DESIGN OF AN AUTOMOTIVE IOT DEVICE TO IMPROVE DRIVER FAULT DETECTION THROUGH ROAD CLASS ESTIMATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

By

Matt Murray

June 2022

COMMITTEE MEMBERSHIP

TITLE:	Design of an Automotive IoT Device to Improve Driver Fault Detection Through Road Class Estimation
AUTHOR:	Matt Murray
DATE SUBMITTED:	June 2022
COMMITTEE CHAIR:	Dennis Derickson, Ph. D Professor of Electrical Engineering
COMMITTEE MEMBER:	Dale Dolan, Ph. D Department Chair of Electrical Engineering
COMMITTEE MEMBER:	Fred Depiero, Ph. D Professor of Computer Engineering

ABSTRACT

Design of an Automotive IoT Device to Improve Driver Fault Detection Through Road Class Estimation

Matt Murray

Unsafe driver habits pose a serious threat to all vehicles on the road. This thesis outlines the development of an automotive IoT device capable of monitoring and reporting adverse driver habits to mitigate the occurrence of unsafe practices. The driver habits targeted are harsh braking, harsh acceleration, harsh cornering, speeding and over revving the vehicle. With the intention of evaluating and expanding upon the industry method of fault detection, a working prototype is designed to handle initialization, data collection, vehicle state tracking, fault detection and communication. A method of decoding the broadcasted messages on the vehicle bus is presented and unsafe driver habits are detected using static limits. An analysis of the initial design's performance revealed that the industry method of detecting faults fails to account for the vehicle's speed and is unable to detect faults on all roadways. A framework for analyzing fault profiles at varying speeds is presented and yields the relationship between fault magnitude and speed. A method of detecting the type of road driven was developed to dynamically assign fault limits while the vehicle traveled on a highway, city street or in traffic. The improved design correctly detected faults along all types of roads and proved to greatly expand upon the current method of fault detection used by the automotive IoT industry today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

1.1: Research Significance

Technology has been integrated into almost every facet of our lives creating a significant worldwide demand for data. A broad network of smart devices is vital to enhancing the standard of living. The integration of IoT devices has increased dramatically over the last decade and have become indispensable to daily life. Something as simple as a smart thermostat or doorbell are now widely accepted and used. The applications in both the private and commercial sectors and the low barrier to entry are driving companies to redesign products to include new smart capabilities. One example is in the automotive industry which has moved to fuse old and new technology to improve driver safety.

Automotive manufacturers are pushing electric, autonomous and connected vehicles that boast a full suite of smart technologies. The advancements in these new vehicles highlight the strides made in this technological era. But to what level do these advancements improve the standard of living? In 2018 the World Health Organization(WHO) reported that the deaths from traffic accidents had increased by 1.35 million people per year, equating to 3700 deaths per day[1]. Typical causes of these accidents include speeding, drunk driving and failure to wear seatbelts. One of the technologies developed to make the road safer is the Advanced Driver Assistance Systems (ADAS). It is a vehicle control system that uses sensors to help the driver recognize habits and react to potentially dangerous situations. Not every vehicle on the road today is equipped with these ADAS systems. In 2020 there were 289.9 million vehicles in the US alone[2]. One in four of these vehicles are 16 years of age or older[3]. Meaning that there are over 72 million vehicles manufactured before 2004. Vehicles of this age have only a rudimentary level of computerization. It was not until 2006 that the US government allowed full public access to the GPS satellite network[4]. It can be concluded that a vast number of vehicles on the road today do not have the technology to track vehicle location let alone assist the driver in recognizing dangerous habits.

A current solution to provide information to the driver is to utilize the On Board Diagnostic (OBD) port. The initial OBD-II protocol was tested in the United States in 1994 and became standard on all cars and small trucks in 1996[5]. The protocol communicates diagnostics and status of vehicle engine components. Most modern vehicle tracking products attach a cellular and GPS package to the OBD port to report the location of the vehicle. This combination of vehicle data and tracking technology is used to improve routing, dispatching, theft prevention, vehicle recovery, fuel monitoring and any other service where vehicle data is needed. These devices normally require newer vehicles where data is abundant and struggle with older legacy vehicles who have limited OBD traffic.

1.2: Research Motivation

The motivation behind this research is to first mimic the hardware implementation of a modern automotive tracking device. Utilizing the current methods of data collection and reporting, develop a driver safety algorithm that can monitor a driver's habits and report them to an overseer. Keeping older vehicles in mind, that have limited OBD traffic, identify a method of decoding the broadcasted data on the OBD port and use the information to aid in the detection of faults. Finally, by analyzing the performance of the first fault detection model, determine a method to improve the process by which faults are detected.

1.3: Research Goals

As this thesis revolves around the design, analysis and improvement of a physical device, a set of research goals are laid out in Table 1.1 to guide the direction of the thesis.

Table 1.1: Research Goals

#	Research Goal
1	Design and build an IoT device that follows the current trends in the automotive IoT sector.
2	Develop a method of automatically decoding the vehicle bus for key parameters related to the operation of a vehicle.
3	Evaluate the effectiveness of the industry method of fault detection
4	Improve upon fault detection by detecting the type of road traveled and adaptively changing fault limits.

Chapter 2: Technology Background

2.1: Controller Area Network (CAN)

The On Board Diagnostics (OBD) is one of the inputs to modern day automotive trackers. The OBD is a vehicle communication system that monitors the emission control and emission related components/systems within the vehicle. By monitoring and evaluating the various components and systems, the on board computer is able to determine the presence of a malfunction and illuminate the "check engine" or "service engine soon" light[6]. The terms "OBD" and "OBDII" are used interchangeably to describe the second generation of On Board Diagnostics. The OBD system is a high layer protocol that specifies the OBDII connector as well as a set of four protocols that can report data. The connector as well as the protocols can be referenced below in Figure 2.1.

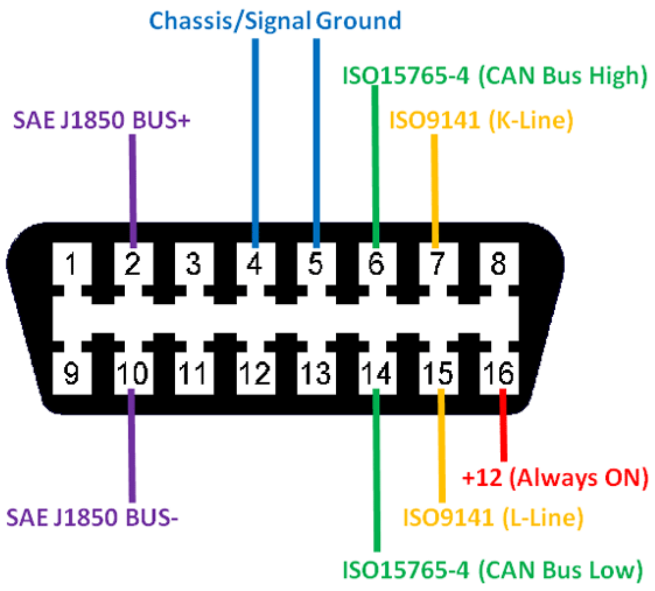


Figure 2.1: OBDII Connector Pinout

Since 2008, the Controller Area Network (CAN) bus has been the mandatory protocol for OBDII in all cars sold in the United States essentially eliminating the other three protocols shown in the specification. As such, the protocol used to collect data from the vehicle will be the CAN bus. CAN is a vehicle bus standard designed to allow microcontrollers within the vehicle to communicate to one another without using a host computer. It is a multi-master serial bus where all nodes are connected to a two wire communication line. Designed to eliminate the analog and digital interconnects between electronic control units (ECU's) the CAN bus creates a simple and lightweight network to transfer data within the vehicle's different sensors and modules.

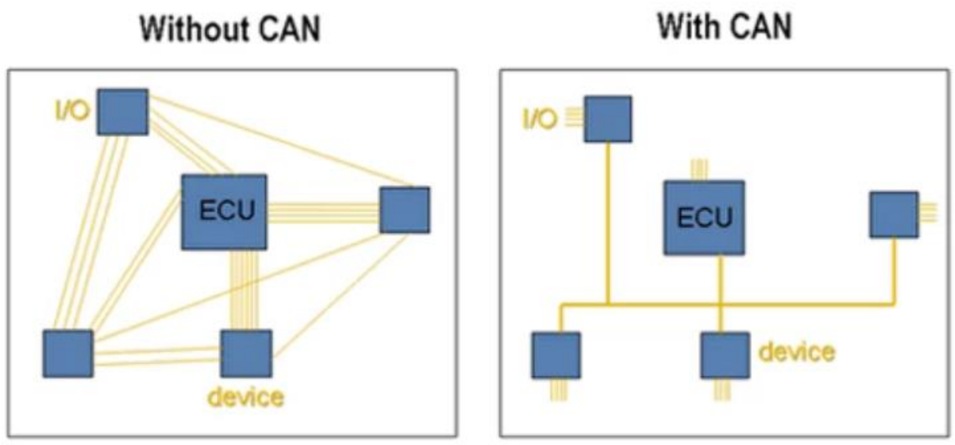


Figure 2.2: Wiring Advantages of CAN

Devices connected to the CAN bus, also called nodes, communicate by sending message frames over the common bus. Any message sent on the network can be read by any node connected. These messages fall into four categories. The first being the remote frame that is used to request data from a different node. The second is the data frame that echoes back the received remote frame with the data populated. The third and fourth are the error and overload frames that are used to report bus errors and give spacing between messages when needed[7].

The two types of messages that are critical to this analysis are the remote and data frames as they are the method where data is requested and received from the vehicle's engine control unit. These two types of frames have roughly the same structure and can be referenced below.

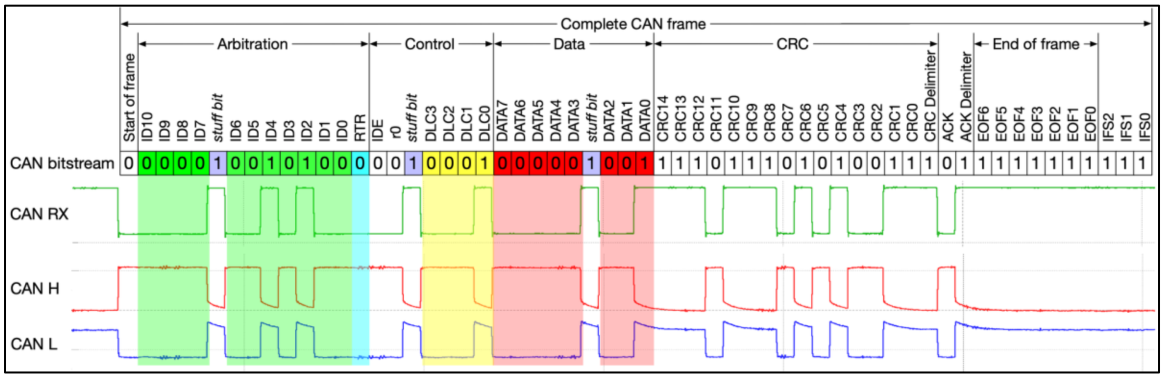


Figure 2.3: Complete CAN Data Frame

The above Figure shows the format of a complete CAN data frame. The frame begins with an 11 bit arbitration field that identifies the message and sets message priority. The lower the message identifier the higher the priority of the message. This field is commonly known as a Parameter Identifier (PID). The next field is the control field that signals the number of bytes in the data frame. The final section is the data field that has a bit length corresponding to the length specified in the control field. These three fields characterize the CAN bus frame.

