openOBD

Automotive Telematics Unit

Design - v3.3

Authors:

- Nicholas Mulvenna
- Kaibo Ma
- Ehsan Ahmadi
- Isaiah Thiessen

Table of Contents

History of Revisions

Version	Authors	Date	Changes
1.0	Isaiah Thiessen Kaibo Ma Nicholas Mulvenna Ehsan Ahmadi	2016-11-28	Initial Draft
1.1	Isaiah Thiessen	2017-02-13	- Updated Hardware Architecture diagram - Changed GPS section to reflect new chip being used - Updated Appendix V with new chip information
1.2	Ehsan Ahmadi	2017-02-13	- New Can transceiver chip - Remove level shifter - New power supply - Added appendix VIII
1.3	Isaiah Thiessen	2017-02-13	- Added component schematics in Appendix III (changed heading to Circuit Diagrams) - Added Appendix VII - PCB design schematic - Added Section 3.1.5 - PCB - Added Appendix IX for the CAN node pinout
1.4	Nicholas Mulvenna	2017-02-13	- Updates software section with details on `Manager` and `Configuration` classes

			- Edited sections on dependency inversion and single responsibility principles - Removed abstraction layers section
2.0	Isaiah Thiessen Ehsan Ahmadi Nicholas Mulvenna Kaibo Ma	2017-02-13	Final edits before v2 document release with some added figures to appendix
2.1	Nicholas Mulvenna	2017-02-26	Corrects overlapping lines in high-level UML class diagram
3.0	Nicholas Mulvenna	2017-03-18	- Style and wording in various sections - Describes the `DeviceCollection` class and removes references to the `Shell` class - Simplifies the high-level UML class diagram and adds new diagrams for the `DeviceCollection` and `Manager` classes
3.1	Isaiah Thiessen	2017-04-02	- Justified amperage in power supply section - Fixed mistaken section heading in t.o.c. - New Hardware architecture Diagram

			- Edited section numbers - Added hardware enclosure section and appendix
3.2	Ehsan Ahmadi	2017-04-03	- Updated power supply section to reflect new design - Added attiny25 section and schematic - Updated current/complete hardware setup - Updated list of tables & fig
3.3	Nicholas Mulvenna	2017-04-17	- Changes to wording

List of Figures and Tables

1 - Introduction

This document is a comprehensive map of the technical design choices used in developing the Automotive Telematics Unit.

1.1 How to Use this Document

Aside from section 2, this document is of a highly technical nature. For both the technical or non-technical reader, section *2.0 - High-Level Architecture* for hardware and software is the recommended starting point.

Following the high level discussion is a low-level technical breakdown of each component in section 3: *Low-Level Design*. Each subsection provides the reasoning behind the design choices, and guides the reader to a corresponding set of relevant appendices where various design artifacts are presented.
Subsection 3.1 covers hardware components, while 3.2 discusses software design.

2 - High-Level Architecture

2.1 - Hardware Architecture

Looking at figure 2.0, one can understand the hardware components involved as well as how each interacts within the design. Photos of the setup, in prototype stage are shown in Appendix IV, while Appendix XI shows the complete setup.

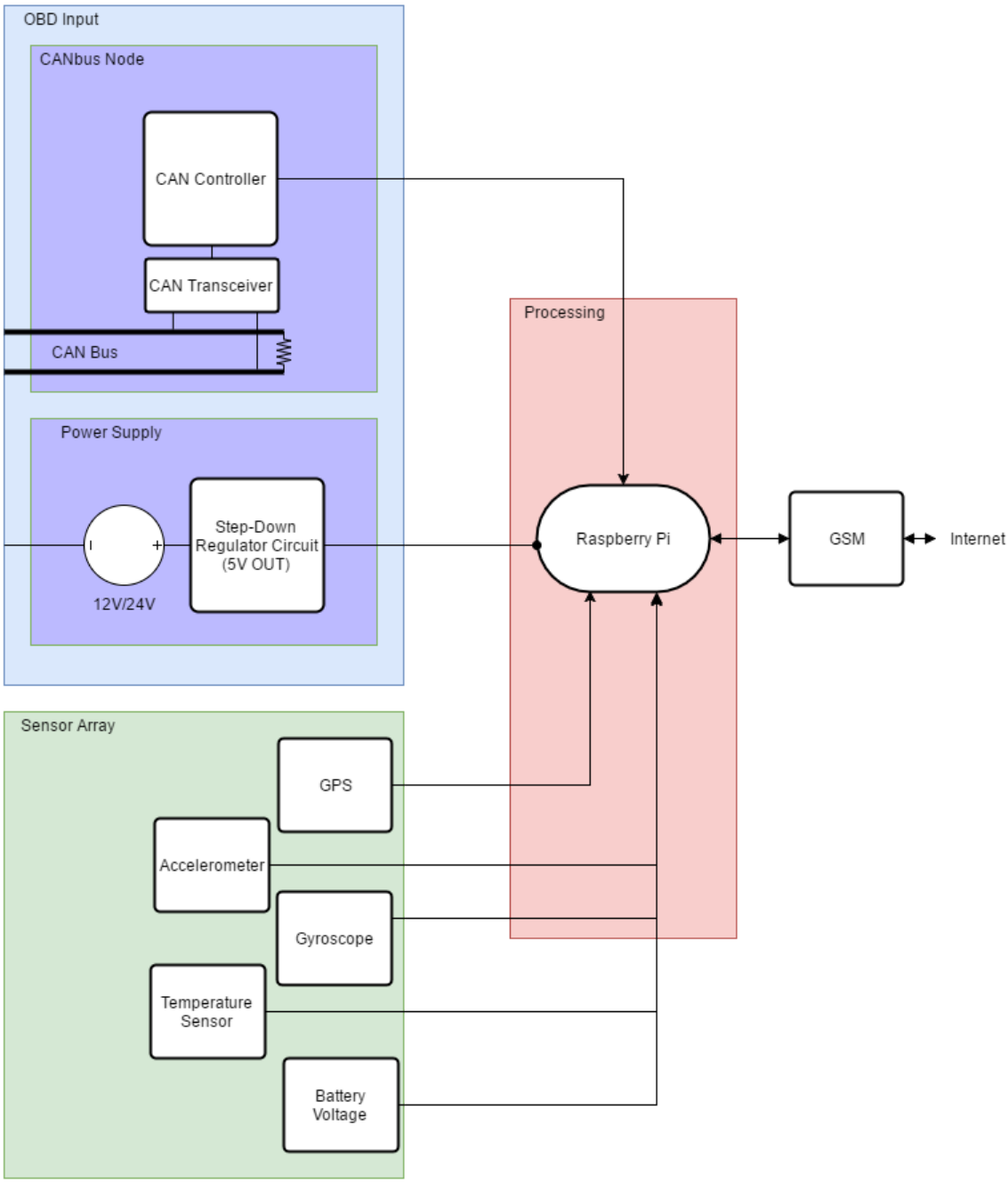


Figure 2.0: High-Level Hardware Architecture

On the following page is a brief description of the function of each component.

Processing Component:

A Raspberry Pi Zero interfaces with sensors and the OBD input block to collect, parse, and process the data.

OBD Input - CAN bus Component:

A CAN Bus device, following the CAN standard, includes a CAN controller and transceiver in order to read and write data to the CAN bus and perform arbitration and identification. It communicates with the processing block over a serial peripheral interface. The CAN bus is chosen as the focus of this project since every vehicle produced in the US since 2008 implements it.

OBD Input - Power Component:

Power for the processing block is drawn from the OBD port at 12 or 24 volts, depending on the vehicle. A power converter is used to regulate the power down to the levels required by the system

Sensors Component:

An array of onboard sensors for motion, temperature, and location readings.

A low level breakdown of each component is provided in section 3.1.

2.2 - Software Architecture

Our architecture, described in the UML class diagram in figure 2.1, is based on abstract classes. Our main executable class `App` instantiates our concrete device classes (based on a configuration file) and aggregates them in a `DeviceCollection` instance. Also, it instantiates the `Manager` class which automatically reads, interprets, and displays data. `App` provides `Manager` with the `DeviceCollection` (an implementation for an I`DeviceCollection`) to allow it to read data from our sensors and the OBD port. Since our major components are all constructed by `App`, `Manager` and `DeviceCollection` depend on abstract classes instead of being coupled to specific implementations.

