 107

A Android Framework **110**
A.1 Definitions and Abbreviations 110
A.2 APIs 112

B Final Schematics **114**

C Copyright Permissions **130**

Bibliography **133**

List of Figures

3.1 Urban Dynamometer Driving Schedule 10
3.2 Highway Fuel Economy Test Driving Cycle 10
3.3 Gallons per 1,000 miles vs. Miles per Gallon 11
3.4 OBD-II Port Pin Layout Image License CC0 1.0 13
3.5 SPI BUS: Master/Slave Connections 18
3.6 Android Layouts and Views 27
3.7 Android Activity Lifecycle Source: `http://developer.android.com/guide/components/activities.html#Creating` 28

5.1 Sparkfun OBD-II Connector - Image courtesy of SparkFun.com . . . 49
5.2 Breakout Board 49
5.3 Filter Stage 50
5.4 3.3 V Supply 51
5.5 5 V Supply 51
5.6 Switched Power Supply 52
5.7 STN1110 Configuration 53
5.8 Voltage Sense Circuit 55
5.9 CAN Transceiver 56
5.10 ISO Transmitter 57
5.11 ISO Receiver 58
5.12 SAE J1850 PWM Receiver 58
5.13 SAE J1850 VPW Receiver 59
5.14 SAE J1850 BUS+ Transmitter 60
5.15 SAE J1850 BUS- Transmitter 61
5.16 JTAG Interface Design 62
5.17 RN4020 Connections 63
5.18 MicroSD Connections 64

6.1 Launching the GEM application 68
6.2 Landscape orientation 69

6.3 Application Menu 70
6.4 Application activity diagram 71
6.5 DeviceScanActivity class diagram 72
6.6 MainActivity class diagram 73
6.7 BluetoothLeService class diagram 74

7.1 Main Firmware State Machine 93
7.2 Profile Subsystem 94
7.3 Bluetooth Connection Subsystem 95
7.4 MicroSD Storage Subsystem 96
7.5 Data Pipeline Subsystem 96
7.6 Power Subsystem 97
7.7 Final Firmware Design 98

8.1 Freematics OBD-II Emulator Interface 100
8.2 Bus Pirate UART Connection Guide 102

9.1 Schedule Page 1 108
9.2 Schedule Page 2 109

C.1 Sparkfun Electronics 130
C.2 OBD-II Pinout 130
C.3 Android Attributions 130
C.4 Creative Commons Attribution License 131
C.5 UDDS and HWFET driving schedules 132

List of Tables

3.1 OBD-II Pinout 12
3.2 OBD-II Modes 15
3.3 Mode 01 PID's 15
3.4 CAN Transceiver Comparison 16
3.5 MCU Comparison 20
3.6 Bluetooth Classes 21
3.7 Decoding ECU response of FD3FC28B from PID request 00 37
3.8 PIDs to retrieve data from ECU 38
3.9 Useful AT Commands 39
3.10 Useful AT Command Set Continued 39
3.11 Additional ST commands 40
3.12 SD Library Functions 40
3.13 Set Commands for the RN4020 41
3.14 Action Commands for the RN4020 42
3.15 MLDP Commands for the RN4020 42

5.1 STN1110 Pinout 54

7.1 Final BOM 91
7.2 PCB Vendor Comparison 92

8.1 Hardware Testing Results 104

Chapter 1

Executive Summary

In 2013 the U.S. consumed over 134 billion gallons of gasoline. On that scale even slight improvements in efficiency can save millions of dollars and prevent tons of CO_2 from entering the atmosphere. Automobile manufacturers are constantly striving to develop more efficient engines in order to meet the needs of consumers as well as regulatory requirements. Unfortunately purchasing a new vehicle in order to realize better fuel economy isn't always viable or practical. In fact purchasing a new vehicle in order to receive an incremental increase in efficiency is actually a bad idea. It is rare that the efficiency gains pay for themselves except over an exceptionally long period of time. Not to mention the large environmental impact and high energy consumption required to build a single car. In order to make progress in this realm what we need is a simple, low-cost solution that is available to every driver.

This is why we are proposing "Gasoline Electronic Management" or GEM. GEM is a simple system that is designed to be available to nearly any driver who uses a smart phone. GEM is a small device that plugs into the OBD-II port that is standard on all vehicles sold in the U.S. after 1996. The OBD-II port is a standardized hardware port through which vehicle diagnostic codes and other vehicle information can be read. We can use this information to help develop more fuel efficient driving habits. While it's prohibitively expensive to retrofit older vehicles with modern fuel efficient systems it can be very cheap and very effective to reprogram an individual's driving habits. Simple changes such as driving the correct speed and avoiding excessive acceleration and braking can increase fuel efficiency by up to 5%. For a driver that drives 15,000 miles a year if they can go from 20 miles per gallon to 21 one they save 36 gallons per year. At three dollars a gallon that's a yearly savings of $108. Not only will it save money and resources it's designed to be easy for anyone to use. When a person gets a GEM it will be very simple for them to connect the device to their vehicle and pair

it with their phone. All they have to do is connect the GEM to the OBD-II port, turn on their car and start the phone application. The phone application provides a simple, hands off user interface that provides feedback to help a driver adjust their driving habits in order to enjoy greater fuel economy. The device itself is powered by the vehicle's battery and when the vehicle is turned off the GEM enters a sleep mode so that it won't drain the battery. When active the GEM automatically pairs with the driver's phone via Bluetooth LE or, if the driver doesn't have their phone available, the GEM will store driving information so that the next time the driver connects their smart phone fuel economy and other statistics will be transferred to the application. While the GEM is designed for semi-permanent installation it also has the ability to store vehicle profiles. If a driver finds themselves behind the wheel of a different car every day they'll still be able to take advantage of the provided feedback.

Chapter 2

Project Description

2.1 Motivation

There is a strong market for for easy to use smart phone connected devices. From fitness trackers to home automation systems people are becoming accustomed to making their devices smarter. We are interested in experimenting with technologies that promote these uses. Through the design and build process for the GEM our team will develop skills in

- Android application development
- Firmware development
- Automotive electronics
- Power efficient and space saving designs
- Schematic design
- PCB layout

When looking over the options for what kind of project we could work on the most compelling ideas were those where we had to design a system that could interface with something in out in the world. Our primary motivation is to work on a project that poses some serious technical challenges. While solving these challenges we will be developing strong hardware and software skill set. On the hardware side there are a variety of subsystems that need to communicate with each other effectively. There are limitations on the radiated emissions that the device can have. Automotive power poses unique challenges, it's not always consistent and the charging and starting system can cause large voltage spikes. Not only must the device withstand

a harsh environment but it must not interfere with the vehicle or other nearby vehicles. Finally of the most challenging hardware aspects of this project is to make sure that the device can communicate with a wide variety of vehicles that contain many different implementations of the OBD-II hardware port.

On the software side the challenge is developing two systems in parallel that can communicate effectively and reliably with one another. The firmware on our device and the application on the smart phone. In order to be successful the device should connect seamlessly with minimal interaction from the device operator. This design requires that our team become very familiar with both low level C code as well as high level object oriented Java code. An important requirement of our software is that it must not interfere with the driver. This means that it must be reliable and when the vehicle is in motion must operate autonomously. If the driver is constantly having to adjust settings or changes menus on the software the system is dangerous and will end up doing more harm than good.

2.2 Objectives

We have identified several key objectives for the development of the GEM system:

- A robust, light-weight enclosing
- Vehicle profiles
- Low power
- Instantaneous measurements
- Autonomous fuel monitoring
- Simplistic interface
- Easy to read fuel data
- Applicable driving suggestions

As stated before, GEM includes a small device plugged into a vehicle's OBD-II port. Many similar fuel monitoring systems and OBD-II readers are very clunky and inconvenient to have permanently installed on a vehicle. As such, we wish to provide a small and robust enclosing which would not disturb the driver during their daily routines. Even though GEM is ideal for permanent installation, we also want to allow users the ability to easily transfer the device to other vehicles. For cases when a new vehicle is purchased or if multiple vehicles are owned. In order to accommodate for multiple vehicles GEM provides profiles for different vehicles via high-level software. In addition to these user-driven objectives, other key considerations have been identified in order for GEM to be feasible and reliable in a vehicle environment. Most

importantly the device must work for long periods of time, while drawing very low power. Furthermore, GEM records driving data in real-time, and provide autonomous fuel monitoring. Evidently, it would be hazardous to require user input, or setting adjustments, while the user is driving.

All of the information is relayed to the user via an application on their phone. As such, it is critical that the interface is intuitive and well organized. We expect users of all technical backgrounds to be able to use GEM. There is plenty of data, which must be organized and displayed in way that is understandable for users. Finally, the primary goal of GEM is to provide a solution to users for the question "What can I do to save gas?" While GEM may be recording an ample amount of data, and employing various fuel optimization algorithms in the background, actual applicable driving suggestions must be presented to the user. The ultimate objective of GEM is that it shall collect and analyze raw vehicle data, and translate it into practical advice for every day drivers.

2.3 Hardware Requirements and Specifications

The hardware systems for GEM can be broken down into the following subsystems

- Power supply
- OBD-II to serial
- On board processing
- Data storage and transmission

2.3.1 Power

Power is the most challenging aspect of the GEM. The primary requirement is that the GEM should be plug and play. The user only needs to plug the device into the OBD-II port on the car and then can leave the device plugged in without drawing down the battery when the vehicle is not in use. In order to determine that maximum allowable current draw while inactive a goal of 183 days (half a year) was set.

The energy storage of batteries manufactured in the U.S. is not specified as automotive batteries are designed for starting the vehicle only. Starting batteries are classified by cold cranking amps (CCA). Approximations for current capacity were anywhere from 30 A h to 60 A h. However, in the European Union labeling with amp hour capacity is a requirement. Varta brand batteries were taken as a typical example of

an automotive battery. The mid-range Varta "Black Dynamic" battery has a range of capacities from 40 A h to 90 A h. There is no way of knowing what battery is being used so the minimum capacity was chosen as the baseline. The Varta "Black Dynamic" type A16 battery has a capacity of 40 A h. The GEM should not draw down more than half the capacity within six months. In order to achieve this the power draw when the device is idle must be less than 5 mA.

$$\frac{20\,000\,\text{mA h}}{5\,\text{mA}} = 4000\,\text{h} \approx 167\,\text{d}$$

2.3.2 OBD-II to Serial Interface

While the port configuration is standardized there are many different communications protocols available for use. The GEM is designed to be able to communicate using all legislated protocols. This means that the GEM will be able to interface with any consumer vehicle sold within the U.S. since 1996. These protocols include but are not limited to

- SAE J1850 PWM/VPW
- ISO 9141-2
- ISO 14230 (KWP2000)
- ISO 15765 (CAN)

2.3.3 On Board Processing

In order to keep the vehicle hardware simple, low cost, and low power on board data processing will be minimal. The primary function of the MCU is to interconnect the various on-board components and prepare the data to be sent to the smart phone for processing. As such the key features of the MCU are

- µA-range idle power consumption
- serial interface
- simple to program

2.3.4 Data Storage and Transmission

The GEM is designed to operate in conjunction with smart phone software that will perform the majority of the data analyses. Because the user of the device may not

always have their smart phone with them there will be some on board storage that can cache driving data until the GEM can connect to the smart phone device again.

Caching will be to an on board SD card or other low-cost memory solution and the data will only be cached or transmitted from the cache when the device is operating. As with the other subsystems the current draw while the device is idle must be as low as possible.

The data will be streamed to the users smart phone (or other Android device) via Bluetooth LE. Since the OBD-II specification requires that the port be within 0.61 m of the driver the range of the GEM does not need to be more than 1 m. This opens up the possible choice for Bluetooth modules and antennas. In addition since there's no need for the device to negotiate a connection when idle the power consumption when idle should be less than 1 mA.

2.4 Software Requirements and Specifications

2.4.1 User Application

2.4.1.1 Bluetooth Connectivity

To setup a Bluetooth LE connection our Android application searches for devices in range and also displays previously connected devices. Once the devices have been discovered they may be paired. It is also possible for a user to terminate a Bluetooth connection with GEM from within the application. All of these functions are accessible from the first screen of the application. In the event that a Bluetooth connection is lost the application handles the error by displaying a notification and attempts to repair the connection. Once a connection is established it remains connected and receives broadcasts until GEM is unplugged or the vehicle is turned off..

2.4.1.2 Data Storage

Data sent to the Android device is stored on internal storage by writing to application files. These data files read by the application enabling the calculation of fuel economy statistics. To allow for more permanent data to be saved and recalled later the application manages data across sessions using shared preferences. This allows the application to track a users fuel economy as long as the data is stored on their device.

2.4.1.3 Interface and Display

The interface of the application consists of a display view and settings view. Only one view is displayed on the screen at a time. The "display" view contains the information the user wishes to be displayed. The settings menu controls all functions of the application. Because this application is designed for a mobile device everything is based upon a touch interface suited for the hdpi resolution and greater screens. The colors used to display text and graphics allow users that are color blind to effectively use the application.

2.4.2 MCU

The microcontroller unit in the GEM system will serve primarily as a bridge between other on-board modules. It will transmit and receive data from an OBD-II transceiver, communicate with a Bluetooth module, and handle a cache memory in an SD card. First off, the MCU must be able to request and store the vehicles identification number (VIN). Based on the VIN, the software must determine the correct OBD-II protocol and use the appropriate subset of PID's. While the vehicle is turned on, the MCU must poll the ECU, through a serial interface, continuously, with a minimum delay of 1 second and a maximum delay of 2 seconds in between polls.

The MCU shall determine if the Bluetooth module has established a connection to a device, and provide a continuous data stream if a connection has been established. If it is determined that no device is in range of the Bluetooth module, the data will not be flushed. Instead, a frame of data will be cached in an SD card. OBD-II message lengths range from 12 to 255 bytes, the minimum delay between messages received is 1 second. We expect GEM to be able to store up to 24 hours of driving data, in cases when users forget their phones or similar scenarios. Considering the most conservative scenario, where all messages are the maximum length, and are delayed by the minimum length of time we have:

$$255\,bytes \cdot 60\,\frac{polls}{minute} \cdot 60\,\frac{minutes}{hour} \cdot 24\,hours = 22.032\,MB$$

Thus, allowing some leeway, the maximum frame length shall be 25 MB, and the on-board storage must be a minimum of 25 MB.

Chapter 3

Research Related to Project Description

3.1 Similar Devices

Conducting research on products similar to our own showed that there is no shortage of devices that provide motorists with data relevant to their fuel economy. Some newer cars do come equipped with software that displays fuel economy, but our product is aimed at consumers that would like to add this functionality to their 1996 and newer vehicles. As of late 2014, the company that offers the best add-on OBD II Bluetooth device is Vinli. This Dallas company has just begun shipping their product and has placed heavy emphasis on the multitude of apps that communicate with the Vinli Basic/Complete. The apps that are available take full advantage of the On-Board Diagnostics with features such as: crash detection, GPS monitoring, fuel economy, and of course vehicle fault codes. Similar to our device, their product plugs directly into the OBD II port and transmits data using Bluetooth making it available to any smartphone within close proximity. The exact hardware specifications of their device are unknown, but it has been confirmed that the device does use an accelerometer. The starting price of the Vinli Basic is $49.99 and comes equipped with Bluetooth. The Vinli Complete is $149.99 and has Wifi and GPS capabilities in addition to Bluetooth.

3.2 Fuel Economy

To determine the fuel economy of a vehicle the Environmental Protection Agency designed a series of tests to estimate fuel consumption. The fuel economy of a vehicle based on the tests performed by the EPA is weighted such that the final value is 55% city and 45% highway. Because the EPA uses a dynamometer which neglects real world driving conditions, they expect actual fuel economy to be less than calculated. The Urban Dynamometer Driving Schedule (UDDS) in Figure 3.1 represents the driving cycle used to estimate city driving. To simulate highway fuel economy the EPA uses the Highway Fuel Economy Test (HWFET) shown in Figure 3.2.

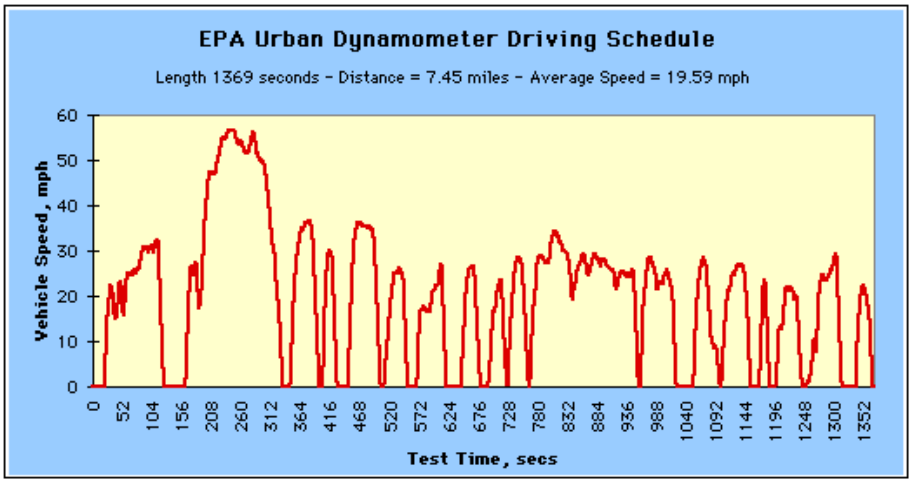


FIGURE 3.1: Urban Dynamometer Driving Schedule

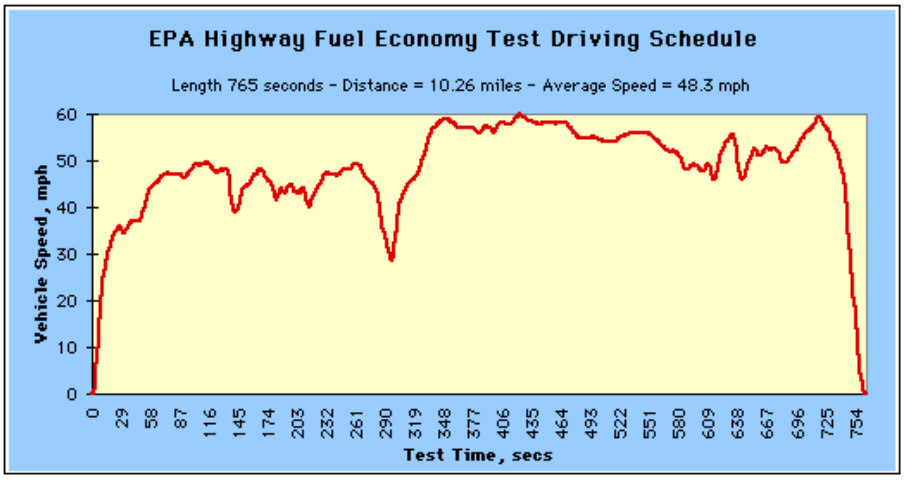


FIGURE 3.2: Highway Fuel Economy Test Driving Cycle

In 2011 the EPA and National Highway Traffic Safety Administration (NHTSA) adopted a new fuel economy label that is displayed on new cars. The most notable change is the addition of a vehicle's fuel consumption rate which is the gallons consumed per 100 miles. Figure 3.3 shows the relationship between miles per gallon and gallons per 1,000 miles. The takeaway from this chart is that there is a notable difference in fuel consumption from 10 mpg to 15 mpg (about 33 gallons) and a less significant improvement from 30 mpg to 35 mpg (about 5 gallons). This new method of measuring fuel economy is used to allow more accurate comparisons among vehicles. Our device will displays mpg and gallons per hour because our primary goal was for consumers to increase their fuel economy no matter how they measure it.

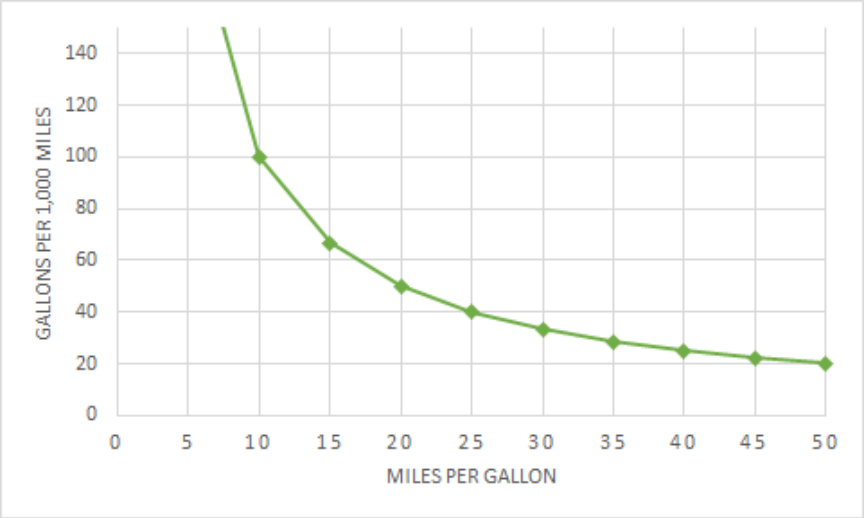


FIGURE 3.3: Gallons per 1,000 miles vs. Miles per Gallon

The EPA encourages people to increase their fuel economy to reduce CO_2 emissions, promote energy alternatives, and save money. The Gasoline Economy Management device that our team is designing shares these same goals. To increase gas mileage the U.S. Department of Energy and EPA offers the following tips:

- Drive the speed limit
- Inflate tires to the proper tire pressure
- Remove unnecessary weight from the vehicle
- Keep the engine maintained and use the correct grade of motor oil
- Use cruise control and overdrive gears
- Drive more efficiently

All of these recommendations can help increase gas mileage and lower fuel consumption, but driving more efficiently can have the greatest impact. This is why our

device provides consumers with the ability to track their fuel economy on a more regular basis. For vehicles that are not equipped to track fuel efficiency consumers can only calculate the distance traveled on a tank of gas divided by the amount of fuel consumed. This formula will provide the miles per gallon (mpg), but does not offer the level of detail necessary to improve driving efficiency. Tracking fuel economy in real-time gives feedback as to the optimum travel speed, acceleration, and effects of cruising for a vehicle.

3.3 Hardware Research

3.3.1 OBD-II Interface

In 1996 the OBD-II port was made standard on all vehicles sold in the U.S.. The OBD-II port is a 16 pin female SAE J1962 connector. The pinout is shown in Table 3.1. The layout of the pins is shown in Figure 3.4. The connector is required to be located in the passenger cabin and within 0.69 m of the steering column, except where requested by exemption.

Table 3.1: OBD-II Pinout

Pin	Signal Name	Pin	Signal Name
1	Manufacturer Discretion	9	Manufacturer Discretion
2	SAE J1850 - Bus Positive	10	SAE J1850 - Bus Negative
3	Manufacturer Discretion	11	Manufacturer Discretion
4	Chassis Ground	12	Manufacturer Discretion
5	Signal Ground	13	Manufacturer Discretion
6	CAN Bus - High	14	CAN Bus - Low
7	ISO 9141-2 & ISO 14230-4 K-Line	15	ISO 9141-2 & ISO 14230-4 L-Line
8	Manufacturer Discretion	16	Battery Voltage

There are five possible signaling protocols used by the OBD-II port. In vehicles 2008 and newer the standard is the CAN Bus (ISO 15765) and is required as one of the protocols.

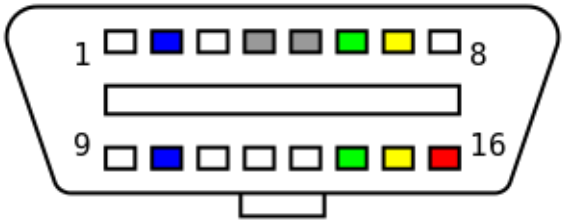


FIGURE 3.4: OBD-II Port Pin Layout
Image License CC0 1.0

3.3.1.1 SAE J1850 PWM/VPW

SAE J1850 can be broken down into two sub classes. The first is variable pulse width (VPW). VPW is a single wire bus protocol that utilizes only pin 2 of the connector. It operates at either 10.4 Kbps or 41.6 Kbps. High signal voltage is a nominal 7 V with a minimum and maximum of 6.25 V and 8 V respectively. Low signal voltage is a nominal 0 V with a minimum and maximum of 0 V and 1.20 V respectively. Start of frame is issued by a 200 µs high signal. A 1 bit is issued by a 128 µs low signal or a 64 µs high signal. A 0 bit is issued by a 64 µs low signal or a 128 µs high signal. Messages may be up to 12 bytes.

The second class of SAE J1850 is pulse width modulation (PWM). PWM is a two wire protocol that utilizes both pin 2 and pin 10 of the connector. PWM supports a speed of 41.6 Kbps. High signal voltage is a nominal 5 V with a minimum and maximum of 3.80 V and 5.25 V respectively. Low signal voltage is a nominal 0 V with a minimum and maximum of 0 V and 1.20 V respectively. The active bus state occurs when pin 2 (BUS+) is pulled high and pin 10 (BUS-) is pulled low. Start of frame is issued by an active bus state for 48 µs. A 1 bit is issued by a 8 µs bus active state within a 24 µs period. A 0 bit is issued by a 16 µs bus active state within a 24 µs period. Messages may be up to 12 bytes.

3.3.1.2 ISO 9141-2

The ISO 9141-2 standard is a two wire serial communication protocol. It operates at 10.4 Kbps. This protocol utilizes pins 7 and 15 on the connector. Pin 7 is referred to as the K-line. Pin 15 is referred to as the L-line and is optional. The K-line is the communication line and is bidirectional. The L-line is used to send a signal to the ECU on older cars as a wake-up so that communication could start on the K-line. In newer cars the L-line is not used and all signaling occurs on the K-line. A high voltage signal is a nominal 12 V with a minimum and maximum of 9.60 V and 13.5 V

respectively. Signaling is similar to RS-232 (Though with the obvious difference in voltage level). The serial settings are 10.4K baud, 8 data bits, no parity, 1 stop bit. Messages may be up to 12 bytes.

3.3.1.3 ISO 14230 (KWP2000)

The KWP2000 protocol is the same as the ISO 9141-2 protocol in all respects except that the data rate is variable from 1.2K baud to 10.4K baud. Messages may also be up to 255 bytes in length.

3.3.1.4 ISO 15765 (CAN)

The CAN protocol is modern standard for vehicle messaging. It is required in all vehicles sold in the United States since 2008. CAN utilizes pin 6 and pin 14 on the connector. Pin 6 is CAN high. Pin 14 is CAN low. The CAN high signal voltage is a nominal 3.5 V with a minimum and maximum of 2.75 V and 4.5 V respectively. CAN low signal voltage is a nominal 1.5 V with a minimum and maximum of 0.5 V and 2.25 V respectively. CAN is in the recessive state when neither pin is being driven. In this state both lines sit at around 2.5 V. CAN is in the dominant state when both lines are being driven and there is a difference of 2 V between the lines.

3.3.1.5 OBD-II PID's

Once the messaging protocol has been determined and the GEM can communicate with the vehicle messages will be sent and received using the OBD communication protocol. Most of the data that is available via OBD-II is related to emissions as the OBD-II protocol was mandated for use in emissions inspections. The GEM will be able to request data related to vehicle speed, engine RPM, fuel level, and other information useful to calculate the current efficiency of the vehicle. In order to retrieve the data from the vehicle ECU a message needs to be sent to the data bus. This message takes the form of a "Parameter ID" or PID. The PID takes the form of four bytes. The first two bytes identify the mode of the query as shown in Table 3.2 and the second two bytes indicate the actual query if applicable. For example, to request the current diagnostic trouble code (DTC) a value of 0300 is sent to the data bus. The ECU responsible for that code will respond with the currently stored DTC.

Since not all vehicles support all modes the GEM is limited to a specific subset. The only two modes that will be used by the smart phone software will be modes 01 and

Table 3.2: OBD-II Modes

Mode	Description	Mode	Description
01	Show Current Data	06	Test Results
02	Show Freeze Frame Data	07	Show Pending DTC
03	Show Stored DTC's	08	Special Control Mode
04	Clear DTC's	09	Request Vehicle Information
05	Test Results	10	Permanent DTC's

09. Mode 1 is used to retrieve specific data related to fuel efficiency as shown in Table 3.3 and mode 9 is used to retrieve the VIN (Vehicle Identification Number) to verify that the GEM has not been moved to another vehicle.

Table 3.3: Mode 01 PID's

PID	Description
00	Supported PID's (1-20)
04	Engine Load %
0C	Engine RPM
0D	Vehicle Speed
20	Supported PID's (21-40)
2F	Fuel Remaining %
40	Supported PID's (41-60)
5E	Engine Fuel Rate

3.3.2 Communication

In order to send and receive signals the GEM must be able to receive a signal using any one of the five protocols. This means that there needs to be four different transceivers that can take the voltages and signals and convert them to something that can be read by the digital input on a microcontroller. The microcontroller must then be able to take the signal, determine what protocol is being used and then receive and transmit data using that protocol. For the SAE J1850, ISO 9141-2, and ISO 14230 buses a simple comparator can be used to compare the line voltages and then output a 3.3 V signal to the OBD-II to serial interpreter.

The remaining challenge is the CAN bus. Because the CAN bus allows for speeds of up to 1 Mbps timing and control is critical. In order to transmit and receive

messages on the CAN bus a commercial transceiver is necessary. There are a wide variety of CAN receivers due to the fact that CAN is used in both automotive and industrial applications. Since there are so many options and many special features are unneeded the simplest and easiest to use in this case is the optimal solution. Two different products stand out as being inexpensive and easy to use. The first is the Microchip MCP2551 and the second is the TI SN65HVD230. Table 3.4 offers a quick comparison of the two devices.

Table 3.4: CAN Transceiver Comparison

	MCP2551	SN65HVD
Voltage	5 V	3.3 V
Signaling Rate	1 Mbps	1 Mbps
Standby Current	365 µA	370 µA
Operating Current (Dominant)	75 mA	10 mA
Operating Current (Recessive)	10 mA	10 mA
Price	$1.12	$2.30

This table demonstrates just a few key points to show how similar the two products are, further comparison of their data sheets show many more common features. Given that the two chips are so similar in their general operating parameters and both have excellent reference designs the primary deciding factor goes to price. In this case the MCP2551 is half the cost of the alternative and so it was selected for the final prototype.

Once the signal has been received from whatever bus the vehicle is using it must then be converted into a usable format. There are two different ways of handling the data. The first is to use the microcontroller to handle all of the data processing. The second option is to use a data interface chip to convert the incoming data to serial and then pass is off to the microcontroller. Each option has benefits and drawbacks.

In the case of using the microcontroller to handle the data from the transceivers the benefits are a simplified hardware design and lower hardware cost. This comes at the cost of a far more complex software design and an increased cost of software. If the MCU handles all of the data there is one less microcontroller that needs to be integrated into the design. Plus these specialized automotive data microcontrollers are typically in the ten to twenty dollar range. The drawback is that a complex software system that can detect which protocol is being used and then communicate using that protocol must be developed. It's not clear at this time how complex this part of the project would be and how much time it would take.

The benefits of using the preconfigured automotive OBD-II to serial interface are a simplified software ecosystem at the cost of hardware complexity and additional device cost. There are already industry standard chipsets on the market that handle incoming data from an OBD-II port. The most common is the classic ELM327 based on the 8-bit PIC18F2580. This device has pins for each of the individual bus lines coming in from the transceivers and a UART to connect to a host microcontroller. It accepts a standardized AT command set for serial communication and it handles protocol negotiation automatically. This last feature is a critical feature for this design. There are also clones of this chip that use faster PIC MCU's and have more features. Examples include the STN1110 and STN1170 from OBD Solutions. Ultimately, realizing that we had a very short windows in which to design an dbuild the GEM we opted to use the STN1110.

3.3.2.1 I^2C

Various devices on the GEM must be able to communicate with each other. There are some standardized ways that these components can communicate. The first of which is I^2C. I^2C is a low speed serial protocol that is used in embedded systems.

This protocol is a two wire protocol. One line is for data, abbreviated SDL and the 2nd line is for clock information, abbreviated SCL. The standard data transmission speed is 100 kbit s though in later revisions higher speeds are supported. Because of its low speed and simplicity I^2C is useful for applications where cost is a concern but speed is not.