As mentioned the data frame and remote frame use the same format. When sending a remote frame to request data the arbitration field or PID corresponds to the requested message and the control field sets the number of bytes to send. The data field is removed and the Remote Transmit Request(RTR) bit is set high signaling that the frame is a remote frame.

The challenge with using the vehicle CAN bus as a data source is not understanding the communication protocol but decoding the arbitration fields and identifying the scaling factor used on the data. This information is proprietary and closely held by automotive manufacturers because a user or intruder can damage the vehicle if data is sent incorrectly and in some cases the vehicle can even be started without a key if the ignition sequence is known. Fortunately, an intruder gaining access to the vehicle CAN bus cannot directly perform a targeted attack. In fact, despite the networks lack of encryption and authentication protocols the CAN bus is encoded according to a specific format designed by the car manufacturer. The only way to obtain the format for a specific vehicle is through the reverse engineering of the vehicle buss [8]. This reverse engineering can often be a time consuming and laborious task if the number of parameters is large. For example, for the PIDs that contains RPM and speed, reverse engineering can be simplified due to the accessibility of data from the vehicle's dashboard. Further discussion on reverse engineering can be found in Chapter 5.

In most cases, some form of reverse engineering is needed to acquire the correct parameters and scaling factors. There is in fact a standardized list of PIDs for SAE-J1979(CAN)[??]. Not all vehicles will support all PIDS and depending on the year make and model of the vehicle, the PIDs may not be accessible by simply requesting the data. The list of PIDs is given so that mechanics and vehicle inspectors, using certified equipment, can analyze a vehicle for emissions inspections or diagnosis. This thesis will focus on the automatic decoding of broadcasted messages on the bus due to the volatility of what parameters are supported, accessible or readable. Allowing for the passive reading and decoding of data without the need to write vehicle specific parameters to the bus.

2.2: Cellular technologies

According to ARM, one of the leading microprocessor producers, an IoT device is defined as “Pieces of hardware, such as sensors, actuators, gadgets, appliances or machines that are programs for certain applications and can transmit data over the internet or other networks”[10]. Therefore, no one technology defines IoT devices. Instead, a common thread of connectivity is the underlying similarity of all such devices. These options include wireless personal area networks (WPAN) like Bluetooth, wireless local area networks (WLAN) like Wi-Fi, and cellular or the third generation partnership project (3GPP) standards[9]. These cellular standards fall into 2G, 3G, 4G and 5G. Where 5G is the leading edge and 2G/3G are currently being discontinued in the United States but are still prevalent in other parts of the world. The 3GPP family of cellular network technologies is the leading platform for worldwide wireless communication and is therefore the standard for the vast majority of all IoT applications. The wide range of form factors and performance metrics ensures that every IoT application can be paired with a corresponding cellular chipset to best match the needs for said device.

The leading technology for connecting smartphones and other advanced mobile devices is 4G(LTE). Supporting data rates of more than 100Mbps[11], 4G can meet the bandwidth and data requirements for virtually any mobile IoT application. 4G LTE is fully optimized for high speed data communication with low latency with higher power consumption[9]. To better match the requirements of a wider range of IoT use cases that operate with low data requirements LTE CAT-1 was released in 2014 and was optimized to reduce the data rate to 10Mbs[11] and include a power saving feature that can support greater battery lifetimes[9]. Furthermore, to accommodate more IoT devices on the network LTE-M was created to improve on LTE CAT-1. The LTE-M supports a further reduced data rate of 800kbps[11] with a reduced transmitting power and extended coverage options. LTE-M is considered the standard for most IoT applications as it balances both power consumption and data rate. The final technology used in IoT applications is narrow band IoT also called NB-IoT. NB-IoT is utilized in extremely low bandwidth applications such as smart parking and smart meters[9]. With data rates capped as 100kbps NB-IoT is designed to support a massive number of low throughput devices[11]. While more exotic modules exist for specialized applications, the vast majority of all IoT devices rely on one of the above discussed technologies.

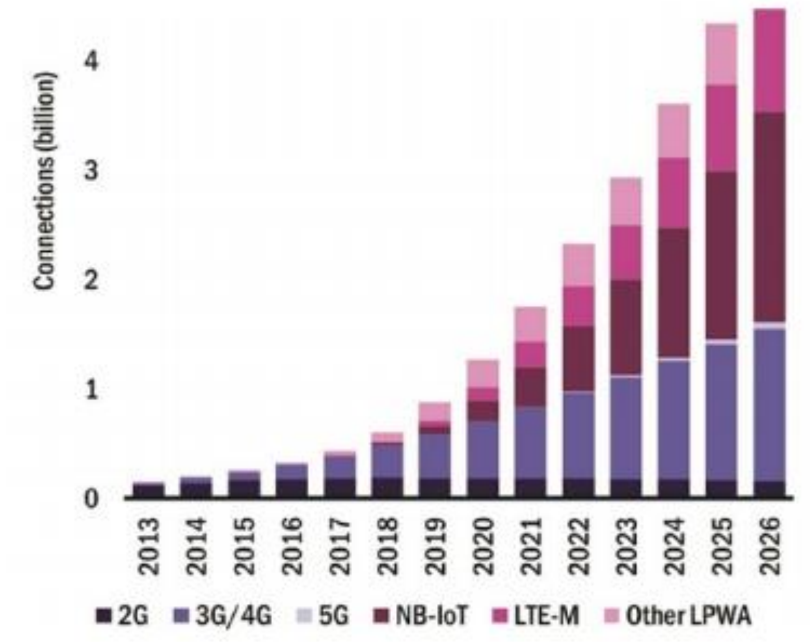
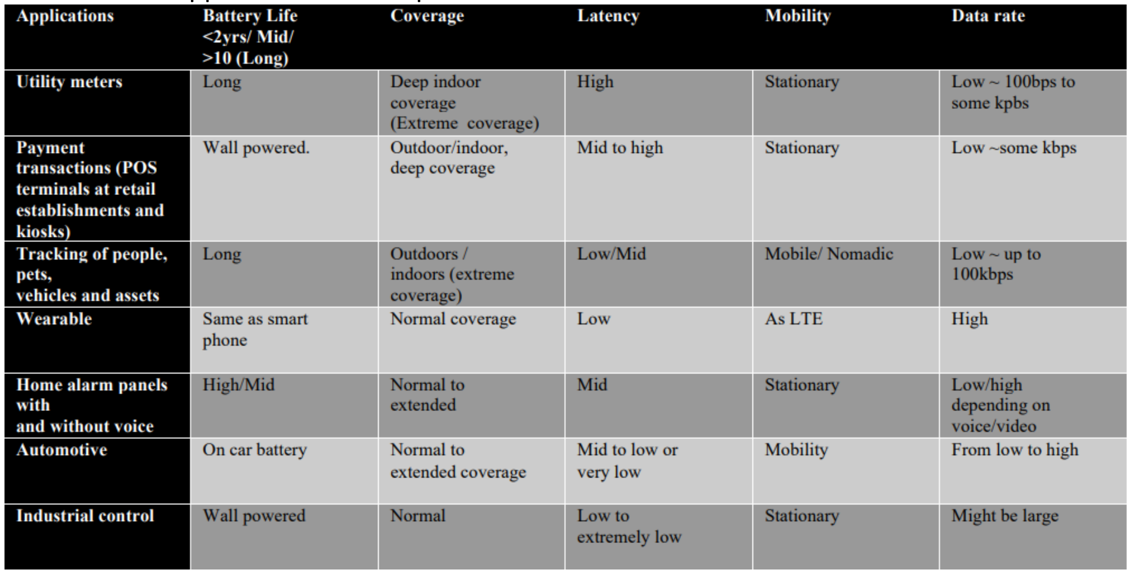


Figure 2.4: Cellular Connections Forecast[9]

Figure 2.4, drawn from a 2018 report that forecasts the growth of the IoT market[11], indicates a steadily growing market of annual connected devices. Each respective technology shown above covers a range of applications defined by the needed data rates, power consumption and latency. A brief summary of these applications and their requirements can be seen below in Table 2.1[12].

Table 2.1: IoT Applications and Requirements



Applications	Battery Life <2yrs/ Mid/ >10 (Long)	Coverage	Latency	Mobility	Data rate
Utility meters	Long	Deep indoor coverage (Extreme coverage)	High	Stationary	Low ~ 100bps to some kpbs
Payment transactions (POS terminals at retail establishments and kiosks)	Wall powered.	Outdoor/indoor, deep coverage	Mid to high	Stationary	Low ~some kbps
Tracking of people, pets, vehicles and assets	Long	Outdoors / indoors (extreme coverage)	Low/Mid	Mobile/ Nomadic	Low ~ up to 100kbps
Wearable	Same as smart phone	Normal coverage	Low	As LTE	High
Home alarm panels with and without voice	High/Mid	Normal to extended	Mid	Stationary	Low/high depending on voice/video
Automotive	On car battery	Normal to extended coverage	Mid to low or very low	Mobility	From low to high
Industrial control	Wall powered	Normal	Low to extremely low	Stationary	Might be large

The table above shows automotive falling within two different categories containing widely different requirements. One category includes the communication of a vehicle over the cellular network, and the other a device tracking the movement of a vehicle. Each of these categories are defined by their respective latencies and data rates. A vehicle may require firmware updates or out of band computation, thus requiring low latencies and a high data rate. Conversely, a device tracking a vehicle that sends occasional location pings requires very little data to be transmitted. Furthermore, the tracking data does not relate to the immediate safety of the driver or vehicle and therefore does not require a low latency connection. While each use case covers the extremes in the automotive field, most automotive cellular applications fall between these two. The standard for most automotive tracking devices is the LTE-M technology that, as mentioned, provides data rates of 800kbps while minimizing power consumption and cost[11].

2.3: GNSS technologies

As stated in the previous chapter, IoT devices are a combination of hardware, sensors, actuators, and gadgets that perform specific functions and can transmit data over the internet or other networks[10]. Just as cellular connectivity is a general standard for automotive IoT devices, Global Navigation Satellite System (GNSS) receivers are one of the most common sensors found within automotive IoT. The benefits of using GNSS have been known for years and are an integral part of everyday life. GNSS receivers are a highly versatile device that provide worldwide coverage for 9.6 billion devices[13].

GNSS receivers rely on a system of satellites to provide real time positioning data. These satellite systems are referred to as constellations and are maintained by different countries around the world. They are, GPS(United States), BeiDou(China), Galileo(EU), GLONASS(Russia), IRNSS(India) and QZSS(Japan)[13]. The network of satellites use microwave signals that are transmitted to GNSS devices to give information on location, speed time and direction. All data originating from the satellites is timestamped with GPS time that is accurate to roughly 4ns or three billionths of a second. The receiver uses the time difference between the signal's reception and the timestamp to compute the transition time from the satellite to the receiver. The distance to the satellite can then be calculated with the speed of light and the formula $Distance = Rate * Time$. By computing this distance with a minimum of 4 other satellites the receiver is able to compute its own three-dimensional position as well as time. While four is the minimum number of satellites needed to determine (x, y, z, t), a receiver will use all satellites within its field of view to increase accuracy[14].

The main focus of modern day GNSS receivers is the mitigation of the many sources of error that degrade the accuracy of a location fix. GNSS signals are very low in power and are prone to several sources of noise and errors. When the distance to each satellite measured by the GNSS receiver is contaminated by these errors, the accuracy of the location fix is decreased[15]. While there are many forms of error within GNSS tracking such as timing errors, relativistic errors and orbital errors, the most significant of all are propagation errors. These propagation errors occur in the upper regions of the atmosphere and at the final approach to the receiver. When the signal reaches an altitude of roughly 1000km above the Earth's surface it penetrates the ionosphere and is no longer traveling through empty space. This layer of the atmosphere includes gasses that are readily ionized by the sun's radiation. The intensity of the solar radiation is a key factor in determining the refractive properties of the ionosphere[16].