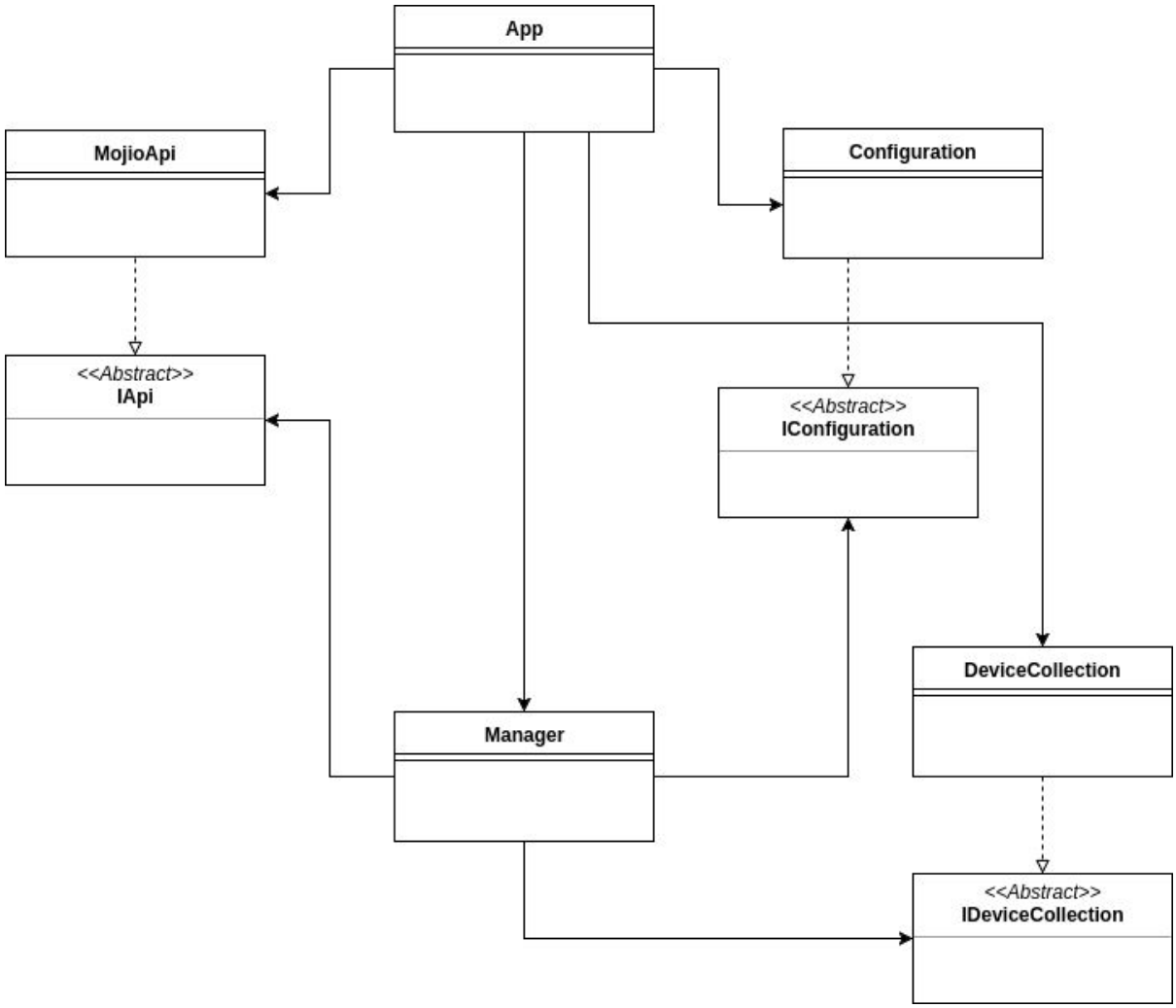


Figure 2.1: UML class diagram focused on the App class

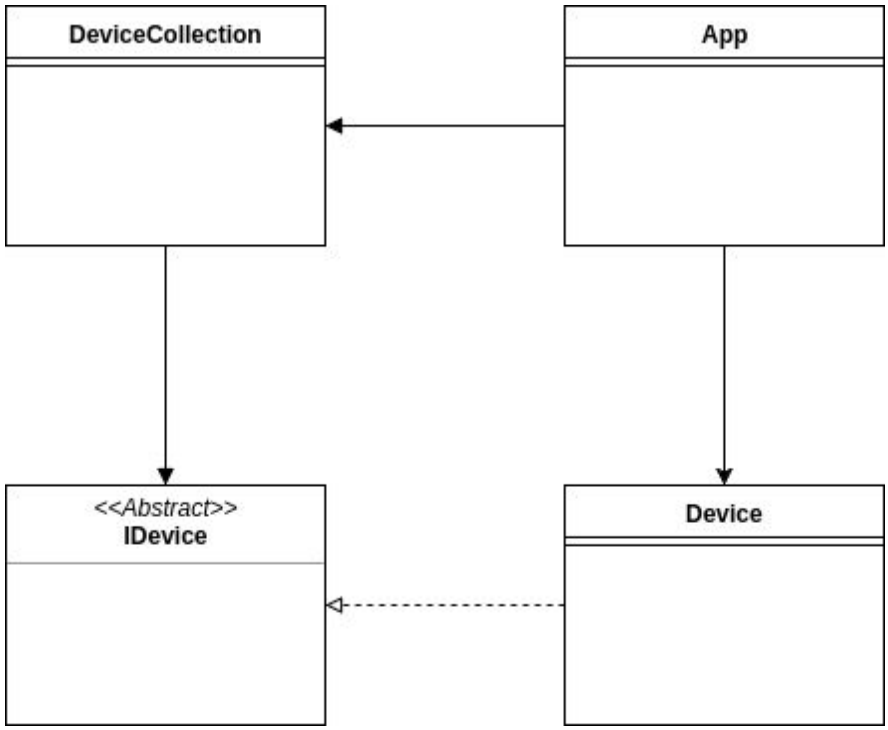


Figure 2.2: UML class diagram focused on the DeviceCollection class

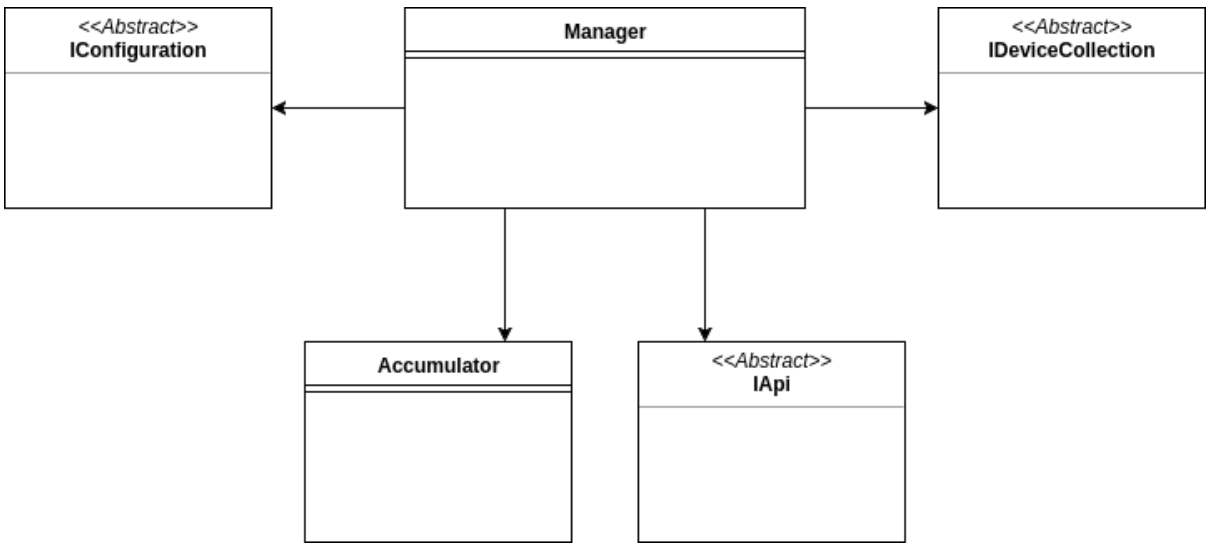


Figure 2.3: UML class diagram focused on the Manager class

3 - Low-Level Design

The next sections go into detail around the design choices of each component. In order to justify these choices, references are made to functional specifications (coded FS##), non-functional specifications (coded NS##), and constraints (coded C##) which are defined in the accompanying requirements document.