3.3.2.2 SPI

The other popular bus for communication between devices is SPI or Serial Peripheral Interface. SPI is a four wire bus. SCLK is the serial clock. MOSI is master output, slave input. MISO is master input, slave output. and SS is slave select. The connections between master and slave are shown in Figure 3.5.

SPI is a full duplex communication system. On each clock cycle the master can send a bit on the MOSI line where the slave will receive it and the slave can send a bit on the MISO line where the master will receive it. In addition there is no upper limit on the clock speed which allows for higher throughput that I^2C. In the GEM this will be important for communication with the SD card. Given the two options, rather than having multiple protocols on the device we will use SPI for communication between components.

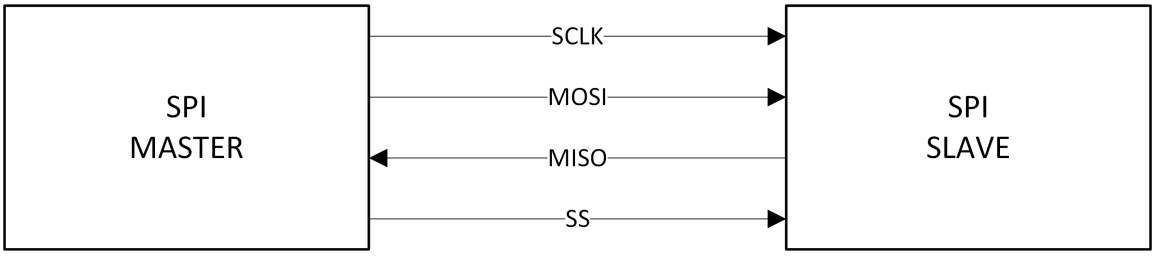


FIGURE 3.5: SPI BUS: Master/Slave Connections

3.3.3 MCU

In the GEM the MCU primarily acts as glue between the different subsystems. There is very little data processing occuring on the device. The processing and analysis is left to the smart phone application. Because of this the main features of the MCU are size, cost, power consumption and ease of use.

Because of the use of Bluetooth for communication there are different options for the MCU. There are a wide variety of Bluetooth modules that have built in 8051 microcontrollers. There is also the option to use an independent microcontroller and then use a standalone Bluetooth module or chip. The advantage of using a single chip solution that includes the microcontroller and Bluetooth SOC is that the board layout is simplified as there is one less chip that needs to be set up. This comes at the cost of a more expensive or complicated design ecosystem because some of the common tools used for programming the 8051 are extremely expensive even with educational discounts. For a standalone microcontroller the minimum requirement is that it have two available UART's, one for communicating with the OBD-II to serial interface and one for communicating with the Bluetooth module. For writing to storage it must also have SPI.

3.3.3.1 MSP430 Family

The MSP430 family of processors from Texas Instruments is easily the microcontroller that the senior design team is most familiar with. It has the advantage of having been used by all three team members. There are hundreds of variations available ranging from super simple devices with only 4 GPIO pins all the way up to controllers with 90 GPIO pins and multiple ADC's. The primary search parameters used to determine which MSP430 would be used for comparison were the UART and SPI requirements and secondarily price. Given those parameters the MSP430F23/24 family was chosen.

This series has four universal serial communication interfaces (USCI's), operates with ultra low power consumption, and has an ultra fast (¡1 µs) wake up time. In low quantities it runs about $6.00 per chip. The drawback is that there are far more GPIO and accessories than are necessary for the design. Unfortunately there is not a version of the chip that meets both the communication requirements and has a low pin count. This makes sense as there are not as many applications for a microcontroller that has a lot of communications options but little in the way of IO or ADC's and timers. An alternative to requiring so many built in communications systems would be to use a chip with only two UART's available and then bit bang the SPI in order to write to the SD card.

3.3.3.2 PIC24F16KA102 Family

PIC microcontrollers are one of the most widely used microcontrollers in the world. Again, the basic requirements were used in a parametric search to find a family of microcontrollers that would meet the needs of the project. The primary advantage we found with the PIC microcontrollers was that there were a wide variety of low pin count controllers available. Many of these were twenty to thirty pins. This would keep the board layout somewhat more simple. The specific family we chose for this comparison is the PIC24F16KA102.

The PIC24F16KA102 family has features that would be very useful to the design of the GEM. It has very low power consumption, simple design, and low pin count. The drawback is that the SPI interface shares pins with one of the UART's. In and of itself this isn't a serious problem because in any situation the system will only be using SPI to write to the SD card or the UART to communicate with the Bluetooth module. If this system were to be used though there would be more complicated hardware in order to switch between the modes as well as additional software.

3.3.3.3 ATmega Family

The ATmega and AT32 family of microcontrollers is perhaps the most familiar to hobbyists and home developers as it is the family found in Arduino boards. However the stock chip from Atmel does not include the Arduino bootloader and software. Being one of the most popular microcontrollers there is a large support community and thousands of pages of documentation on using and developing using Atmel microcontrollers.

3.3.3.4 CC2541

The Texas Instruments CC2541 is the outlier among the choices for the microcontroller. The CC2541 is a Bluetooth LE module with a built in 8051 microcontroller. The advantage here is obvious. Using the CC2541 allows us to reduce the overall complexity of the design by consolidating the MCU and the Bluetooth chip. After reviewing the other microcontrollers this seemed like the obvious choice but there is one major drawback developing using this chip. While the hardware is cheaper and simpler than the other option the software used to develop and debug is prohibitively expensive. In order to develop for and debug the CC2541 a license for IAR Embedded Workbench is required. Even for an educational license this software is many thousands of dollars. The only alternative would be to use a code size limited version or a time limited version both of which are free. For the code size limited version we would not be able to compile the Bluetooth stack if necessary as it exceeds the size limitation. For the time limited version the license is only good for 30 days. This is a dangerous limitation because even if we were able to complete the embedded software within the time limit if a bug arose at a later time or we needed additional functionality there would be no way to add to the code base.

3.3.3.5 MCU Summary

A brief comparison of the microcontrollers under consideration is shown in Table 3.5.

Table 3.5: MCU Comparison

	MSP430	PIC	ATmega	CC2541
Voltage	3.3 V	3.3 V	5 V	3.3 V
Current-Active	270 µA	8 µA	8 mA	18.2 mA
Current-Standby	0.3 µA	2 µA	0.8 µA	1 µA
Package	QFP/QFN	DIP/SOIC/QFN	DIP/QFN	QFN
I/O	4 USCI	2 USART	2 USART	2 USART+BLE
RAM	4 kB	1.5 kB	4 kB	8 kB
Flash	56 kB	16 kB	64 kB	256 kB
Price	$5.92/ea	$2.88/ea	$8.65/ea	$6.25/ea

The price and the voltage/current requirement are enough to knock the ATmega out of the running. If the proper software was available the ideal choice would be CC2541 however without IAR Embedded Workbench we also have to ignore that as an option. This leaves the PIC or the MSP430. If it were only over price we would go with the

PIC but since our team has familiarity with the MSP430 and it also has the four USART's it is ideal for our project. If we were to be designing this for production the choice would be more difficult and we would likely lean towards

3.3.4 Bluetooth

In order to get data from our device to the smart phone application we chose to use Bluetooth. Bluetooth is a short range wireless standard originally intended for wireless RS-232 serial data. It operates in the 2.4Ghz range. There are three power classes for Bluetooth as shown in Table 3.6.

Table 3.6: Bluetooth Classes

Class	Maximum Power	Range
1	100 mW	100 m
2	2.5 mW	10 m
3	1 mW	1 m

3.3.4.1 Pairing

Bluetooth devices operate as a master and a slave. Typically in a Bluetooth use case a user wants the two devices in communication to communicate securely but at the same time avoid having to manually connect the devices whenever they are in range. The two devices should automatically connect and start sharing data when in range. In order to achieve this devices go through a pairing operation which is usually started by a specific request from the user. Once the devices have gone through the pairing operation they are securely bonded and can connect to each other whenever they're in range without further interaction from the user.

There are different ways to pair Bluetooth devices depending on the version of Bluetooth being used. Up to Bluetooth V2.0 the pairing mechanism was to have each device provide a 4 digit pin code and if both codes matched the devices would bond. The limitation is that the device without an input, say a headset or an OBD-II module will rely on a preset PIN code stored in memory. This opens up security concerns and the possibility of man in the middle attacks. However it is easy to set up and maintain. In Bluetooth V2.1 and greater the pairing mechanism is known as Secure Simple Pairing (SSP). SSP offers multiple levels of pairing from "Just Works" where two devices can discover each other and pair without any user input to "Passkey

Entry" where a user must enter a specific code on at least one of the devices in order to pair them.

In our use case, since the device connects directly to the data bus on the vehicle and the vehicle data bus is notorious for its lack of security, we are opting for the passkey entry. The device can store a code that the user will have to enter on their smart phone the first time the device is paired.

3.3.4.2 Profiles

Bluetooth devices use the concept of "profiles." Profiles define the possible applications and behaviors that two devices may use to communicate with each other. According to the Bluetooth Developer Portal the minimum information required by a Bluetooth protocol is

- The dependency on other profiles
- User interface formats
- What parts of the Bluetooth protocol stack are used by the profile

There are a wide range of profiles from printing profiles to video imaging. In our case the data being transmitted is all text based and very basic so we will likely be implementing the SPP or Service Port Profile that defines how to act as a virtual serial port connection between two Bluetooth devices.

3.3.4.3 Implementation

Since Bluetooth is such a common standard there are many ways that we can implement it on our device. Initially we had planned on using Bluetooth Low Energy but the number of smart phones that can use BLE is still limited. Instead we will be using Bluetooth Classic. This opens up the number of devices we will be compatible with. This leaves us with two options. Should we integrate a prebuilt Bluetooth module or should we develop a module based upon an existing Bluetooth chip? Texas Instruments makes the CC2564 Bluetooth controller chip that is ideal for our application. The difficulty is that it requires advanced PCB layout and soldering techniques. The chip package is a QFN package which is a no lead package. In order to solder the chip to the board we would need to use either hot air or a reflow oven which our team has limited experience with. In addition we also would need to move to a four layer board so that we can easily create controlled impedance traces for the antenna. This is a

daunting proposition. The alternative is to buy a module that already contains the CC2564 as well as the antenna and then just connect to it using the UART interface. This costs about four times as much, the modules are approximately $20.00 while the CC2564 is roughly $5.00. The benefit to designing an implementing the Bluetooth solution is that it's a more cost effective design and the team members gain experience in areas that may otherwise be lacking.

Given that our schedule plans on a prototype PCB being complete by February 2015, if relatively bug free an attempt can be made to develop a Bluetooth solution based on the CC2564.

One of the modules that could be used in this case is the Panasonic ENW-89842A2KF. This device incorporates the CC25xx series Bluetooth module so that swapping the module for our own system would not require extensive redesign or reprogramming.

3.3.5 Antennas

The CC2564 requires an external antenna. The antenna recommended by Texas Instruments is an "inverted F" type antenna for which they provide exact dimensions and board layout in their design guide. This design requires the ground plane and all others layers to be clear underneath the antenna. In addition it requires a $50\,\Omega$ impedance trace to connect to the CC2564. The advantage of this antenna is that it is omni-directional so we don't need to be concerned with the orientation of the GEM when it is connected to the vehicle.

The alternative to having the antenna drawn out on the PCB is to use an external chip type antenna. The Fractus Compact Reach Xtend Chip antenna is a small form factor chip antenna design for the 2.4Ghz range. It utilizes space-filling fractal shapes to shrink the size of the antenna while maintaining good output properties. Compared with the F Antenna it is 57% less efficient but it takes a fraction of the board space.

3.3.6 Storage

The GEM needs to be able to store data when the user does not have their smart phone available. On board storage in the MCU memory is not sufficient because should the device become unplugged or otherwise disconnected from the battery the memory will be lost. A simple solution is a cheap, easily available, easy to use, non-volatile memory. SD cards meet all of these requirements. In order to interface with

the SD card the MSP430 can utilize the SPI bus. Texas Instruments provides a SD card library which can be used to write data to the card.

3.3.7 Power

The power requirements for this design are somewhat complicated. The system requires a vehicle battery voltage, nominally 12 V, a 7 V supply for the comparators for the SAE J1850 VPW system, a 5 V supply for the comparators for the SAE J1850 PWM system, a 3.3 V supply for the microcontrollers, and finally a 1.8 V supply for the CC2564. Since the two SAE J1850 protocols use the same input and output and only differ in their voltages it is required that the power supply on those inputs is switchable between 5 V and 7 V

In addition because of the nature of the power supply on a vehicle we also need to have filtering and transient voltage suppression. Filtering can be done by capacitors on the input to the power stage. Transient voltage suppression can be done with the addition of TVS diode pairs on the input stage to shunt current when the voltage rises beyond a set point.

In order to meet the power requirements multiple voltage regulators will need to be used. For these stages we need high efficiency as we have ultra low power requirements when the vehicle is powered off and we have limited heat dissipation when the vehicle is on and the device is powered.

Given the typical operating current for the various on board devices we arrive at a current consumption of 223 mA, add a 50% safety margin and we have 334 mA consumed while the device is fully active. If we were to drop the battery voltage to 3.3 V via a linear regulator we would have to dissipate 2.9 W.

The optimal solution in this case it to regulate the voltage in different ways in different sub-systems. For the microcontrollers and comparators a switching regulator can be used to bring the batter voltage down to 5 V and then a linear regulator to bring it down to 3.3 V and 1.8 V. A separate system can be used to to generate and switch between the 7 V and 5 V signaling voltages required by the SAE J1850 protocols. These systems will be examined further in the power design section of this document.

3.4 Software Research

All research conducted related to software, whether it is high level mobile application software, or low level MCU firmware is contained in this section.

3.4.1 Android

The following sections describe development within the Android mobile operating system.

3.4.1.1 Development Environment

The entire front-end system for GEM will be in an Android mobile phone application. Android is an open source framework for a multitude of mobile devices whose development is led by Google. An SDK, API libraries, and several developer tools are provided by Google, free of charge. Apps are developed in Java using XML layouts for visual elements. There are two primary development environments to choose from. The more traditionally used is the Eclipse IDE with the Android Developer Tools (ADT) bundle. This provides all the basic tools needed to develop an Android app, bundled into a very popular IDE in Eclipse. The build has been around for several years, is heavily tested and stable. The other environment is the more recently released Android Studio. It is built on the IntelliJ platform and provides more features, a better feel, and more tools, but it is currently in beta testing and may have some bugs developers will have to work around. We have opted for the increased functionality and ease of use of Android Studio to develop the mobile application for the GEM system.

3.4.1.2 User Interface

The Android library provides a set of built in components to build graphical user interfaces, including UI objects and layouts, as well as a flow framework to guide development. The framework is based on a hierarchy of 'ViewGroup' objects, which contain 'View' objects. A 'View' is essentially a visible rectangular element on the screen, while a 'ViewGroup' is an invisible container composed of Views or other ViewGroups.

Using XML, the look, feel, and alignment of all visual elements can be controlled by the developer. XML layouts control the default appearance of application elements, and can be dynamically modified programmatically. The Android XML vocabulary allows developers to use a variety of layouts, Views, and Widgets. Documents must contain one root View or ViewGroup which define the positioning of all child content. Some of the most used layouts include linear layout, relative layout, table layout, list view, and grid view, shown in Figure 3.6.

Each layout provides its own set of parameters, and attributes which can be used to control positioning, visibility, size, margin, padding, and several other characteristics. Linear layouts, relative layouts, and table layouts are primarily used for predetermined non-dynamic layouts. Using a ListView or GridView allows for dynamic population of elements at runtime. In order to do so an Adapter class must be used. An Adapter serves as a bridge between external data sources and an AdapterView. ListView and GridView both provide an AdapterView implementation. AdapaterView has methods that facilitate interaction, such as click listeners and other event handlers.

All of these UI elements are bound together by an 'Activity.' Activities are the driving principle of Android development. An activity is just one screen which a user can view or interact with. Activities are visibly populated by the UI elements previously described, but may also perform background computations and tasks not visible to the user. Activities are governed by a unique lifecycle, in which each state is accessible using a different method. When an activity is first launched it looks to its onCreate method, this is where the UI is constructed and any essential objects or values are initialized. Activities may be displaced to a background stack when another activity is initiated. Before the other activity initiates, the onPause method is called in the activity that is about to shift to the background. This state is usually used to quickly save any important data the activity modified. When an activity returns to execution from a paused state the onResume method is called. When an activity signals another activity to terminate, the onDestroy method of the activity to be destroyed is called. There are several other lifecycle states that are executed with a unique method, which are all displayed in Figure 3.7.

Activities can't implicitly communicate with other activities, even if they stem from the same application. As such, any piece of data that is needed by more than one activity must be explicitly passed via Intents. An Intent is an Android specific abstract structure which can carry an action to be performed and data. Activities can start other activities through an Intent. Primitive data types can also be passed through an Intent, and so can objects as long as they are serializable.

FIGURE 3.6: Android Layouts and Views

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

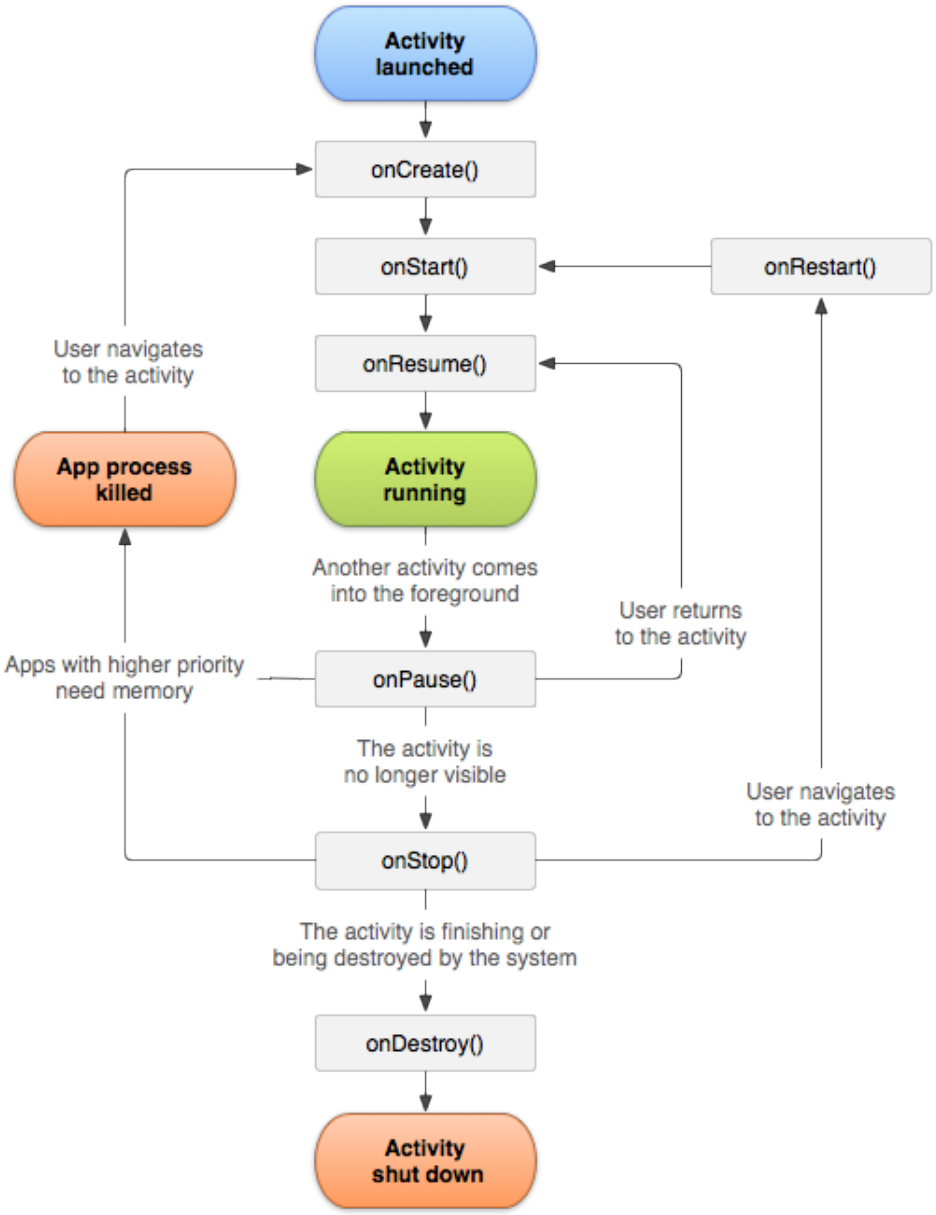


FIGURE 3.7: Android Activity Lifecycle
Source: `http://developer.android.com/guide/components/activities.html#Creating`

3.4.1.3 Fragments

While some data can be passed from activity to activity through an Intent, and each activity can contain its own UI layouts, GEM aims for a simple and sleek design with modularity across all activities. The use of 'Fragments' may greatly facilitate these design goals. A fragment is an Android object, which can be viewed as a piece of an activity. It can be a certain computation, or a UI portion. Activities can be built from a collection of fragments, and fragments can be re-used in different activities. Fragments have their own independent lifecycle and layout, but still abide to the state of the activity they reside in. As such, fragments are particularly useful for duplicating a component in all activities in an application (such as a menu) and providing a consistent look and feel throughout an application, which GEM hopes to achieve. Fragments are also useful for performing background computation, separate from a UI thread. This can also be employed in GEM while fuel calculations are made, saving some time by making calculations while fetching data.

3.4.1.4 Bluetooth Connectivity

Android provides Bluetooth network stack support through a native API. This includes Classic Bluetooth and, starting from Android 4.3, Bluetooth Low Energy (BLE). BLE is of interest in the development of GEM because it offers much lower power consumption than Classic Bluetooth. The downside, however, is that only about 24% of Android phones are running version 4.3 or higher. This can alienate many potential users due to not having a phone which can support BLE. On the other hand 99% of all Android phones have Classic Bluetooth support. Thus, the tradeoff is consumer outreach for power efficiency.

All API tools to establish, configure, and manage a Bluetooth connection are available in the package 'android.bluetooth.' The classes and interfaces of significance are as listed below.

- BluetoothAdapter
- BluetoothDevice
- BluetoothSocket
- BluetoothServerSocket
- BluetoothClass
- BluetoothProfile
- ServiceListener

BluetoothAdapter is the object used to represent the phones local adapter. To retrieve an instance of it the methods getDefaultAdapter or getSystemService are used (depending on the API version). Once there is access to a BluetoothAdapter object, it can be used to discover devices, retrieve all currently paired devices, and interact with already paired devices. To retrieve the list of connected device the getBondedDevices method is used. The startDiscovery method is used to start searching for devices. All of these methods require special permissions in order to be used in an Android device. The BLUETOOTH and BLUETOOTHADMIN permissions must be specified in the application manifest file. The application should also check if Bluetooth is enabled on the device by calling the isEnabled method, and use an Intent to request for Bluetooth to be turned on and enable discovery if it isn't. Once startDiscovery is called, all Bluetooth enabled devices in the area will be queried for a device name and a unique MAC address. This information can be used to establish a connection. After the devices are paired for the first time, all information is saved within the Bluetooth API and future connections can be performed without discovery.

Once devices are paired, a list of objects called BluetoothDevice can be retrieved with the method getBondedDevices. The BluetoothDevice class essentially just holds all the physical information of a device including its MAC address and name. Just having devices paired doesn't mean that a connection for communication is established. A client-server mechanism has to be established also. One device has to have an open server socket while the other device connects to it. This is done through the BluetoothSocket and BluetoothServerSocket interfaces. These interfaces establish a Serial Port Profile (SPP) between devices that allow streaming of data through Bluetooth. One device should create a listening BluetoothServerSocket and the other device will create a BluetoothSocket to connect to available servers.

With a connection established the read and write methods can be used to write bytes to an input stream and output stream. This data can be accessed using the getInputStream and getOutputStream methods. The BluetoothClass class is useful in managing connected devices. The method getBluetoothClass is used to attain the class for a device, and the class provides information about the device type, services, and capabilities. Additionally, the Bluetooth API has support for what is known as Bluetooth profiles, which are interface specifications for Bluetooth-based communication between devices. The BluetoothProfile and ServiceListener interfaces are used to support a Bluetooth profile on an Android device.

3.4.1.5 Bluetooth Low Energy

BLE is able to transmit small packets of data, called attributes, under a generic attribute profile (GATT). The GATT sits on the Attribute Protocol layer (ATT) of the Bluetooth stack. These attributes are identified by a Universally Unique Identifier (UUID). Data from the UUID is transported between paired devices as characteristics and services. Similar to the client-server mechanism in Classic Bluetooth, BLE devices have a central-peripheral system. The Android API only allows for mobile devices to be configured as the central role. Peripherals advertise themselves and a central device looks for advertisement. This applies to the BLE connection at the physical layer. Once the physical connection has been established, BLE devices communicate using GATT servers and clients.

Using the Android API to establish a BLE connection requires a similar initial setup to Classic Bluetooth. The BluetoothAdapter object is still used and the same permissions have to be set in the Android manifest file, with the addition of the BluetoothLE permission. From there on out, a different set of methods specific to BLE are available in the API. Also a new class (API level 18), BluetoothManager, is used to retrieve the BluetoothAdapter. To scan for BLE devices the method used is startLeScan rather than startDiscovery in Classic Bluetooth. In order to interact with discovered devices, the connectGatt method must be used, which returns a BluetoothGatt object used to relay attributes between devices. Once communication is established activity is monitored via the broadcastUpdate method, and attributes can be read and written. Additionally, the app can be notified when a characteristic on the GATT server is changed using the setCharacteristicNotifaction method.

3.4.2 MCU Firmware

The subsequent sections include all information relevant to the construction of firmware for the GEM system.

3.4.2.1 ECU Data

As previously discussed, vehicle data from the engine control unit (ECU) is accessible via PID requests sent through the OBD-II data bus. In the GEM system PID requests and message retrieval will be handled with microcontroller software. As such, ECU message encoding and decoding is of paramount importance. Since vehicles provide support for different modes and PIDs, the MCU must determine which of them are

supported for the vehicle GEM is currently connected to. This information can be retrieved with PIDs 00, 20, 40, 60, and 80 in mode 01. These PIDs each return four bytes of data representing which of the next 32 PIDs are supported, with the MSB representing PID number 1 and the LSB representing PID number 20 (for the PID 00 request). An example of decoding PID request 00 is shown in Table 3.7

GEM is only equipped to handle OBD-II data, but not all vehicles comply with OBD-II standards. The PID request 1C (mode 01) returns a single byte describing which OBD standards the vehicle's ECU complies with. Return values of 1, 3, 7, 9, 11, and 13 all represent OBD-II compliance. There are also a plentiful amount of PIDs relevant to fuel efficiency. PID 04 (mode 01) returns a calculated engine load value in one byte, given as a percentage up to 100. PID 02 (mode 01) returns the engine RPM in a two byte value. PID 0D (mode 01) returns the vehicle speed in two bytes measured in kilometers per hour. PID 10 (mode 01) returns the MAF air flow rate in two bytes in units of grams per second. PID 11 (mode 01) returns the vehicles throttle position as a percentage in a one byte value. PID 1F (mode 01) returns the run time since engine start, measured in seconds, in a two byte value. PID 2F (mode 01) returns the fuel level, as a percentage, in a one byte value. PID 5F (mode 01) returns the engine fuel rate in a two byte value measured in units of liters per hour. Additionally, PID 02 in mode 09 is used to obtain the vehicle identification number (VIN). The VIN is returned in 17-20 ASCII-encoded bytes, with null characters padding if necessary. Mode 09 also has a PID 00 which returns supported PIDs of the ECU (just like mode 01's 00 PID). Also, PID 01 (mode 09) provides the VIN message count for PID 02. A summary of all PIDs is shown in Table 3.8.

3.4.2.2 Implementing OBD Requests

The MCU will be performing OBD requests via an OBD to UART interpreter chip. Most interpreter chips provide a set of similar commands which can be issued through software. One candidate OBD interpreter chip for the GEM system is the STN1100 which provides an additional command set from a host MCU. The MCU can communicate with the interpreter chip via ASCII commands. Once a connection is established the interpreter chip prompts for an input by transmitting the '>' character. The MCU can then respond with a set of AT commands by outputting the ASCII characters 'AT' through a UART. A set of useful AT commands are shown in Tables 3.9 and 3.10. For the STN1100 specifically there's an additional ST command set, shown in Table 3.11. Any data sent to the interpreter chip which does not begin with 'AT' or 'ST' is assumed to be an OBD commands. Possible OBD commands are shown in Table 3.8. The MCU only needs to send 2 bytes of data to the interpreter

chip to perform an OBD request, the first byte is the mode, and the second is the PID.

3.4.2.3 Interfacing an SD Card

Most SD Cards sold are shipped partitioned with a FAT file system. A file system provides an architecture for writing and deleting files on a storage device. SD cards need a file system so that an operating system can read its data. This is standard for applications such as viewing pictures taken from a camera on a personal computer. However, in an embedded environment, a file system results in a significant amount of memory overhead. Apart from taking up space on the SD card itself, the microcontroller software would need to import additional libraries for opening and closing files. This can be detrimental to small scale designs where there is no operating system, such as GEM. Alternatively, the SD card can be formatted and raw data may be written to it in SPI mode via software. Only the host MCU will have access to this raw data, which is all that is needed for an embedded storage application. Raw data is usually written to the SD card in 512 byte blocks. An SD card can be controlled by a host MCU through a synchronous 1-bit command line. Commands are defined by the SD standard as 48 bits in length. The SD protocol works as command-response system, where all commands are initiated by the master. Commands take the form of "CMDXX," "XX" being the unique command number. In SPI mode a 6 byte frame is sent through the data in port. The frame is always initialized with a 0 bit, and ends with a 1. Most commands, such as a card write, will return a 1-byte token providing status information. Texas Instruments provides a library for interfacing a microcontroller with an SD card via the SPI interface. A set of functions is available to make initialization, card reads and writes, and commands easy to implement. All that needs to be adjusted are the pin settings specific to the microcontroller in use. The most useful of these functions are highlighted in Table 3.12.

3.4.2.4 Bluetooth Stack

Bluetooth connections require a standard protocol which must be followed in order to achieve connectivity between devices. This protocol is an onion-like set of layers (stack) implemented in software. As defined by the standard, a Bluetooth stack needs to provide a minimum of four layers.

- A physical transport layer for transferring data bits
- An HCI layer for the management of physical conditions to and from devices
- An L2CAP layer for managing logical channels of established connections

- Any additional Bluetooth services on top of L2CAP layer that adds functionality, usually the Service Discovery Protocol (SDP)

The Bluetooth protocol stack is prevalent in laptops and smart phone devices, and fairly easy to implement on higher level applications. However, implementing a Bluetooth stack in an embedded environment with limited recourses leads to some difficulties. There are however, open source libraries designed for limited memory applications. One such API is called BTstack. BTstack can be used in conjunction with a CC2564 chipset, or a PAN1323 module to provide Bluetooth connectivity for an embedded system. There is also the option of using a Bluetooth module which has a dedicated chip which implements the Bluetooth stack, this avoids the issue of implementing a Bluetooth stack with limited recourses, but accrues higher hardware costs and adds an additional chip that takes board space and has to be interfaced with. Two such modules are the RN4020 low energy, and the RN-42 class 2 Bluetooth modules.

3.4.2.5 Bluetooth Low Energy Protocol

Bluetooth Low Energy (BLE) is an extension of the Bluetooth stack protocol starting from the Bluetooth 4.0 standard. BLE provides far less power consumptions than Classic Bluetooth at the expense of throughput. BLE provides easier initialization, discovery and connection. BLE is a connectionless client/service architecture as opposed to Classic Bluetooth, and uses a Generic Attribute Profile to simplify software. Interfacing is also meant to be simpler with BLE. In the BLE protocol the L2CAP protocol serves as a backend for simpler services to be bound to. One of these is the Low Energy Attribute Protocol (ATT), which provides a similar feature to the SDP but is simplified and specially adapted for Low Energy Bluetooth applications. The generic access profile (GAP) also sits on the L2CAP layer. GAP is used to manage, scan, and establish connections between devices. Most BLE implementations also rely on the generic attribute profile (GATT), which provides a framework for communications between paired devices. All additional services are built on top of the GATT, and their characteristics are defined by the GATT.