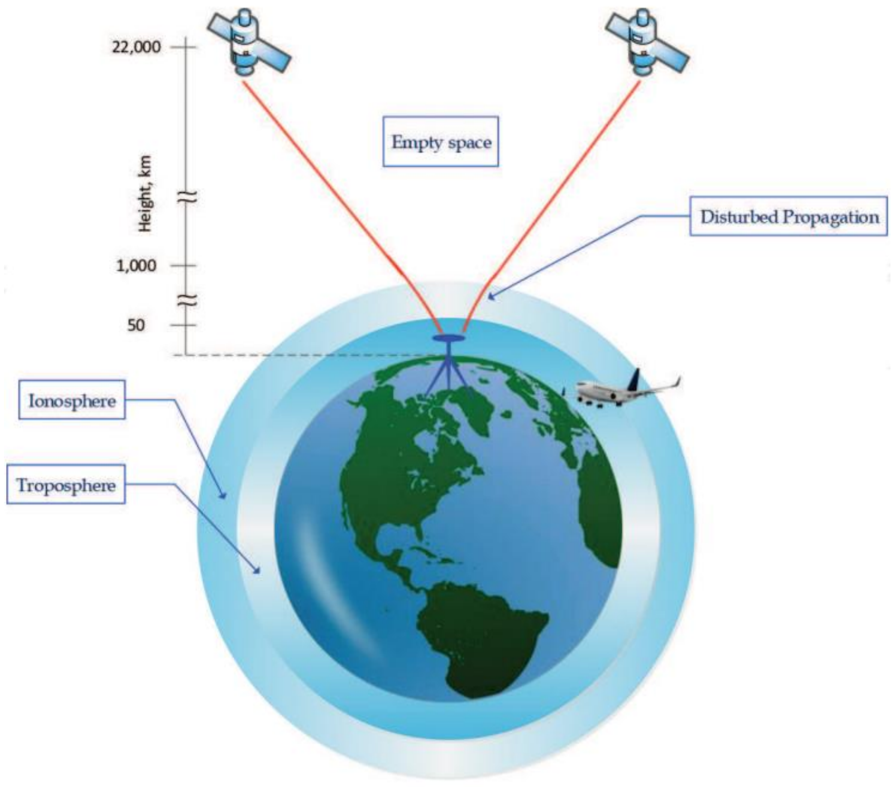


Figure 2.5: GNSS Signal Propagation Mediums

The refraction of the signal in the upper atmosphere effects the signal transit time measured by the receiver. The ionosphere acts as a dispersive medium, meaning that the ionospheric delay is frequency dependent. This delay is one of the most significant ranging error in GNSS positioning and can cause an error up to 100m in extreme circumstances[17]. Once through the ionosphere the signal must then pass through the troposphere, roughly 20km above the surface of the Earth, where nearly all weather conditions take place. Comprised of dry gasses and water vapor, it too induces delays to the GNSS signals. However, being electrically neutral, this layer is nondispersive and not dependent on GNSS frequencies. Due to this, the tropospheric error is mitigated by mapping the tropospheric delay in the local area and factoring in satellite elevation angle to correct for the total delay[18]. Without the modeling of the troposphere, the tropospheric delay can induce 2.5m-25m of range error. The last major source of error is called multipath error. Shown in Figure 2.6, multipath error is caused by the combination of direct line of sight (LOS) signals with one or more of its echoes that are delayed versions of the original LOS signal[15].

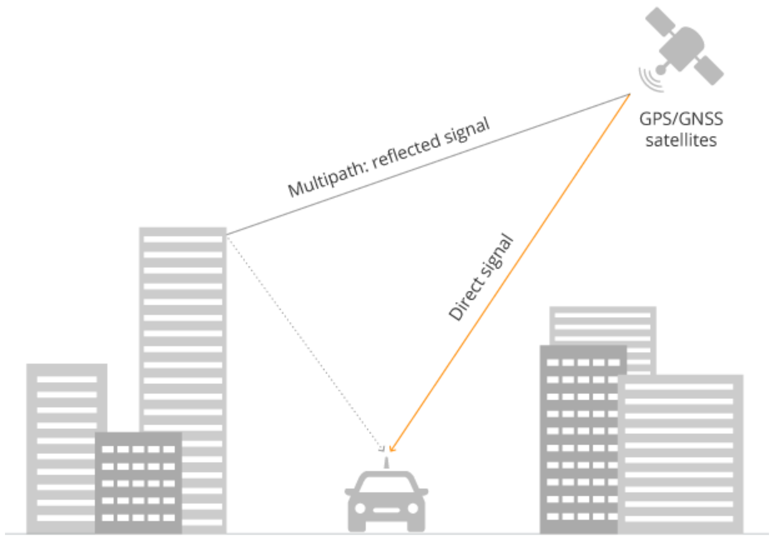


Figure 2.6: Multipath Reflections

The multipath effect depends on the surrounding environment. GNSS receivers are evaluated in what is called “open sky” conditions where there are no obstacles in the way to block LOS from the receiver in any direction. The multipath effect is normally observed in mobile applications where the receiver will enter an urban area that blocks LOS from most, if not all, of the satellites in view. In the most severe conditions multipath can induce up to 100m of error[19]. Mitigation of this error is the most challenging of the sources presented. Some modern receivers use techniques relying on antenna arrays, where the receiver can tune itself to only track the LOS signal and block all other replicas of the signal[20]. Furthermore, the signal adds phase as it echoes off each surface on its way to the receiver. By detecting the differences in phase from the incoming signal and echoed signal, the receiver can adjust satellite weighting or even reject measurements with severe multipath effects[21]. Even with these two techniques to mitigate multipath errors, there are no methods to ensure accurate measurements in all situations. As a result, cities with urban canyons create inaccurate data. Other technologies such as dead reckoning and remote rover systems assist in providing location data when a GNSS signal becomes inundated with multipath errors. However, for the scope of this paper these advanced technologies will not be employed as they require further hardware in place to operate.

Note: While there is a separation between cellular and GNSS technologies within this thesis, the current trend of the market combines both the cellular and GNSS receiver onto the same chipset.

2.4: Edge vs. Cloud Computing

For resource limited devices, it is important to define where the computation of data occurs. Depending on where that occurs the hardware needed to support the needed functionality can change drastically. This decision is where cloud computing and edge computing are primarily discussed.

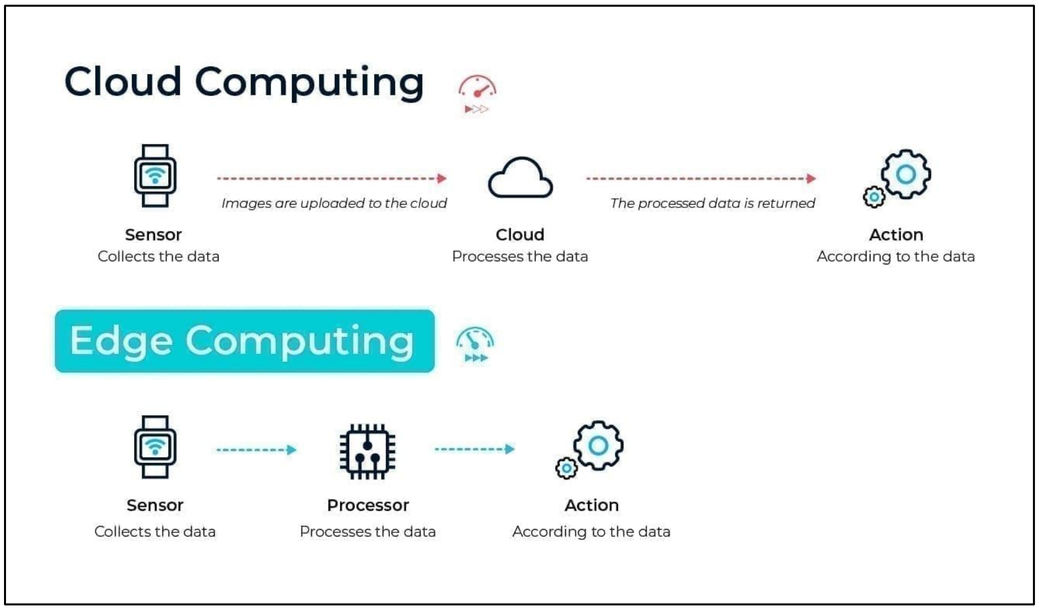


Figure 2.7: Cloud Computing vs. Edge Computing

Cloud computing utilizes a simpler data collection tool that is enabled to record data and stream it to the cloud over a network such as the cellular network or Bluetooth. This type of architecture relies on computational power closer to the "core" of the network. Therefore, actions and decisions do not occur in real time. Depending on the use case, cloud computing can greatly increase latency as well as demand a larger amount of bandwidth[22]. With automotive IoT devices, real time processing is a major aspect of its functionality and therefore adopt an edge computing architecture. This architecture utilizes a microprocessor to do the majority of the processing on the live data that is collected. This allows the device to make decisions and react in real time.

Many forms of automotive IoT devices share the computational load with the cloud. As the device collects raw data it can find insights such as the vehicle’s location, movement, operation, and health. Software in the cloud can be focused on how to best present the information to the user. This is especially useful in applications where hundreds or thousands of devices are in the field. The processed data sent to the cloud may then be used in applications such as fleet management, route planning, vehicle health and any other application that utilizes data collected from a vehicle.

Chapter 3: Hardware Design

3.1: Block Diagram

To evaluate the effectiveness of driver safety methods within the automotive environment, a physical device must be built as a proof of concept and as a data collection tool. As discussed in Chapter 2, typical automotive IoT devices utilize edge computing to allow for real time processing and reduced bandwidth usage. While the hardware implementation of these devices varies from application to application, they all implement some form of microprocessor, cellular modem and GNSS chipset. Below is a block diagram that incorporates all the mentioned hardware elements with the addition of a CAN receiver and an accelerometer.

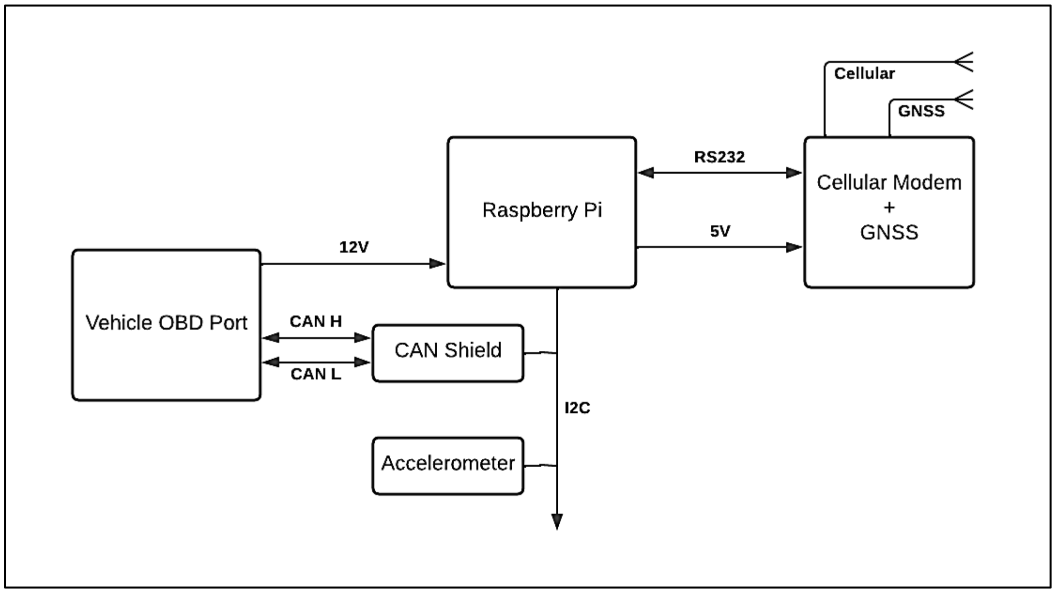


Figure 3.1: Hardware Architecture Block Diagram

With the microcontroller controlling data collection from all sensors, the device is able to quickly collect and process data in real time. Upon collecting data, the device will then be able to preprocess the data converting it to known metrics and evaluate the current state of the vehicle with the driver safety model that will be developed in further chapters.