3.1 - Hardware Components

3.1.1 - Raspberry Pi Zero

The Raspberry Pi Zero is a small, inexpensive single board computer used as the main processing unit of the system. It runs a 1GHz ARM processor, supported by 512MB of ram, providing plenty of processing power. An array of GPIO pins with SPI, I2C and UART capabilities makes interfacing with sensors and hardware components very easy and satisfies the FS02 specification. The Pi Zero is widely adopted by the open source community globally and extensively documented. This means open source libraries are available for almost any application and support is readily accessible resulting in short development ramp-up time. The Pi Zero runs Raspbian Jessie (A Debian Linux based operating system optimized for the Pi) which provides a terminal interface to the user over USB, satisfying the FS03 specification.

3.1.2 - CAN Node

The CAN node implements the standard CAN communication protocol and facilitates interfacing with the CAN bus and satisfying FS01. It gives the system access to the vehicle's on board diagnostics (OBD-II) network which can provide information about and control over different components of the vehicle. The CAN node is comprised of a CAN controller, CAN transceiver and two level converters A detailed description of the OBD-II and CAN are provided in appendix I and II respectively. See Appendix III for a circuit diagram and pinout information.

3.1.2A MCP2515 CAN Controller

As detailed in Appendix I the CAN communication protocol implements error correction, bit timing, and synchronization and arbitration of different nodes on the

CAN bus. To achieve this, complex message frames are used to transmit data on the can bus. The CAN controller acts as a configurable node on the can bus and deals with the low level implementation of the CAN protocol. It extracts the data contained in a CAN message frame received from the bus and provides it to the processor over an SPI interface. It can also convert data provided by the processor and converts it to a CAN message frame that is ready to be transmitted on the bus. This chip was chosen as it provides the best combination of functionality and low cost, and is commonly used in the design of CAN devices. [7]

3.1.2B TCAN332 CAN Tranceiver

CAN message frames provided by the can controller as a serial signal of ones and zeros is converted to a differential signal and transmitted on the CAN bus (appendix I) by the CAN transceiver. At the same time differential signals received on the can bus are converted to a serial signal and provided to the CAN controller. This chip was chosen as it provides the best combination of functionality and low cost. Since the chip uses 3.3 volts, it uses less power the need for level shifting between it and the controller is eliminated resulting in a smaller pcb footprint.[8]

3.1.3 - Power Supply

An LMR16030 Buck controller, combined with external components (7 capacitors, Inductor, a Diode and a few resistors) is used as the power supply. The power supply is designed for a wide input voltage range of 6V to 40V. It steps down voltage from the OBD port (typically 12V or 24V) to a constant 5V to power the Pi Zero which in turn powers all the other components with the exception of the sim808 (powered through a 5V to 4V linear regulator), satisfying the FS07 requirement. The power supply is designed to provide up to 3A (15W). 1A is required by the Pi to function reliably [16] leaving enough power for the sim808 chip which peaks at 2A. The LMR16030 is a flexible converter with an internal high side mosfet running at 500KHz, providing the ability to use very small external components, saving both space and cost. A switching supply is used instead of a linear regulator in order to reduce power losses and prevent excessive heating, in line with C04 and C06.

3.1.4 - Sensor Inputs

3.1.4A - Sim808 GSM + GPS Cellular Module

The Sim808 module was chosen in order to reduce footprint consumption for the PCB design, while retaining ease-of-setup and short development overhead. This component provides all necessary GPS location functionality over UART to the processing unit in a “plug-and-play” fashion at relatively low cost, in addition to providing 2G Cellular network connectivity. See Appendix V for pinout information and links to official documentation.

3.1.4B - MPU6050 MEMs Motion Sensor Breakout Board

The MPU6050 was chosen with the same justifications as the Adafruit GPS board discussed above. It provides accelerometer, gyroscope and temperature data to the processing unit over I2C. See Appendix VI for pinout information and links to official documentation.

3.1.4C - ATtiny25 Microcontroller

One of the limitations of the Pi is that it does not include an ADC (analog to digital converter). To overcome this limitation, a cheap ATtiny25 microcontroller was added to the design. Some desired functions of the ATtiny include ADCs, SPI communication and general purpose GPIOs. The ATtiny senses the voltage of the vehicle’s battery and makes it available to the Pi over SPI. It is also able to safely shut down and turn on the Pi. The ATtiny runs independent of the Pi, however, it can be programmed on-the-go using the Raspberry Pi, making it a very flexible and customizable power management solution.

3.1.5 - Printed Circuit Board (PCB) Design

The design schematic for the PCB layout is included in Appendix VII. It was designed to approximately match the form factor of the Raspberry Pi Zero in order to minimize the overall size of the device. Further technical information is included with the schematic. All values for items such as minimum hole size, trace size and spacing, were chosen in order to meet the capabilities of the PCB manufacturer

while achieving the desired form factor. Since the PCB combines a high frequency switching power supply and multiple high speed communication devices in a small area. Strict guidelines were followed when laying out the board in order to ensure signal integrity and reliable communication. These guidelines were It is highly recommended to follow these guidelines detailed in appendix X when reproducing the PCB with minor modifications. The pcb has an exposed ground pad as well as several test points for the output and feedback pin of the power supply, 4V linear regulator, SPI data lines, as well as multiple power rails of the Sim808 chip. As a result, an oscilloscope can be easily attached in order to debug the circuit if need be.

3.1.6 - Hardware Enclosure

Given that this is an open-source, modular project, an example hardware enclosure is shown in Appendix XI. This part's design would be modified depending on changes made to the hardware.

The example design was chosen for simplicity and ease of fabrication. It holds all of the components together and provides a grip point for ease of setup or removal.

3.2 Software Components

3.2.1 - Python 3

Python 3 is a robust and popular language. There is significant community-based documentation and a wide variety of available modules. Python also includes convenient `Cmd` and `ConfigParser` classes that we use for debugging and in our `Configuration` class implementation.

In addition, our team has picked Python 3 over Python 2. In the following Table, key main differences between the two versions and why we believe Python 3 is more advantageous:

Python 2.7.12	Python 3.5.2
- Legacy - Will have no more major feature releases	- Long term supported version - Present and future of the language - Ongoing expansions and support for the language

Table 3.0: Python 2 vs Python 3

3.2.2 - Dependency inversion principle

In object oriented programming, there are various types of branches of design principles. One of which is dependency inversion principle. This refers to the decoupling between different software modules. In other words, two different libraries are not dependent on each other. When following this principle, higher level classes/modules will be independent from lower level modules implementation details. For our project, that means that our `Manager` class (through its `DeviceCollection`) depends only on abstract classes for devices. This allows different device class implementations to be substituted without requiring any changes to classes such as the `DeviceCollection`, `MojioApi`, and `Manager`.

In future prototypes and products, it will be easier to change hardware modules (such as our GPS or sensors) because only the module/class responsible for that hardware's drivers need to be modified and not the entire application. Software modules can be extracted and placed in a whole new application without worrying about dependencies of a particular module being extracted. This allows a "plug and play" convention for mixing and matching modules. For example, we could extract the modules for GPS and import them in a completely unrelated project that uses the same GPS hardware.

Our architecture is based on abstract classes to fulfill requirement NS02. Python supports multiple inheritance so it does not have Java-style interfaces. To ensure that they cannot be instantiated, our abstract classes extend the `abc.ABC` class.[14]

This is equivalent to an interface-oriented architecture because our abstract classes do not contain implementations.