3.4.2.6 BTstack

BTstack is an open source portable Bluetooth stack available on Google Code. It was designed to be implemented on recourse limited devices. It works well in 16 bit embedded environments and is highly configurable. The entire stack footprint

only needs 32 kB of flash memory and 4 kB of RAM. While this is exponentially smaller than a typical Bluetooth stack, it most likely still needs a microcontroller dedicated to it only. BTstack supports Classic Bluetooth as well as Bluetooth 4.0 Low Energy specifications and works as a peripheral or in a central role. BTstack is a collection of interacting state machines, which can be single threaded or multi-threaded. Protocols and services can be initiated through library functions provided by BTstack.To initialize and allocate memory for services, active connections, and remote devices the function *btstack_memory_init* is called. A run loop is then initiated (in single thread mode) by the function *run_loop_init*. Bluetooth hardware control is done through a provided struct *bt_control_t*. Handling of the HCI Trasnport layer is done through the *hci_transport_t struct*. To change the BAUD rate and for other UART controls the *hci_uart_config_t* struct is modified. Remote device connection information is stored in *remote_device_db_t*. For each additional layer needed, its own specific initiation function is provided by BTstack. Packets are handled by the L2CAP layer via the *l2cap_register_packet_handler* function. For a remote device to become discoverable the application calls *hci_discoverable_control* passing in an input parameter 1. The *hci_write_local_name* command is used to set a device name. In order to scan for remote devices the *hci_inquiry* command is triggered. Finally, devices can be paired depending on which profiles and layers the user wants to implement. For a minimum requirement connection, a device provides an L2CAP service, and BTstack initiates communication with the service by using the *l2cap_init* function, followed by creating an outgoing channel using *l2cap_create_channel_internal*.

3.4.2.7 RN4020

The RN4020 is a complete Bluetooth module with an onboard Bluetooth Low Energy 4.1 stack. Using the RN4020 or the RN-42 would reduce Bluetooth related MCU software to the task of interfacing with the module. The RN4020 has a command API, which are issued by the host microcontroller as ASCII characters. Commands are sent from the host MCU through a UART control interface. The RN4020 divides all commands into 8 types, of which 3 may prove useful for the development of the GEM system:

- Set/Get Commands
- Action Commands
- Microchip MLDP Commands

Set/Get commands start with the ASCII character 'S' followed by an identifier and an input parameter, separated by a comma. They are used to configure specific

module functions. All set commands have a corresponding Get command to output the set configurations. Get commands have the same format as its corresponding Set command excluding the input parameter. A module reboot is required to guarantee new settings are in effect. A set of useful Set commands are shown in Table 3.13. Action commands are each unique but are mainly used to display information or initiate functionality, a set of action commands are displayed in Table 3.14. A private Microchip Low-Energy Data Profile (MLDP) is embedded into the RN4020 and used to send UART data bytes wirelessly to connected devices. A couple of commands are also available for this profile, shown in Table 3.15.

3.4.2.8 RN-42

The RN-42 is another all-in-one Bluetooth module. This module supports only Classic Bluetooth (standard 2.1). In contrast to the RN4020, the RN-42 provides more built in profiles and flexibility. The RN-42 includes the commonly used GAP, SDP and SPP profiles instead of a private MLDP profile. A Get/Set command syntax, comparable to the RN4020's, is used by this module. However instead of MLDP commands the RN-42 operates in two basic modes, command mode or data mode. When in command mode, similar commands to the RN4020 can be given. When in data mode, the module simply operates transparently as a data path between the MCU and a connected device, constantly sending out Bluetooth packets. A simpler command set and initialization is provided by the RN-42, but is considerably more expensive and doesn't support BLE.

Hex	Binary	PID #	Supported?
F	1	1	Yes
	1	2	Yes
	1	3	Yes
	1	4	Yes
D	1	5	Yes
	1	6	Yes
	0	7	No
	1	8	Yes
3	0	9	No
	0	0A	No
	1	0B	Yes
	1	0C	Yes
F	1	0D	Yes
	1	0E	Yes
	1	0F	Yes
	1	10	Yes
C	1	11	Yes
	1	12	Yes
	0	13	No
	0	14	No
2	0	15	No
	0	16	No
	1	17	Yes
	0	18	No
8	1	19	Yes
	0	1A	No
	0	1B	No
	0	1C	No
B	1	1D	Yes
	0	1E	No
	1	1F	Yes
	1	20	Yes

TABLE 3.7: Decoding ECU response of FD3FC28B from PID request 00

Mode	PID	Description	Decoding
01	00 20 40 60 80	Returns which PIDs are supported by the vehicle's ECU	4 bytes for the next 32 PIDs. See Table 3.7
	1C	Describes which OBD standards the vehicle's ECU complies with	Returns a 1 byte value. Values 1, 3, 7, 9, 11, 13 indicate OBD-II compliance
	04	Provides an engine load percentage	1 byte value
	02	Engine RPM	2 byte value
	0D	Vehicle speed in km/h	2 byte value
	10	MAF air flow rate in gm/s	2 byte value
	11	Vehicle throttle position as a percentage	1 byte value
	1F	Run time since engine start in seconds	2 byte value
	2F	Fuel level input percentage	1 byte value
	5F	Engine fuel rate in L/h	2 byte value
09	00	Same use as mode 01	See Table 3.7
	02	Returns VIN	17-20 bytes. ASCII- encoded. Null character padding when necessary
	01	VIN message count for PID 02	1 byte value

Table 3.8: PIDs to retrieve data from ECU

AT Command	Functionality
Carriage Return (0x0D)	Repeat the last performed command. Provides the fastest rate for obtaining consecutive values, such as consistently polling for the engine's RPM.
AL	Allow long messages. The default number of data bytes in a message is 7. However some OBD messages such as the VIN request are longer. AL lifts the 7 bytes limitation. Default is AL off.
NL	Normal length messages. Limits all sent and received data to seven bytes.
D	Sets all settings to default factory settings. Last stored protocol is retrieved.
DP	Describe current protocol. Returns the vehicle's OBD protocol name, automatically determined by the IC.
DPN	Describe the current protocol by number. Same as DP but uses a number to represent the protocol instead.

Table 3.9: Useful AT Commands

AT Command	Functionality
E0, E1	Echo off or on. Controls whether characters transmitted to the interpreter should be retransmitted back to the host. Default is echo on (E1).
H0, H1	Headers off or on. Controls whether to display the OBD message header information bytes. H1 turns headers on.
I	Identify. The chip provides its name to the host. Can be used by the MCU to initially determine which IC is communicating with it.
L0, L1	Linefeeds off or on. L1 turns linefeeds on which will generate a linefeed after every carriage return is transmitted.
LP	Low power mode. Enable low power mode, where only essential functionality is allowed.
S0, S1	Printing of spaces off or on. When on (S1), ECU responses, which are a serious of hex characters, are separated by a space to provide increased readability. Default setting is S1. Messages can be transferred faster if spaces are off.
Z	Reset all. Causes a complete reset including power. All settings and values are returned to default.

Table 3.10: Useful AT Command Set Continued

ST Command	Functionality
BR value	Switches the UART baud rate to the given value. Returns a '?' character if the specified value cannot be achieved within a 3% accuracy.
S@1 string	Sets the device description from an ASCII encoded string. Max length of 47 characters.
SATI string	Set device ID string. Can provide an ASCII encoded string to be the device ID. One time programmable.
DVI	Print the device ID string.

Table 3.11: Additional ST commands

Function	Description
mmcInit	Used to initialize the SD card in SPI mode. Has no parameters and returns an error/success code character.
mmcping	Checks if a card is present. Has no parameters and returns an error/success code character.
mmcSendCmd	Sends a command to the SD card. Takes 3 parameters: a command, data for that command, and a checksum. No return value.
mmcGoIdle	Puts the SD card in an idle low power mode. Has no parameters and returns an error/success code character.
mmcReadBlock	Read a block from the SD card. Takes 3 parameters: the start address of the data to read, number of bytes to read, and a pointer to read buffer. Returns an error/success code character.
mmcWriteBlock	Writes a 512 byte block to the SD card. Takes 3 parameters: Start address of data to write on the card, number of bytes to be written, and a pointer to write buffer. Returns an error/success code character.
mmcReadCardSize	Reads the size of the SD card from an internal register. Has no parameters and returns the detected card size as a long integer.

Table 3.12: SD Library Functions

Command	Description
S-, <string>	Sets the name of the device as a serialized Bluetooth friendly string. Supports a length of up to 15 characters (ASCII alphanumeric), and appends 2 bytes from the MAC address to ensure unique numbering.
SB,<n>	Sets the baud rate based on a given number (0-7) to a value between 2400 and 921K. 0 represents 2400 and 7 represents 921k.
SF,<n>	Resets to factory settings on next device reboot. Can take a number, either 1 or 2, as a parameter. 2 is a hard reset, while 1 resets a majority of the settings excluding device name and information.
SR, <v>	Sets the supported features of the RN4020 module. Takes in a 32-bit value.

TABLE 3.13: Set Commands for the RN4020

Command	Description
+	Toggles echo on and off. Default state is off.
|O, <v>|I, <v>	Used to set the output of the digital I/O pins (O) or get their inputs.
A, <v>, <v>	Commands the device to begin advertisement when it is in a peripheral role. Two values are passed in as parameters, the first is the advertisement interval, and the second is the total advertisement time (both in milliseconds).
B	Used to bond two connected devices. Once bonded, reconnection between devices doesn't require authentication.
D	Dumps information about the device over the UART. Transmits the device MAC address, name, connection role, connected devices, bonded devices, and server services.
E, <0>,<v>	Establishes a connection with a discovered peripheral device. Takes the 6- byte MAC address of the peripheral device as a parameter.
K	Kills the active BLE link.
N,<v>	Places the RN4020 in a peripheral role, the A command can then be used to broadcast. Takes in an "advertising content" as a parameter, which is a hexadecimal value up to 25 bytes in length.
R, 1	Forces a complete reboot. All setting changes take effect upon startup.

Table 3.14: Action Commands for the RN4020

Command	Description
I	Places the RN4020 in MLDP simulation mode. All UART data received will be transmitted wirelessly to the connected device. The only way to exit is to assert CMD/MLDP low.
SE,<n>	Sets the security mode for MLDP communications. Three possible parameters: no security required (0), data is encrypted (1), and data is authenticated (2).

Table 3.15: MLDP Commands for the RN4020

Chapter 4

Design Constraints and Standards

4.1 Design Constraints

In this section we will analyze and discuss constraints as they apply to the GEM device. All engineering projects must deal with constraints in their design and implementation.

4.1.1 Economic

The GEM device is designed as an efficiency increasing and cost saving device. As such it's important that the device remain relatively low cost or else the initial cost of the device will outweigh and savings that it may be able to provide. In order to bring a device like the GEM to market we would require that the device be able to pay for itself within one year. Overall our device used only one specialized component (The STN1110) which is something that could be done away with in another revision by having a more robust firmware. Our initial prototypes while expensive still ended up coming in under budget. With further refinement a sub fifty dollar unit would certainly be feasible.

In addition we had a budgetary constraint as our budget was provided by The Boeing Corporation. We opted to stay within that budget and successfully designed and built the GEM for less than $700.00.

4.1.2 Environmental

The primary design constraint for this project as well as all modern electronic projects deal with the manufacturing process. Components should be soldered using lead free solder and components that contain lead or other heavy metals should be limited or eliminated from the design completely.

4.1.3 Social

Constraints involving social issues could not be identified in relation to the GEM.

4.1.4 Political

As the GEM is designed to be an automotive device there are many rules and regulations regarding it's operation. It must not interfere with the operation of the vehicle and as with any device that intentionally radiates energy it must comply with FCC regulations.

4.1.5 Ethical

The GEM is a device that can potentially record personal information that an individual may not want revealed. Since GEM is designed to continually log data about a person's driving habits there is always the possibility that that information could be search by law enforcement or other government officials. Because of this it should be a requirement that the data recorded by the GEM is always available for the user and can be easily deleted if the user requests or requires it.

4.1.6 Health and Safety

The GEM is an automotive accessory. As such it must be designed in such a way that it is not distracting to the driver while fulfilling it's main design requirements. We accomplished this by creating an ultra simple Android application for the user interface and requiring no user interaction with the device hardware.

4.1.7 Manufacturability

The GEM prototype was not designed with manufacturability in mind. However there are no features which would cause problems in manufacture other than excessive part count. In order to optimize for manufacture the GEM would need to have a simplified bill of materials. For example rather than using an 866 ohm resistor, we would try to see if a 1 kilo-ohm resistor would work in it's place to reduce the overall number of different components.

4.1.8 Sustainability

Sustainability would require further analysis of our chosen parts as well as the manufacturing process. In order to make sure that our product is robust enough to survive the rigors of automotive use we would require specialized testing. To optimize for long term production we would need to verify with manufacturers that there is a guaranteed lifetime for the parts as well as find out what the process the manufacturer goes through when parts reach the end of the manufacturing lifetime. Given this information we could plan redesigns around a parts end of life.

4.1.9 Legal

Since the GEM is similar to a number of other products on the market it would be important to verify that the device is not in violation of any patents. As noted in the political constraints sections we would also need to verify that the device is operating within all government standards in order to bring it to market.

4.2 Related Standards

In this section all relevant and relatable industry standards pertaining to the GEM system are listed and briefly described.

4.2.1 SAE J 2561-2001

A Bluetooth wireless protocol for automotive applications by the Society of Automotive Engineers. The report defines the functionality of Bluetooth applications used by electronic devices within vehicles.

4.2.2 ANSI/CEA 2040-2011

The SD Card Common Interface Standard by the Consumer Electronics Association. Describes the interface between a microSD card and a terminal device, including hardware, signaling, and application interface.

4.2.3 RS-232

Standard for serial data transmission. Defines electric characteristics, timing, and meaning of signals. As well as the physical size and pinout of connectors.

4.2.4 IEEE 802.15

The Institute of Electrical and Electronics Engineers' standards regarding wireless personal area networks including Bluetooth.

4.2.5 Software Design Standards

The GEM Android application was developed based on the best practices suggested by the Android Development Team at `https://developer.android.com/training/index.html`.

4.2.6 Software Documentation Standards

Code documentation was generated using Javadoc because of its availability in Android Studio and our team's familiarity with this documentation generator. All documentation followed the conventions provided by Oracle at `http://www.oracle.com/technetwork/articles/java/index-137868.html`.

4.2.7 Software Coding Standards

The GEM Android application adhered to the code style guidelines distributed by the Android Development Team at `https://source.android.com/source/code-style.html`.

Chapter 5

Hardware Design Details

5.1 Enclosure

The GEM is intended to be left connected to the vehicle semi-permanently. Because of this it needs to be small enough that it won't get in the way of the driver. Similar devices are roughly two inches wide and from from 1.5 to 2.5 inches deep. Our design is in line with these devices. The connecting portion of the housing is an OBD-II connector from Sparkfun Electronics. The connector is shown in 5.1.

The outer dimensions of the rectangular portion are 37.5 mm by 17.5 mm. The standard connector has two rows of pins separated by 9 mm and each row has a 4 mm pin pitch. Because of how the pins are separated there is not an direct way to attach the port to our main board. In order to get around this a smaller daughter board that will breakout the eight pins we will be using. These will come off the daughter board using right angle pin headers. The breakout board is shown in 5.2. The main board will then be attached to the breakout board via the pin headers. The main board attached to the breakout board can then be placed in a standard enclosure with a cutout provided for the OBD-II plug. The selected enclosure is the Bud Industries 563-PB-1575 available from Mouser.

5.2 Power Supply

There are multiple voltages that the GEM requires, 12 V, 5 V, 3.3 V and for the SAE J1850 bus switchable 5 V / 7 V. 12 V can be supplied by the battery but due to noise

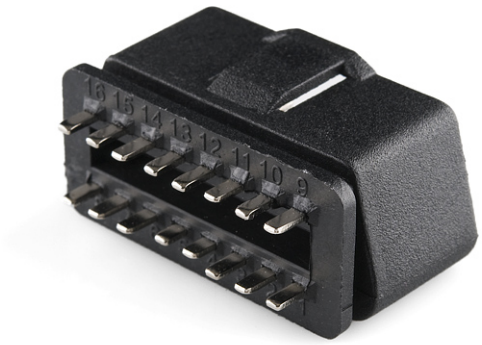


FIGURE 5.1: Sparkfun OBD-II Connector -
Image courtesy of SparkFun.com

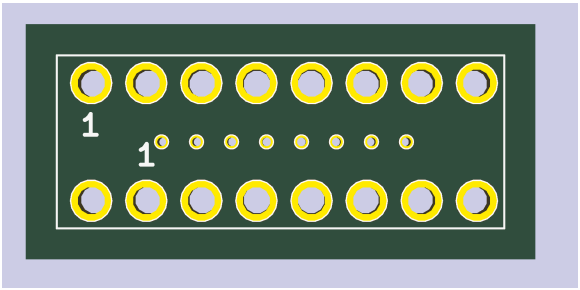


FIGURE 5.2: Breakout Board

on the bus as well as the possibility of inductive voltage spikes the battery voltage must be filtered before it is usable. The filter and TVS stage of the power supply is shown in figure 5.3. This design is based off the reference design from OBD Solutions.

RAW_BAT is the direct connection to the battery via the OBD-II port. For system protection two devices are used. F1 is a "positive temperature coefficient device" or resettable polyfuse. D10 is a transient voltage suppression diode. The polyfuse provides over current protection, should the device draw too much current (>0.7 A). In order to reset the fuse the GEM must be removed from power. The TVS diode protects the system from over voltage. The TVS diode allows a typical operating voltage but if the voltage spikes the TVS diode goes into breakdown and current flows to ground. Diode D11 is used for reverse voltage protection, should someone hook up

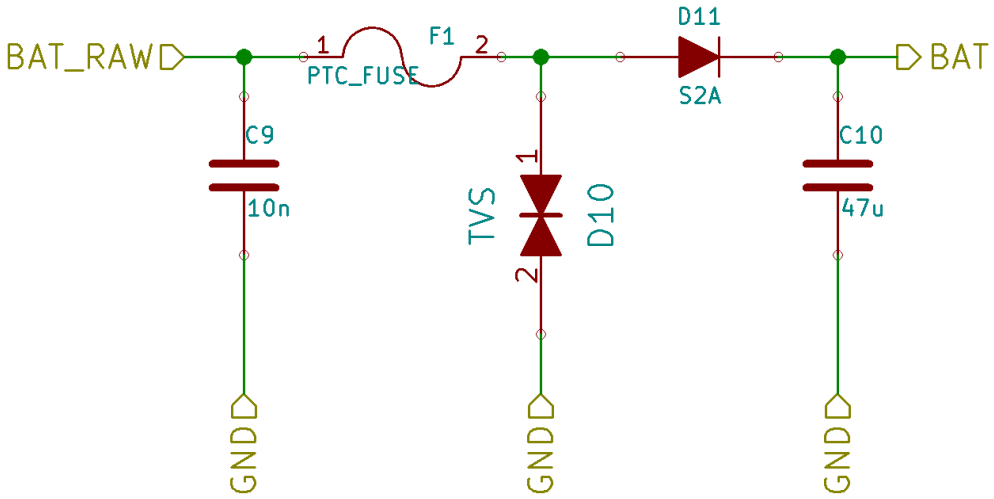


FIGURE 5.3: Filter Stage

their battery backwards while the GEM is connected to the OBD-II port. Finally, C9 and C10 are used to filter high frequency noise. These values are recommended by the manufacturer of the STN1110.

From this stage the filtered and protected battery supply is passed to two different voltage regulators. One for the the 5 V supply and one for the 3.3 V supply. These two supplies are based off of designs created using the Texas Instruments WEBENCH power designer. Both circuits use the LMR16006 buck regulator. The schematics for the 3.3 V and 5 V regulator circuits are shown in figures 5.4 and 5.5 respectively. Other than component values the designs are essentially the same.

In both cases the resistor divider sets the output voltage by:

$$V_{out} = V_{fb} \times (1 + \frac{R_4}{R_5})$$

where $V_{fb} = 0.765\,\text{V}$, R_4 is the top resistor and R_5 is the bottom resistor. All component values were selected based on recommended values from the TI WEBENCH power designer.

The final portion of the power supply is the switched power management circuit. This part turns on and off the BAT and 5V supply to the GEM based on the status of the STN1110. This allows for a lower power consumption as the active components that use these supplies will draw no quiescent current. The switched power management circuit is shown in figure 5.6.

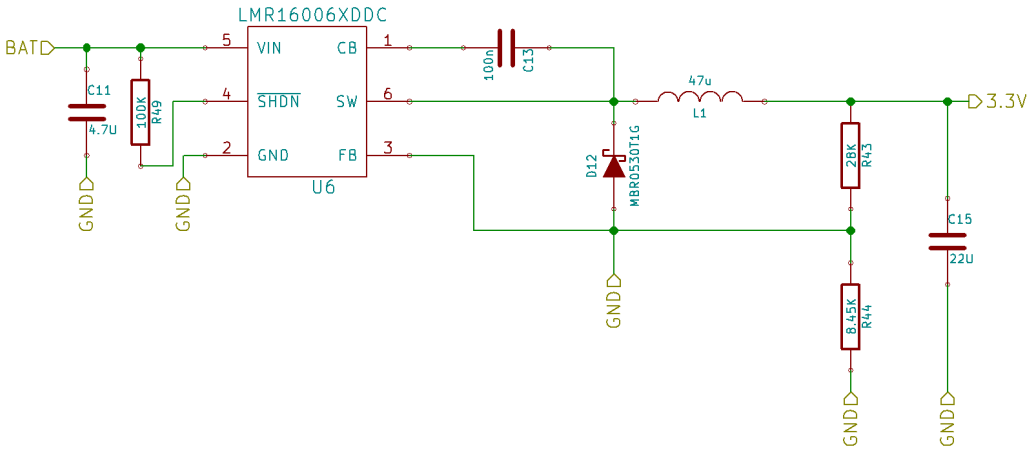


FIGURE 5.4: 3.3 V Supply

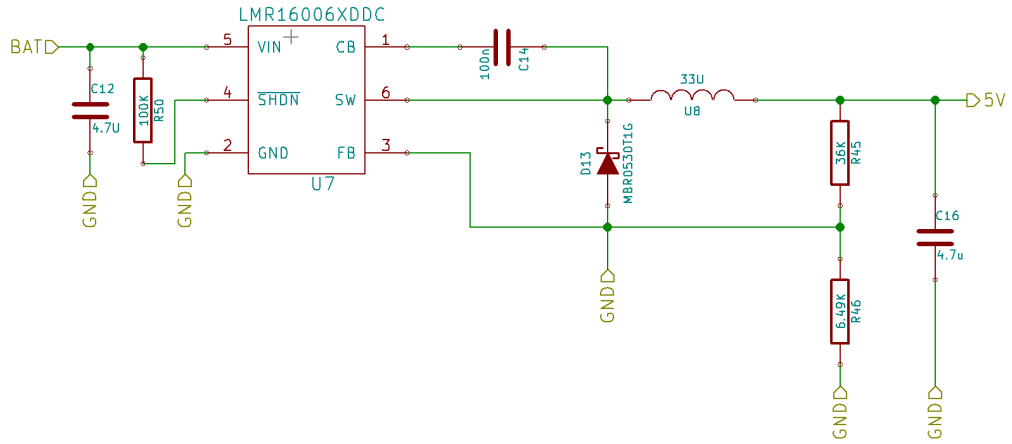


FIGURE 5.5: 5 V Supply

In this circuit when $\overline{\text{PWR_SAVE}}$ is low, Q7 is off which drives the gate voltage on the two MOSFETs low. When $\overline{\text{PWR_SAVE}}$ is high, Q7 is on which drives the gate voltage on the two MOSFETs low. When the gate voltage is high the MOSFET is off and when the voltage is low the MOSFET is on.

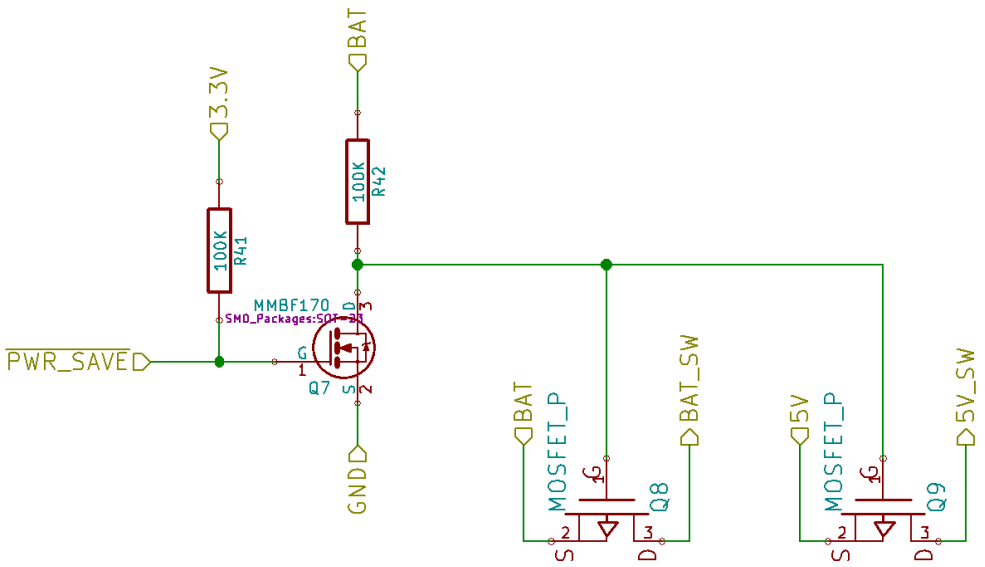


FIGURE 5.6: Switched Power Supply

5.3 OBD-II to Serial Interface

5.3.1 STN1110

The OBD-II to serial interface is based on STN1110. The STN1110 takes the incoming data from the vehicle, converts it to a serial stream and then forwards the serial data to the MSP430 for further processing. The connections are shown in figure 5.7. The pinout for the STN1110 is given in table 5.1 The connections are based on the reference design supplied by OBD Solutions.

$\overline{\text{RESET}}$ is not used in this design and so it is tied high. ANALOG_IN is connected to the voltage sense circuit shown in figure 5.8. If there is any change in voltage such as when the engine is started or stopped the GEM can detect the voltage change and react by entering the appropriate power state. PWM/$\overline{\text{VPW}}$ connects to the J1850 bus transmitter circuit, figure 5.14 in the transceivers section. Because the J1850 bus operates using either 5 V or 7 V there needs to be a way to select the voltage level. This pin is used to select between the two voltages depending on whether VPW or PWM is being used. J1850_BUS+_TX & $\overline{\text{J1850_BUS-_TX}}$ are connected to their appropriate transceivers. When VPW is in use only the J1850_BUS+_TX is used. When PWM is in use both lines are used. VSS is connected to ground.

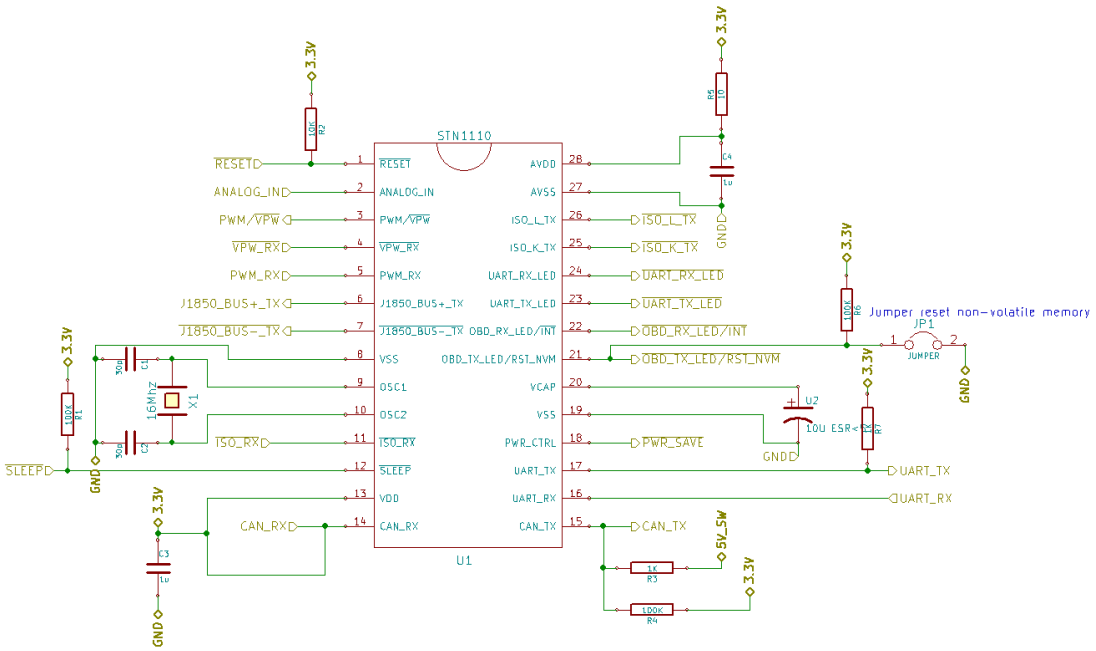


FIGURE 5.7: STN1110 Configuration

OSC1 and OSC2 are connected to the crystal oscillator. For the board layout the crystal oscillator needs to be placed within 12 mm of pins 9 and 10. The capacitors need to be placed right next to the crystal. All components should be on the same side of the board. The crystal and capacitors should be surrounded by a grounded copper pour and no signal or power traces should run inside the pour. No signal or power traces should be routed underneath the crystal oscillator circuit.

$\overline{\text{ISO_RX}}$ connects to the ISO K line transceiver. $\overline{\text{SLEEP}}$ is tied high. If needed it is available to be used by the MSP430 to put the STN1110 into low power mode.

VDD is connected to 3.3 V and can support a voltage range of 3.0 V to 3.6 V. CAN_RX receives input from the CAN transceiver(figure 5.9) using 5 V logic. CAN_TX is an open drain output and requires a pull-up to 5 V. The value of the pull-up resistor depends on the CAN baud rate and trace length. The manufacturer recommends a 1 kΩ pull-up resistor. UART_RX is connected to the UART_TX of the MSP430. UART_TX is connected to the UART_RX of the MSP430. UART_TX is an open drain output and requires a pull-up to VDD. PWR_CTRL is used to power down devices connected to BAT_SW and 5V_SW. PWR_CTRL is an open drain output and requires a pull-up to VDD. The pull-up resistor is shown in the switched power management circuit, figure 5.6. VSS is connected directly to ground. VCAP must be connected to ground via a low ESR ceramic capacitor, typically 10 µF.