3.1: Device Selection

Following the above block diagram, devices must be chosen to fulfill the needed functionality of the automotive IoT device. At the center of the device sits a microcontroller. Responsible for interfacing and collecting data from all other periphery devices, the microcontroller requires a processor and interface suite that is robust enough to handle communication to each sensor over varying protocols. Due to this, the Raspberry Pi 3B+ was chosen as the microcontroller for this device. With a 40 pin GPIO header and 4 USB 2.0 ports the Raspberry Pi can support many external sensors and communication busses. It also boasts a 64bit Arm Cortex-A53 processor running at 1.4GHz[23]. By using a microcontroller that over exceeds the complexity of the total IoT device, the use of a real time operating system can be avoided and Linux can be used as the operating system of the device. By allowing the use of Linux, the device can be written in Python and the vast extension library can be utilized to decrease the development time of the device. The Raspberry Pi also has the advantage of allowing extensions through the 40pin header which is increasingly useful for adding a CAN bus communication interface.

The Seeed Studio two channel CAN bus driver is a Raspberry Pi shield that adds two CAN I/O ports to the Raspberry Pi[24]. The shield is capable of flexible data rates up to 8Mbs and communicates with the Raspberry Pi through the Serial Peripheral Interface (SPI). The advantage of the two channel implementation is that one channel can be connected to a male OBD-II connector for use in a vehicle and the other can be looped back into the first for bench testing and the simulation of log files. The shield also extends the Raspberry Pi's two SPI interfaces to allow further sensors, like an accelerometer, to be attached. Figure 3.2 shows the communication interfaces of the Raspberry Pi as well as the Seeed Studio CAN shield. Figure 3.3

shows the block diagram of the CAN shield and how it extends the Raspberry Pi's header to allow for the interfacing of the dual CAN and vehicle power.

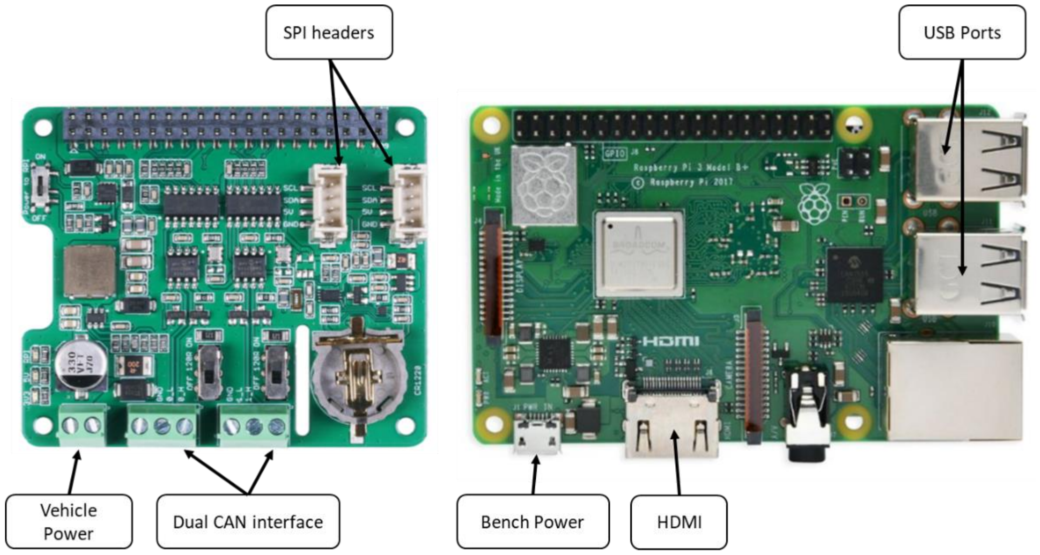


Figure 3.2: Raspberry Pi and CAN Shield Interfaces

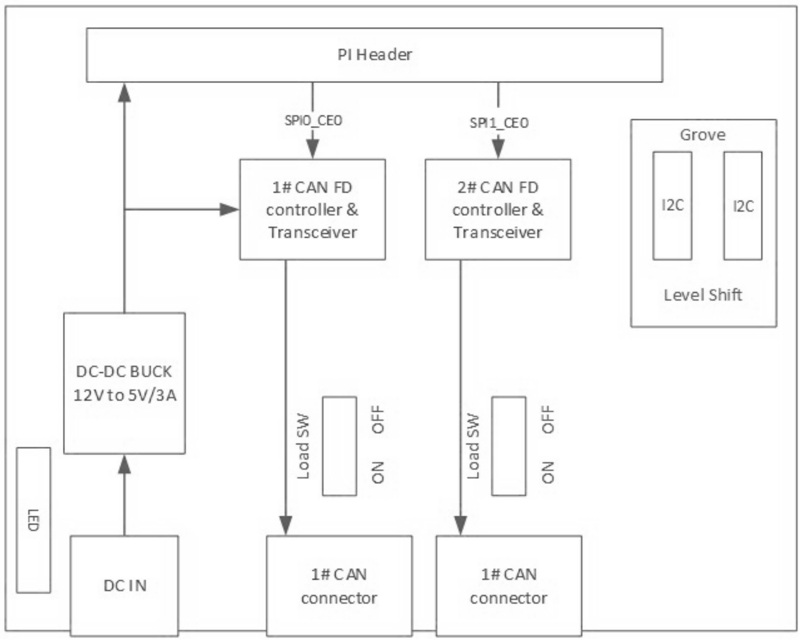


Figure 3.3: Seeed Studio Dual Can Shield Block Diagram

The next peripheral chosen was the accelerometer. With many different flavors on the market and its low cost it is easy to choose one that fits any need. The device selected is the Lis3dh accelerometer. As with many others, it communicates over I2C and uses a FIFO buffer to store measurements. Furthermore, it comes implemented on its own board with pin headers so that integration needed is to connect it to the Raspberry Pi's I2C bus.

Following the accelerometer, the next device to be chosen is the cellular and GNSS chipset. These chipsets are widely sold around the world. However, due to the differing regulations and local carriers, certain manufacturers are more prevalent in certain regions. The dominating vendor in the current 2021 market is Quectel. They lead in six out of the eight global markets and control 21.7 percent of the IoT market[25].

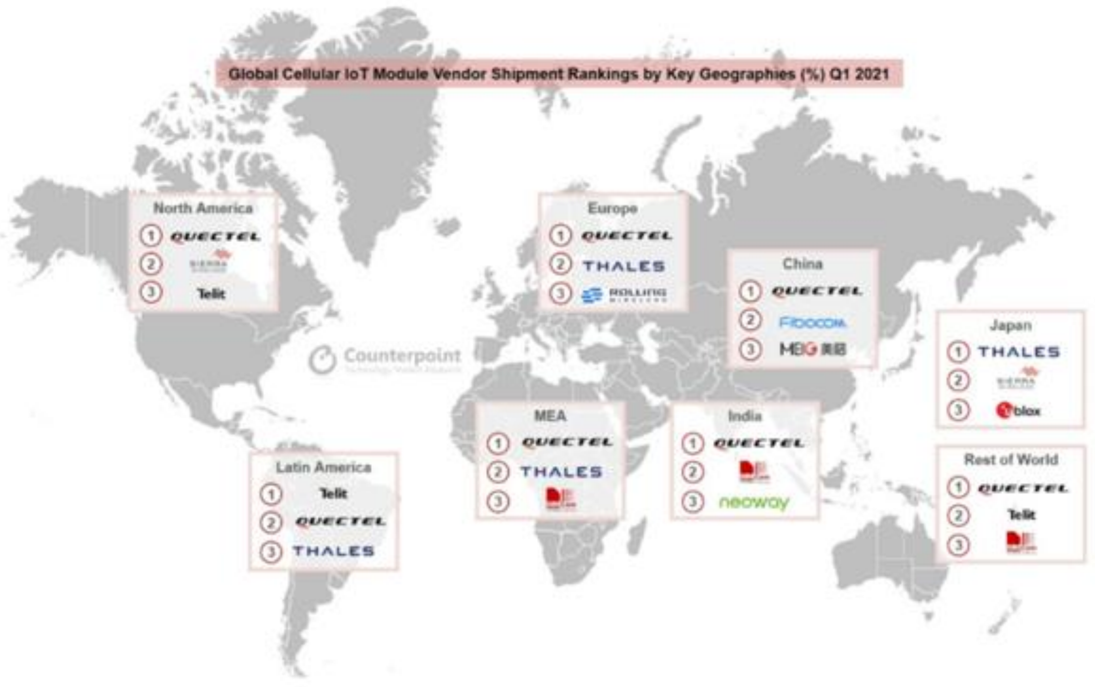


Figure 3.4: Worldwide Cellular IoT Vendors

With the goal of mimicking the hardware implementation of a standard automotive IoT device, the most logical vendor to use would be Quectel as they lead the IoT market. Using one of Quectel's chipsets will allow the IoT device, and the driver safety model, to reflect the current market today. Out of all Quectel's chipsets, the BG96 is their leading IoT solution. Their low cost, CAT-M1 LTE chipset boasts a wide array of features that allows it to be extremely useful in an IoT environment. With features such as integrated differential GNSS, 2G fall back for worldwide applications and low power consumption the BG96 is capable of meeting the telematic requirements of an automotive IoT device[25]. Typically, the chipset is integrated into a custom PCB, but for the sake of this design, the chipset will be used with an evaluation board that extends all communication protocols as well as provides connections for power, SIM cards and debug ports. By using the chipset with an evaluation board, the time needed to integrate the chipset into the automotive IoT device is drastically lowered. Figure 3.5 shows the BG96 integrated onto the evaluation board.

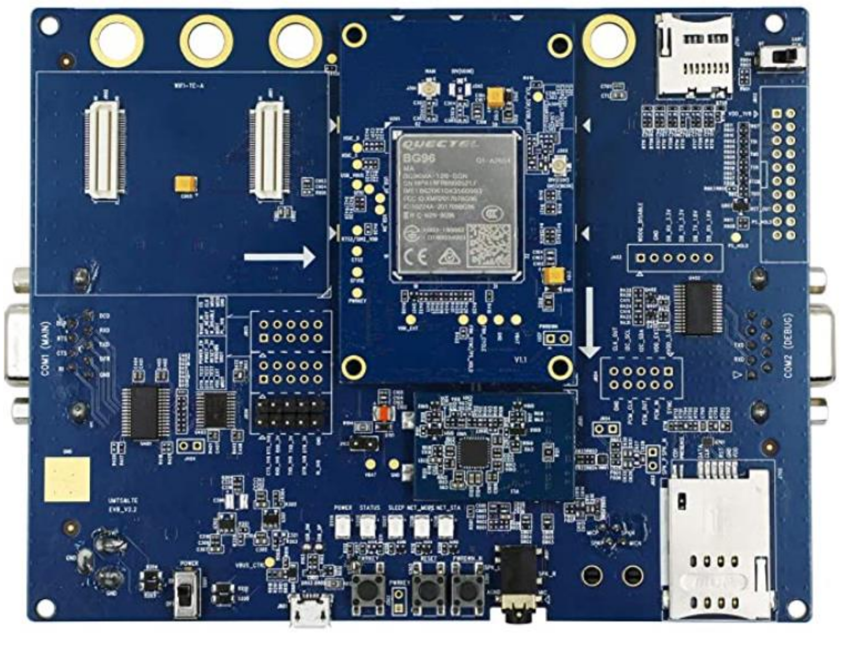


Figure 3.5: BG96 Evaluation Board

To properly use the device within an automotive environment, each component must be securely attached into a portable housing. Figures 3.6 and 3.7 depict the mounting of the device within the upper and lower halves of the enclosure. Figure 3.8 and 3.9 show the final assembly of the device.