3.2.3 - Single responsibility principle

This is a principle in object oriented design which states that modules, classes, and functions should only have one responsibility or purpose within the software. The functionality should be completely encapsulated within the class or function. For example, the device classes are responsible only for taking readings from the hardware modules but not for working with those readings. Similarly, the `Manager` class decides when to take readings and displays them to the user but does not directly interact with the hardware. Indirectly, this principle enforces the dependency inversion principle as well.

4 - Conclusion

The overall design and theme of the product is based on community learning. Both hardware and software use open source solutions and libraries. As a result our project will also be open source with a public repository for anyone else to access.

Our hardware and software are tailored for customizability. The hardware can be interchanged with relative ease because the software adopts the single responsibility and dependency inversion principles. This allows modularity and independence of different modules and devices like OBD and GPS. Hardware can be swapped out for faster and even better devices with minimal changes to the software.

References

[1] Guttag, John V. (2013-01-18). *Introduction to Computation and Programming Using Python* (Spring 2013 ed.). Cambridge, Massachusetts: The MIT Press. ISBN 9780262519632.

[2] "Abstract Methods and Classes". *The Java™ Tutorials*. Oracle. Retrieved 27 November 2016.

[3]"Dissection of the Single Responsibility Principle - dylanwilson.net", Dylanwilson.net, 2016. [Online]. Available: http://www.dylanwilson.net/dissection-of-the-single-responsibility-principle. [Accessed: 27- Nov- 2016].

[4] "Dependency Inversion Principle | Object Oriented Design", Oodesign.com, 2016. [Online]. Available: http://www.oodesign.com/dependency-inversion-principle.html. [Accessed: 27- Nov- 2016].

[5] “Should I use Python 2 or Python 3 for my development activity?”, wiki.python.org, 2016. [Online]. Available: https://wiki.python.org/moin/Python2orPython3 [Accessed: 27- Nov- 2016].

[6] “MPU-6000 and MPU-6050 Product Specification Revision 3.4”, Sparkfun, 2016. [Online]. Available: http://43zrtwysvxb2gf29r5o0athu.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf [Accessed: 27-Nov-2016]

[7] “Stand-Alone CAN Controller with SPI Interface”, microchip.com, 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21801G.pdf [Accessed: 27-Nov-2016]

[8] “High-Speed CAN Transceiver”, microchip.com, 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/20005167C.pdf [Accessed: 27-Nov-2016]

[9] “Adafruit Ultimate GPS Breakout - 66 Channel W/10 HZ Updates”, adafruit.com, 2016. [Online]. Available: https://www.adafruit.com/product/746 [Accessed: 27-Nov-2016]

[10] “Triple Axis Accelerometer Gyro Breakout MPU6050 v12”, sparkfun.com, 2016. [Online]. Available:

https://cdn.sparkfun.com/datasheets/Sensors/IMU/Triple_Axis_Accelerometer-Gyro_Breakout_-_MPU-6050_v12.pdf [Accessed: 27-Nov-2016]

[11] “GPS_FGPMMOPA6H_v0.3”, adafruit.com, 2016. [Online]. Available: https://cdn-learn.adafruit.com/assets/assets/000/022/494/large1024/gpssch.png?1421536475 [Accessed: 27-Nov-2016]

[12] “Introduction to the Controller Area Network (CAN)”, ti.com, 2008. [Online]. Available: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf [Accessed: 27-Nov-2016]

[13] “On-board diagnostics”, Wikipedia, The Free Encyclopedia, 2016. [Online]. Available: https://en.wikipedia.org/wiki/On-board_diagnostics#SAE_standards_documents_on_OBD-II [Accessed: 27-Nov-2016]

[14] "Abstract Base Classes in Python – dbader.org", Dbader.org, 2016. [Online]. Available: https://dbader.org/blog/abstract-base-classes-in-python. [Accessed: 28-Nov-2016].

[15] “Switching Power Supply PCB Layout Considerations – Towards a Better Switcher”, optimumdesign.com, 2017/ [online]. Available: http://blog.optimumdesign.com/switching-power-supply-pcb-layout-considerations-towards-a-better-switcher. [Accessed: 13-Feb-2016]

[16] “Power Supply”, raspberrypi.org, 2017 / [online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md [Accessed: 02-Apr-2016]

[17] “Program an AVR...Raspberry Pi GPIO”, Adafruit.com, 2017 / [Online]. Available: https://learn.adafruit.com/program-an-avr-or-arduino-using-raspberry-pi-gpio-pins/overview [Accessed: 03-Apr-2016]

Appendix I - On Board Diagnostics - OBD-II

Onboard diagnostics (OBD) is a computer based system built into all 1996 and later light-duty vehicles and trucks to monitor and in newer vehicles control their different sub components. The OBD-II standard defines the OBD II diagnostic connector and 5 standard communication protocols for interfacing with it, one of which is the CAN protocol and is standard on all US produced vehicles since 2008. The CAN is described in detail in appendix II and is the target protocol for this project. For more information on the OBD, see [13].

Appendix II - Controller Area Network - CAN

A Controller Area Network is defined by its two components described in this appendix, the CAN bus and the CAN communication protocol (ISO-11898). This appendix is based on ti's introduction to CAN document [12] and introduces a few basics of the CAN in order to demonstrate its complexity. This complexity is handled by the MCP2515 CAN controller, however, a hardware designer needs to be aware of these complexities in order to use the controller.

The CAN bus is a multi-master, message broadcast system that specifies a maximum signaling rate of 1 megabit per second (Mbps). As shown in figure A1, multiple masters (nodes) are connected to a differential line that is terminated by two resistors. Signals on the CAN bus are differential and highly immune to noise. A logic high is dominant and represents a 0, while a logic-low is recessive and represents a 1. This means that a 0 written on the bus by one node, overwrites a 1 that is simultaneously written by another node. This property is key to arbitration process described in the communication protocol. For effective communication, every node on the bus must agree to uniform nominal bitrate.

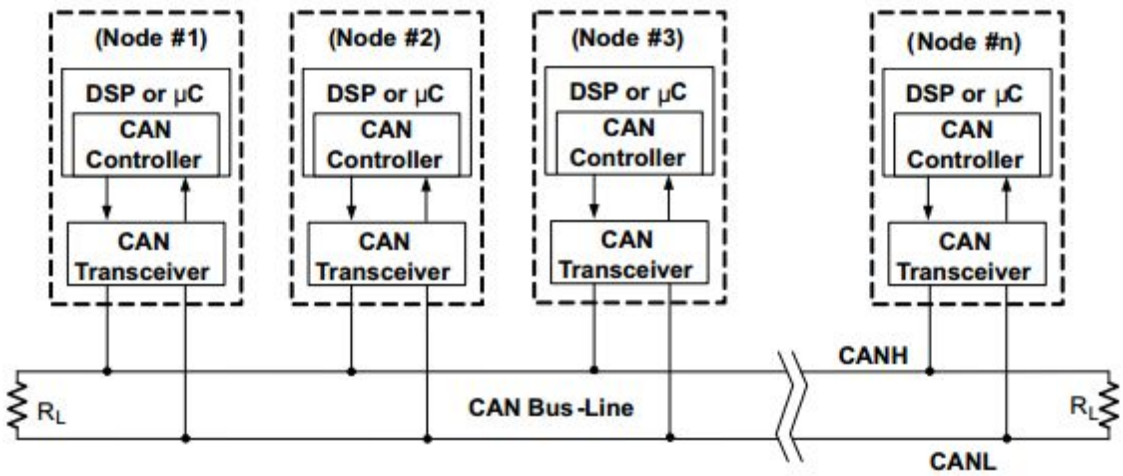


Figure A1.0: CAN bus diagram

The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP). Since every node on the bus is a master and has read/write capability, each node on a bus must wait for a prescribed period of inactivity before attempting to send a message. Usage of the bus granted to the node that is transmitting a message with the highest priority. Messages are identified using a 11-bit (standard CAN) or 29-bit (extended CAN) identifier which allows for up to 2048 or 537 million unique prioritization levels respectively. Since the transmitted messages are broadcasted to all the nodes, the identifier can also be used to target the messages to specific nodes. The receiving nodes perform error checking on the message and requests a resend if failed. Due to different delays times and different clock speeds across the CAN, The actual bitrate of the messages is dynamically adjusted slightly above or below the nominal bit rate to keep the nodes synchronized and resolve errors