Table 5.1: STN1110 Pinout

Pin Number	Pin Name	Pin Description
1	$\overline{\text{RESET}}$	Reset Input
2	ANALOG_IN	Analog Voltage Measurement
3	PWM/$\overline{\text{VPW}}$	SAE J1850 Voltage Select
4	$\overline{\text{VPW_RX}}$	VPW Receive Input
5	PWM_RX	PWM Receive Input
6	J1850_BUS+_TX	SAE J1850 Bus+ Transmit
7	$\overline{\text{J1850_BUS-_TX}}$	SAE J1850 Bus- Transmit
8	VSS	Ground Reference
9	OSC1	16.000 MHz Oscillator Input
10	OSC2	16.000 MHz Oscillator Output
11	$\overline{\text{ISO_RX}}$	ISO K-Line Input
12	$\overline{\text{SLEEP}}$	External Sleep Control
13	VDD	Positive Voltage Supply
14	CAN_RX	CAN Receive Input
15	CAN_TX	CAN Transmit Output
16	UART_RX	UART Receive Input
17	UART_TX	UART Transmit Output
18	PWR_CTRL	External Power Control Output
19	VSS	Ground Reference
20	VCAP	CPU Logic Filter Cap
21	$\overline{\text{OBD_TX_LED}}$/$\overline{\text{RST_NVM}}$	Activity LED and Reset to Defaults
22	$\overline{\text{OBD_RX_LED}}$/$\overline{\text{INT}}$	Activity LED and Interrupt Output
23	$\overline{\text{UART_TX_LED}}$	Activity LED
24	$\overline{\text{UART_RX_LED}}$	Activity LED
25	$\overline{\text{ISO_K_TX}}$	K-Line Output
26	$\overline{\text{ISO_L_TX}}$	L-Line Output
27	AVSS	Analog Ground Reference
28	AVDD	Analog Positive Supply

Pin 21, $\overline{\text{OBD_TX_LED}}$/$\overline{\text{RST_NVM}}$, is a double duty pin. When JP1 is connected this pin is grounded which resets the non-volatile memory. When JP1 is not connected this pin controls the OBD transmit activity LED. pin 22, $\overline{\text{OBD_RX_LED}}$/$\overline{\text{INT}}$, also has two uses, it either controls the OBD receive activity LED or acts as an interrupt output. $\overline{\text{UART_TX_LED}}$ and $\overline{\text{UART_RX_LED}}$ control the UART_TX and UART_RX activity LED's.

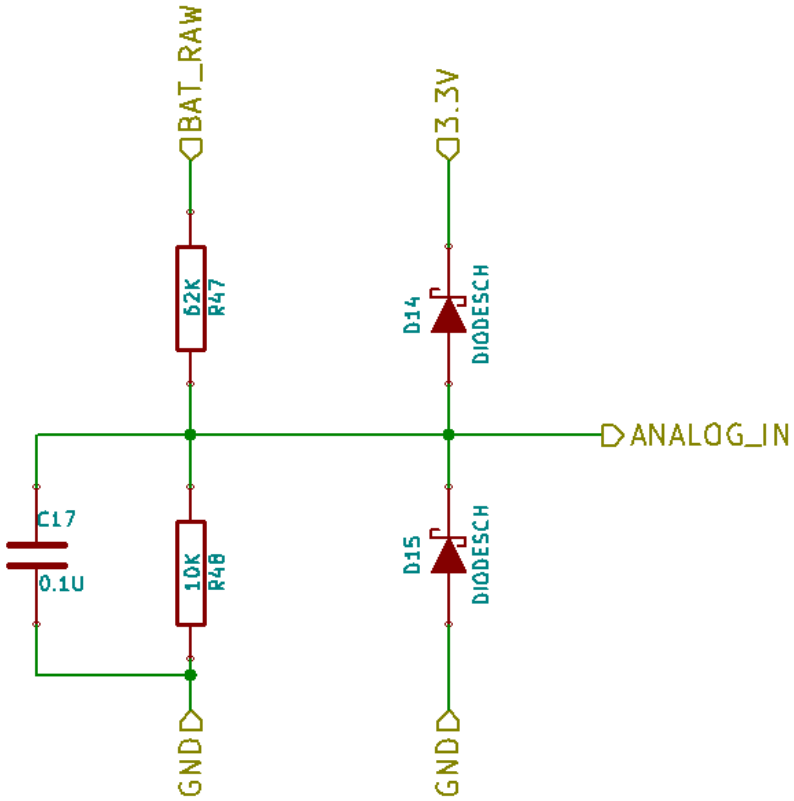


FIGURE 5.8: Voltage Sense Circuit

$\overline{\text{ISO_K_TX}}$ is an active low pin connected to the K-Line transceiver. $\overline{\text{ISO_L_TX}}$ is an active low pin connected to the L-Line transceiver. AVSS is connected directly to ground and AVDD is connected through a 10 Ω resistor. The resistor is recommended by the manufacturer to decouple AVDD from VDD.

5.3.2 Transceivers

The OBD-II to serial interface relies on three different transceivers, one for the CAN bus, one for the J1850 VPW and PWM bus and one for the ISO bus.

The CAN transceiver used for the GEM is the Microchip MCP2551. The MCP2551 takes input from the CAN bus in the form of a differential voltage and outputs a high or low signal via the RXD pin. The schematic design is shown in figure 5.9.

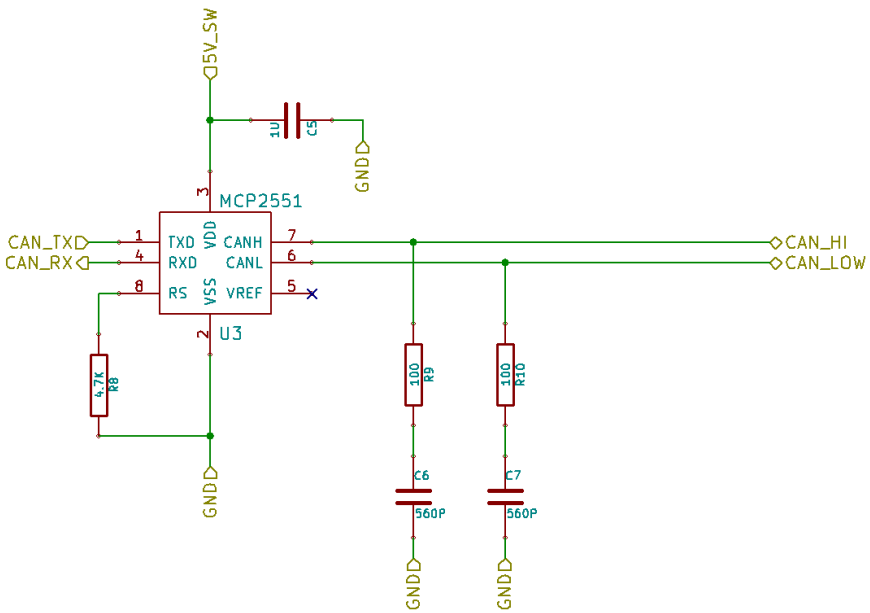


FIGURE 5.9: CAN Transceiver

The ISO transceiver has two sub-circuits. The transmitter for the K-line and the L-line is shown in 5.10. In both cases the high voltage is equal to the battery voltage, far too high for the STN1110 to handle on it's own. In these circuits the output from the STN1110 turns on and off the transistor which in turn sets the voltage on the line to either BAT which is nominally 12 V or to ground.

The receiver uses an LM339 comparator to compare the voltage on the K-line to the battery voltage. As the signal voltage on the K-line is either 12 V or 0 V a voltage divider is used on the battery voltage to reduce the voltage by half. With this the comparison is between 12 V and 6 V or 0 V and 6 V. When the K-line is low the open collector output of the comparator is pulled to 3.3 V. When the K-line is high the output of the comparator is pulled to ground. This output is then fed to the $\overline{\text{ISO_RX}}$ pin of the STN1110. The receiver circuit is shown in figure 5.11.

The SAE J1850 transceiver must deal with two different protocols and so is the most complex of the three transceivers. Both receivers use the LM339 comparator. The PWM receiver takes an input of the J1850 BUS+ to the positive input and BUS- to the negative input. When BUS+ is pulled high and BUS- is pulled low the open collector output of the comparator is pulled to 3.3 V. Otherwise the output is pulled to ground. The schematic for the PWM receiver is shown in figure 5.12

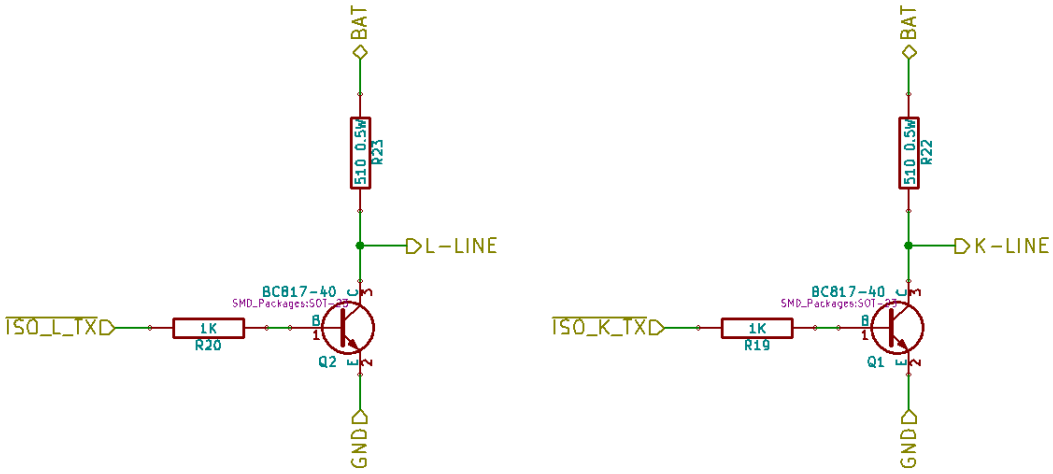


FIGURE 5.10: ISO Transmitter

For the VPW receiver only the BUS+ line is used. The positive input of the LM339 is connected to 5 V through a voltage divider that reduces it to 4 V as the decision point between high and low for VPW is set at 3.5 V. BUS+ is connected to the negative input of the comparator. When BUS+ is less than 4 V the open collector output of the LM339 is pulled to 3.3 V and when the BUS+ is greater than 4 V the output is pulled to ground. The schematic for the VPW receiver is shown in figure 5.13.

For the transmitter portion of the SAE J1850 bus the BUS+ line has to be able to operate at two different voltages. The voltage is selected by the STN1110. When PWM is high transistor Q3 is on which bypasses the 374 Ω resistor. This sets the output of the LM317 voltage regulator to 5.75 V. When PWM is low Q3 is off and the output of the LM317 is 7.6 V. When the J1850 BUS+_TX pin is high, Q4 is turned on which pulls the base of Q6 to ground turning it on and the output to BUS+ is roughly 7 V after the base emitter drop and the diode drop at D8. When the BUS+_TX pin is low Q4 turns off which causes the base voltage at Q6 to rise and turn off Q6. This leaves the BUS+ output at ground. This schematic for this circuit is shown in figure 5.14.

The BUS- transmitter is far simpler as it is only used for PWM and so is only ever left high when not transmitting or pulled low when transmitting. When the BUS-_TX pin is low transistor Q5 is off and the voltage at BUS- is roughly 5 V. When the pin

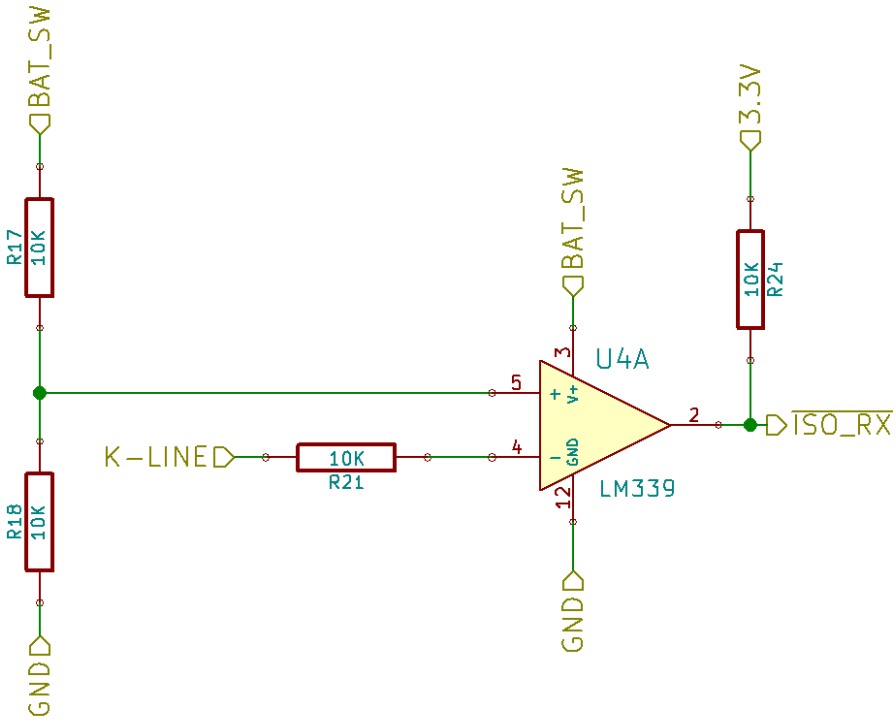


FIGURE 5.11: ISO Receiver

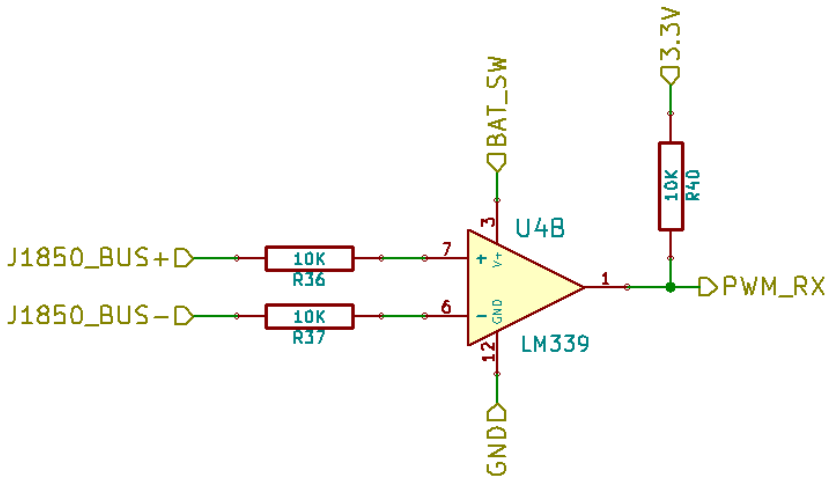


FIGURE 5.12: SAE J1850 PWM Receiver

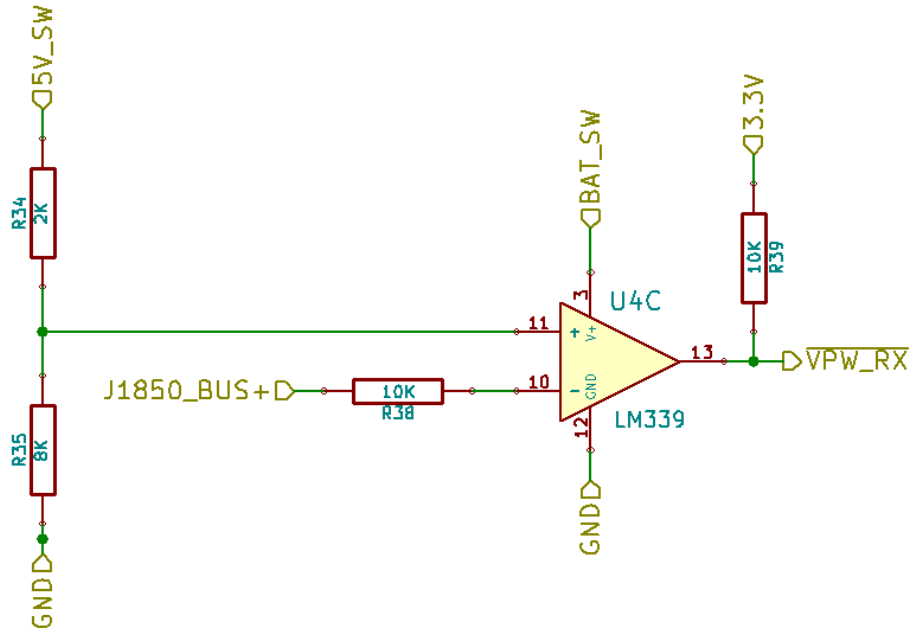


FIGURE 5.13: SAE J1850 VPW Receiver

goes high the transistor is turned on and the voltage on BUS- is pulled to ground. The schematic for this circuit is shown in figure 5.14.

5.4 Microcontroller Hardware Design

The design of the microntroller subsystem, including it's peripherals and any hardware considerations is portrayed in the following subsections.

5.4.1 Microcontroller Choice

The MSP430F247 was the final MCU design choice for the GEM system. A variety of system requirements and personal preferences led to this choice. For our design we needed an ultra-low power 16 bit microcontroller. We narrowed down our choices to an MSP430 chip, based on team members' experience, and comfort, with the device family. We determined an operating system was not necessary for GEM, since the

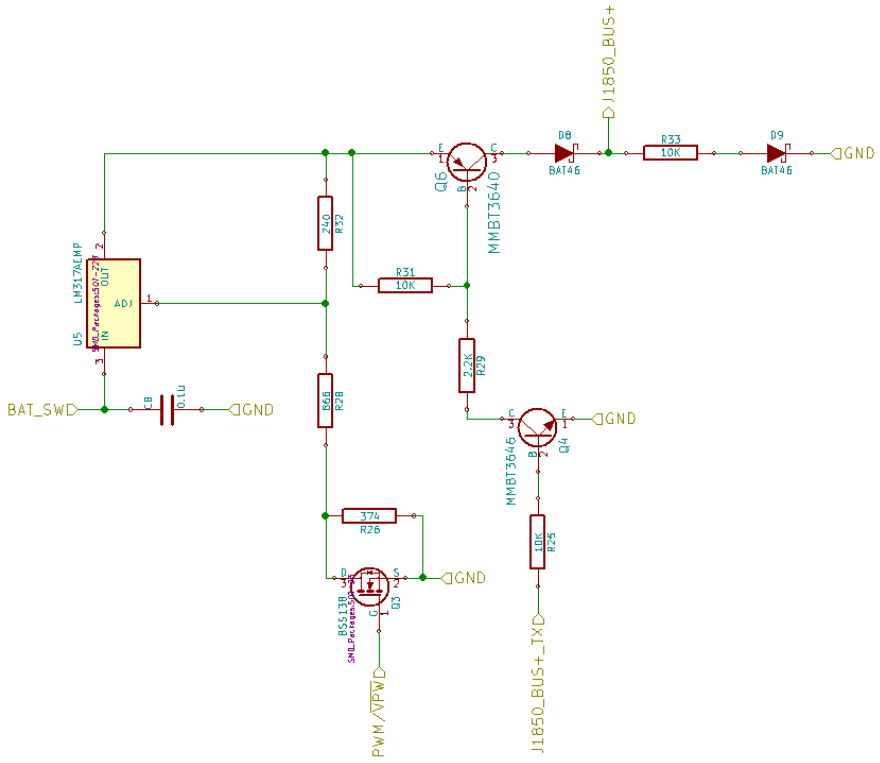


FIGURE 5.14: SAE J1850 BUS+ Transmitter

MCU is only required to interface with several other chips and modules. Using an operating system would also drive up the power usage of the device, and one of GEM's primary goals was to keep the design as power efficient as possible. The MSP430 has the task of interfacing with the STN1110 OBD-II interpreter chip, an onboard microSD card, and a Bluetooth module. Thus, a device with at least 3 serial communication interfaces was needed for our design. The MSP430F247 provides 4 serial communication interfaces, meeting our system requirement. Also, the MSP430F247 doesn't have a DMA, which would have gone to waste in our design. The 247 was specifically chosen among its family processors for the available flash memory and RAM. The MSP430F2XX family includes 6 different processors each providing a variable amount of memory. The 247 device hosts 32 KB of flash memory and 4 KB of RAM. This is the median amount amongst this family of devices. Choosing an MCU with above average on chip memory was important for supporting all our firmware and writing data to the SD card, which must be done in 512 byte blocks. The device is also cheap and samples are provided by Texas Instrument. However, some of the downfalls included an unused on chip ADC, and a large percentage of pins (device has 64 total pins) that where not needed. Although I/O pins where left

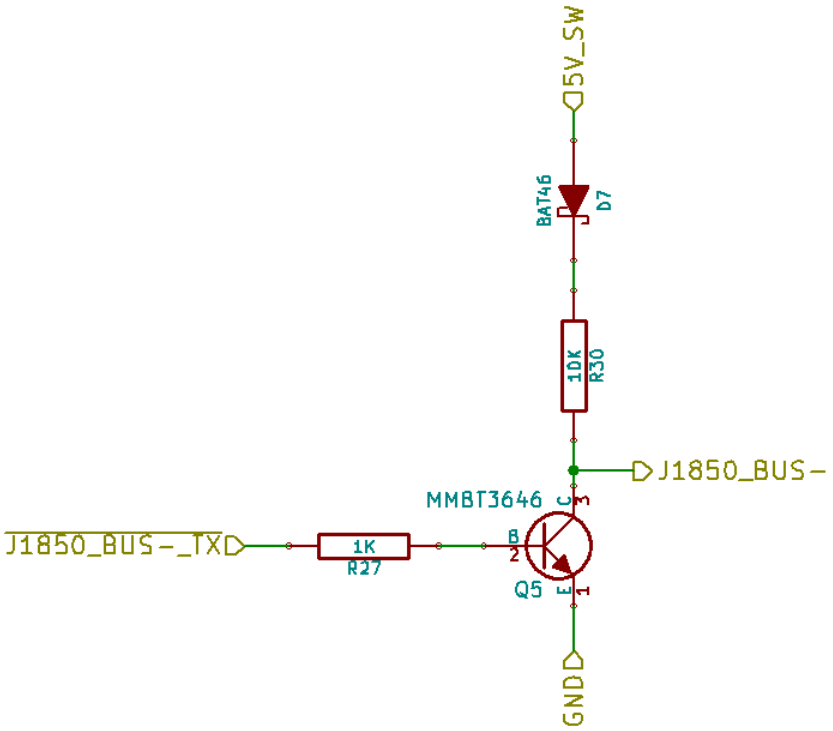


FIGURE 5.15: SAE J1850 BUS- Transmitter

unconnected in case future additions would be necessary. Overall, the MSP430F247 was a good, cheap, power efficient, choice that met all of our design requirements and our team members were all comfortable using.

5.4.2 JTAG Interface

In our design, several pins were broken out from the MSP430 in order to make use of the JTAG interface. The JTAG interface is used for on system debugging and flash programming. Without the JTAG interface we would have to resort to very expensive flashers to program the MCU. Our system was designed for a Spy Bi-Wire (2 Wire JTAG) connection shown in Figure 5.16. Our design is based off the reference design from TI's wiki page. A 14-pin header is used, but 8 of the pins are unused for 2 Wire JTAG connections with an MSP430. The JTAG header has two power inputs which are set via a jumper. J1 on the figure is the onboard 3.3 V power supply, J2 is used when programming the device and power is given by the programming device (a computer). Pin 9 is grounded, and pin 7 outputs a test clock to a dedicated JTAG pin

on the MSP430F247. Pin 8 is the device protection fuse, tied to a TDI/TCLK input on the MCU. And finally, pin 1 is the bidirectional TDO/TDI (test data in/out) used for programming the MSP device.

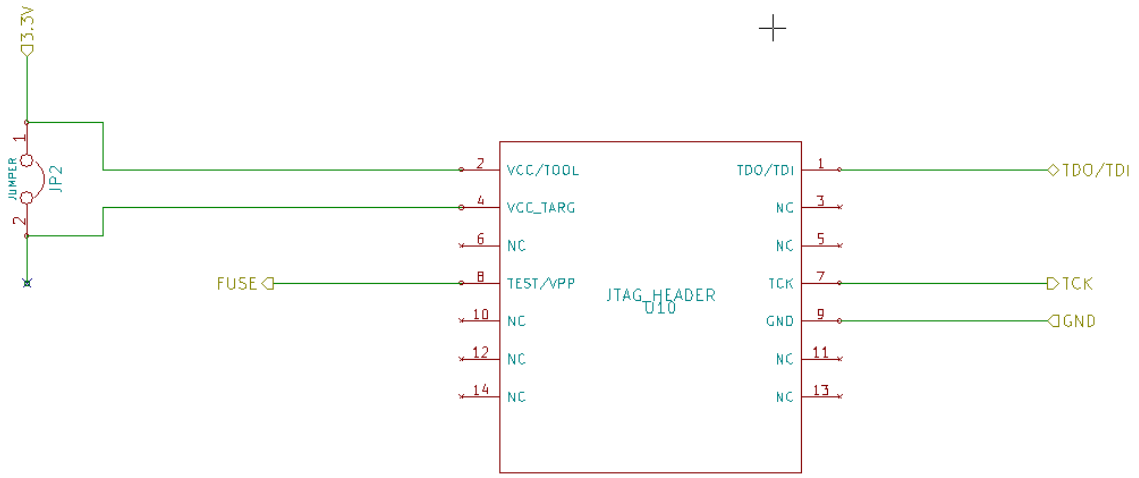


FIGURE 5.16: JTAG Interface Design

5.4.3 RN4020

Since low power and low cost drove all our previous design decisions. We wanted to maintain consistency when deciding how to implement Bluetooth on the GEM system. Ultimately, we decided to use BLE for its power efficiency, and avoided having to implement a memory intensive Bluetooth stack on our MCU. Thus, the RN4020 BLE module with an onboard Bluetooth 4.1 stack made the most sense for our design needs. The pin interface between the RN4020 and the MSP430F247 is displayed in Figure 5.17. The module has 24 total pins, 4 of which must be grounded and 3 which must remain unconnected as specified by the manufacturer's data sheet. 3.3 V are supplied to pin 23 from GEM's power supply. The UART_TX_BT and UART_RX_BT lines are connected to one of the MSP430's UART transmit and receive lines (respectively). Pins 2-4 are analog inputs to the RN4020, but are unused in our design, and as such, are grounded. CMD is connected to a standard digital I/O port on the MSP430, and is used to toggle the RN4020 between command mode and MLDP mode. WAKE_SW and WAKE_HW are also connected to standard I/O ports, and are used to control resets and power states in combination with software. SPI/PIO sets the modes of pins 10-13, asserted high for SPI mode. Finally, two status LED's are connected to

the device for debugging purposes. BT_CNT_LED signals when a device is connected, while BT_ACT_LED signals when an activity is performed (data transfer).

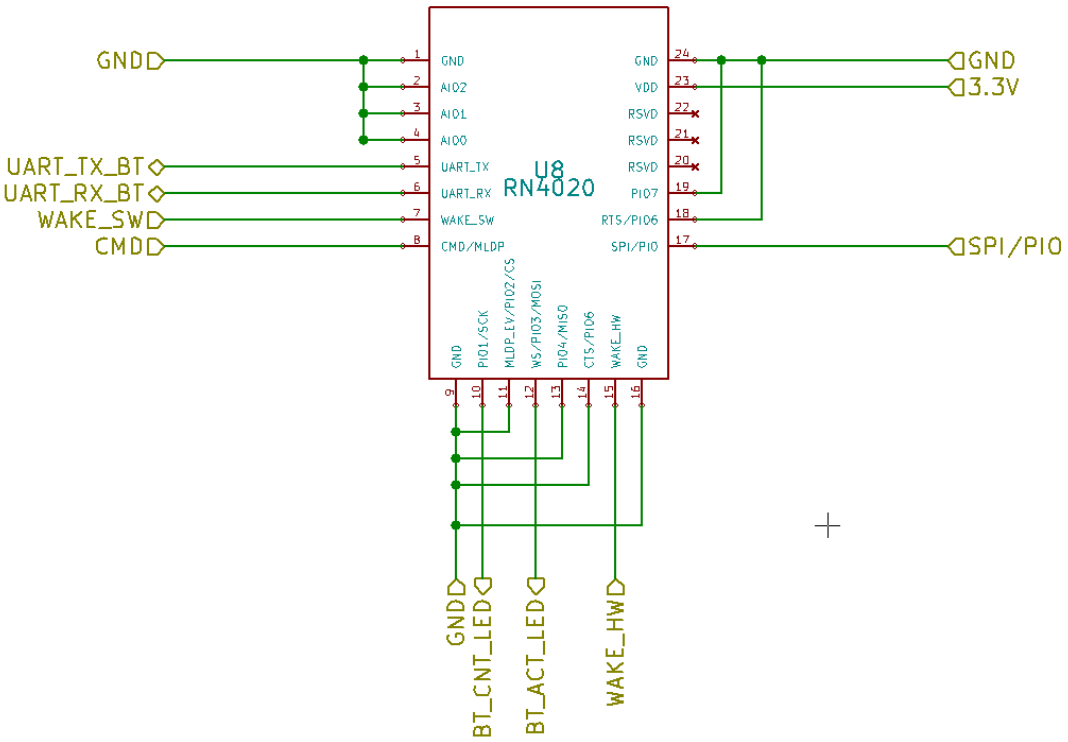


FIGURE 5.17: RN4020 Connections

5.4.4 MicroSD Card

GEM was required to have an on board storage device in order to store data when a Bluetooth connection is not established. Any storage media would have sufficed for this application. A microSD card was chosen mainly for its low power consumption and pinout as well providing a small form factor. For storing and receiving raw data in an SD card an SPI bus connection must be established with the host MCU. A microSD card is an 8 pin device. The connections for this device are shown in Figure 5.18. Two of the pins are required to be left unconnected, there is 1 grounded pin, and a 3.3 volt power input. Thus, the remaining 4 pins provide the interface to the host controller. Three of the connections are the standard SPI connections: slave in, master out (SIMO); slave out master in (SOMI), and an SPI clock provided by the master. The final connection, chip select (CS), is controlled by a standard digital I/O port on the MSP430. Chip select is used in conjunction with software to reset the SD card and select the operating mode.

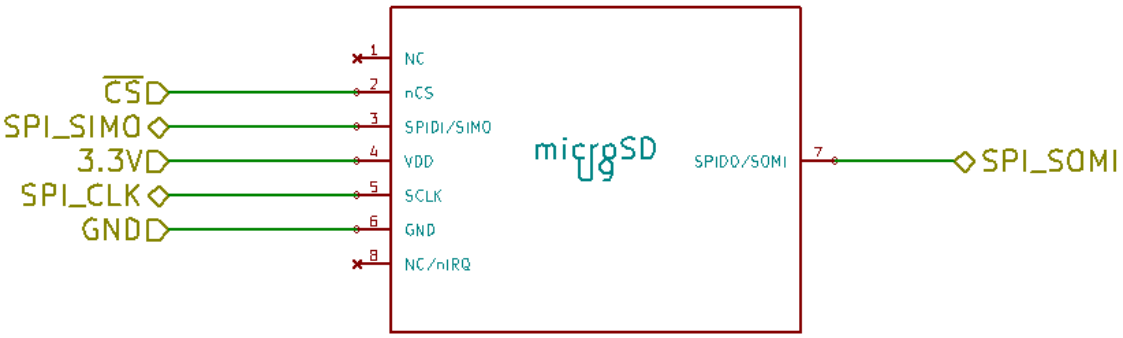


FIGURE 5.18: MicroSD Connections

Chapter 6

Software Design Details

6.1 Introduction

This chapter describes the architecture and software system design of the GEM Android application. It was our intention to create a reference for ourselves and others that outlined the procedures necessary to develop the system.

6.2 System Overview

Our goal was to produce software that meets the expectations of mobile application users with performance and presentation similar to other fuel economy applications. The Android application displays vehicle metrics received from the GEM device in real-time. Some of which are:

- Vehicle speed
- RPM
- Instant fuel economy
- Tripometer

This software is able to display these values for the user within range of the GEM device.

6.2.1 Software Development Tools

In addition to the Android development environment described in subsubsection 3.4.1.1 the following tools were used:

- SublimeText2 - A cross-platform text and source code editor.
  `http://www.sublimetext.com/2`
- GitHub - An online version control system.
  `https://github.com`
- Adobe Illustrator - A vector graphics editor.
  `http://www.adobe.com/products/illustrator.html`

6.3 Design Considerations

6.3.1 Reusability

The reusability of the system was a top priority from the beginning. The system is designed to take information from the GEM device and provide the application with the data necessary to produce meaningful statistics to the user. The design architecture is flexible enough to allow any subsystem or feature to be modified/improved in future releases.

6.3.2 Maintainability

Maintainability is crucial to any system that may be modified or examined in the future, and proper programming practices must be employed throughout the application development lifecycle. The Agile software development model was deemed the most effective as design challenges may have forced us to change directions quickly, but proper documentation throughout the build was absolutely necessary.