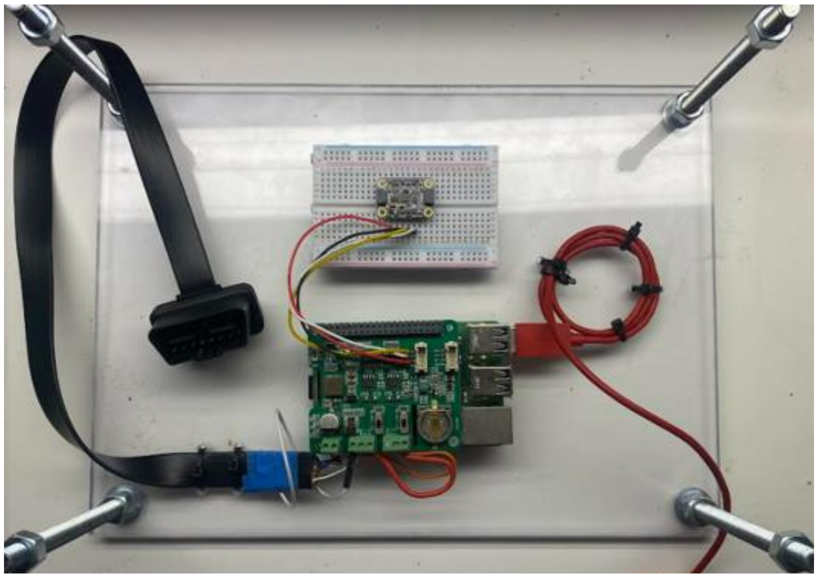


Figure 3.6: Lower Half Device Enclosure

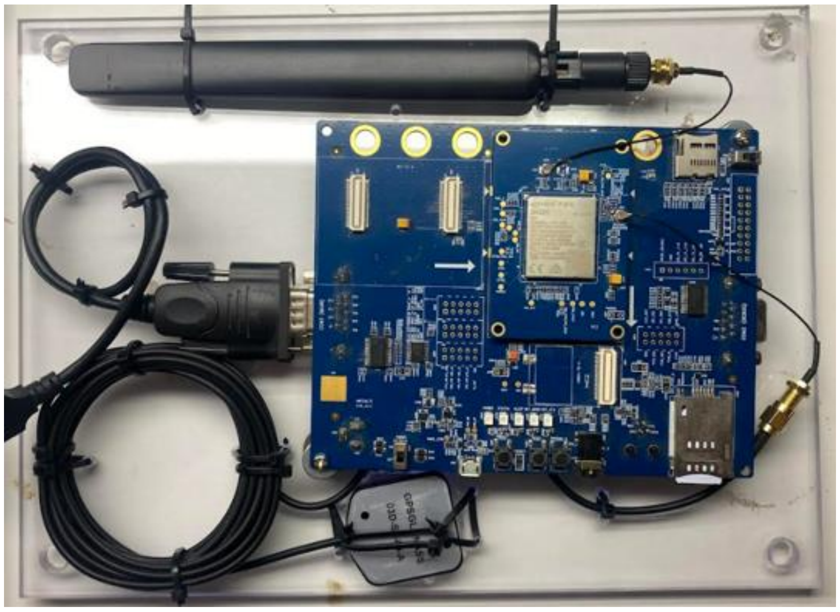


Figure 3.7: Upper Half Device Enclosure

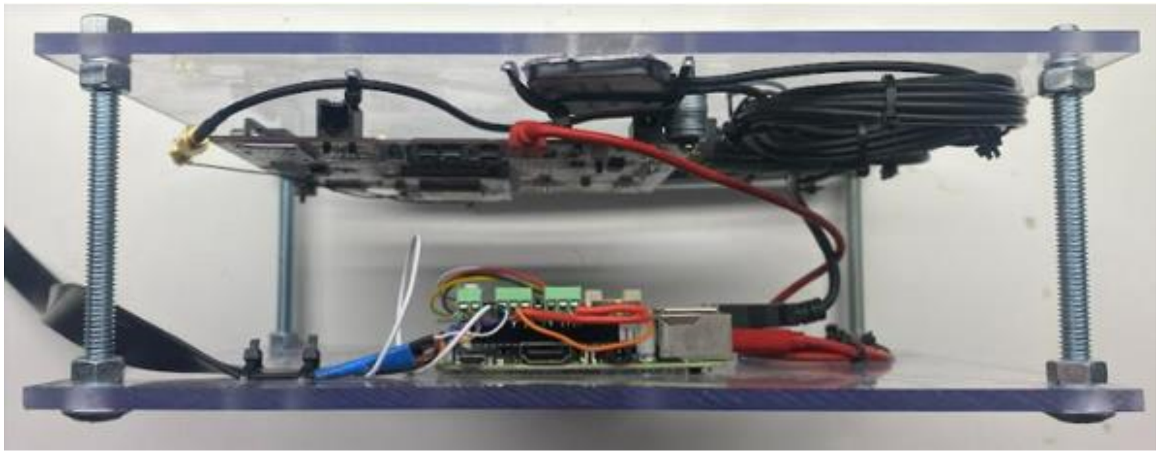


Figure 3.8: Final Device Implementation Side View

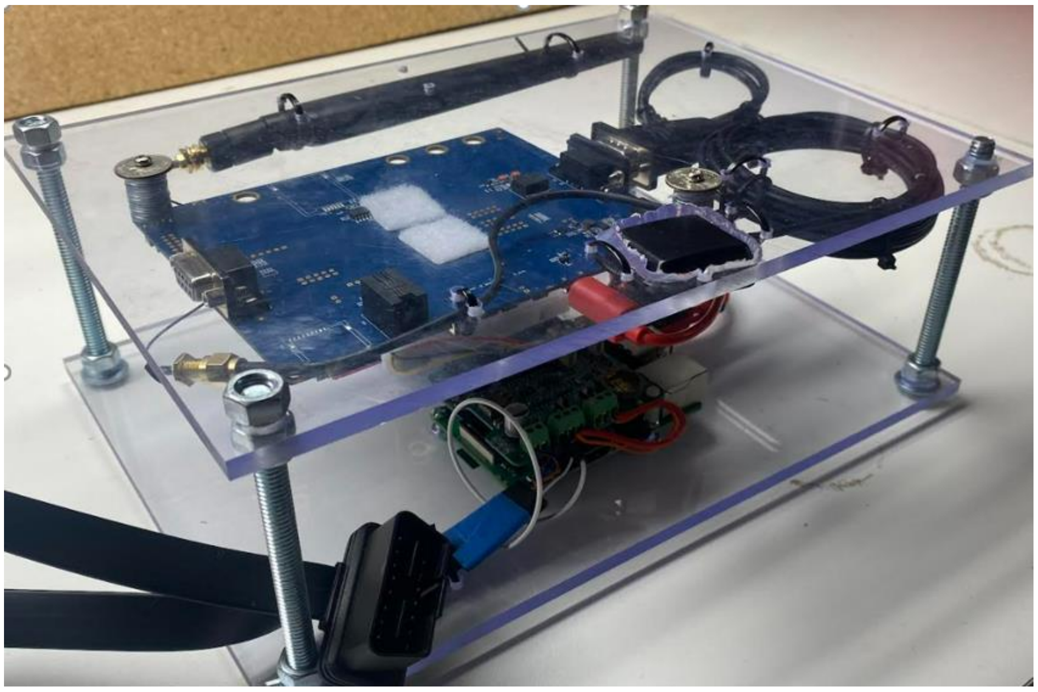


Figure 3.9: Final Device Implementation Top View

Chapter 4: Unsafe Driver Habits

4.1: What is Unsafe Driving?

As the goal of this paper is to develop an automotive tracking device capable of detecting a drivers unsafe habits, it is important to define what kinds of actions are linked to unsafe driving. Unsafe driving is a broad term that is referenced many ways to legally define the operation of motor vehicles on the roadway. Terms such as risky, unsafe, and aggressive driving are used to categorize the actions that result from traffic violations to criminal offences. The National Highway and Transit Safety Association (NHTSA) defines unsafe driving as, "The operation of a motor vehicle in a manner that endangers or is likely to endanger persons or property."[27] Just as how unsafe driving can be defined in many ways, the actions that characterize this behavior cover many forms of driving. According to the National Conference of State Legislatures (NCSL), "Speeding, tailgating, weaving in and out of traffic, running red lights or any combination of these activities generally are considered unsafe driving[28]". Factors such as inexperience, lack of skill, distraction, congested roadways, and lack of concern for others can either directly or indirectly contribute to the cause of unsafe driving. While these are important factors to monitor, the device to be constructed will be focused on detecting the physical maneuvers of the vehicle. This focus ensures a feasible way to evaluate and test the effectiveness of the research.

There are also many ways motorist put themselves and others in danger. For example, speeding has been involved in approximately one third of all motor vehicle fatalities over the last two decades. In 2019, speeding was a contributing factor in 26 percent of all traffic fatalities, resulting in 9,478 deaths[29]. Furthermore, harsh braking or rapid forward acceleration further puts motorists in danger. The challenge of this research is to detect as many unsafe driving practices as possible with the limited number of data sources at hand. Some of the most common unsafe driving habits are listed below:

- Speeding or racing
- Harsh braking or harsh forward acceleration
- Improper or erratic lane changing
- Tailgating
- Excessive cornering
- Operating vehicle in erratic, reckless, or negligent manner
- Failure to yield right of way

A total of 36,096 people died in motor vehicle crashes in 2019[30]. The U.S department of Transportation's most recent estimate of the economic cost of crashes is $242 billion per year[30]. Detecting and reporting unsafe driving habits can minimize the number of yearly crashes and deaths.

4.2: Detecting Unsafe Driving

Depending on the number of sensors available, unsafe driving can be detected in a multitude of ways. Companies approach the topic in a way that fit their use case. For example, automotive companies utilize their existing onboard sensor suite, the ridesharing industry uses data gathered from the driver's cellphone app, and Insurance companies utilize OBD scanners to detect driver habits. In each of these examples, the challenge of detecting driver habits is approached from a different angle with varying levels of complexity. However, the underlying data is all similar. In each case the algorithm to detect unsafe driving operates on the vehicle's speed and acceleration. Cameras and LiDAR can be used to detect other aspects of unsafe driving but may not be applicable in every situation or use case. this paper can be aligned with the industry trends by focusing on parameters related to the vehicles speed and acceleration.

Chapter 5: System Design

An overarching system must be designed to manage the tasks of a normal vehicle tracking device. These tasks include initialization, vehicle states, trip tracking, fault detection, communication, and logging. It is only with all of these tasks running correctly that improvements can be made to fault detection.