Appendix III - Circuit Diagrams

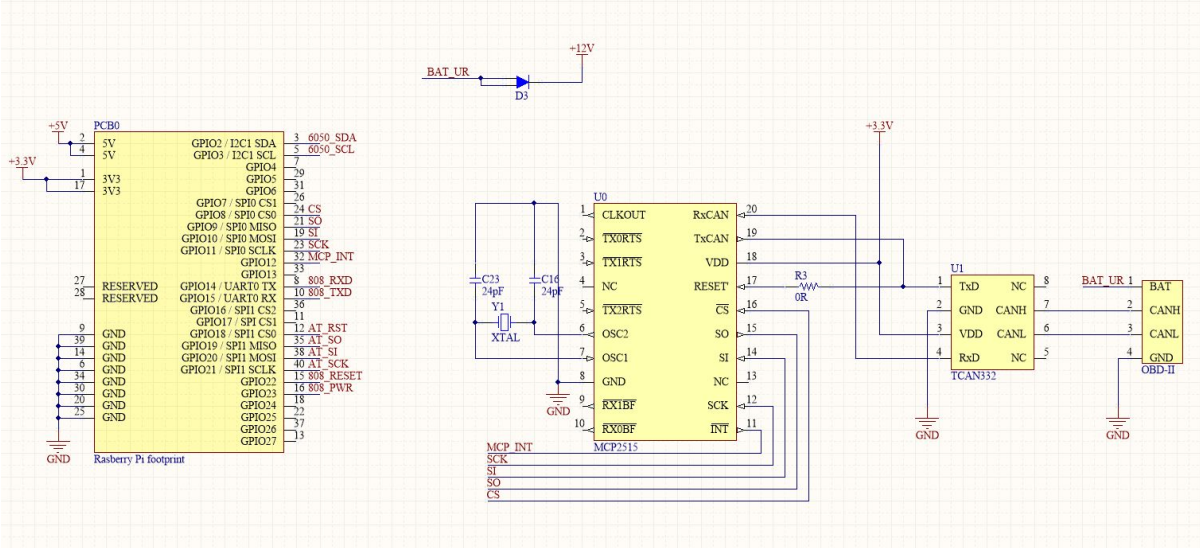


Figure A3.0: CAN node circuit diagram

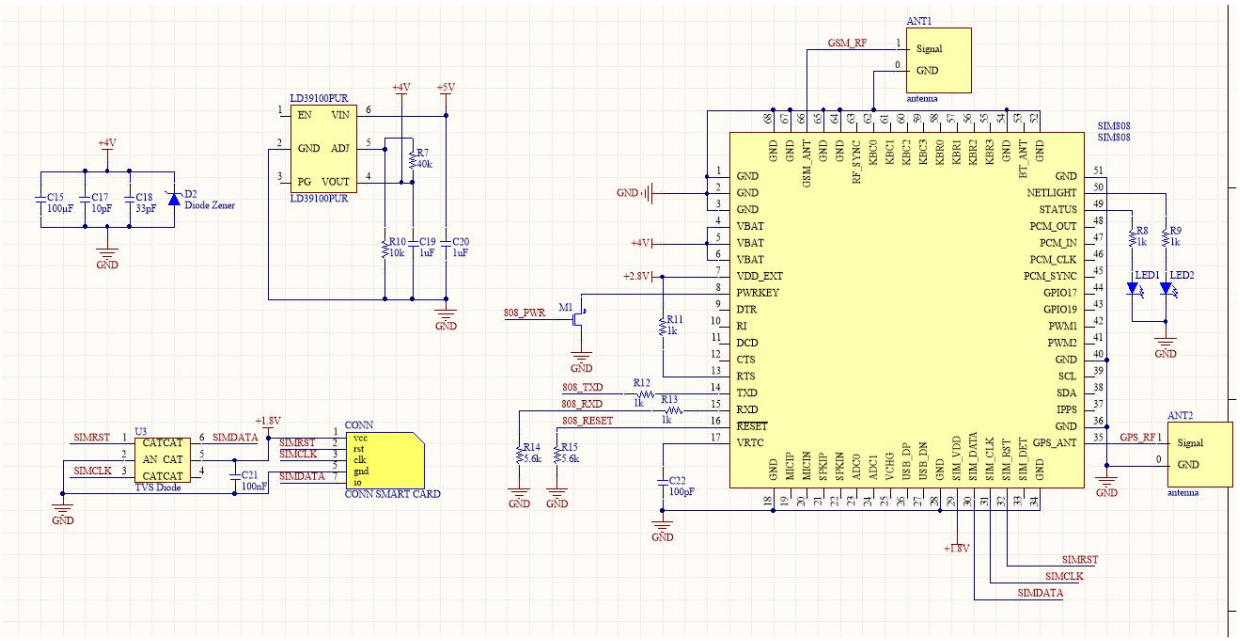


Figure A3.1: SIM808 circuit diagram

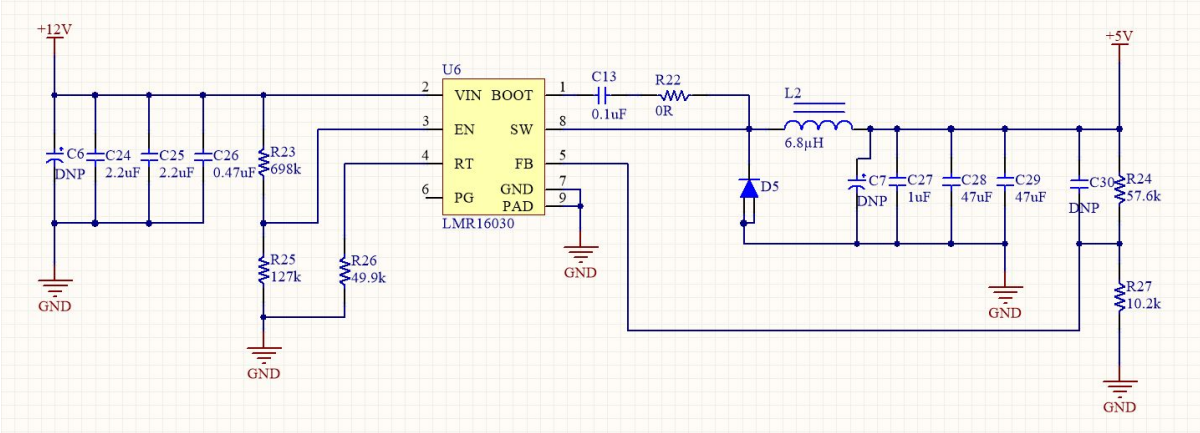


Figure A3.2: Power circuit diagram

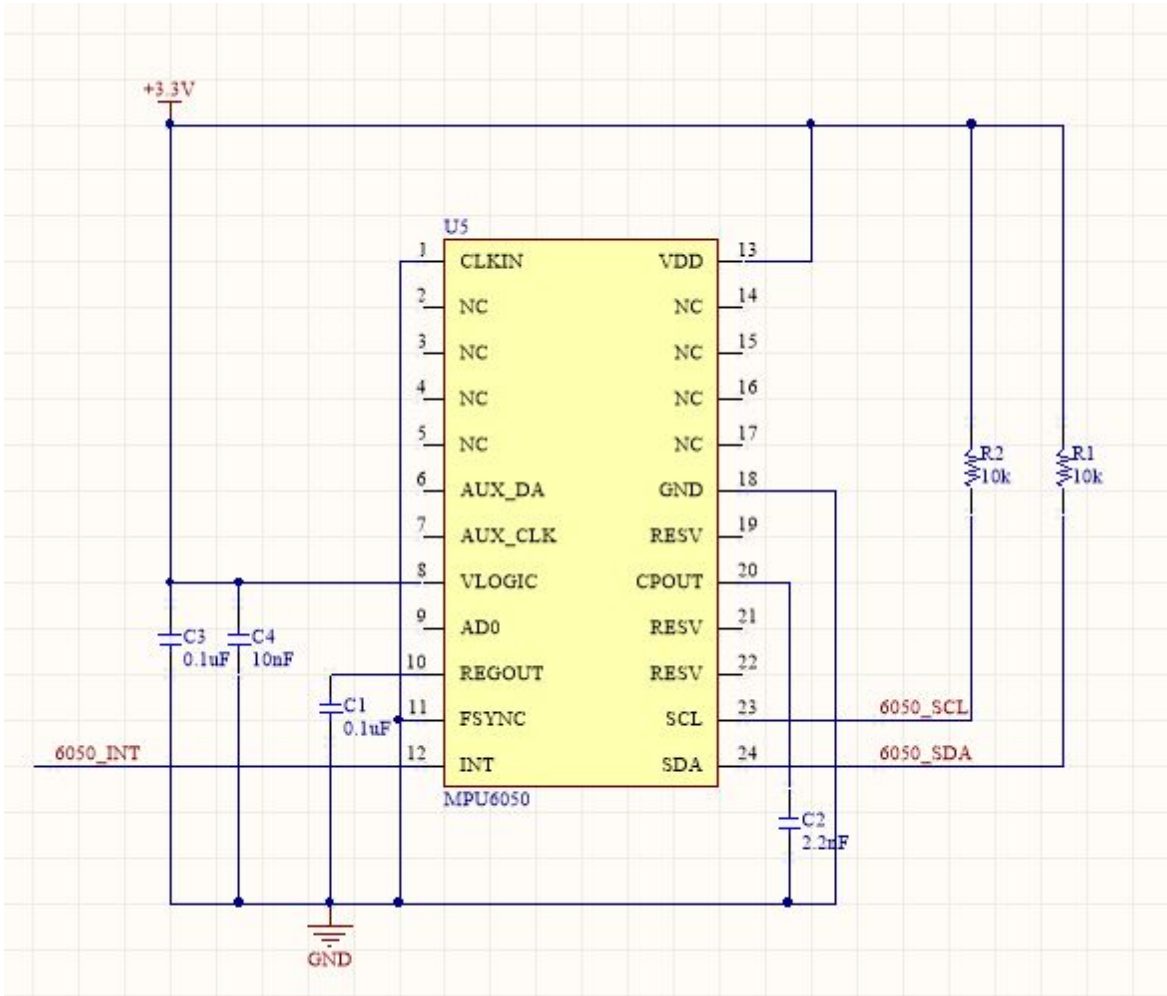


Figure A3.3: MPU6050 circuit diagram