6.3.3 Testability

Testing the system on mobile devices required access to several different Android devices with varying hardware. The Android Virtual Device (AVD) emulator increased

productivity, but did not allow Bluetooth emulation. Our chosen IDE, Android Studio, is built specifically for the creation or Android apps and offered much of the same testing frameworks as Eclipse.

6.3.4 Performance

Our system was designed to be relatively light and requiring Android 4.3 guaranteed that a given mobile device's hardware could handle our application. No performance issues arose during the development process.

6.3.5 Portability

Developing the system for mobile devices introduced many variables (i.e. screen size, processor, RAM), but the languages and APIs used to program our system are standardized and subsume previous releases which made portability a relatively simple feat to achieve.

6.3.6 Safety

The information stored and transmitted by our device is not considered sensitive information. The range of the Bluetooth transmission is short enough that it was unlikely that unintended users would have access to the information transmitted. In the event that data was intercepted, there was no danger that a users vehicle could be altered or harmed in any way. The biggest concern for safety that our team had was that users would not be distracted by the application while operating their motor vehicle. Careful consideration was put into the design of the application so as not to require input from a user while driving.

6.3.7 Assumptions and Dependencies

It was assumed that the user would use a device with the Android operating system and Bluetooth LE capabilities. The Android application targeted as many devices as possible running API level 18 or greater, but no guarantees are made on the stability or functionality therein. Overall, it is expected that the user will use the software for its intended purposes.

6.4 System Architecture

6.4.1 User Interface

A main goal of the application was ease of use and its ability to effectively convey information to the user without distractions. This is why the application launches from the icon directly to the Bluetooth LE devices screen shown in figure 6.1.

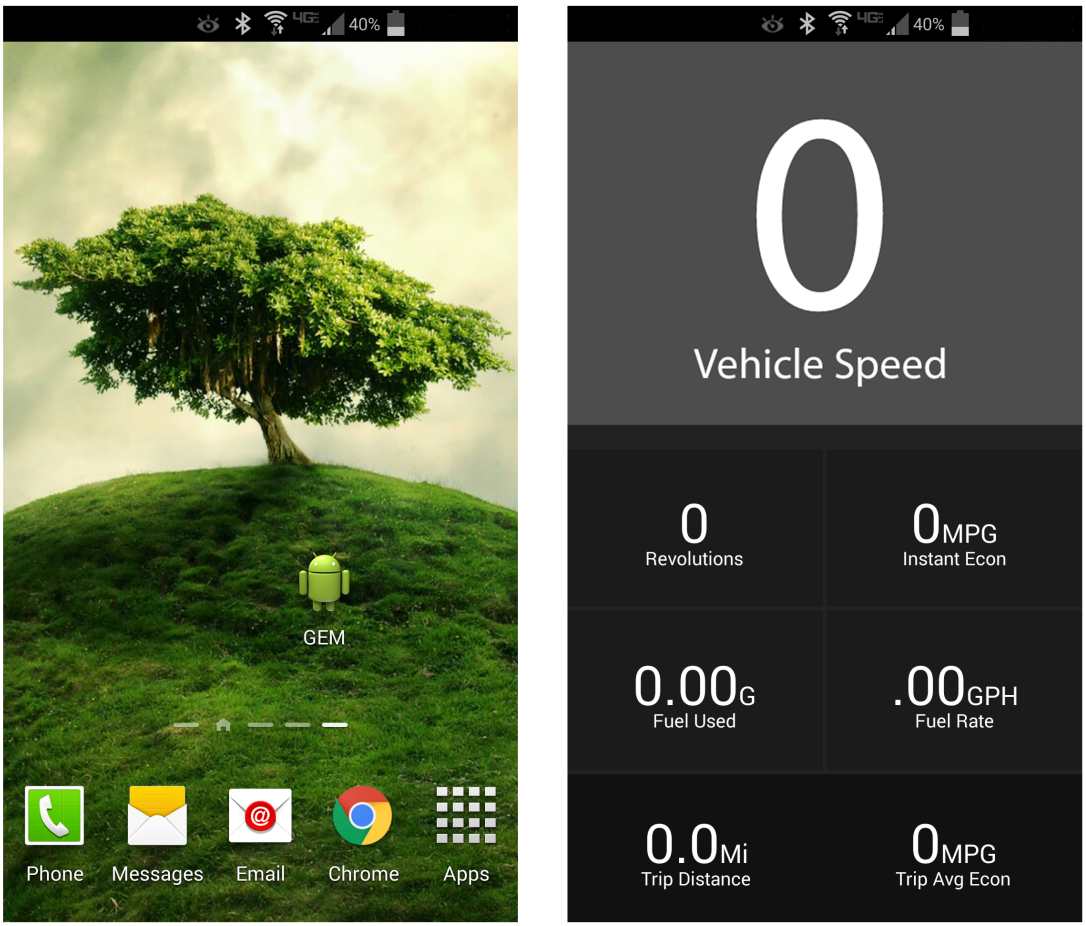


FIGURE 6.1: Launching the GEM application

The color scheme of the application was chosen for its contrast, paired with the automatic brightness feature of the mobile device makes it ideal for day and night driving. Vehicle speed is displayed significantly larger than the rest of the metrics because a digital readout is convenient since the majority of vehicles do not come equipped with digital speedometers. Also shown on the Home screen is Revolutions (RPM) which is

an important value to monitor for increased fuel economy. Next to Revolutions is the Instant Econ representing the MPG at that instant. Fuel Used displays the amount of fuel consumed during a trip and Fuel Rate shows the fuel consumed in gallons per hour (GPH) as an additional method of tracking fuel economy. The Trip Distance tracks the distance traveled and can be reset by the user. The Trip Avg Econ allows users to measure their average MPG over the course of a trip (Trip Distance).

For maximum flexibility the application is able to operate in a landscape orientation as shown in 6.2. This allows users to mount the device in their vehicle as they see fit.

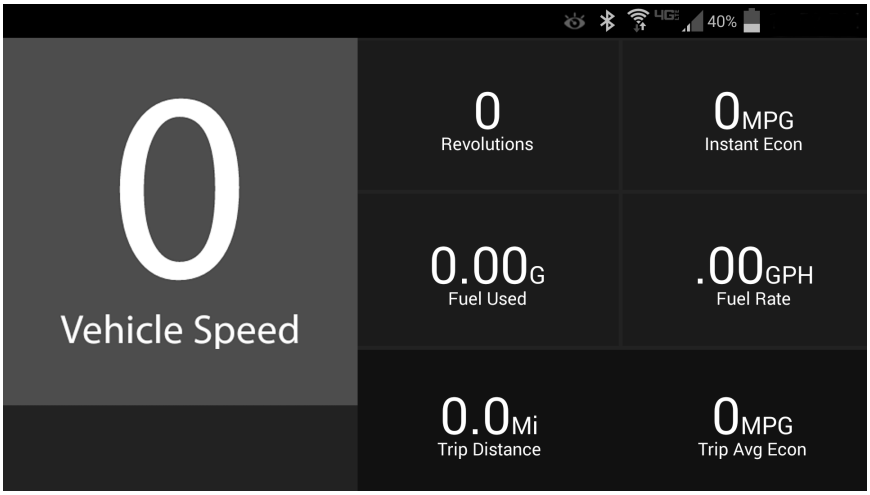


FIGURE 6.2: Landscape orientation

The menu button on a users mobile device displays the application Menu shown in 6.3. Selecting Start begins communication and data transmission with the GEM device. The Exit function disconnects the mobile device from the GEM device and closes the application.

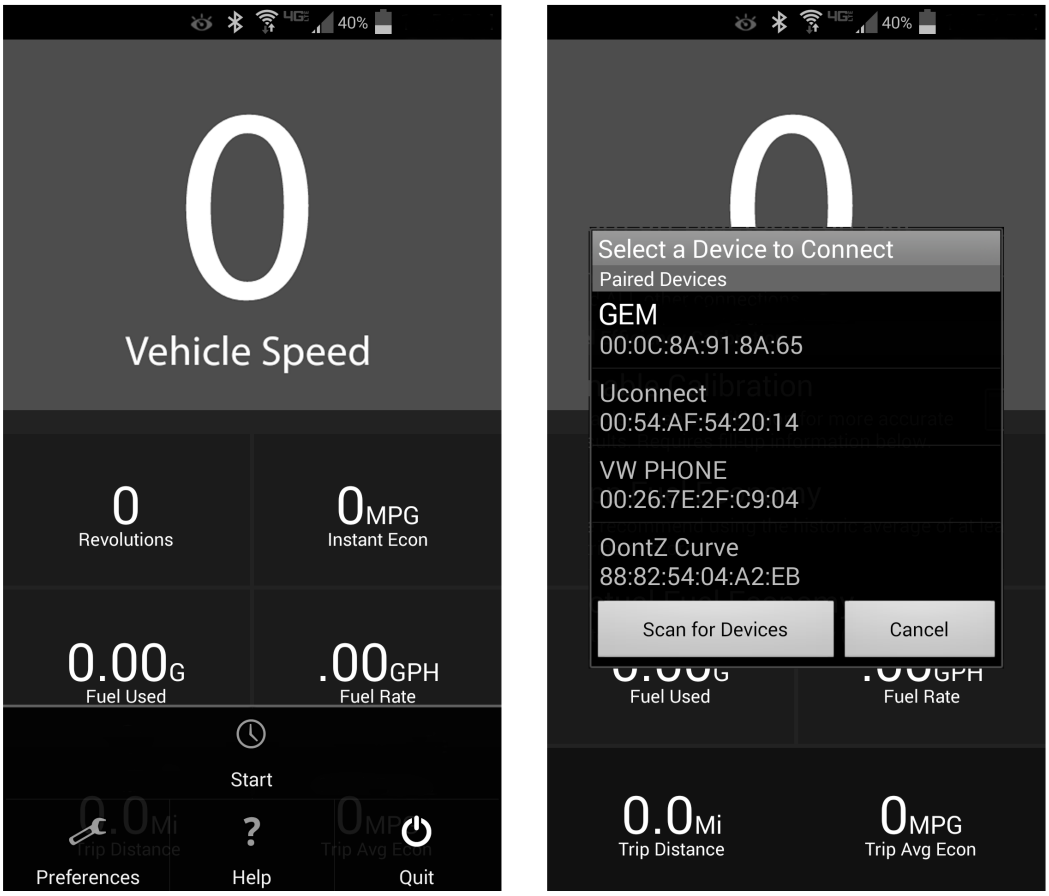


FIGURE 6.3: Application Menu

6.4.2 Decomposition Description

The activity diagram in 6.4 shows the flow of the GEM application.

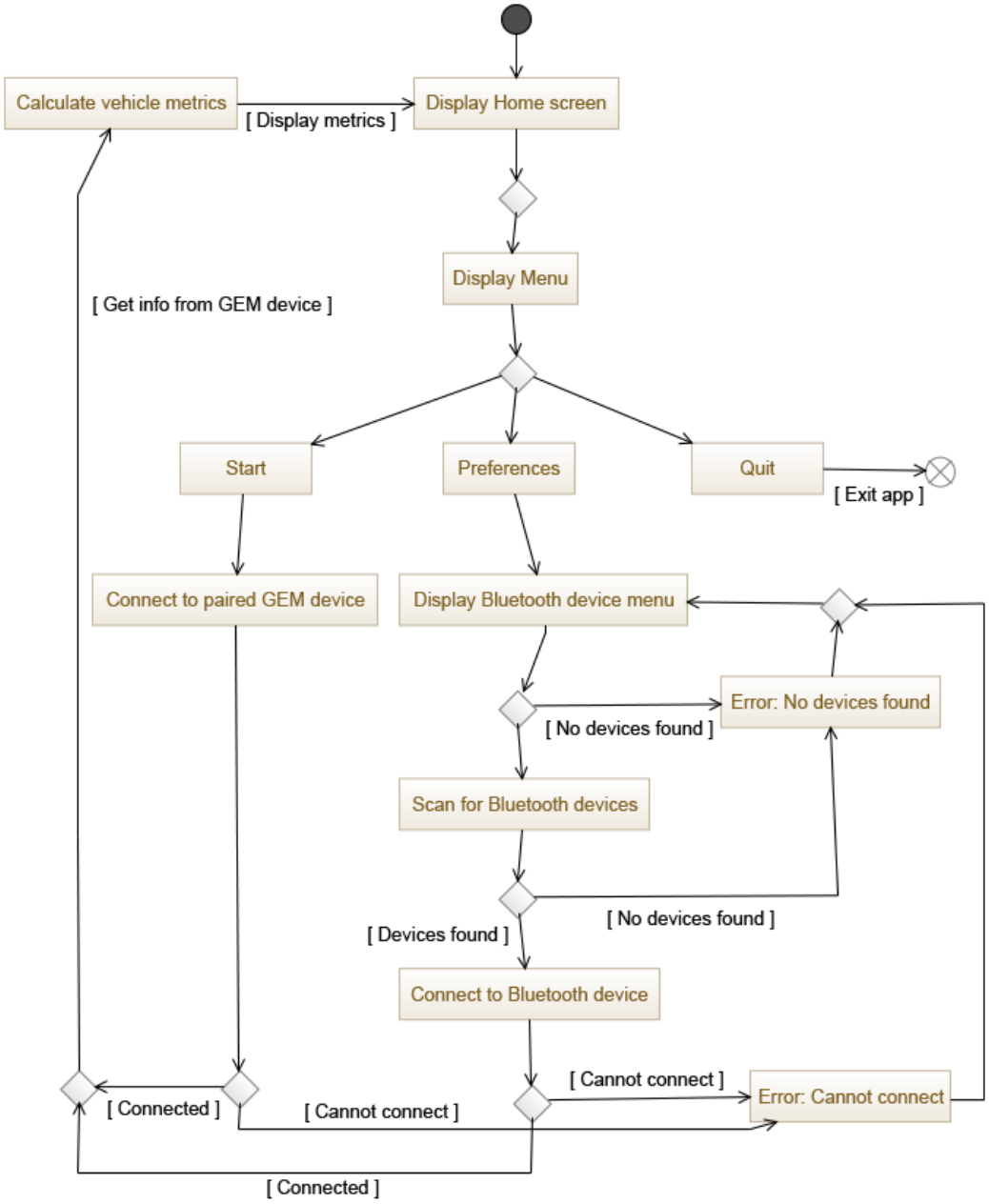


FIGURE 6.4: Application activity diagram

DeviceScanActivity is responsible for launching the application, scanning and displaying Bluetooth LE devices. Once connected to a Bluetooth LE device the application control is given to MainActivity.

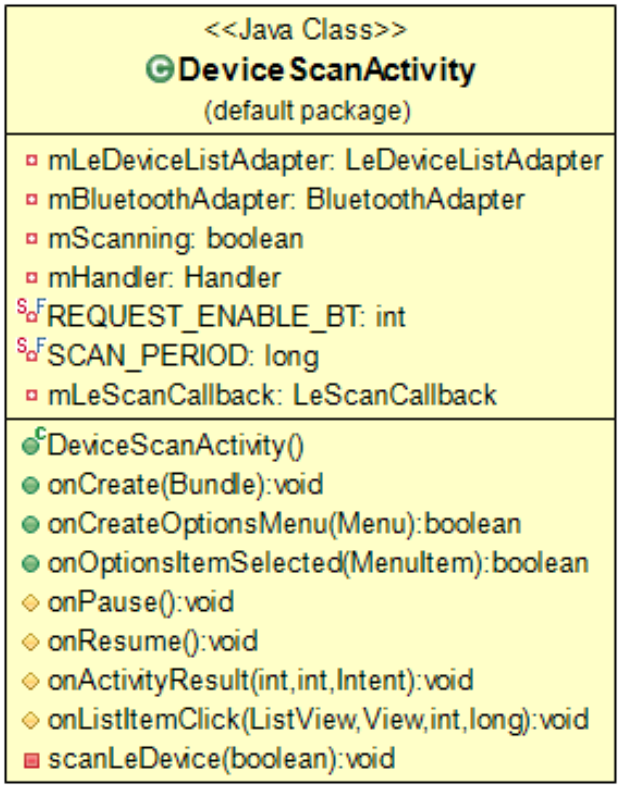


FIGURE 6.5: DeviceScanActivity class diagram

MainActivity is called upon when a user establishes a connection to a Bluetooth LE device from the DeviceScanActivity. This UI activity communicates with Bluetooth-LeService to manage and display vehicle data received from a Bluetooth LE device.

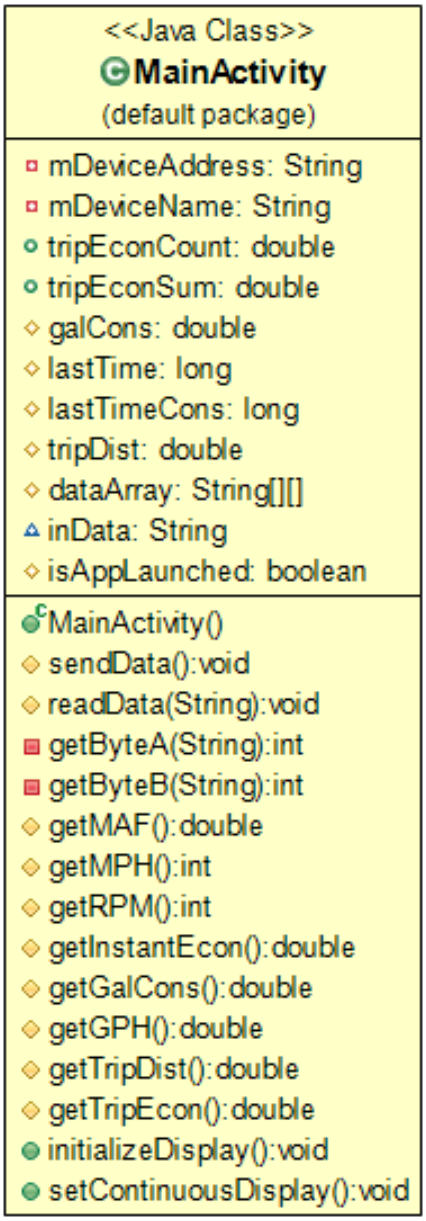


FIGURE 6.6: MainActivity class diagram

Once the GEM device is connected a Service that handles data transmission with a Generic Attribute (GATT) server hosted on a Bluetooth LE device is started. BluetoothLeService broadcasts data to the UI (MainActivity) where the values for vehicle speed, RPM, MPG, and other relevant information are calculated.

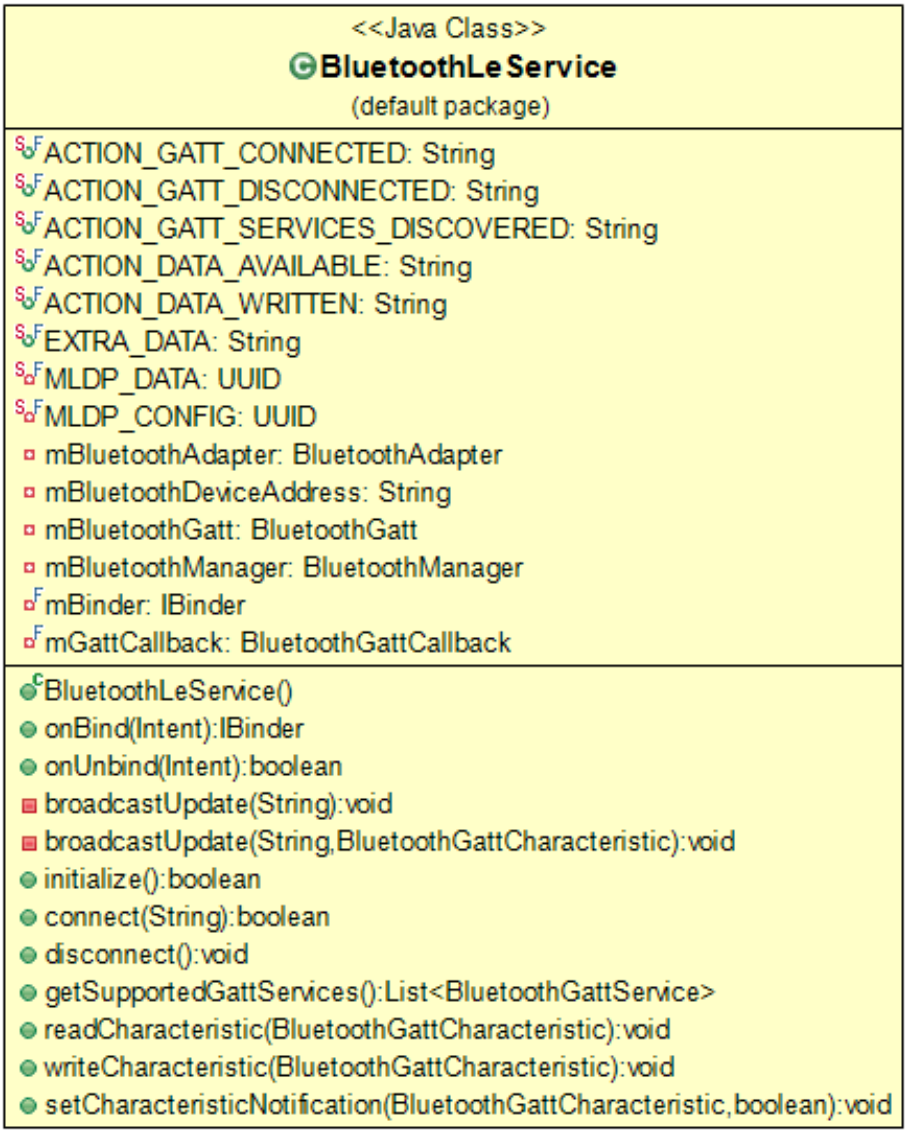


FIGURE 6.7: BluetoothLeService class diagram

6.5 Detailed System Design

6.5.1 Component Descriptions

Class **DeviceScanActivity**

Responsible for launching the application and then scanning for new or paired Bluetooth LE devices.

Declaration

public class DeviceScanActivity **extends** ListActivity

Fields

- private LeDeviceListAdapter mLeDeviceListAdapter
  - List adapter to hold the list of Bluetooth LE devices found from a scan.
- private BluetoothAdapter mBluetoothAdapter
  - Represents a local Bluetooth adapter.
- private boolean mScanning
  - Tracks whether there is a scan in progress.
- private Handler mHandler
  - Handler to stop scanning after a time delay.
- private static final int REQUEST_ENABLE_BT = 1
  - Identifies a response from Activity that enables Bluetooth.
- private static final long SCAN_PERIOD = 3000
  - Length of time in milliseconds to scan for Bluetooth LE devices.
- private LeScanCallback mLeScanCallback
  - A discovered device during a scan.

METHODS

- *onCreate*
  public void **onCreate**(Bundle **savedInstanceState**)
  - **Usage**
    Initializes the Bluetooth LE device activity.

- *onCreateOptionsMenu*
  public boolean **onCreateOptionsMenu**(Menu **menu**)
  - **Usage**
    Initializes the menu activity.

- *onOptionsItemSelected*
  public boolean **onOptionsItemSelected**(MenuItem **item**)
  - **Usage**
    Called when an item in the menu is selected.

- *onPause*
  protected void **onPause**( )
  - **Usage**
    Called to pause an activity.

- *onResume*
  protected void **onResume**( )
  - **Usage**
    Called to resume a previous activity.

- *onActivityResult*
  protected void **onActivityResult**(int **request**, int **result**, Intent **data**)
  - **Usage**
    Test response from request to enable Bluetooth.

- *onListItemClick*
  protected void **onListItemClick**(ListView **l**, View **v**, int **position**, long **id**)
  - **Usage**
    Device selected in list adapter.

- *scanLeDevice*
  private void **scanLeDevice**(final boolean **enable**)
  - **Usage**
    Scan for Bluetooth LE devices.

Class **MainActivity**

Responsible for displaying the Home screen and its metrics, handling exceptions, interfacing with the other components, and exiting the application.

Declaration

public class MainActivity **extends** ActionBarActivity

Fields

- private String mDeviceAddress
  - Bluetooth device MAC address.
- private String mDeviceName
  - Bluetooth device name.
- public double tripEconCount
  - Number of times the trip economy has been updated.
- public double tripEconSum
  - Sum of the trip economy.
- protected double galCons
  - Total number of gallons of gasoline consumed.
- protected long lastTime
  - Time since the vehicle metrics were updated.
- protected long lastTimeCons
  - Time since the number of gallons consumed were updated.

- protected double tripDist
  - Total number of miles traveled during this application session.
- protected String [ ][ ] dataArray
  - Array of OBD commands to send to the GEM device.
- String inData
  - Data received from the GEM device.
- protected boolean isAppLaunched
  - Tracks if the GEM application has been started.

Methods

- *sendData*
  protected void **sendData**( )
  - **Usage**
    Sends OBD commands to the GEM device.
- *readData*
  protected void **readData**(String **data**)
  - **Usage**
    Reads the data received from the GEM device.
- *getByteA*
  private int **getByteA**(String **str**)
  - **Usage**
    Reads first byte of data from the received data.
- *getByteB*
  private int **getByteB**(String **str**)
  - **Usage**
    Reads second byte of data from the received data.
- *getMAF*
  protected double **getMAF**( )
  - **Usage**
    Gets the MAF value from the data array.

- *getMPH*
  protected int **getMPH**( )
  - **Usage**
    Gets the vehicle speed value from the data array.

- *getRPM*
  protected int **getRPM**( )
  - **Usage**
    Gets the RPM value from the data array.

- *getInstantEcon*
  protected double **getInstantEcon**( )
  - **Usage**
    Calculates the instant fuel economy in mpg.

- *getGalsCons*
  protected double **getGalsCons**( )
  - **Usage**
    Calculates the number of gallons consumed during the application session.

- *getGPH*
  protected double **getGPH**( )
  - **Usage**
    Calculates the number of gallons consumed per hour.

- *getTripDist*
  protected double **getTripDist**( )
  - **Usage**
    Calculates the total distance traveled in miles during the application session.

- *getTripEcon*
  protected double **getTripEcon**( )
  - **Usage**
    Calculates the average fuel economy in mpg over the total distance traveled during the application session.

- *initializeDisplay*
  public void **initializeDisplay**( )
  - **Usage**
    Sets all values displayed in the UI to zero.

- *setContinuousDisplay*
  public void **setContinousDisplay**( )
  - **Usage**
    Updates the values displayed in the UI during the application session.

Class **BluetoothLeService**

Responsible for creating a GATT server and broadcasting data received from the GEM device.

Declaration

public class BluetoothLeService **extends** Service

Fields

- public final static String ACTION_GATT_CONNECTED
  - Represents an action to broadcast to MainActivity that the GATT server is connected.

- public final static String ACTION_GATT_DISCONNECTED
  - Represents an action to broadcast to MainActivity that the GATT server is disconnected.

- public final static String ACTION_GATT_SERVICES_DISCOVERED
  - Represents an action to broadcast to MainActivity that GATT services have been discovered.

- public final static String ACTION_DATA_AVAILABLE
  - Represents an action to broadcast to MainActivity that data from the GEM device is available.

- public final static String ACTION_DATA_WRITTEN
  - Represents an action to broadcast to MainActivity that data has been sent to the GEM device.
- public final static String EXTRA_DATA
  - Represents an action to broadcast to MainActivity that more data from the GEM device is available.
- public final static UUID MLDP_DATA
  - Represents MCU data UUID.
- public final static UUID MLDP_CONFIG
  - Represents MCU configuration UUID.
- private BluetoothAdapter mBluetoothAdapter
  - Represents a local Bluetooth adapter.
- private String mBluetoothDeviceAddress
  - Connected Bluetooth device MAC address.
- private BluetoothGatt mBluetoothGatt
  - Controls the Bluetooth communication link.
- private BluetoothManager mBluetoothManager
  - Used to get the Bluetooth adapter.
- private final IBinder mBinder
  - Binds the BluetoothLeService to MainActivity.
- private final BluetoothGattCallback mBluetoothGattCallback
  - Tracks GATT callback events.

Methods

- *onBind*
  public IBinder **onBind**( )
  - **Usage**
    Binds the BluetoothLeService to MainActivity.

- *onUnbind*
  public boolean **onUnbind**(Intent **intent**)
  - **Usage**
    Closes Bluetooth GATT server connection.

- *broadcastUpdate*
  private void **broadcastUpdate**(final String **action**)
  - **Usage**
    Broadcasts an intent to MainActivity.

- *broadcastUpdate*
  private void **broadcastUpdate**(final String **action**, final BluetoothGattCharacteristic **c**)
  - **Usage**
    Broadcasts an intent and data to MainActivity.

- *initialize*
  public boolean **initialize**()
  - **Usage**
    Gets BluetoothManager and BluetoothAdapter.

- *connect*
  public boolean **connect**(final String **address**)
  - **Usage**
    Opens a Bluetooth GATT connection to a Bluetooth LE device.

- *disconnect*
  public void **disconnect**()
  - **Usage**
    Ends an existing Bluetooth GATT connection to a Bluetooth LE device.

- *getSupportedGattServices*
  public List¡BluetoothGattService¿ **getSupportedGattServices**()
  - **Usage**
    Retrieves the list of supported GATT services on the connected Bluetooth LE device.

- *readCharacteristic*
  public void **readCharacteristic**(BluetoothGattCharacteristic **c**)
  - **Usage**
    Reads a given characteristic from the connected Bluetooth LE device.

- *writeCharacteristic*
  public void **writeCharacteristic**(BluetoothGattCharacteristic **c**)
  - **Usage**
    Writes to a given characteristic of the connected Bluetooth LE device.

- *setCharacteristicNotification*
  public void **setCharacteristicNotification**(BluetoothGattCharacteristic **c**, boolean **enabled**)
  - **Usage**
    Enables notification when a given characteristic changes.

6.6 Fuel Optimization Algorithm

The firmware on the GEM system deals with polling the vehicle for as much fuel data as possible on every trip a user makes. This raw data is then transmitted to a user's mobile device with some basic organization. An analysis of the data is left to the mobile application with the use of high level programing. It is our intent to design a high level fuel optimization algorithm to efficiently process the data and achieve our team goal of providing practical fuel advice to users. While some vehicles provide unique PID's to measure specific fuel injection, others don't, therefore we can't rely on this data when designing fuel algorithms. We instead will opt to use widely available ECU data to meet the GEM system's requirements. The useful messages for this task are engine RPM, speed, air flow rate, throttle position, and fuel rate.

By synchronizing instantaneous data with fixed-interval time polls, average rates can be efficiently calculated. For instance, instantaneous air flow rate and fuel rate can be used to calculate an instantaneous miles per gallons (MPG) measurement. Synchronizing several measurements over equally divided time segments would provide an average MPG measurement. This technique can be generalized for acceleration and de-acceleration data as well. Thus, the first stage of the algorithm deals with synchronizing instantaneous data and time. The data is then normalized to lower error percentages and placed in a histogram-like data structure. The device will then compare this to, static, "optimal rates" pre-stored in the application software. These

optimal rates will be derived from readily available fuel research. A point-by-point distance formula will be used to generate a correlation coefficient between the gathered average rates and optimal driving rates. A set of thresholds will then be used to trigger fuel advice based on several correlation coefficients for each driving parameter measured by the GEM system.

Chapter 7

Project Prototype Details

7.1 BOM

The bill of material for the GEM is shown in Table 7.1. It includes prices, part numbers, and quantities as well as reference designators that match the schematics in Appendix B.

7.2 PCB Vendor and Assembly

For the final design and build of the GEM we chose OSHPark for the PCB run and we purchased a hot air reflow station for the soldering. The following sections detail why those choices were made.

7.2.1 Printed Circuit Board

Thanks to the hobbyist market there are many printed circuit board (PCB) vendors to choose from. The primary requirements for this project as it is on a very limited time frame are cost and turnaround time. Table 7.2 gives an overview of some of the more popular PCB prototyping services. Due to impedance matching requirements for the Bluetooth system a four layer design will be be used unless we can optimize for a two layer board. Our initial budgeting considered a four layer board. The expected board size is roughly 2.5 cm by 7.5 cm. In the final design we chose a two layer board

that was stacked using standard pin headers. This simplified the design process as well as helped us to reduce overall cost.