5.1: Initialization

The hardware block diagram referenced in Figure 3.1 shows devices such as the cellular/GNSS chipset, CAN shield and accelerometer. All need to be initialized correctly to function and communicate properly. The simplest initialization is the accelerometer, needing only a single Python extension to be interfaced. Normally accelerometers need to be calibrated to determine the acceleration along each axis. This involves rotating the device along each of its axis while the device is running. However, in this application, only the deviation from the baseline output is needed. As such the accelerometer requires little to no initialization.

The CAN shield is a far more complex chipset as it is the gateway to the vehicle bus and the power source for the entire device. It also uses Python extensions to interface with the Raspberry Pi. For the Raspberry Pi to recognize the shield, it must first be initialized as an interface port on the Pi. By opening a terminal on the Pi and opening the config.txt file:

```
$ sudo nano /boot/config.txt
```

adding the following at the end of the file will allow the device to install the correct packages at boot. Once added, reboot the Pi.

```
dtoverlay=seeed-can-fd-hat-v2
```

After the device has rebooted and installed the correct packages, the CAN interfaces may then then be initialized by issuing one or both of the following commands. The interfaces can then be viewed with the "ifconfig" command that lists all interfaces on the device.

```
$ sudo /sbin/ip link set can0 up type can bitrate 500000 restart-ms 1000
$ sudo /sbin/ip link set can1 up type can bitrate 500000 restart-ms 1000
```

To use the interfaces within a program, a few different extensions can be used. Within this thesis "python-can" is the preferred method for reading data off the interface. A bus object can be created in code that allows for the reading of messages off the bus. This bus object is then used throughout the code to allow for the collection of raw CAN data.

The last and most complex component to initialize is the cellular chipset. Since it is responsible for all communication to and from the device, special consideration to its initialization must be given to ensure proper operation. It is also responsible for controlling the GNSS component within the module. All cellular modules communicate via a set command list called AT-Commands. These command sets are comprised of industry standard commands and specific vendor commands. The initial challenge of setting up a cellular module is getting the device connected to the cellular network correctly. The commands used to initialize the cellular module can be seen below in Table 5.1

Table 5.1: Cellular Initialization Commands

Command	Action
ATV1	Set response level
AT+IPR?	Set UART data rate to default
ATI	Display Identification Information
AT+GSN?	Request IMEI
AT+CPIN?	Check for SIM pin number
AT+CIMI	Request IMSI
AT+CSQ	Signal quality report
AT+CREG?	Network registration settings
AT+CGREG?	Network registration status
AT+COPS?	Operator selector
AT+CMGF=1	Set message format
AT+QGPSCFG=”outport”,0	Configure GNSS
AT+QGPS?	Query GPS lock

Using these commands in conjunction with an activated SIM card yields a registered cellular module that is searching for a GPS fix. The GPS, under open sky conditions, normally acquires a fix within 45s. However, if the device is located within an urban canyon or an area with poor line of sight to satellites, then the time to first fix can take up to three minutes. Once the cellular module is properly initialized and registered, the above commands in Table 5.1 are no longer required.

5.2: Data Collection

Due to the different types of sensors utilized within the platform, different collection methods must be employed to collect, parse and process the data. Each source has a unique rate at which information is sampled and ready to read. The collection of data from each source must be defined accurately so that all samples are synchronized for future processing and logging.

5.2.1: Collecting and Parsing GPS Data

As seen in Table 5.1, the GNSS module is activated within the cellular initialization. Once the command is given to turn on the GPS, it automatically begins to search for a fix. The time measurement of time to first fix (TFF) is one of the main factors that distinguishes GPS chipsets from one another. Once a fix is established, the device will begin to internally create National Marine Electronics Association (NMEA) sentences at a 1Hz rate. In total there are 19 sentences. Each containing information on the current state of the module, accuracy of the fix and other location parameters. The NMEA sentence $GGA contains information on the current fix, its accuracy and the speed over ground. The format of this string can be seen below:

Table 5.2: NMEA sentence $GGA format

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*42
```

Where:

GPGGA	Global Positioning System Fix Data
123519	Fix taken at 12:35:19 UTC
4807.038,N	Latitude 48 deg 07.038' N
01131.000,E	Longitude 11 deg 31.000' E
1	Fix quality: 0 = invalid 1 = GPS fix 2 = DGPS fix 6 = estimated
08	Number of satellites being tracked
0.9	Horizontal dilution of position
545.4,M	Altitude, Meters, above mean sea level
46.9,M	Height of geoid (mean sea level)
(empty field)	Time in seconds since last DGPS update
(empty field)	DGPS station ID number
*42	The checksum data, always begins with *

As the cellular module handles all communication to and from the GPS, a series of AT-commands are used to communicate and request data. To request the $GGA sentence, the command “AT+QGPSLOC=0” is sent to the device. The response from the device is the current

$GGA sentence. If the command is given at a rate faster than 1Hz, then the response will remain unchanged until one second Universal Time Coordinated (UTC) has passed. UTC is a time standard that measures time as a 24 hour clock measured from Greenwich, England[31].

Once the NMEA sentence has been received, then the only remaining step is to parse the string for the data and convert the data to useful units. The UTC time is converted to UTC seconds, measured from the last epoch (1 January 1972 00:00:00). The Lat and Lon data are converted from degree minutes (ddmm.mmm) to decimal degrees. And the speed in knots is converted to mi/hr. Once the data is parsed and converted, it is then ready for further usage by the device

5.2.2: Decoding and Collecting CAN Data

As mentioned previously in Chapter 2.1, decoding the CAN bus traffic is often a laborious and time consuming task as the data bytes in question reside upon an unknown PID with an often unknown unit and scaling factor. As a result, companies that develop automotive IoT devices often must pay large licensing fees for access to vehicle databases. As previously mentioned, the vehicle bus outputs large amounts of data pertaining to every electrical and mechanical operation within the vehicle. Therefore, only certain parameters within the context of the application are targeted. Some of the most common parameters used within driver safety devices are PIDs that relate to the status of the vehicle's motion as well as the status of the driver. These parameters are, in most cases, broadcasted over the bus while the vehicle is running. A description of these parameters and what they are used for are listed below.

Table 5.3: Commonly Used Vehicle PIDs

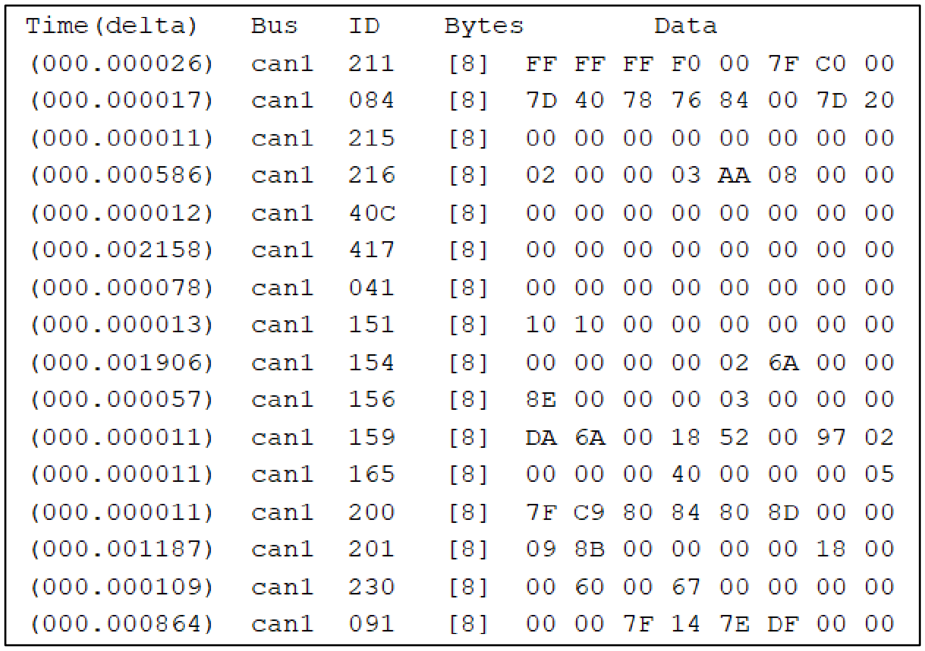
Parameter	Description	Usage	Unit
RPM	Engine speed	Status of vehicle	Revolutions / min
Speed	Vehicle speed	Status of vehicle	km/hr, (mi/hr)
Fuel Level	Fuel remainign in tank	Fuel efficiency	%, (Liters), (Gallons)
Mass Air Flow	Air flow into fuel injectors	Fuel efficiency	kg/s
Seatbelt	Status of seatbelt (driver/passenger)	Status of driver	On / Off
Odometer	Total mileage on vehicle	Status of vehicle	Km, (mi)
Ignition	Ignition status of vehicle	Status of vehcle	On / Off

To align with the goal of developing a driver safety model that is capable of detecting driver faults, the device will target the RPM and speed data from the vehicle. Fortunately, these two parameters are feasible to decode from the CAN bus because their current values are displayed on the dashboard. To decode the parameters by hand, individual logs must be taken at constant values of speed and RPM. Then using the true values recorded for each of the logs and applying the most commonly used scaling factors, the logs may be searched to find the matching hex values. Shown below is a table of some of the most common scaling factors used for RPM and speed.

Table 5.4: RPM and Speed Common Scaling Factors

Parameter	Common Scaling Factors			
RPM	1	0.25	0.1	
Speed	10	1	0.1	0.01

The general procedure to search the CAN bus by hand is to take the true value for RPM and speed and apply each of the scaling factors. The true value for speed has to be converted into both kilometers and miles. Taking each scaled value and converting to hex. Three possible RPM values and eight possible speed values are generated. Searching the logs for each value may return a match that identifies a possible decoding. Due to the large size and many PIDs contained in the logs, shown in Figure 5.1, it is important to use all scaling factors and multiple versions of the logs so that the correct PID, scaling factor and byte location can be confirmed. It is only after these three parameters are found that the data can be used in future applications