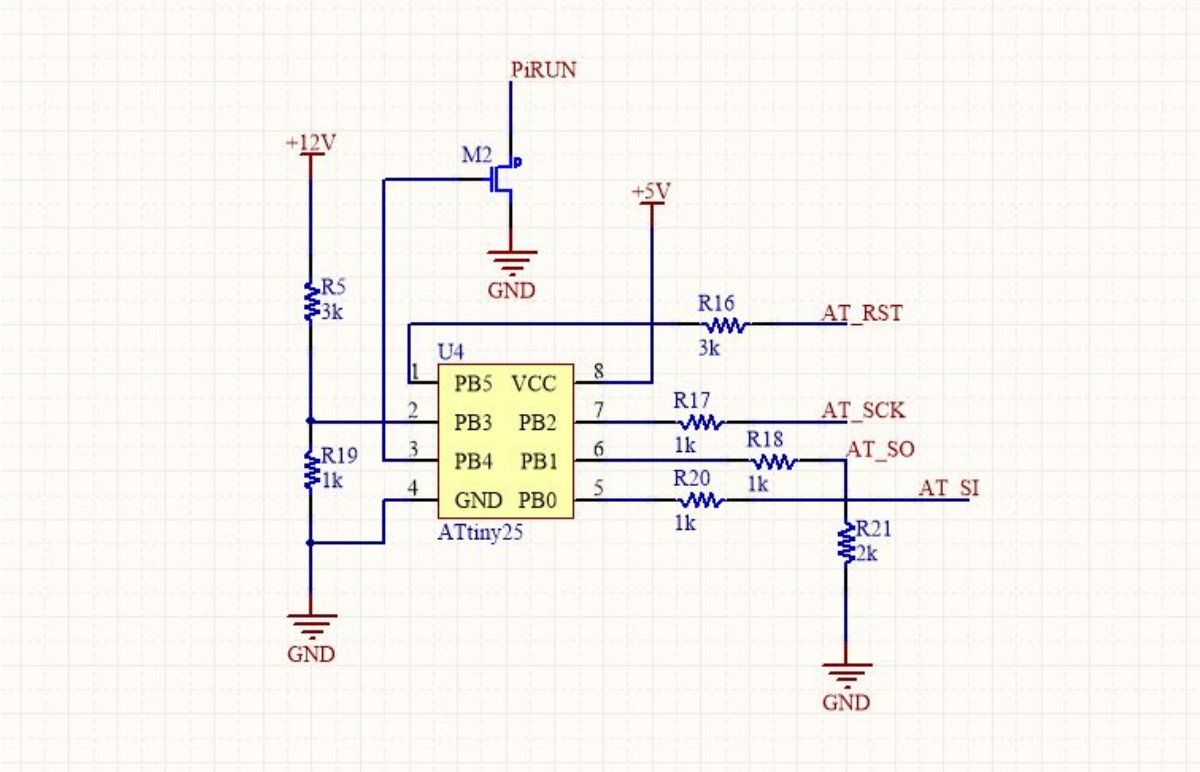


Figure A3.4: ATtiny circuit diagram

Appendix IV - Prototype Hardware Setup

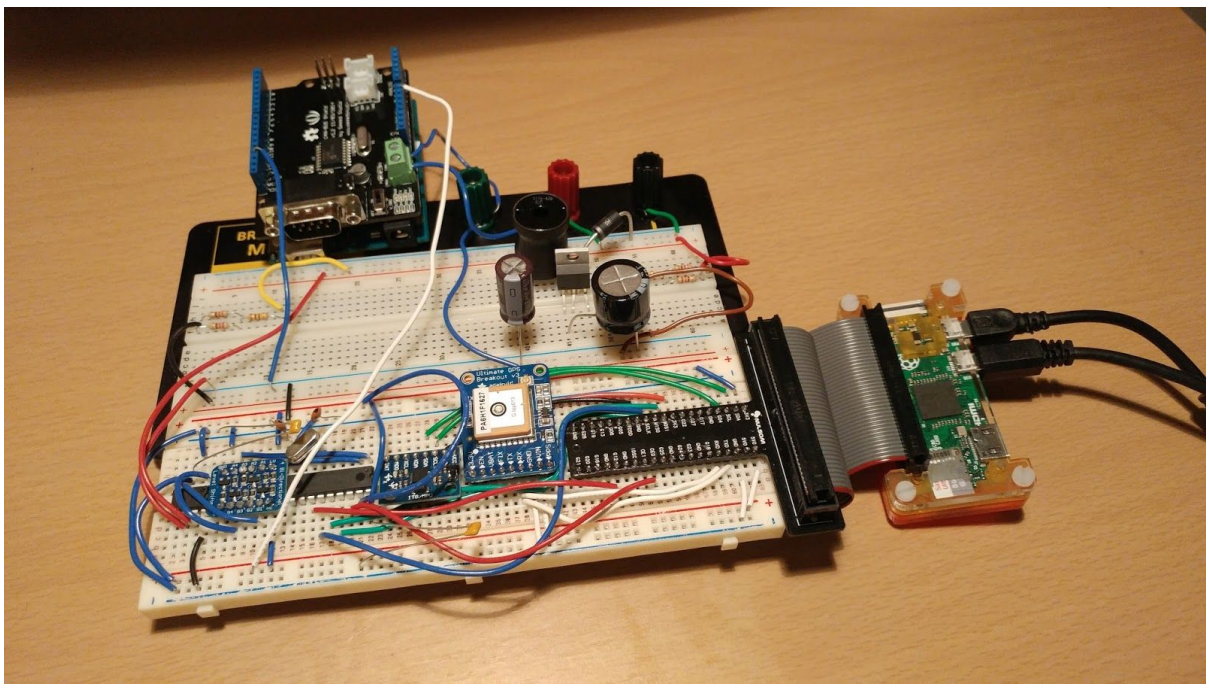


Figure A4.0: Prototype hardware setup

Appendix V - Sim808 Connections

https://www.adafruit.com/product/2637

Sim808 Connections	
Sim808 Pin	**Raspberry Pi pin #**
RESET	15
PWR	16
GND	6
VIN	2
TXD	10
RXD	8