Companies like 4PCB and OURPCB give flat rates for boards up to a certain size. The PCB for the GEM will be very small so paying by the inch is a better solution. OSHPark charges $10.00 per square inch for a four layer board with free shipping. The only disadvantage is the long lead time. However if we stick to a schedule and plan ahead the long lead time won't effect the project. If it turns out that we need a quicker turn around one of the other companies will be selected on a case by case basis.

7.2.2 Assembly

The GEM utilizes many surface mount passive components and some of the remaining parts are only available in the QFN package. QFN stands for quad flat pack no lead. This means that there are connection points on the bottom of the chip but no lead for us to hand solder. Due to the nature of the parts and the packages that are available assembly of the prototype will require the use of reflow soldering. Reflow soldering involves attaching the parts to the PCB using a solder paste and then heating the assembly to melt the paste to provide a secure mechanical and electrical connection.

Ideally a PCB assembly house would be used in order to machine assemble or professionally hand assemble the board. However the cost of assembly in our case would be prohibitively expensive. On the order of $300 to $600. This would consume almost the entire sponsorship budget.

There are many different techniques that can be used for reflow soldering. The ideal process is to use a specially designed reflow oven that can be set to use a predefined temperature profile to bake the assembly at the manufacturers guideline temperatures for the proper times. In lieu of an oven designed for reflow it is possible to use a household toaster oven though in that case control over the temperature profile is lost and consistent heating is not guaranteed. The third technique is to use a hot air rework station and heat each of the parts or areas of the PCB individually. This technique is the most readily available so will be used unless reflow ovens are found to be available.

Because of the budgetary concerns attempts will be made at having the team members perform the soldering. However if there are unexpected problems or assembly is too difficult additional resources will be found to have the board professionally assembled.

7.3 Software

All software implemented in the prototype of the GEM system is described in this section.

7.3.1 Initial Firmware Design

The following sections describe the initial firmware design used in the first revision of the GEM system.

7.3.1.1 Main System

The MCU firmware was designed with modularity, portability, and reusability in mind. The design revolves around five interacting subsystems, gelled together by a main subroutine. A finite state machine best describes the design of the main routine which is shown in Figure 7.1. The subsystems are implemented as modular subroutines and each handle an independent task as follows:

- Bluetooth Connection
- Vehicle Profiles
- Onboard Storage
- OBD-II Data Pipelining
- Power Control

Upon powering on the device for the first time, an initial setup state is entered by the MCU firmware. During this initialization stage, the MSP430 establishes a connection, and configures the STN1110, RN4020, and microSD chips to enable the core functionality of GEM. An "initial state" flag is cleared to ensure that these operations don't have to be repeated. Upon all subsequent device powering's a default operation state is executed instead. In this default state the VIN is always retrieved and passed on to the Profiles subroutine. The Profiles subroutine passes profile data to the Bluetooth subroutine. The main subroutine decides whether to proceed to either the Storage or Data subroutines based on connection status parameters provided by the Bluetooth subroutine. If a connection is not available or if there are previously unsent packets in the microSD card, the Storage subroutine is initiated. The Data subroutine is only initiated if a connection is available and there isn't any unsent data. Finally, the Power subroutine interrupts any executing routine when the Vehicle is turned in order to cache important data and power off the device.

7.3.1.2 Vehicle Profile Subsystem

There is a small footprint profile system onboard that serves two primary functions. The first is to permit GEM to be a portable system. The second is to enable quicker start up times when the vehicle is already known. The system software operates in either a Fetch state or a Create state. Up to 5 vehicle profiles may be stored, identified by the VIN and some other parameters. The system software compares the VIN to existing profiles. If there is a match, the corresponding profiles cached data is provided to the main subroutine. If the VIN doesn't match an existing profile, a new profile is created, or an older one is replaced if 5 unique profiles already exist.A system diagram is shown in Figure 7.2.

7.3.1.3 Bluetooth Connection Subsystem

The Bluetooth subroutine is triggered based on the Profile system's parameters, or status flags for non-transmitted packets provided by the Data system. When the system software in its default state it scans for connectable devices, if none are discovered a flag is set and execution is passed to the Storage system. If a device is found, the system proceeds to a Paring state which determines if a previously connected device is available. If an unknown device is paired with the GEM system, an authentication step is required, which is handled in a Connection state. Once authentication is achieved or if a recognized device is connected, the MLDP state is triggered. In this state the device prepares the RN4020 to blast OBD data to the mobile device. Once this is complete, a connection report frame is used to initiate the Data subroutine. A system diagram is shown in Figure 7.3.

7.3.1.4 Onboard Storage Subsystem

The Storage subroutine is triggered after the Bluetooth subroutine finishes execution. This subroutine operates in either a Write or Read state, based on flags set by other routines. When a connection is not available the system is placed in a write state, storing all OBD data in microSD blocks. The read state is entered when there are blocks in the microSD card that have not been transmitted yet. When a connection is available, and there aren't any non-transmitted blocks the system cedes execution. A system diagram is shown in Figure 7.4.

7.3.1.5 OBD-II Data Subsystem

When several conditions have been met the main routine transfers execution to the Data subroutine. These conditions are passed in through several status flags. The system software alternates between two states, a Request state and a Pipeline state. In the Request state the ECU is constantly polled through the STN1110 and a small local frame is built. Once the frame is ready, it is sent to the Pipeline stage, which deals with transmitting to the RN4020 in MLDP mode via UART. Both states monitor for any status flags changes, such as loosing Bluetooth connection, and terminate the subroutine if there are any changes, passing control back to the main routine. A system diagram is shown in Figure 7.5.

7.3.1.6 Power Control Subsystem

The Power subroutines main task is to handle the power states of all peripherals. Any of the other subroutines execution can be interrupted by the Power subroutine, when it is determined that the vehicle has been turned off. Coming from an interrupted routine the Power system initiates in the Interrupt state. In this state, important data (status flags) necessary for the next startup is cached, peripheral devices power states are altered, and any other tasks are interrupted. Then, an Idle no power state assumed. Once the device is powered on again, the Idle state wakes up peripheral devices and returns execution to the main routine.A system diagram is shown in Figure 7.6.

7.3.2 Final Firmware Design

Following the completion of the first revision of GEM, several optimizations were necessary to meet initial design specifications. A new firmware design was fully constructed to be implemented on the final GEM system. The complexity of the design was trimmed down to a main routine with two receive buffers rather than the inefficient five sub-routines of the initial design. A diagram of the final firmware is shown in Figure 7.7. The key modifications which allowed an exponential speedup in GEM's communication including cutting out the profile and data logging subsystems. The vehicle profile system was intended to speed-up device power on by matching the VIN with an existing profile and loading flags and device states from memory. However, it was discovered that with the implementation of the new firmware, fetching and storing this information from memory was actually a bottleneck in speed for device initialization. Rather than store profiles, all the vehicle data and device setup can be

re-retrieved from the ECU on every power on, much faster than it can be retrieved from MCU or SD card memory.

Additionally, SPI communication with an SD card with two UART communications was causing unwarranted synchronization and slowdown issues. This added with the difficulty of dealing with raw read and writes, in a low memory environment, and the fact that data-logging is not an essential feature of the GEM device lead to the removal of the data logging system as well. Thus, as seen in the state diagram the final firmware design of the GEM system relies on two buffers, implemented as circular queues. One buffer is for incoming RN4020 UART messages, the other for STN1110 messages. The queues forces the MCU to handle messages in the order that they were received, resolving any synchronization issues. Based on the received messages the MCU transitions between three states: Interpreting Bluetooth responses, interpreting STN messages, and Bluetooth transmission.

Table 7.1: Final BOM

Designator	Quantity	Part Number / Value	Part Number Link	Price Each	Total
C1,C2	2	30pF	C0805C300J1GACTU	0.27	0.54
C3,C4,C5	3	1uF	77-VJ0805Y105KXQTBC	0.05	0.15
C6,C7	2	560pF	77-VJ0805A561KXXCBC	0.06	0.12
C8,C17,C20	3	0.1uF	581-08055C104J4T4A	0.15	0.45
C9	1	10nF	C0805C103K5RACAUTO	0.12	0.12
C10	1	47uF	963-JMK212BJ476MG-T	0.77	0.77
C11, C12	2	4.7uF	81-GRM426X475K10L	0.33	0.66
C13,C14	2	100nF	80-C0805C104K5R	0.1	0.2
C15	1	22uF	581-08056D226M	0.1	0.1
C16	1	4.7uF	581-08056D475K	0.1	0.1
C18	1	2.2n	VJ0805Y222JXJPW1BC	0.06	0.06
C19	1	10U	VJ0805G106KXQTW1BC	0.09	0.09
D1, D2, D3, D4,D5,D6	6	LED	APT2012ZGCK	0.47	2.82
D7, D8, D9, D14, D15	5	BAT46	78-BAT46W-HE3-08	0.15	0.75
D10	1	SMAJ16CA	SMAJ16CA	0.5	0.5
D11	1	S2A	512-S2A	0.47	0.47
D12,D13	2	MBR0530T1G	863-MBR0530T1G	0.36	0.72
F1	1	1206L075/13.2	1206L075/13.2WR	0.49	0.49
L1	1	SDR0604-470KL	SDR0604-470KL	0.43	0.43
L2	1	SRN3015-330M	SRN3015-330M	0.36	0.36
Q1,Q2,Q5	3	MMBT2222	863-MMBT2222ALT1G	0.12	0.36
Q3	1	BSS138	863-BSS138LT1G	0.26	0.26
Q4	1	MMBT3646	MMBT3646	0.21	0.21
Q6	1	MMBT3640	512-MMBT3640	0.61	0.61
Q7	1	N-Channel Mosfet	SQ2310ES-T1-GE3	0.68	0.68
Q8,Q9	2	ZXMP6A13F	ZXMP6A13FTA	0.58	1.16
R1, R6, R41, R42, R49, R50	6	100K	71-CRCW0805J-100K-E3	0.08	0.48
R2, R17, R18, R21, R24, R25, R30, R31, R33, R36, R37, R38, R39, R40, R48	15	10K	71-CRCW0805J-10K-E3	0.08	1.2
R3, R7, R19, R20, R27	5	1K	71-CRCW0805J-1K-E3	0.08	0.4
R4	1	47k	CRCW080547K0JNEA	0.08	0.08
R5	1	10	71-CRCW080510R0JNEA	0.08	0.08
R8	1	4.7K	71-CRCW0805J-4.7K-E3	0.08	0.08
R9,R10	2	100	71-CRCW0805100RJNEA	0.08	0.16
R11, R12, R13, R14, R15, R16	6	330	CRCW0805330RFKEA	0.08	0.48
R22, R23	2	510 / 0.5W	667-ERJ-P06J511V	0.24	0.48
R26	1	374	CRCW0805374RFKEA	0.08	0.08
R28	1	866	CRCW0805866RFKEA	0.08	0.08
R29	1	2.2K	71-CRCW08052K20JNEB	0.08	0.08
R32	1	240	71-CRCW0805240RJNEA	0.08	0.08
R34	1	2K	71-CRCW08052K00JNEA	0.08	0.08
R35	1	8K (8.2K)	CRCW08058K06FKEA	0.08	0.08
R43	1	28K	71-CRCW0805-28K-E3	0.08	0.08
R44	1	8.45K	71-CRCW0805-8.45K-E3	0.08	0.08
R45	1	36K	71-CRCW080536K0JNEA	0.08	0.08
R46	1	6.49K	71-CRCW0805-6.49K-E3	0.08	0.08
R47	1	62K	71-CRCW080562K0FKEA	0.08	0.08
U2	1	10u ESR<5	TPSR106K006R1500	0.83	0.83
U3	1	MCP2551	579-MCP2551-I/SN	1.12	1.12
U4	1	LM339	926-LM339AMX/NOPB	0.92	0.92
U5	1	LM317AEMP	926-LM317AEMP	1.12	1.12
U1	1	STN1110	STN1110	9.99	9.99
U11	1	MSP430F247	MSP430F247TPMR	6.93	6.93
U12	1	RN4020	RN4020-V/RM	10.61	10.61
U13	1	microSD	DM3CS-SF	1.5	1.5
U14	1	JTAG_HEADER	2514-6002UB	2.7	2.7
U6	1	LMR16006XDDCT	LMR16006XDDCT	3.86	3.86
U7	1	LMR16006YDDC	LMR16006YDDCT	3.86	3.86
X1	1	16Mhz	403C11E16M00000	1.1	1.1
P1	1	CONN_01X06_MALE	68000-406HLF	0.24	0.24
P3	1	CONN_01X06_FEMALE	929870-01-06-RA	0.97	0.97
P2	1	CONN_01X08_MALE	AMP 825433-8	0.84	0.84
P4	1	CONN_01X08_MALE_90	68016-208HLF	0.53	0.53
P5 P6	2	CONN_01X08_FEMALE	929870-01-08-RA	1.22	2.44
JP2	1	JUMPER3	FCI 69190-403	0.14	0.14
TOTAL ITEMS	116		TOTAL PRICE	56.77	64.62

Table 7.2: PCB Vendor Comparison

Company	PCB Cost	Shipping Cost	Turnaround
4PCB	$66.00	$20.00 (avg.)	5 Business + Shipping
OURPCB	$236.00	International Rates	2 Days + Shipping (China)
Sunstone	$52.00	Free	5 Days + Shipping
Seedstudio	$49.90	Air Mail	7 Days + Shipping
OSHPark	$30.00	USPS (Free)	21 Days
ITeadStudio	$88.00	Ground	Not Listed

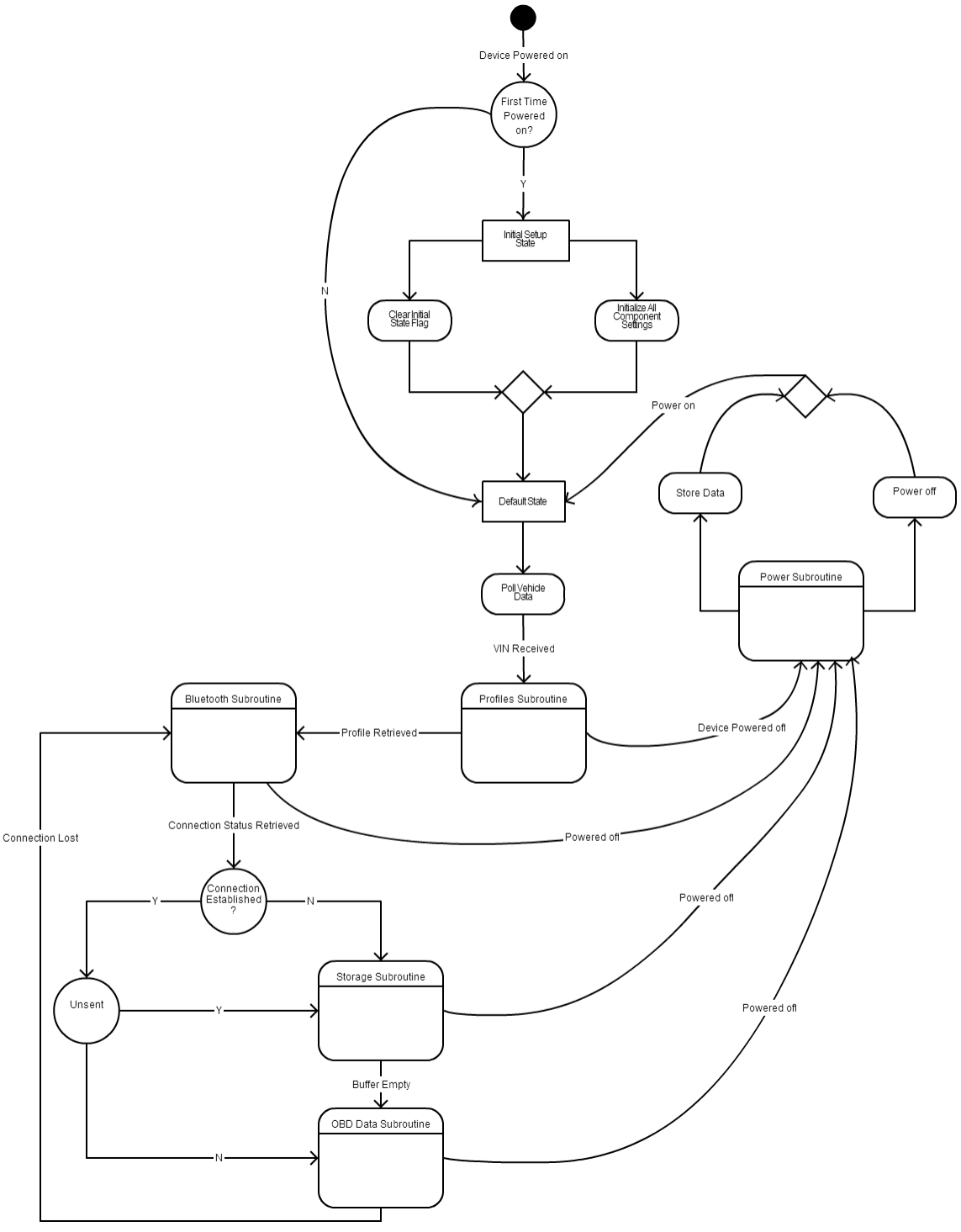


FIGURE 7.1: Main Firmware State Machine

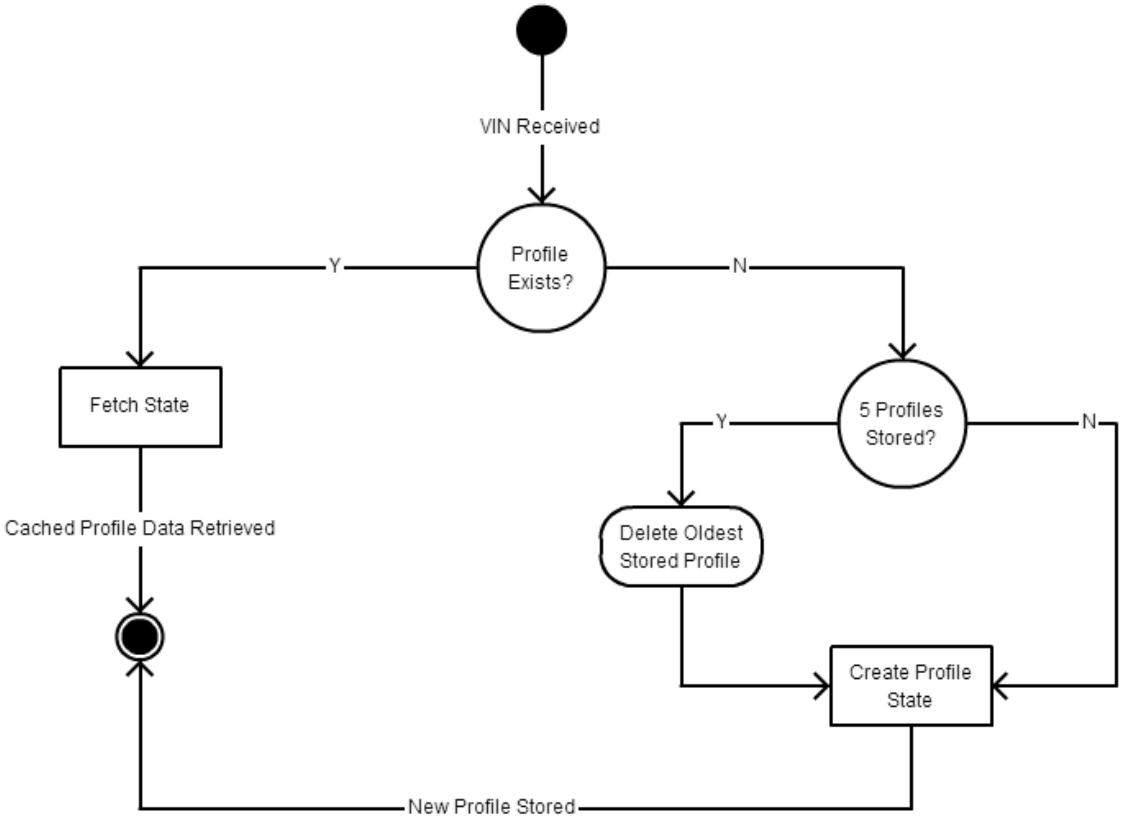


FIGURE 7.2: Profile Subsystem

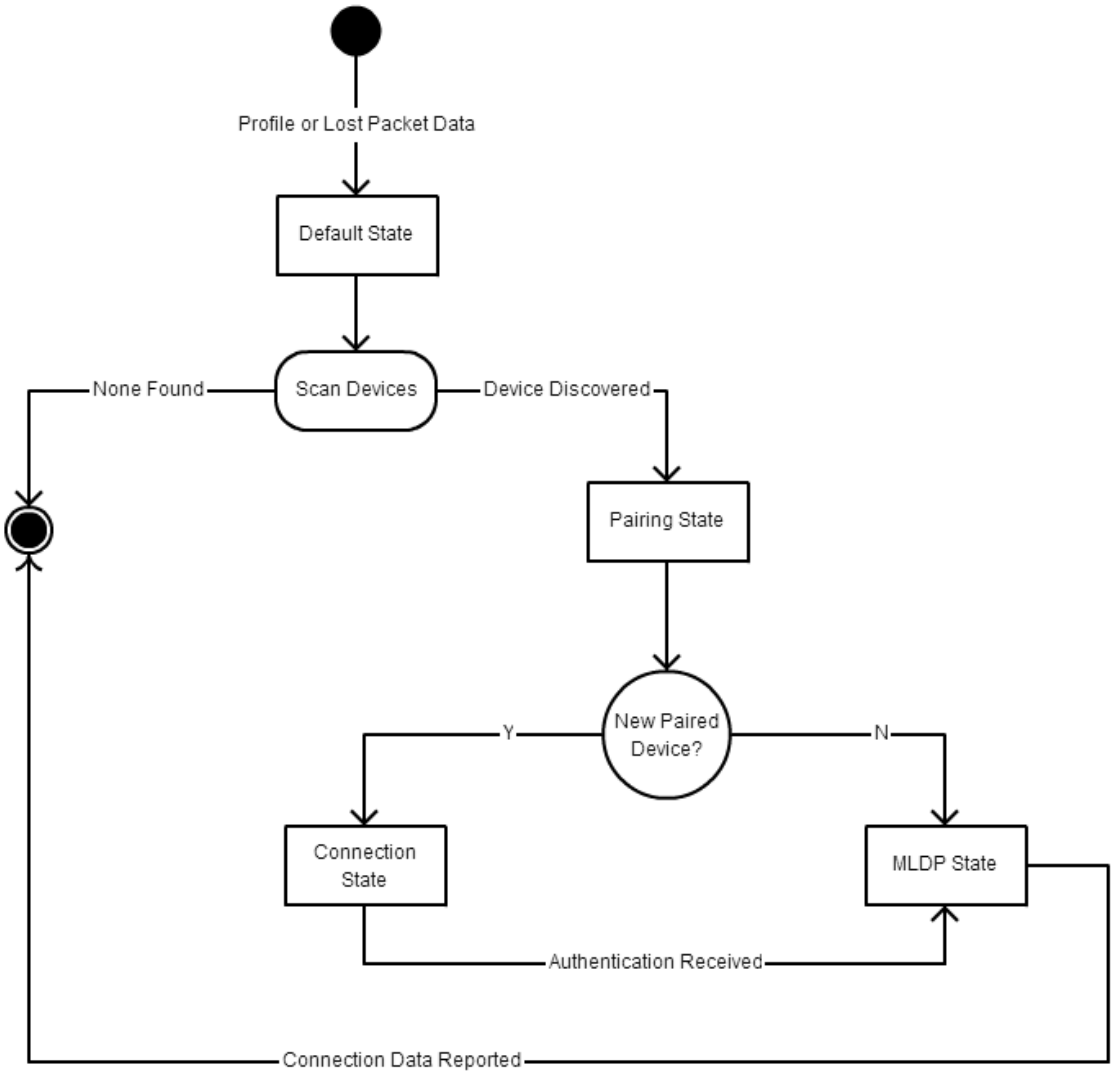


FIGURE 7.3: Bluetooth Connection Subsystem

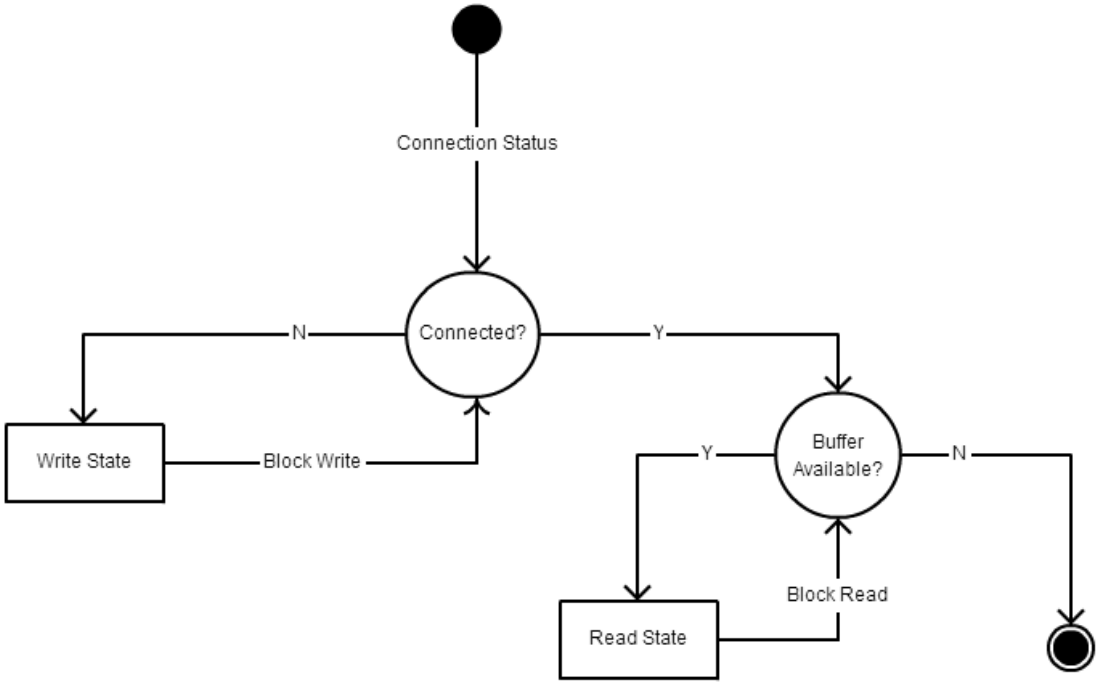


FIGURE 7.4: MicroSD Storage Subsystem

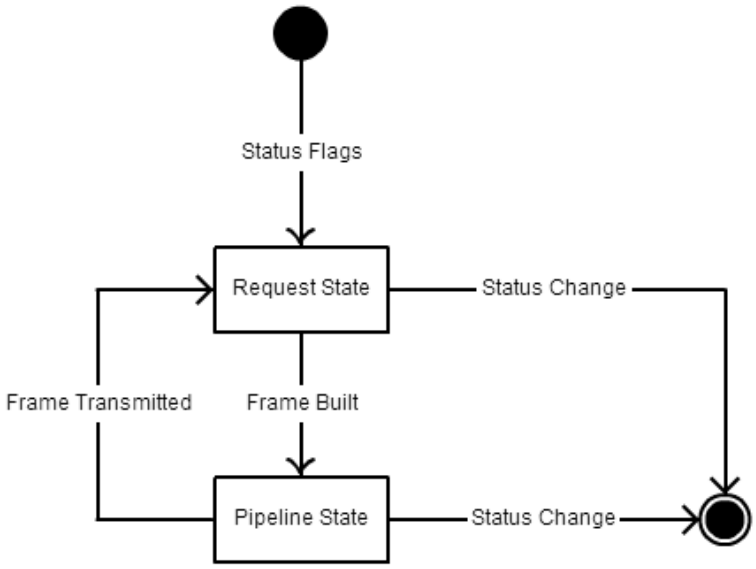


FIGURE 7.5: Data Pipeline Subsystem

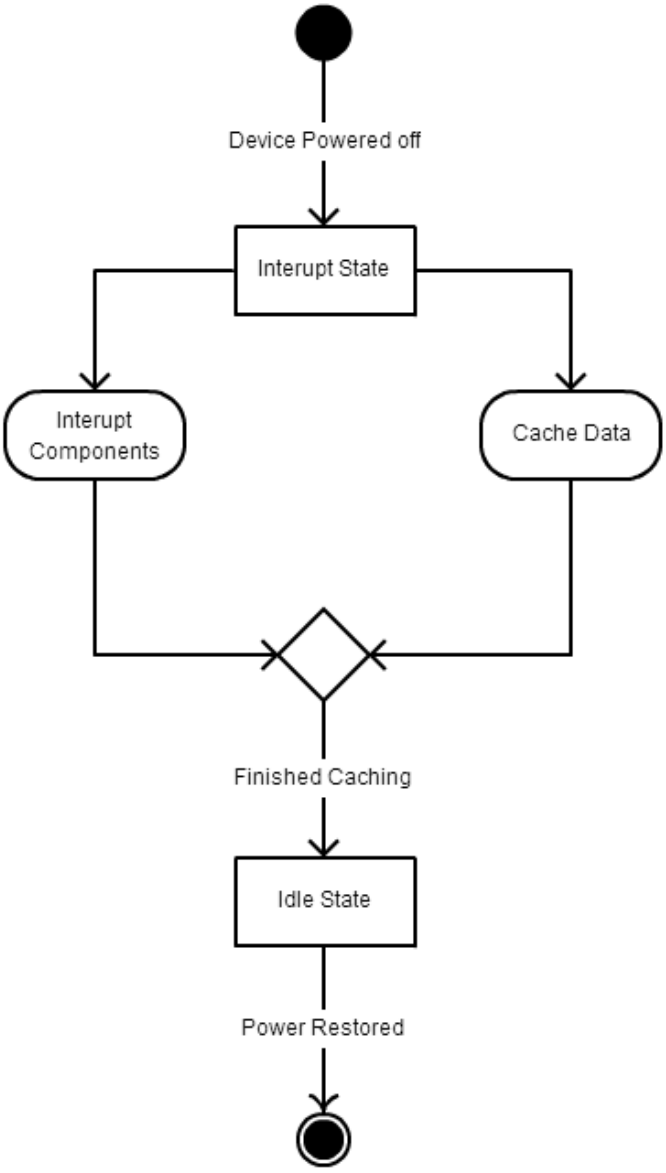


FIGURE 7.6: Power Subsystem

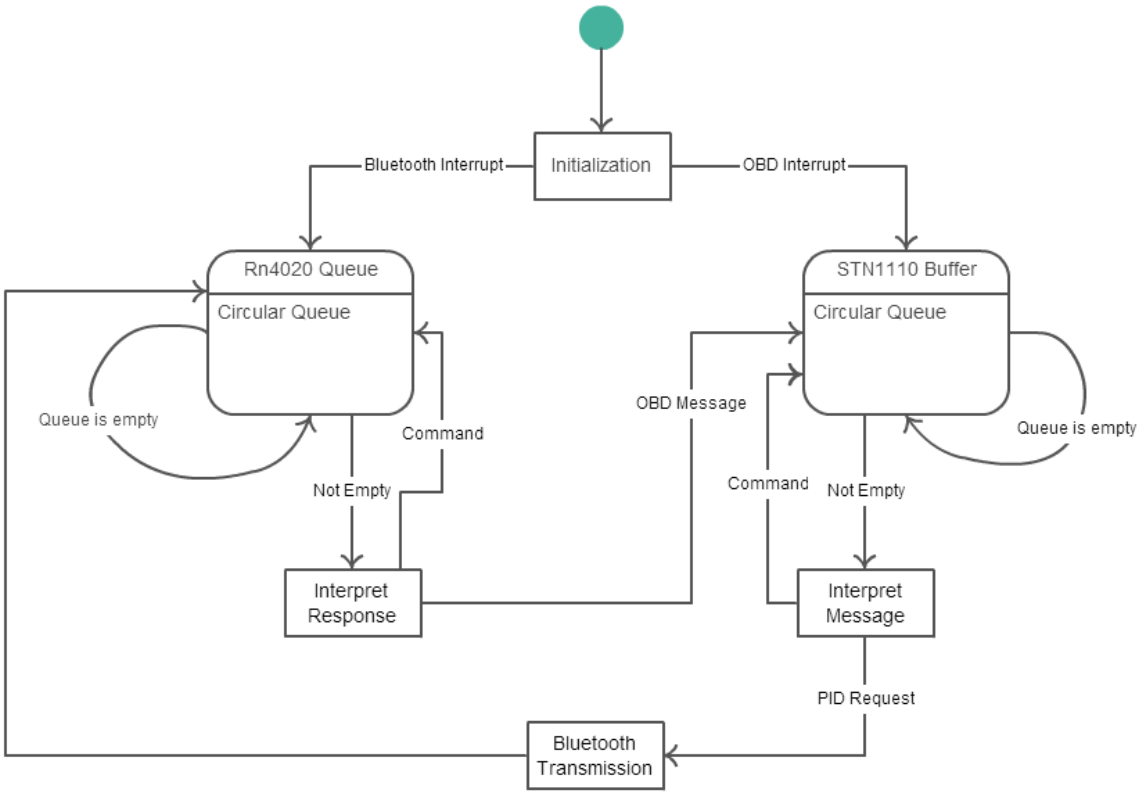


FIGURE 7.7: Final Firmware Design

Chapter 8

Project Prototype Testing

8.1 Hardware

Due to the complexity of the project specialized test equipment will be required in order to test the device in the laboratory. Fortunately the most expensive equipment is available in the senior design lab. However there are some pieces of test equipment that will be required that is not available in the lab. First and foremost is an OBD-II test bench so that testing can be done on the GEM without having to hook it up to an actual vehicle to verify operations. This is a requirement not only for convenience but also to allow other pieces of test equipment such as an oscilloscope or a logic analyzer to be connected to the device while it is operating. The senior design lab at UCF has a variety of equipment available such as oscilloscopes, power supplies, and digital multimeters. However some of the specialized equipment we will require for debugging and design is not available and will need to be acquired.