```
Time(delta)      Bus    ID     Bytes         Data
(000.000026)    can1   211    [8]   FF FF FF F0 00 7F C0 00
(000.000017)    can1   084    [8]   7D 40 78 76 84 00 7D 20
(000.000011)    can1   215    [8]   00 00 00 00 00 00 00 00
(000.000586)    can1   216    [8]   02 00 00 03 AA 08 00 00
(000.000012)    can1   40C    [8]   00 00 00 00 00 00 00 00
(000.002158)    can1   417    [8]   00 00 00 00 00 00 00 00
(000.000078)    can1   041    [8]   00 00 00 00 00 00 00 00
(000.000013)    can1   151    [8]   10 10 00 00 00 00 00 00
(000.001906)    can1   154    [8]   00 00 00 00 02 6A 00 00
(000.000057)    can1   156    [8]   8E 00 00 00 03 00 00 00
(000.000011)    can1   159    [8]   DA 6A 00 18 52 00 97 02
(000.000011)    can1   165    [8]   00 00 00 40 00 00 00 05
(000.000011)    can1   200    [8]   7F C9 80 84 80 8D 00 00
(000.001187)    can1   201    [8]   09 8B 00 00 00 00 18 00
(000.000109)    can1   230    [8]   00 60 00 67 00 00 00 00
(000.000864)    can1   091    [8]   00 00 7F 14 7E DF 00 00
```

Figure 5.1: CAN Log File Format

Reverse engineering the CAN bus by hand is a time consuming task that can take many trips to and from the vehicle. Most of this process can be automated in software, but the limiting factor in reverse engineering the CAN bus is the lack of true values. To eliminate the need for the user to decode the bus by hand, an iterative system is needed for the real time processing and decoding of the CAN bus.

The first parameter to decode is speed. As mentioned before, the challenge with decoding the bus is the collection of the true value of the parameter. Fortunately, the GPS module gives an estimated value of speed over ground. This measurement of speed is not always accurate while under acceleration as seen in Figure 4.2, but it can give a reasonable estimate for determining the CAN parameter for speed. Figure 5.2 shows a plot of GPS speed and CAN speed. Increased error can be seen in areas of acceleration. This error can be further exasperated when the GPS receiver has poor line of sight or is within an urban canyon. Further proving the usage of GPS only as a fallback source for speed.

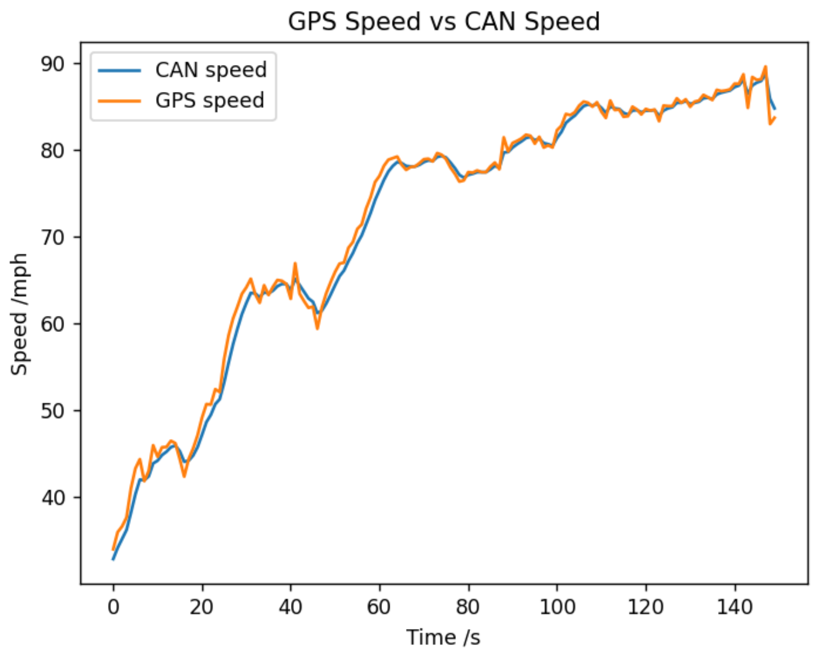


Figure 5.2: CAN Speed vs GPS Speed

While the vehicle is operating at a low acceleration, the device reads these GPS speeds and performs the iterative search on the incoming CAN data. Repeatedly polling the GPS for speed and searching the CAN bus for the defining parameters of PID, scaling factor and byte number. The device is then able to list out the most likely possibilities for the speed data. Because the GPS speed reading is constantly changing, the possibility of a false positive is very low due to the changing GPS speed. But due to the error of the reading, the algorithm searches for a range of speed values. The true PID for speed will maintain a constant scaling factor and bit location while also closely following the GPS reading. Out of the possible PIDs found, the average error from the GPS speed is used to select the result. The flowchart for this algorithm can be seen below in Figure 5.3.

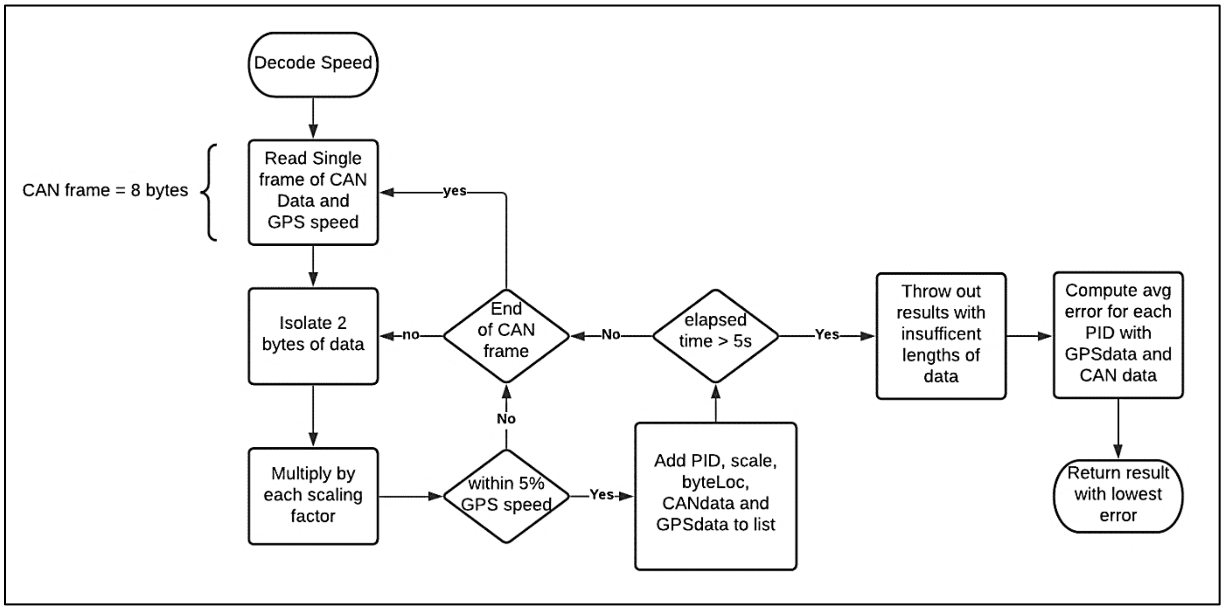
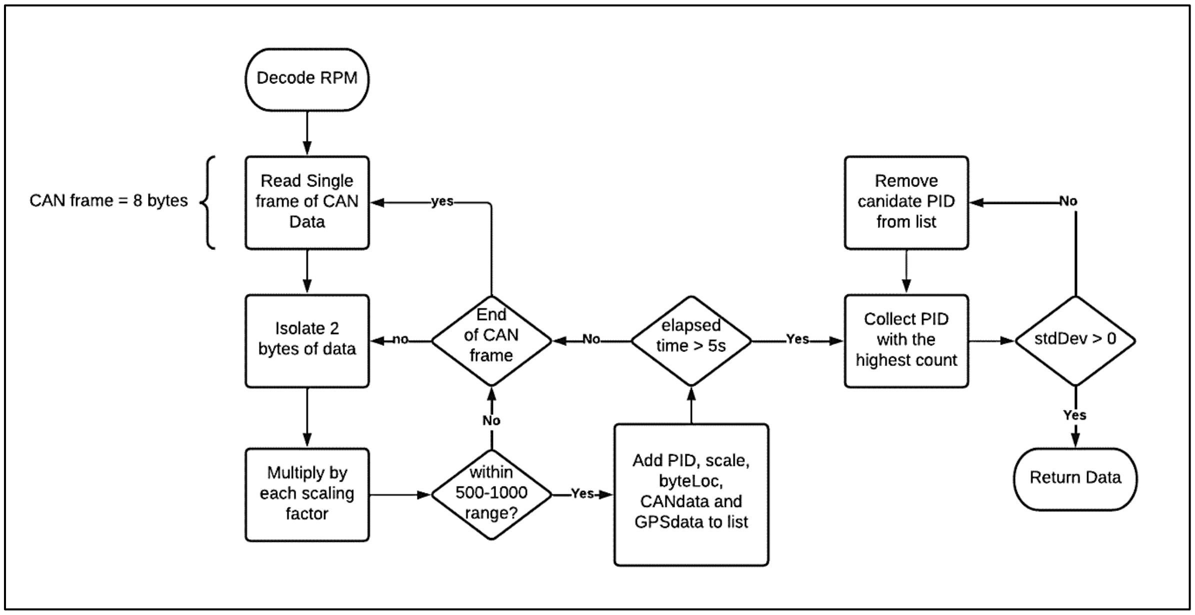

Decode Speed
CAN frame = 8 bytes
Read Single frame of CAN Data and GPS speed
yes
Isolate 2 bytes of data
no
End of CAN frame
No
elapsed time > 5s
Yes
Throw out results with insufficent lengths of data
Compute avg error for each PID with GPSdata and CAN data
No
Multiply by each scaling factor
within 5% GPS speed
Yes
Add PID, scale, byteLoc, CANdata and GPSdata to list
Return result with lowest error

Figure 5.3: CAN Speed Decode Flowchart

Due to the large variation in vehicles, this method is not absolute. In a situation where speed decoding is not successful or the average error is large, the GPS speed can be used as a fallback. As seen in Figure 4.2 the GPS speed is not entirely accurate in areas of high acceleration or when location accuracy is low but is sufficient for use as a fallback. If the GPS speed is used as the data source, then the driver safety model must adapt its limits to account for the possible error induced from the GPS speed.

The next parameter, RPM, can be decoded when the vehicle is turned on and at rest. Unlike with the speed parameter there is no sensor that can provide the vehicle's true idle RPM. This results in a need to search for a range of probable values. Fortunately, most cars idle between 600 and 1000 RPM. While this does not give an exact value, the range is small enough to exclude a large portion of the data on the CAN bus. Without having a true value, the algorithm must search each byte pair and determine if it is a possible match for the RPM reading, resulting in a much larger array of possible decoding results. The logic of scanning the vehicle bus is similar to the speed decoding where if the "True" value is set to 800rpm, the data must be +/- 20 percent of that target. If a pair of bytes is found to match that criteria then its value, PID, location and scaling factor is added to a list of matches. Figure 5.4 shows the flowchart for RPM decoding.



Decode RPM
CAN frame = 8 bytes
Read Single frame of CAN Data
yes
Isolate 2 bytes of data
no
End of CAN frame
No
elapsed time > 5s
Yes
Multiply by each scaling factor
within 500-1000 range?
Yes
Add PID, scale, byteLoc, CANdata and GPSdata to list
Remove canidate PID from list
No
Collect PID with the highest count
stdDev > 0
Yes
Return Data

Figure 5.4: RPM Decode Flowchart

After searching the bus for five seconds, the list is evaluated by the number of matched values for each PID and the one with the most matches is identified as a candidate for the result. With the vast amount of data on the bus, it is highly probable that the candidate is a false positive. Most commonly the false positives show up as constant values that lie within the range searched for. To detect and throw out this false positive, the standard deviation of the PID's data Is calculated. If the result is near zero then a false positive has been found and the algorithm then goes back to the list of matches and selects the next most common PID and performs the same evaluation. This method either results in decoded RPM value, or a failure in decoding. Since the RPM value is a mandatory parameter for the device state machine, the device can poll the user over SMS to supply the true idle RPM. Using the method to decode speed laid out above, the device can try again to decode RPM with the near exact value of the RPM.

5.2.3: Circular Buffers