Table A5.0: Sim808 module connections

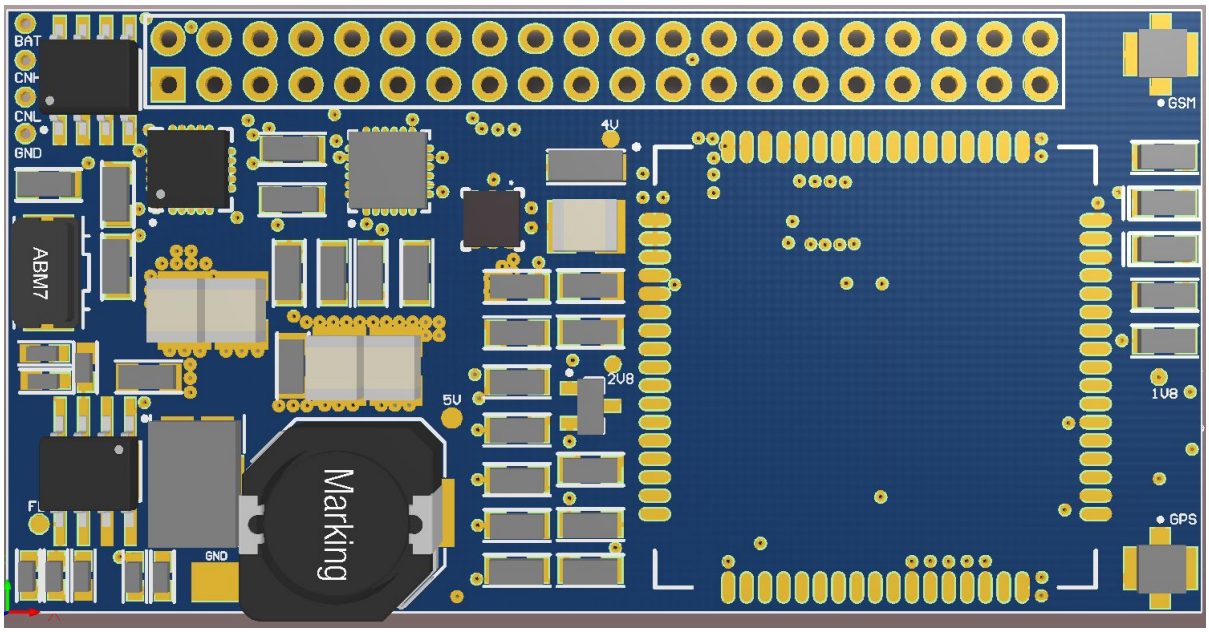
Appendix VI - MPU6050 Connections

https://cdn.sparkfun.com/datasheets/Sensors/IMU/Triple_Axis_Accelerometer-Gyro_Breakout_-_MPU-6050_v12.pdf

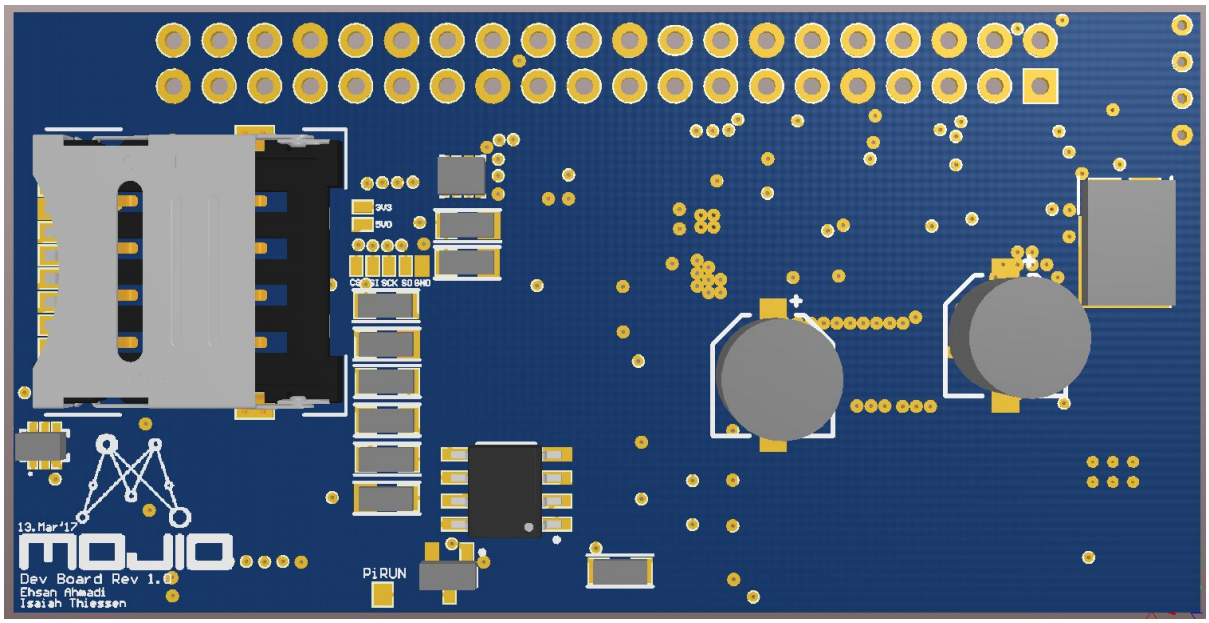
MPU6050 Connections	
MPU6050 Pin	**Raspberry Pi pin #**
VCC	1
GND	6
SCL	5
SDA	3

Table A6.0: Motion sensor module connections

Appendix VII - Printed Circuit Board - PCB



Top View



Bottom View

Figure A7.0 - Printed circuit board layout

Appendix VIII - PCB Layout Considerations

High frequency switching power supplies are efficient and flexible compared to linear regulators, however they can be a significant source of EMI (Electro-Magnetic Interference) which can corrupt low power signals in their vicinity resulting in a malfunctioning board. EMI generated by switching power supplies can be significantly reduced if the following guidelines are followed when laying out the design. The guidelines refer to figure A8.0 which shows the main AC and DC current loops in the buck converter topology.

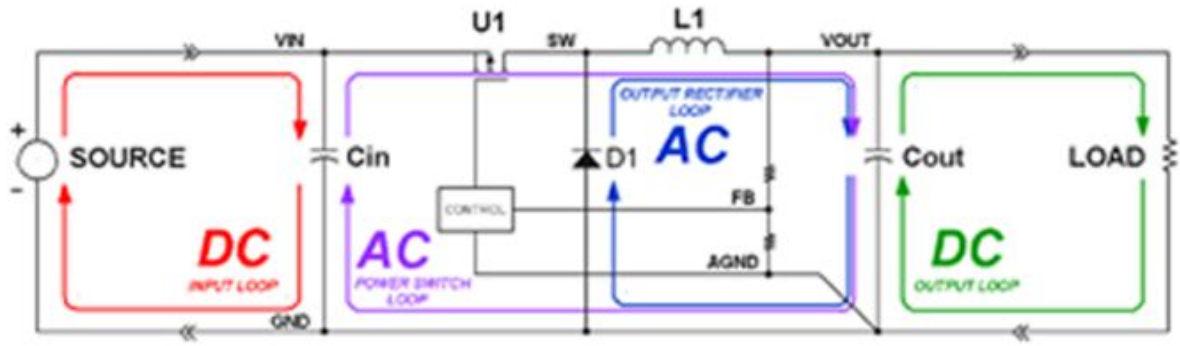


Figure A8.0 - Buck converter topology and current loops [15]

- The SW node carries a high frequency AC current of large magnitude and acts as an antenna, the main source of EMI. A good layout should minimizes the length of the switch node.
- For the AC currents, the length should be minimized with the forward paths on the top layer and the return paths on the bottom layer forming a closed loop. A ground plane on the bottom layer facilitates this.
- Lots of vias have to be used between layers for a low impedance path.
- The ground plane under the power supply should be separated from the signal ground(for the rest of the board) using a small connection in order to keep the high frequency currents contained.
- All the AC and DC loops should be laid out using polygons to ensure low impedance.
- The power supply should be placed as far as possible from sensitive signals, especially the GPS and GSM antennas.