8.1.1 Test Equipment

8.1.1.1 OBD-II Test Bench

Professional OBD-II test benches can cost many thousands of dollars which puts them out of reach our analysis. Though it has it's limitations the OBD-II Emulator MK2 from Freematics is sufficient to test the basic functionality of the device. This

emulator simulates CAN, KWP2000, and ISO9141-2 though it lacks SAE J1850. Other functions that are provided include:

- Can respond to OBD-II PID poll
- Can respond to VIN requests
- Provides power to device under test
- Emulates diagnostic trouble codes
- Supports changing VIN's
- USB control via Windows software package

This test bench will be used as the final lab testing environment. It provides many options for adjusting what is sent to the GEM as shown in figure 8.1. Once the device is confirmed to work on the test bench the device will then be tested in an automobile.

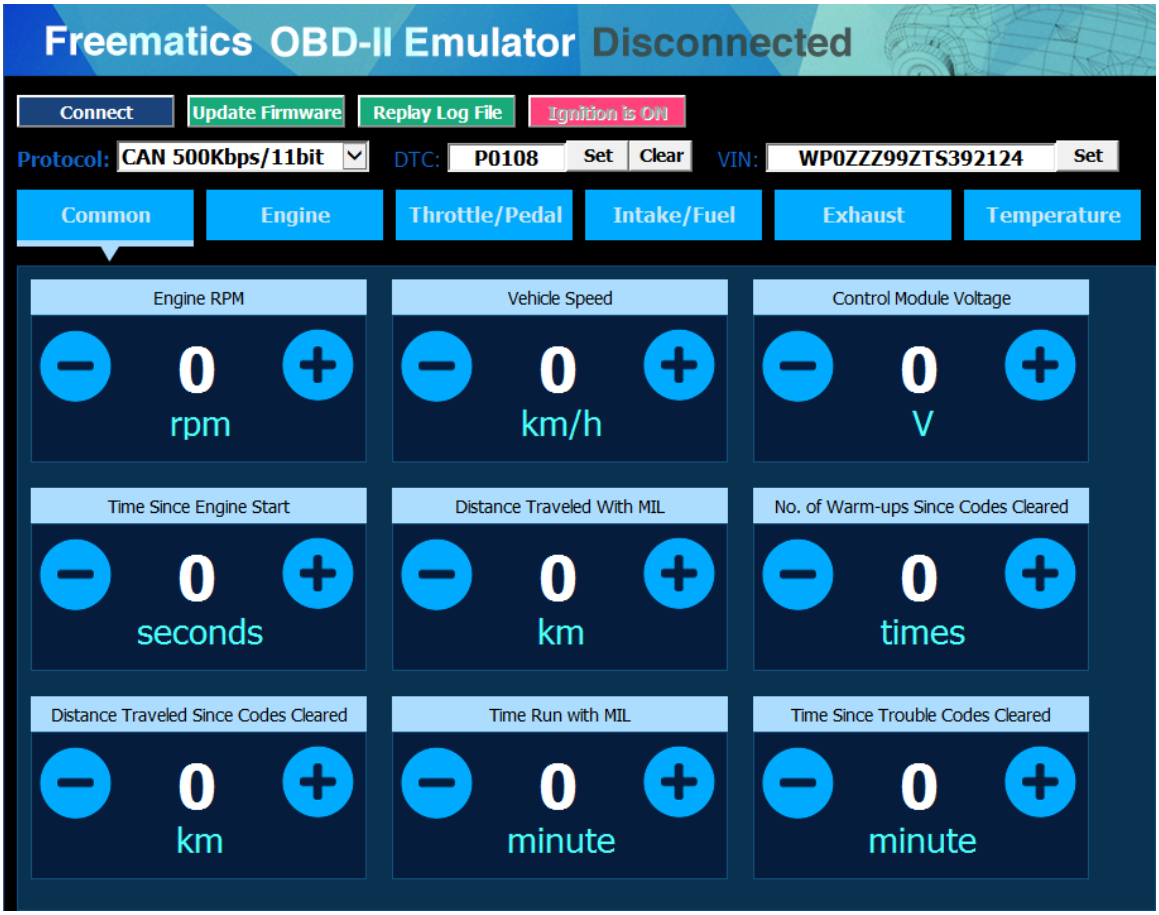


FIGURE 8.1: Freematics OBD-II Emulator Interface

8.1.1.2 Bus Pirate

There are two primary means of communication between the devices on the the GEM. Devices communicate either using their UART or in the case of the SD card SPI. In order to debug each subsystem it will be necessary to communicate with each subsystem individually. In order to faciliate this a simple device from Dangerous Prototypes called the Bus Pirate will be used. The Bus Pirate can be connected in circuit and then used as a USB to UART bridge. This will allow reading from and transmitting data to each of the subsystems. For example the Bus Pirate could be connected to the UART on the STN1110 to verify that it is receiving data properly and that it is in the correct mode of operation.

8.1.1.3 MSP430 Flash Emulation Tool

An MSP-FET flash emulation tool is required in order to program the MSP430 device. The MSP-FET connects to the MSP430 via JTAG or Spy-Bi-Wire (2-wire JTAG). The GEM is designed to use the Spy-Bi-Wire interface. This tool not only will allow the programming of the MSP430 device but also debugging while the device is in operation. By having the GEM connected to the host PC via the JTAG port Code Composer Studio can set debug points in the firmware. The firmware can then be stepped through line by line to verify that it is functioning as expected.

8.2 Hardware Testing Procedure

In order to verify that the GEM is operating as expected and performing the expected functions a variety of test procedures will be used.

8.2.1 Power Test

For the power test the following lab equipment will be used:

- Agilent E3630A Triple Output DC Power Supply
- Tektronix DMM 4050 Multimeter

This first test is designed to verify that the GEM is being powered properly. This test should be performed before all other tests and as a first step whenever troubleshooting

the device. Since the GEM uses multiple voltages and two different voltage regulators there is a lot of room for error here. To verify the correct operation of the on board regulators the power supply will be connected to the the power and ground pins on the main OBD-II connector of the GEM. The DMM will be connected to ground and the 5 V test pin. The voltage from the power supply will be slowly increased from 0 V to 12 V. At 12 V the measured output voltage should be 5 V. This procedure is then repeated for the 3.3 V system.

8.2.2 STN1110 Verification

For this verification test the following equipment will be used:

- Freematics OBD-II Emulator
- Dangerous Prototypes Bus Pirate

To verify the correct operation of the OBD-II to serial interface chip (STN1110) the GEM is connected to the OBD-II emulator and the OBD_UART TX and RX jumpers are disconnected. The Bus Pirate is the connected as shown in figure 8.2. The Bus

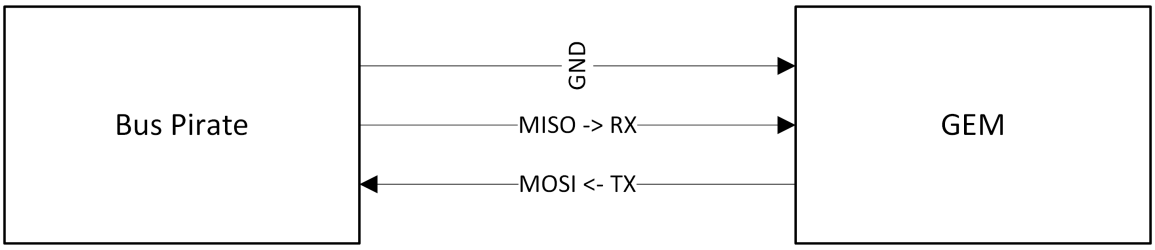


FIGURE 8.2: Bus Pirate UART Connection Guide

Pirate needs to be connected to the host PC. The tester then connects to the Bus Pirate using a terminal program such as PuTTY. If there is no prompt press enter once. when the "HiZ>" prompt is displayed enter the following sequence:

- m – change mode
- 3 – UART mode
- 7 – 38400 Baud
- 1 – 8 data bits, no parity
- 1 – 1 stop bit
- 1 – Idle 1 receive polarity
- 1 – Open drain output

at the "UART>" prompt press '0' for the macro menu then '3' for "Bridge mode." This completes the process for configuring the Bus Pirate. At this point the PC terminal program is connected to the STN1110 via the Bus Pirate and ready to test.

Once power is applied to the GEM the terminal should display:

```
ELM327 v1.3a

>
```

At this prompt the user should enter 'SN' to have the STN1110 display the factory programmed serial number. This number is a unique 12 digit identifier. If the STN1110 returns an acceptable value the user should then enter 'DP' to have the STN1110 display the current communication protocol being used. This should match the protocol set by OBD-II emulator.

8.2.3 Bluetooth Verification

For the Bluetooth test the following lab equipment will be used:

- Android device
- Freematics OBD-II Emulator
- Dangerous Prototypes Bus Pirate

For this test the GEM is connected to the OBD-II Emulator and the Bus Pirate is connected to the GEM by disconnecting the BT_UART TX and RX jumpers and connecting as shown previously in figure 8.2.

The Bus Pirate needs to be connected to the host PC. The tester then connects to the Bus Pirate using a terminal program such as PuTTY. If there is no prompt press enter once. when the "HiZ>" prompt is displayed enter the following sequence:

- m – change mode
- 3 – UART mode
- 9 – 115200 Baud
- 1 – 8 data bits, no parity
- 1 – 1 stop bit
- 1 – Idle 1 receive polarity
- 2 – Normal output

at the "UART>" prompt press '0' for the macro menu then '3' for "Bridge mode." This completes the process for configuring the Bus Pirate. At this point the PC terminal program is connected to the RN4020 Bluetooth module via the Bus Pirate and ready to test.

The Android device needs to be running a Bluetooth serial terminal, for example SENA Bterm. The Android device should attempt a connection with the RN4020. If it connects the connect status LED should light. Once a connection is confirmed the user should send a sting of characters from the serial terminal on the PC. This string of characters should be shown in the Bluetooth serial terminal on the Android device.

8.2.4 Testing Results

The results of the testing procedures for the PID's is shown in Table 8.1. This is the final test after all other basic tests have passed. It shows that the GEM responds to commands, submits them to the ECU and can send the proper response.

Table 8.1: Hardware Testing Results

Test PID	Values set on emulator	Responses	Description
01 1F	{5,100,2000,5000,3000}	411F{ 0005,0064,07D0,1388,7530}	Run time since engine start (in seconds)
01 0C	{1000,2000,3000,4000,5000}	410C {0FA0,1F40,2EE0,3E80,4E20}	Engine RPM in 1/4th value
01 0D	{10,25,50,100,125}	410D {0A,19,32,64,7D}	Speed in km/hr
01 04	{5 ,10, 20, 30, 40, 50, 60, 70, 80, 90, 95}	4104 {0C,19,33,4C,66,7F,99,B2,CC,E5,F2}	Engine load %
01 11	{10,25,50,75,100}	4111 {19,3F,7F,BF,FE}	Throttle position, as a %
01 10	80	4110 1F40	MAF air flow rate %
01 5E	{10,15,20,40,100,150}	415E {0087,008C,0091,00A5,00E1,0BB5}	Engine fuel rate. L/H

8.3 Software

8.3.1 Verification and Validation

The Android application designed for the GEM device was subjected to a multitude of tests throughout development and they all fell into one of two categories: verification and validation. Because our team designed and tested the device we could evaluate the correctness of our methods every step of the way. To verify that our application was built using the accepted methods and practices of Android application development our team relied on the Android Development Team's guides and references. Outlined therein were instructions for using dynamic software testing tools provided in the Android framework.

The testing procedures implemented over the lifecycle of the GEM Android application included but were not limited to: unit, integration, system, and acceptance testing. The focus of this section is to outline the Android application specific tests that were performed.

8.3.2 Activity Testing

Testing every screen or view that a user can interact with falls under the category of activity testing. The activity testing frameworks were used to test:

- Input validation
- Lifecycle events
- Intents
- Runtime configuration changes
- Screen sizes and resolutions

8.3.3 Service Testing

The goal of service testing was to check that services handle calls properly in every state. Service testing is done most reliably in isolation and is best done using a mock application. Ensuring that data transmitted to the Android device is properly written to internal storage and can be accessed in the future requires services.

8.3.4 Content Provider Testing

The GEM application had no need for content providers, but considering the eventual need for a content provider was reasonable. This testing consisted of designing the classes and activities with the expressed notion that content providers could be used in later versions of the application.

8.3.5 Accessibility Testing

Accessibility testing was conducted in an effort to provide this application to users who may or may not actually drive a vehicle. As noted in the "Software Requirements and Specifications" the application was designed in a manner that allows color blind users to use it.

8.3.6 UI Testing

The automation of user interface testing greatly reduced the amount of time that team members spent testing each combination of user actions. The majority of manual UI testing was performed on an actual Android device and included users other than a design team member. This divide and conquer method greatly increased our chances of finding and fixing any bugs in the application and granted us feedback from potential users.

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Chapter 9

Administrative Content

9.1 Budget and Finances

Due to a sponsorship by The Boeing Company the baseline budget for the GEM is $774.08. We were able to design and build the prototype within the constraints of this budget with just over one hundred dollars remaining.

9.2 Milestones

Our original goal was to have the project design and review complete by the end of Senior Design I and to order parts and PCB's in order to have them ready by the beginning of Senior Design II. As this was a complicated system testing was crucial to make sure that we stay on track. In order to facilitate testing we had quick builds with long testing periods in between. We were close to maintaining the schedule and thanks to that the prototype was complete and the demo successful. The original schedule is shown in figures 9.1 and 9.2

Schedule

ID	Task Name	Duration	Start	Finish
1	**Senior Design I**	**61 days**	**Mon 9/8/14**	**Mon 12/1/14**
2	**Research**	**20 days**	**Mon 9/8/14**	**Fri 10/3/14**
3	Bluetooth System	20 days	Mon 9/8/14	Fri 10/3/14
4	Storage System	20 days	Mon 9/8/14	Fri 10/3/14
5	MCU	20 days	Mon 9/8/14	Fri 10/3/14
6	Accelerometer	20 days	Mon 9/8/14	Fri 10/3/14
7	ODBII to Serial Interface	20 days	Mon 9/8/14	Fri 10/3/14
8	Transceivers	20 days	Mon 9/8/14	Fri 10/3/14
9	Power System	20 days	Mon 9/8/14	Fri 10/3/14
10	Android BLE	20 days	Mon 9/8/14	Fri 10/3/14
11	Android UI	20 days	Mon 9/8/14	Fri 10/3/14
12	**Research Complete**	**0 days**	**Fri 10/3/14**	**Fri 10/3/14**
13	20 Pages Written (Each)	0 days	Fri 10/3/14	Fri 10/3/14
14	**Design**	**35 days**	**Mon 10/6/14**	**Fri 11/21/14**
15	Bluetooth System	35 days	Mon 10/6/14	Fri 11/21/14
16	Storage System	35 days	Mon 10/6/14	Fri 11/21/14
17	MCU	35 days	Mon 10/6/14	Fri 11/21/14
18	Accelerometer	35 days	Mon 10/6/14	Fri 11/21/14
19	ODBII to Serial Interface	35 days	Mon 10/6/14	Fri 11/21/14
20	Transceivers	35 days	Mon 10/6/14	Fri 11/21/14
21	Power System	35 days	Mon 10/6/14	Fri 11/21/14
22	Android BLE	35 days	Mon 10/6/14	Fri 11/21/14
23	Android UI	35 days	Mon 10/6/14	Fri 11/21/14
24	**Design Complete**	**6 days**	**Fri 11/21/14**	**Mon 12/1/14**
25	30 Pages Written (Each)	0 days	Fri 11/21/14	Fri 11/21/14
26	Pre Holiday Document Review	3 days	Mon 11/24/14	Wed 11/26/14
27	Documention Complete	0 days	Wed 11/26/14	Wed 11/26/14
28	Turn In Senior Design I Documents	0 days	Mon 12/1/14	Mon 12/1/14
29	Order Parts/PCB's	0 days	Mon 12/1/14	Mon 12/1/14
30	Spring Semester Begins	0 days	Mon 1/12/15	Mon 1/12/15
31	**Senior Design II**	**76 days**	**Mon 1/12/15**	**Mon 4/27/15**
32	Review Hardware	2 days	Mon 1/12/15	Tue 1/13/15
33	Review Documentation	2 days	Mon 1/12/15	Tue 1/13/15
34	Firmware Rev. 1	14 days	Wed 1/14/15	Mon 2/2/15
35	**Hardware Rev. 1**	**23 days**	**Wed 1/14/15**	**Fri 2/13/15**
36	Build	7 days	Wed 1/14/15	Thu 1/22/15
37	Test	14 days	Fri 1/23/15	Wed 2/11/15
38	Respin Board if Needed	16 days	Fri 1/23/15	Fri 2/13/15
39	Firmware Rev. 2	14 days	Mon 1/12/15	Thu 1/29/15
40	**Hardware Rev. 2**	**23 days**	**Mon 2/16/15**	**Wed 3/18/15**
41	Build	7 days	Mon 2/16/15	Tue 2/24/15
42	Test	14 days	Wed 2/25/15	Mon 3/16/15
43	Respin Board if Needed	16 days	Wed 2/25/15	Wed 3/18/15
44	Firmware Final	14 days	Mon 1/12/15	Thu 1/29/15
45	**Hardware Final**	**20 days**	**Thu 3/19/15**	**Wed 4/15/15**
46	Build	5 days	Thu 3/19/15	Wed 3/25/15
47	Test	15 days	Thu 3/26/15	Wed 4/15/15

Page 1

FIGURE 9.1: Schedule Page 1

Schedule

ID	Task Name	Duration	Start	Finish
48	Phone Software Alpha	21 days	Wed 1/14/15	Wed 2/11/15
49	Alpha Integration Testing	3 days	Thu 2/12/15	Mon 2/16/15
50	Phone Software Beta	21 days	Tue 2/17/15	Tue 3/17/15
51	Beta Integration Testing	3 days	Wed 3/18/15	Fri 3/20/15
52	Phone Software Final	21 days	Thu 3/19/15	Thu 4/16/15
53	Final Integration Testing	3 days	Fri 4/17/15	Tue 4/21/15
54	Rev 1 Complete	0 days	Mon 2/16/15	Mon 2/16/15
55	Rev 2 Complete	0 days	Fri 3/20/15	Fri 3/20/15
56	Final Complete	0 days	Tue 4/21/15	Tue 4/21/15
57	Final Documention	7 days	Fri 4/17/15	Mon 4/27/15
58	Final Presentation	5 days	Fri 4/17/15	Thu 4/23/15
59	Print Documentation	0 days	Mon 4/27/15	Mon 4/27/15

FIGURE 9.2: Schedule Page 2

Appendix A

Android Framework

A.1 Definitions and Abbreviations

- Activity - An application component that provides a screen with which users can interact in order to do something. Often presented to the user as full-screen windows, they can also be used in other ways: as floating windows or embedded inside of another activity.
- API - Application programming interface. APIs often come in the form of a library that includes specifications for routines, data structures, object classes, and variables.[1]
- APK - Android application package. The package file format used to distribute and install application software onto Google's Android operating system.[2]
- App - Application software.
- Broadcast receiver - A component that responds to system-wide broadcast announcements.
- Bundle - A mapping from String values to various Parcelable types.
- Dialog - A small window that prompts the user to make a decision or enter additional information.
- Fragment - Represents a behavior or a portion of user interface in an Activity.
- IDE - Integrated development environment.

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

- Input controls - The interactive components in an app's user interface. Android provides a wide variety of controls such as buttons, text fields, seek bars, checkboxes, zoom buttons, toggle buttons, and many more.
- Intent - A messaging object you can use to request an action from another app component.
- Handler - Sends and processes Message and Runnable objects associated with a thread's MessageQueue.
- Layout - Defines the visual structure for a user interface, such as the UI for an activity or app widget.
- Manifest - The manifest file presents essential information about your app to the Android system, information the system must have before it can run any of the app's code.
- Notification - A message that can be displayed to the user outside of the application's normal UI. When the system issues a notification, it first appears as an icon in the notification area.
- OBD - On-board diagnostics.
- Package - Contains information about a Java package.
- Permission - A security permission that can be used to limit access to specific components or features of this or other applications.
- Preference - Preference UI building block to be displayed in the activity.
- Process - When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread).
- Provider - Supplies structured access to data managed by the application.
- RPM - Revolutions per minute.
- Service - An application component that can perform long-running operations in the background and does not provide a user interface.
- Style - A collection of properties that specify the look and format for a View or window.
- Theme - A style applied to an entire Activity or application, rather than an individual View.
- Thread - A concurrent unit of execution.
- Toast - Provides simple feedback about an operation in a small popup.
- UI - User interface. Everything that the user can see and interact with.
- UUID - Universally unique identifier. An identifier standard in software construction which allows for generating identifiers which do not overlap or conflict

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

with other identifiers which were previously created even without knowledge of the other identifiers.[3]
- View - Occupies a rectangular area on the screen and is responsible for drawing and event handling.
- VM - Virtual machine.
- XML - Extensible markup language.

A.2 APIs

android - Contains resource classes used by applications included in the platform and defines application permissions for system features.

android.app - Contains high-level classes encapsulating the overall Android application model. An Android application is defined using one or more of Android's four core application components. Two such application components are defined in this package: Activity and Service.

android.backup - Contains the backup and restore functionality available to applications. If a user wipes the data on their device or upgrades to a new Android-powered device, all applications that have enabled backup can restore the user's previous data when the application is reinstalled.

android.bluetooth - Provides classes that manage Bluetooth functionality, such as scanning for devices, connecting with devices, and managing data transfer between devices. The Bluetooth API supports both "Classic Bluetooth" and Bluetooth Low Energy.

android.content - Contains classes for accessing and publishing data on a device. It includes three main categories of APIs:

- Content sharing (android.content) - For sharing content between application components.
- Package management (android.content.pm) - For accessing information about an Android package (an .apk), including information about its activities, permissions, services, signatures, and providers.

Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

- Resource management (android.content.res) - For retrieving resource data associated with an application, such as strings, drawables, media, and device configuration details.

android.gesture - Provides classes to create, recognize, load and save gestures.

android.graphics - Provides low level graphics tools such as canvases, color filters, points, and rectangles that let you handle drawing to the screen directly.

android.hardware - Provides support for hardware features, such as the camera and other sensors.

android.media - Provides classes that manage various media interfaces in audio and video. The Media APIs are used to play and, in some cases, record media files. This includes audio (e.g., play MP3s or other music files, ringtones, game sound effects, or DTMF tones) and video (e.g., play a video streamed over the web or from local storage).

android.os - Provides basic operating system services, message passing, and interprocess communication on the device.

android.preference - Provides classes that manage application preferences and implement the preferences UI. Using these ensures that all the preferences within each application are maintained in the same manner and the user experience is consistent with that of the system and other applications.

android.test - A framework for writing Android test cases and suites.

android.util - Provides common utility methods such as date/time manipulation, base64 encoders and decoders, string and number conversion methods, and XML utilities.

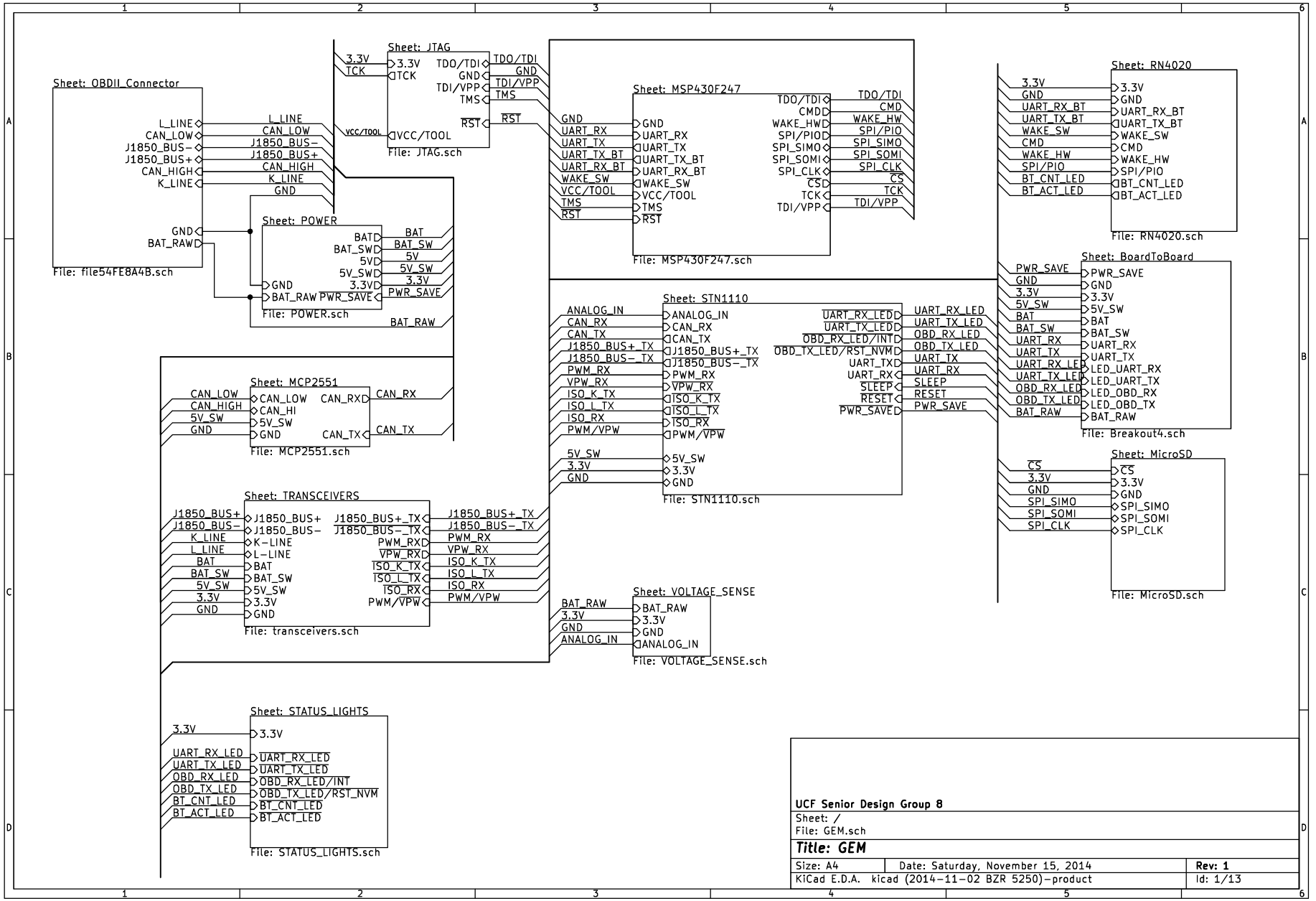
android.view - Provides classes that expose basic user interface classes that handle screen layout and interaction with the user.