Appendix IX - CAN Node Connections

To Raspberry Pi	
Circuit Pin	**Raspberry Pi Pin #**
5V0	2
3V3	1
GND	6
RESET	GND
CS	24
SO	21
SI	19
STBY	Not currently Used

Table A9.0: CAN Node to Raspberry Pi Connections

OBD Connections	
Circuit Pin	**OBD Port Pin #**
CAN HIGH	6
CAN LOW	14
BAT +12V	16
GND	5

Table A9.1: CAN Node to OBD-II Connections

Appendix X - Live OBD2 Readings for Speed and RPM

The following two images are from accelerating a Toyota Camry 2015 from resting and parking to about 21km/h.

```
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966861.330865        ID: 07e8    000    DLC: 8    04 41 0c 25 dd 00 00 00
the responseTimestamp: 1486966861.330865        ID: 07e8    000    DLC: 8    04 41 0c 25 dd 00 00 00
<class 'can.message.Message'>
message Received:2423.25rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966861.609655        ID: 07e8    000    DLC: 8    04 41 0c 2c a7 00 00 00
the responseTimestamp: 1486966861.609655        ID: 07e8    000    DLC: 8    04 41 0c 2c a7 00 00 00
<class 'can.message.Message'>
message Received:2857.75rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966861.867541        ID: 07e8    000    DLC: 8    04 41 0c 32 28 00 00 00
the responseTimestamp: 1486966861.867541        ID: 07e8    000    DLC: 8    04 41 0c 32 28 00 00 00
<class 'can.message.Message'>
message Received:3210.0rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966862.080551        ID: 07e8    000    DLC: 8    04 41 0c 36 b3 00 00 00
the responseTimestamp: 1486966862.080551        ID: 07e8    000    DLC: 8    04 41 0c 36 b3 00 00 00
<class 'can.message.Message'>
message Received:3500.75rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966864.043770        ID: 07e8    000    DLC: 8    04 41 0c 14 8b 00 00 00
the responseTimestamp: 1486966864.043770        ID: 07e8    000    DLC: 8    04 41 0c 14 8b 00 00 00
<class 'can.message.Message'>
message Received:1314.75rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966864.357970        ID: 07e8    000    DLC: 8    04 41 0c 12 c0 00 00 00
the responseTimestamp: 1486966864.357970        ID: 07e8    000    DLC: 8    04 41 0c 12 c0 00 00 00
<class 'can.message.Message'>
message Received:1200.0rpm
> obd_send_test
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0c 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486966866.295198        ID: 07e8    000    DLC: 8    04 41 0c 10 64 00 00 00
the responseTimestamp: 1486966866.295198        ID: 07e8    000    DLC: 8    04 41 0c 10 64 00 00 00
<class 'can.message.Message'>
message Received:1049.0rpm
>
```

Figure A10.0 - Live RPM readings from revving Toyota Camry engine

```
message Received:0km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968912.963185        ID: 07e8    000    DLC: 8    03 41 0d 00 00 00 00 00
the responseTimestamp: 1486968912.963185        ID: 07e8    000    DLC: 8    03 41 0d 00 00 00 00 00
message Received:0km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968913.982939        ID: 07e8    000    DLC: 8    03 41 0d 04 00 00 00 00
the responseTimestamp: 1486968913.982939        ID: 07e8    000    DLC: 8    03 41 0d 04 00 00 00 00
message Received:4km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968915.007164        ID: 07e8    000    DLC: 8    03 41 0d 08 00 00 00 00
the responseTimestamp: 1486968915.007164        ID: 07e8    000    DLC: 8    03 41 0d 08 00 00 00 00
message Received:8km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968916.027145        ID: 07e8    000    DLC: 8    03 41 0d 0c 00 00 00 00
the responseTimestamp: 1486968916.027145        ID: 07e8    000    DLC: 8    03 41 0d 0c 00 00 00 00
message Received:12km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968917.051285        ID: 07e8    000    DLC: 8    03 41 0d 0e 00 00 00 00
the responseTimestamp: 1486968917.051285        ID: 07e8    000    DLC: 8    03 41 0d 0e 00 00 00 00
message Received:14km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968918.076082        ID: 07e8    000    DLC: 8    03 41 0d 13 00 00 00 00
the responseTimestamp: 1486968918.076082        ID: 07e8    000    DLC: 8    03 41 0d 13 00 00 00 00
message Received:19km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
Message recieved on native socketcan channel
The Message recieved is:Timestamp: 1486968919.095646        ID: 07e8    000    DLC: 8    03 41 0d 15 00 00 00 00
the responseTimestamp: 1486968919.095646        ID: 07e8    000    DLC: 8    03 41 0d 15 00 00 00 00
message Received:21km/h
The Message sent is:Timestamp:        0.000000        ID: 07df    000    DLC: 8    02 01 0d 00 00 00 00 00
Message sent on native socketcan channel
```

Figure A10.1 - Log for accelerating on Toyota Camry

Appendix XI - Complete Hardware Setup

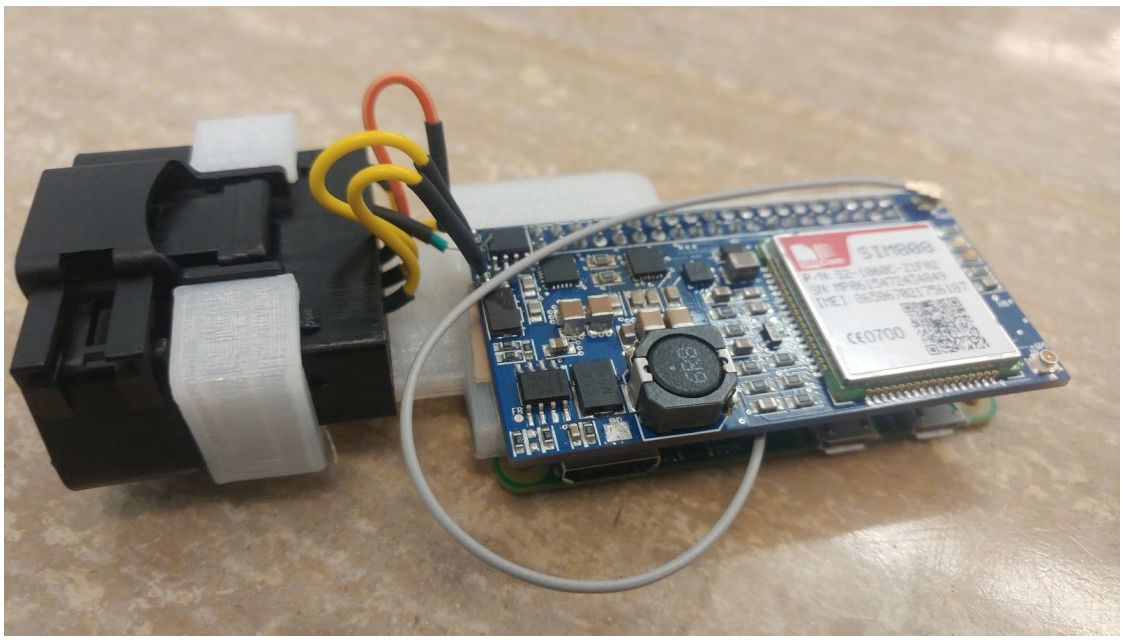


Figure A11.0 - Final setup assembled with the enclosure