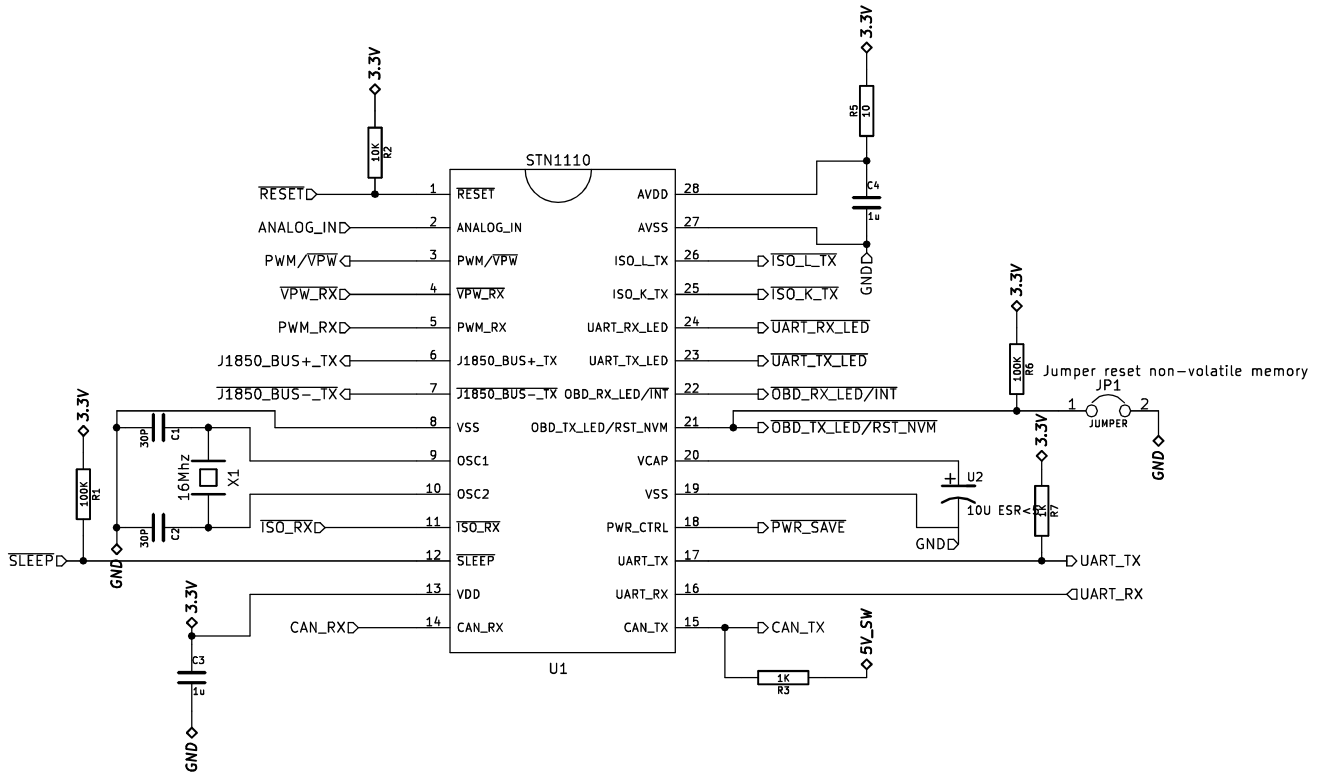
Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Appendix B

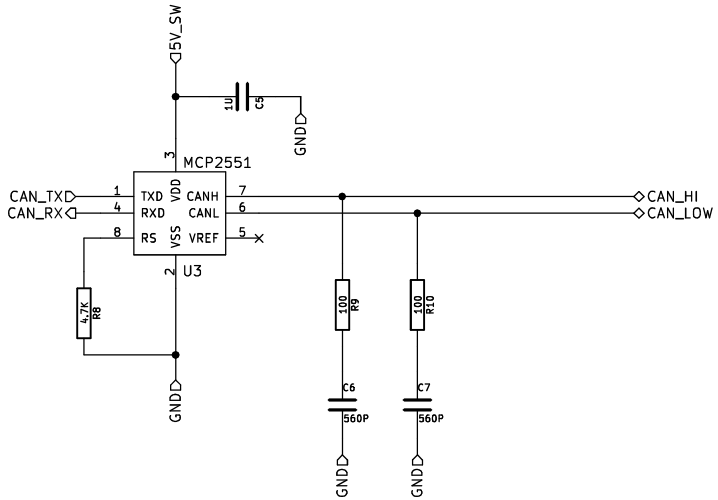
Final Schematics

The following drawings are the final schematics as well as the PCB layout for the GEM. Open source digital versions of these files may be found at the UCF Senior Design website, http://www.eecs.ucf.edu/seniordesign/fa2014sp2015/g08/

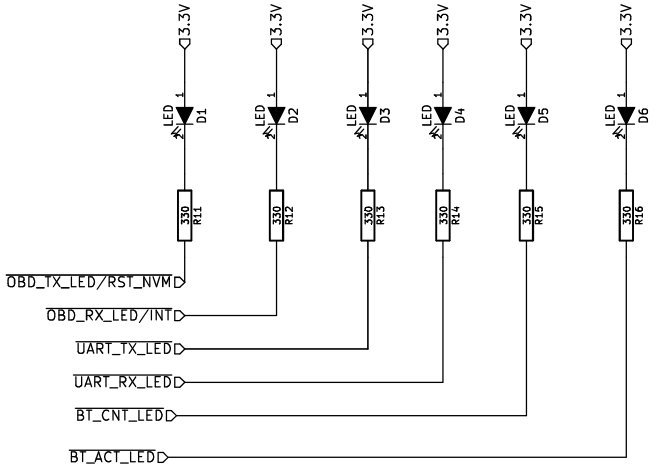

Sheet: OBDII_Connector
File: file54FE8A4B.sch
Sheet: JTAG
File: JTAG.sch
Sheet: POWER
File: POWER.sch
Sheet: MCP2551
File: MCP2551.sch
Sheet: TRANSCEIVERS
File: transceivers.sch
Sheet: STATUS_LIGHTS
File: STATUS_LIGHTS.sch
Sheet: MSP430F247
File: MSP430F247.sch
Sheet: STN1110
File: STN1110.sch
Sheet: VOLTAGE_SENSE
File: VOLTAGE_SENSE.sch
Sheet: RN4020
File: RN4020.sch
Sheet: BoardToBoard
File: Breakout4.sch
Sheet: MicroSD
File: MicroSD.sch
UCF Senior Design Group 8
Sheet: /
File: GEM.sch
Title: GEM
Size: A4
Date: Saturday, November 15, 2014
Rev: 1
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product
Id: 1/13



Sheet: 1/STN1110/		
File: STN1110.sch		
Title: GEM II INTERPRETER INTERFACE		
Size: A4	Date: Saturday, November 15, 2014	**Rev:** 0.1
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 2/13



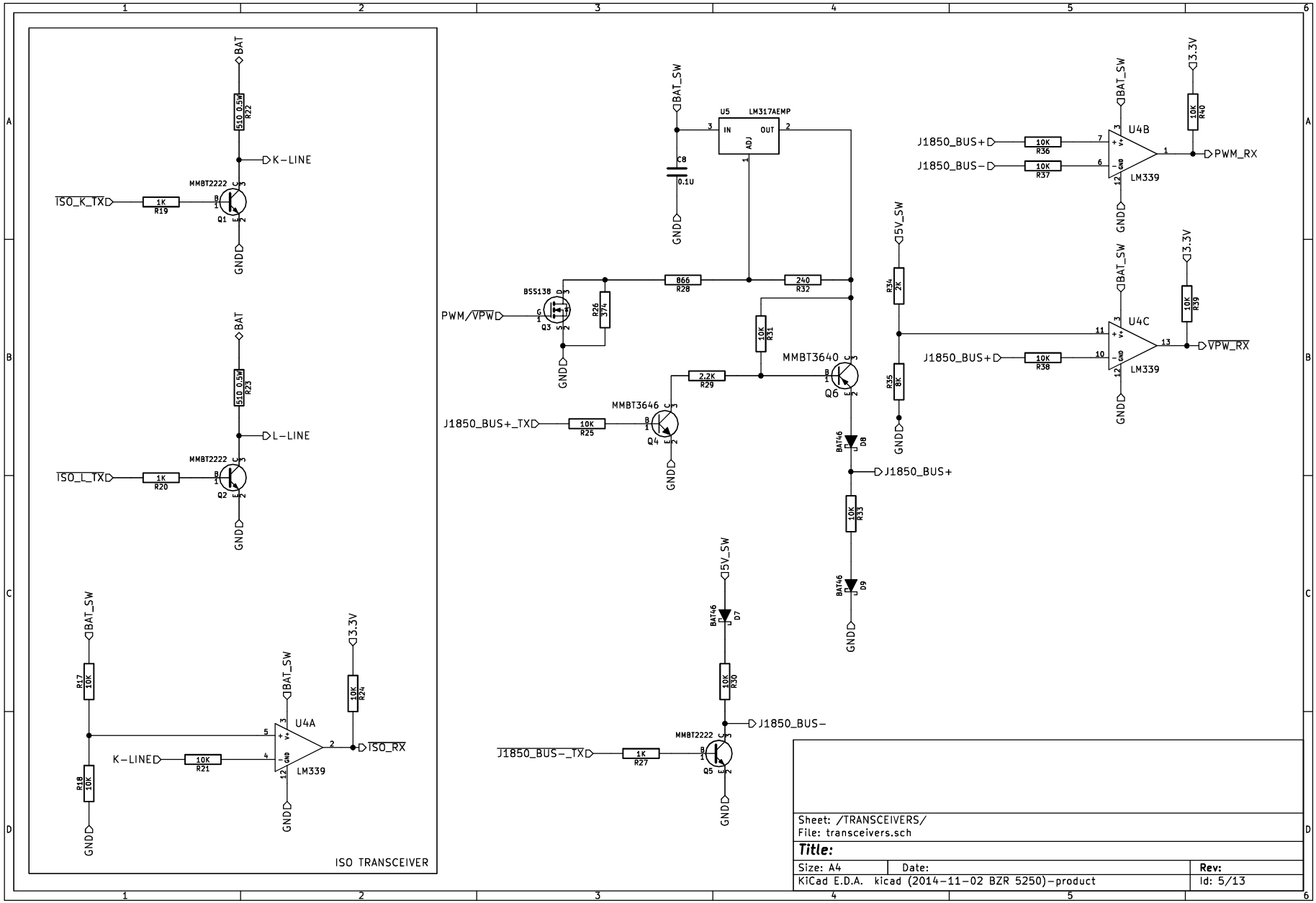
Sheet: /MCP2551/		
File: MCP2551.sch		
Title: GEM		
Size: A4	Date: Saturday, November 15, 2014	**Rev: 1**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 3/13



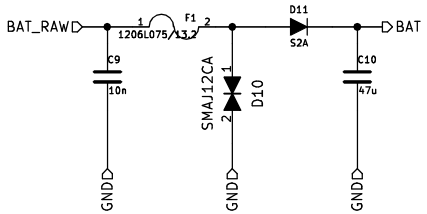
Sheet: /STATUS_LIGHTS/
File: STATUS_LIGHTS.sch

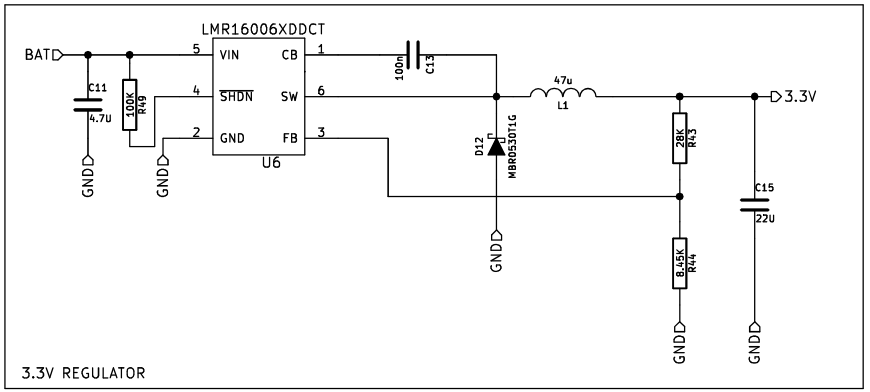
Title: GEM

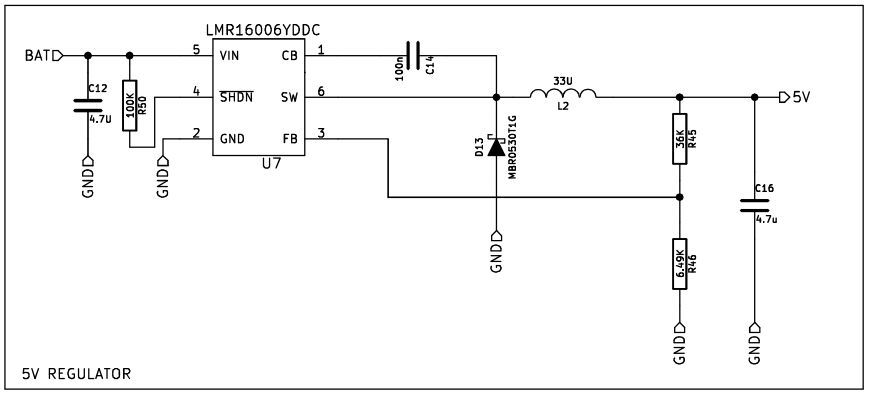
Size: A4	Date: Saturday, November 15, 2014	**Rev: 1**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 4/13

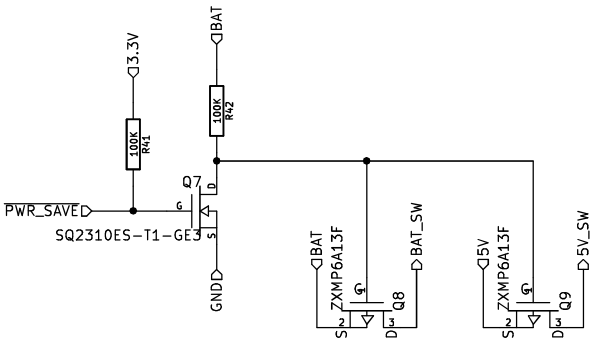


Sheet: /TRANSCEIVERS/		
File: transceivers.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 5/13

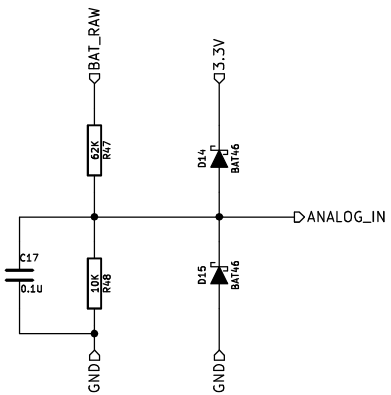




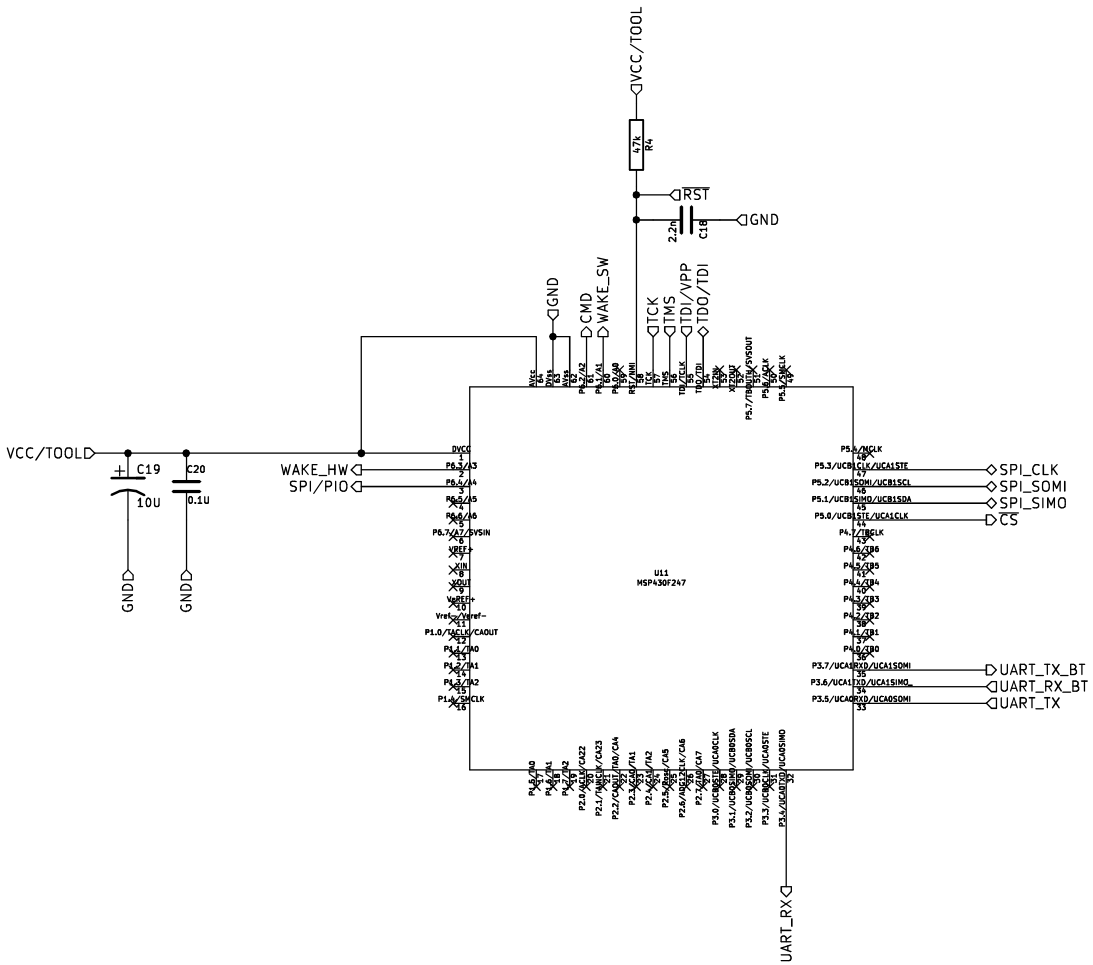




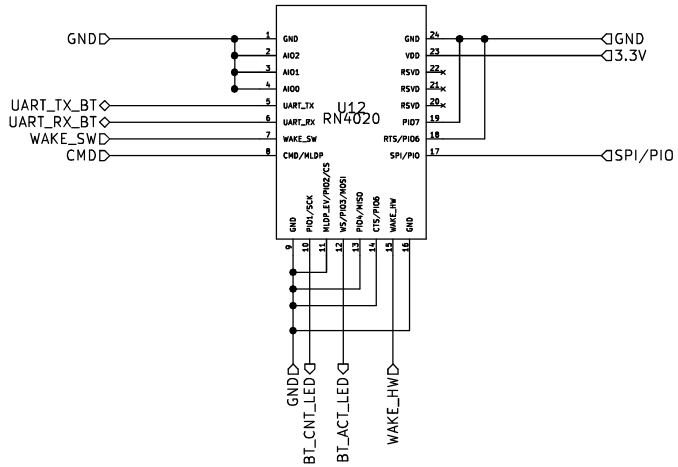
Sheet: /POWER/ File: POWER.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 6/13



Sheet: /VOLTAGE_SENSE/ File: VOLTAGE_SENSE.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 7/13

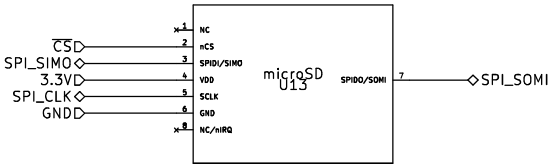


Sheet: /MSP430F247/		
File: MSP430F247.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 8/13

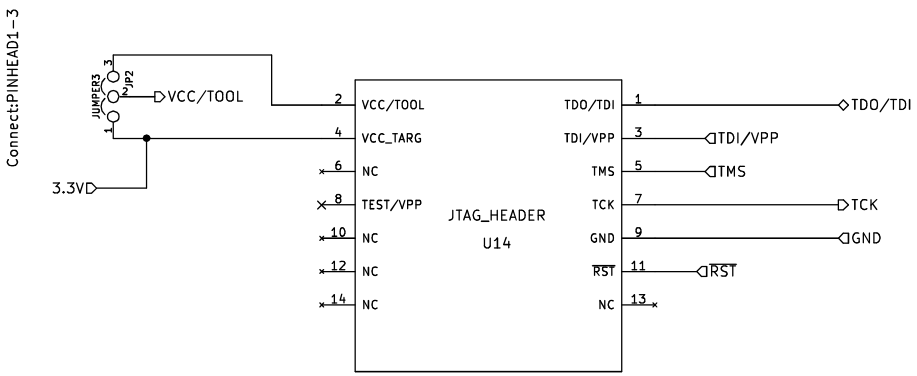


Sheet: /RN4020/ File: RN4020.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 9/13

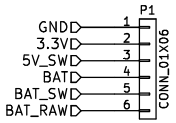
Might Need Pull Up Resistors Not Sure

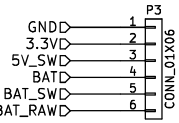


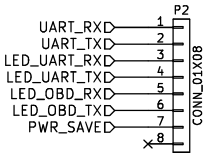
Sheet: /MicroSD/		
File: MicroSD.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 10/13

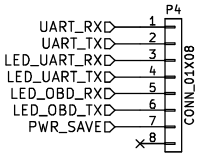


Sheet: /JTAG/ File: JTAG.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 11/13

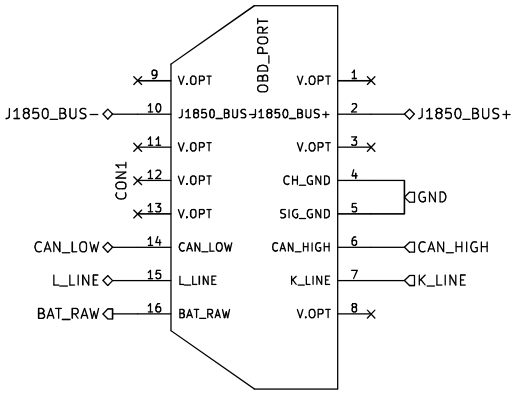


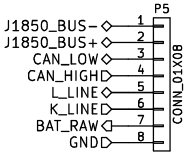


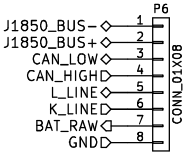




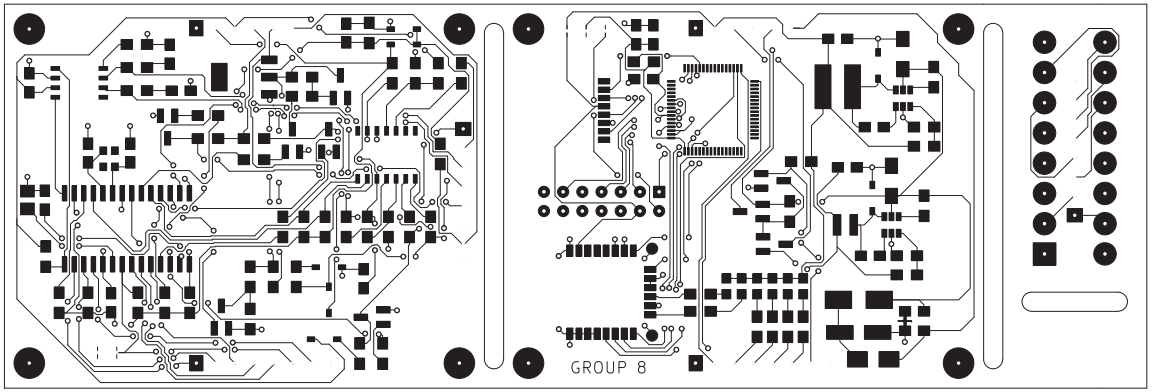
Sheet: /BoardToBoard/ File: Breakout4.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 12/13



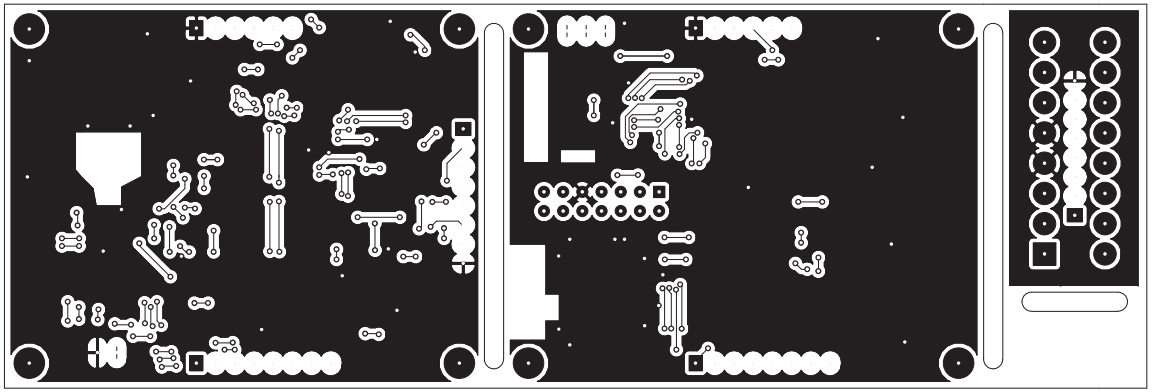




Sheet: /OBDII_Connector/ File: file54FE8A4B.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 13/13



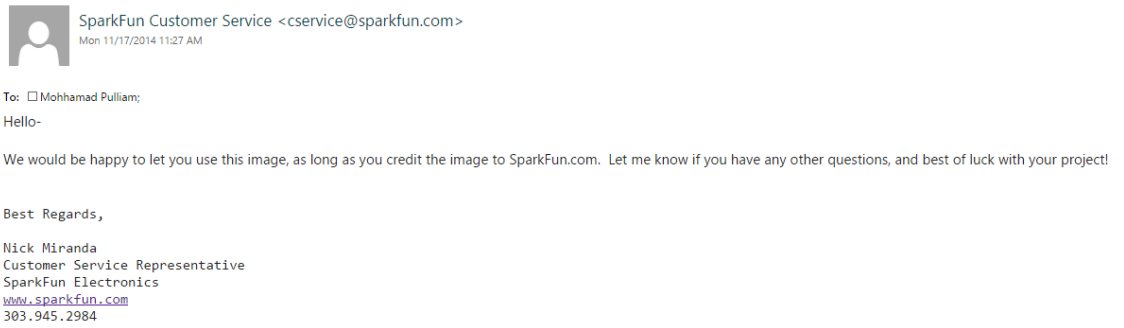
Sheet: File: GEM.kicad_pcb		
Title:		
Size: A4	Date:	Rev:
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 1/1



Sheet: File: GEM.kicad_pcb		
Title:		
Size: A4	Date:	Rev:
KiCad E.D.A. kicad (2014-11-02 BZR 5250)-product		Id: 1/1

Appendix C

Copyright Permissions



SparkFun Customer Service <cservice@sparkfun.com>
Mon 11/17/2014 11:27 AM

To: ☐ Mohhamad Pulliam;

Hello-

We would be happy to let you use this image, as long as you credit the image to SparkFun.com. Let me know if you have any other questions, and best of luck with your project!

Best Regards,

Nick Miranda
Customer Service Representative
SparkFun Electronics
www.sparkfun.com
303.945.2984

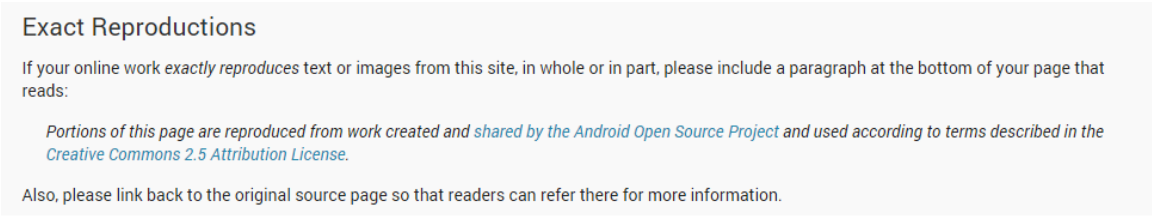
FIGURE C.1: Sparkfun Electronics

I, the copyright holder of this work, hereby publish it under the following license:

This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication.

The person who associated a work with this deed has dedicated the work to the public domain by waiving all of his or her rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission.

FIGURE C.2: OBD-II Pinout
`http://commons.wikimedia.org/wiki/File:OBD_connector_shape.svg`



Exact Reproductions

If your online work *exactly reproduces* text or images from this site, in whole or in part, please include a paragraph at the bottom of your page that reads:

> *Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.*

Also, please link back to the original source page so that readers can refer there for more information.

FIGURE C.3: Android Attributions

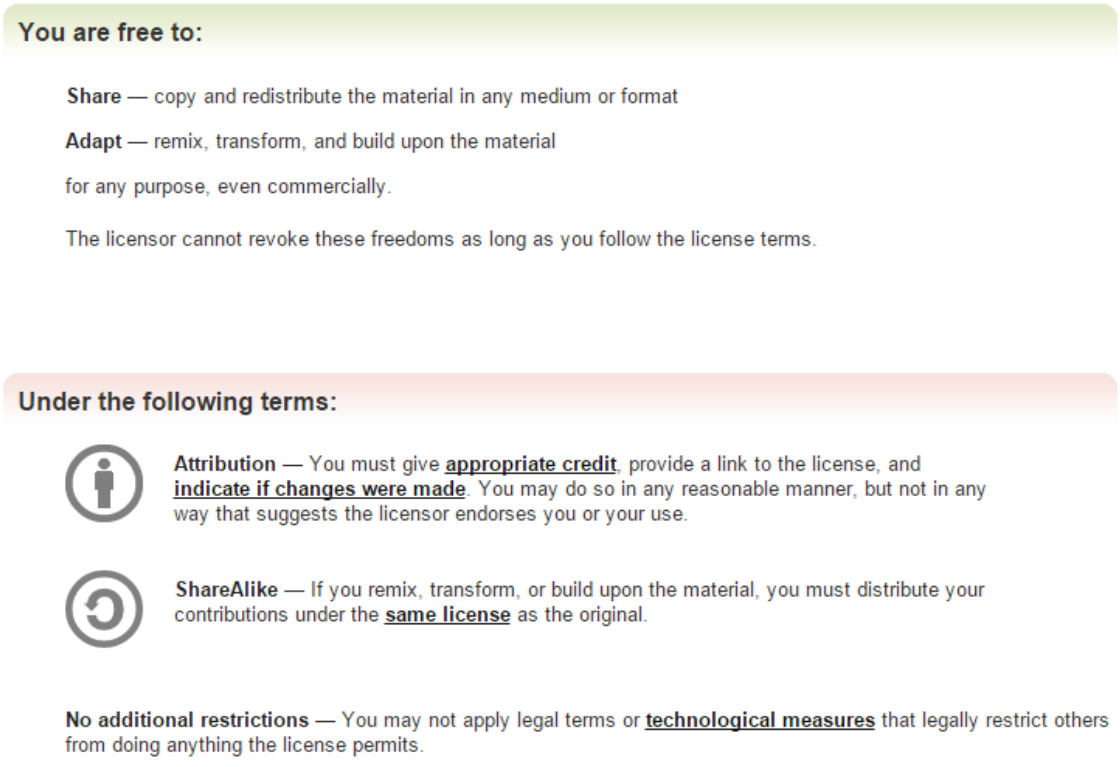


FIGURE C.4: Creative Commons Attribution License

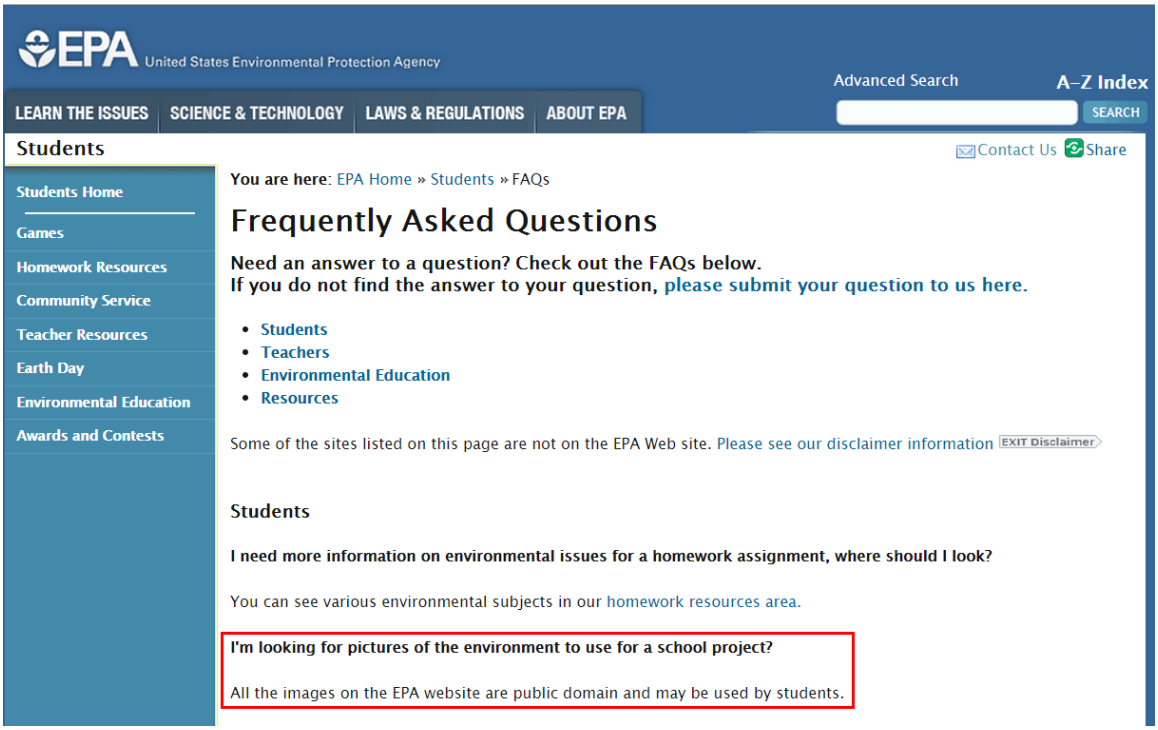


FIGURE C.5: UDDS and HWFET driving schedules
`http://www.epa.gov/oms/emisslab/methods/uddsdds.gif` and
`http://www.epa.gov/oms/emisslab/methods/hwfetdds.gif`

Bibliography

[1] Wikipedia contributors. "Application programming interface," Wikipedia, The Free Encyclopedia. Internet: http://en.wikipedia.org/w/index.php?title=Application_programming_interface&oldid=633502752, Nov. 12, 2014, [Nov. 29, 2014].

[2] Wikipedia contributors. "Android application package," Wikipedia, The Free Encyclopedia. Internet: http://en.wikipedia.org/w/index.php?title=Android_application_package&oldid=629618095, Oct. 14, 2014, [Nov. 29, 2014].

[3] Wikipedia contributors. "Universally unique identifier," Wikipedia, The Free Encyclopedia. Internet: http://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=634823457, Nov. 21, 2014, [Nov. 29, 2014].

[4] Paulo Pires. "android-obd-reader (Version 2.0-BETA2)," GitHub. Computer Program: https://github.com/pires/android-obd-reader, Nov. 14, 2014, [Nov. 20, 2014].