One challenge that comes with using data sources with varying refresh rates is how to synchronize and store data to be processed together. Because the GPS data updates at the slowest rate, the rest of the collections occur at a 1hz rate to ensure that each collection cycle contains the most recent values from each sensor. To ensure that each sample of data is synchronized and stored in a way to be processed later, circular buffers are used to keep a running history of the device’s status. A circular buffer, in this use case, is implemented as an initialized array of length 30. The pointer to the current index is incremented every time a collection cycle has completed (once per second). Once reaching the end of the buffer, the data can then be processed and logged. This allows the device to retain 30 seconds of historical data to process. The index is then reset back to the beginning of the buffer and begins to overwrite the data, beginning the cycle again. Most processes within the device are triggered once the index reaches the end of the buffer, meaning that the length of the buffer defines the time between fault detection, communication and logging.

5.3: Vehicle States and Trip Tracking

As stated before, the device is responsible for a multitude of tasks. Each task must be tailored to how the vehicle is currently being operated. While the device is moving, all functional blocks need to run. But when stopped, only occasional checks need to occur. To control the operation of the device and to decide what functional blocks to run, a set of vehicle states are defined. The states are Move, Idle and Stopped and they define the operations done on the circular buffers. Figure 5.5 shows the vehicle state machine and the conditions that define when the states transition.

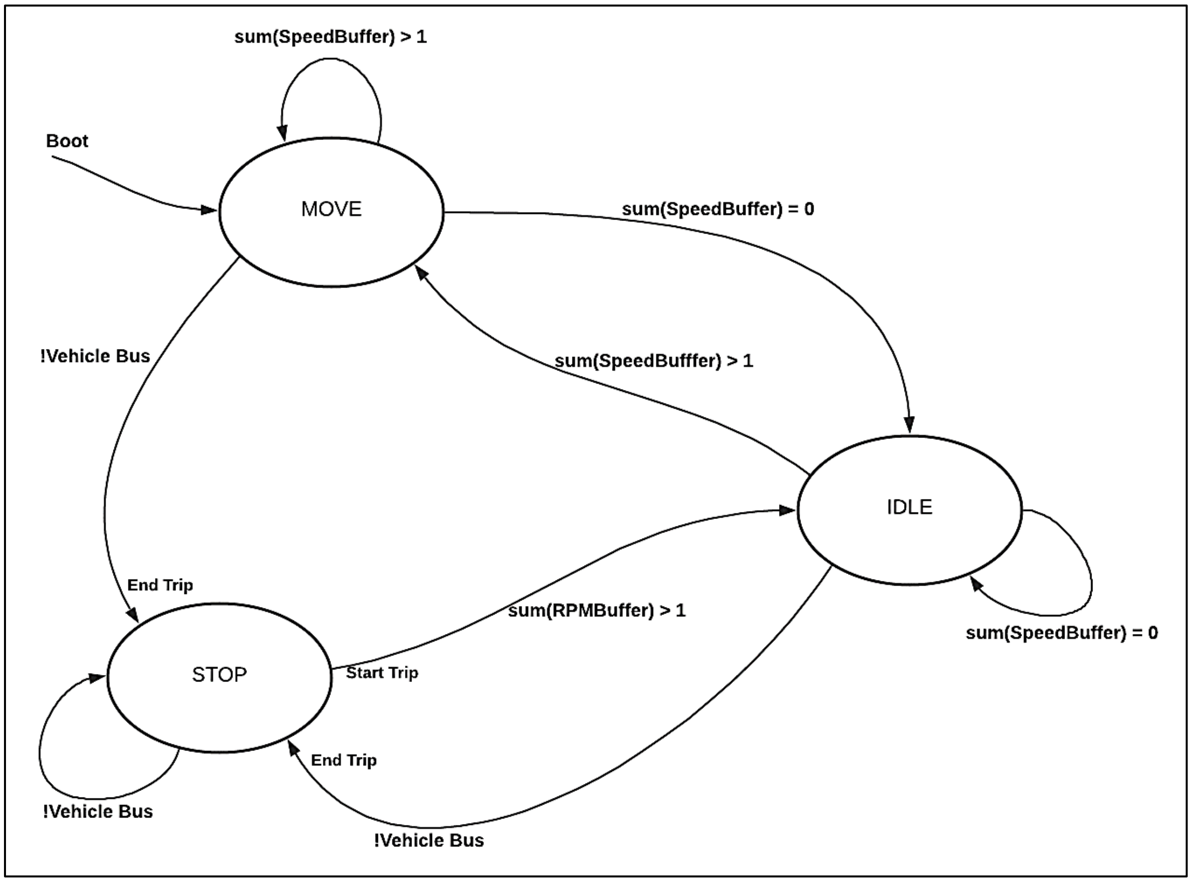


Figure 5.5: Vehicle State Machine

The device is initialized to be in the Move state and the circular buffer containing speed data is populated with ones. This allows the device to boot, and if no motion has occurred it will transition to the Idle state once the speed buffer has been populated with zeros. As mentioned before, the length of the circular buffers defines state transition time. Once the device enters the Idle state, it will remain here until a speed is collected or the when the vehicle is turned off and the bus shuts down. If the latter occurs, then the device enters the stop state and remains there until the vehicle is turned on and an RPM value is read, ensuring that the vehicle has been turned on. The current trip log is ended and closed when the device enters the Stopped state. Once the device leaves the Stopped state, a new trip log is created. Figure 5.6 shows an example state transition from one trip to another.

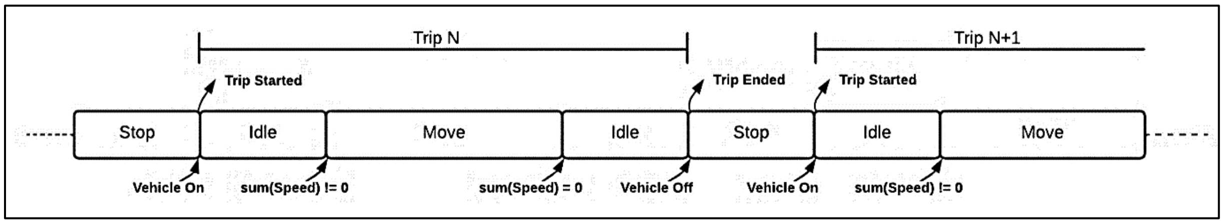


Figure 5.6: Trip State Progression

5.4: System State Machine

As previously defined, the vehicle states determine what functional blocks are executed at any given time. These functional blocks are connected in a master state machine that begins after initialization and can bus decoding. The state machine begins every cycle by evaluating the vehicle state machine to get the current vehicle state. Then once acquiring the vehicle state, the following occurs based on the state:

STOPPED – RPM is collected. Then the cellular module is polled to check for unread commands received over SMS. The device then sleeps for 10sec before beginning again.

IDLE – RPM, speed, telemetry and acceleration are all collected and added to the circular buffers. The cellular module is then checked for SMS. And finally, the data is logged before starting again

MOVE – All data is collected and added to buffers, If the buffers are not fully populated, then the device follows the IDLE path. If they are populated, then the data is checked for faults and if one is found then it is sent and the buffer data is logged before beginning again.

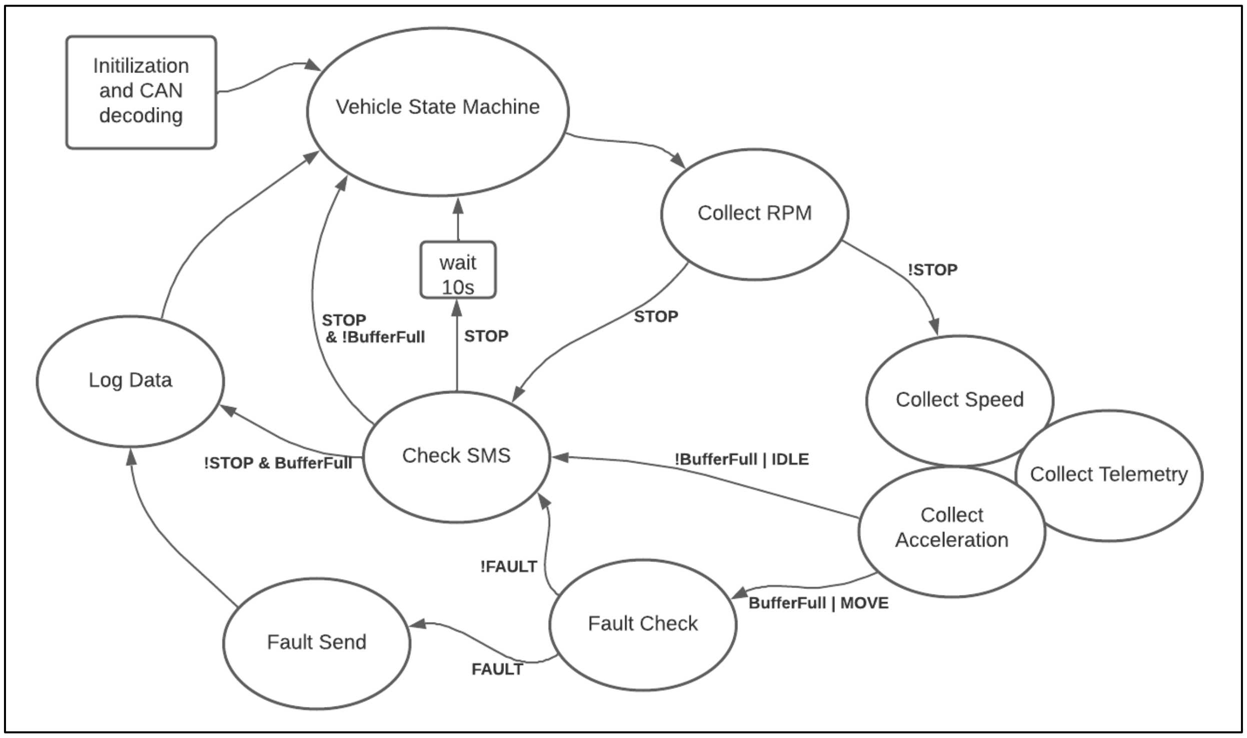


Figure 5.7: Main System State Machine

5.5: Fault Detection

As this device is designed to be a driver safety tool, the fault detection module is one of the most important functional blocks in the system. Each data collection mechanism and communication module are designed to supply this functional block with data and manage its output. Following the main state machine shown in Figure 5.7 above. The fault check block is ran only when the device is in the “Move” state. This ensures that the data coming into the fault check algorithm is data pertaining to the movement of the vehicle.

The fault detection is handled in two parts. The first being a periodic check that occurs only when the circular loops are full, and a second that continually checks the most recent data. The creation of two algorithms allows for the concatenation of sequential driver faults relating to RPM and speed, and the instantaneous faults pertaining to harsh acceleration. The faults in question are as follows:

Table 5.5: Fault Types and Limits

Source	Type	Limit	Method
Speed	Speeding	Speed > 80mph	Periodic
RPM	Over Revving	RPM > 5000rpm	Periodic
Acceleration	Harsh acceleration	yAccel > 3	Instantaneous
Acceleration	Harsh brake	yAccel < -3	Instantaneous
Acceleration	Harsh right corner	xAccel < -3	Instantaneous
Acceleration	Harsh left corner	xAccel > 3	Instantaneous

The faults above encapsulate most driver infractions that occur on the road. Speeding and over revving most normally occur while the vehicle is on the highway and are linked to aggressive driving through their direct measurements of speed and RPM. The acceleration faults most normally occur when the driver is aggressively driving on surface streets and is tailgating, speeding, or harshly coming to a stop. These driver faults result in higher acceleration readings coming from the device. Detecting these faults comes from the direct analysis of the sensor data, but the data that needs to be included in a fault report must also give further information to classify when, where and how the driver fault was created. The device, once identifying the beginning of a fault must begin to populate the data needed to define the fault. For speeding and RPM faults, the start and end time, location data and maximum value reached must be recorded for future use. For the acceleration faults, since they normally only occur for a few moments, need the time and location of occurrence and the max value reached during the fault. The next two sections define how each fault is detected and the fault data is populated.

5.5.1: Periodic Faults

As mentioned, the periodic faults are needed due to the nature of speeding and over revving. These faults often last longer than the duration of the circular buffers, meaning that faults are needed to be left pending and open, waiting for the next batch of data to complete. Both the speed and RPM faults check for the presence of sensor data above the limit defined in Table 5.5. The fault detection algorithm for periodic faults are ran for each data source. For any given batch of data it tries to start a fault by looking for at least five seconds of data past the limit. If a fault has already begun, the algorithm looks for five seconds of data below the limit to end the fault. The flowchart for this algorithm can be seen below in Figure 5.8

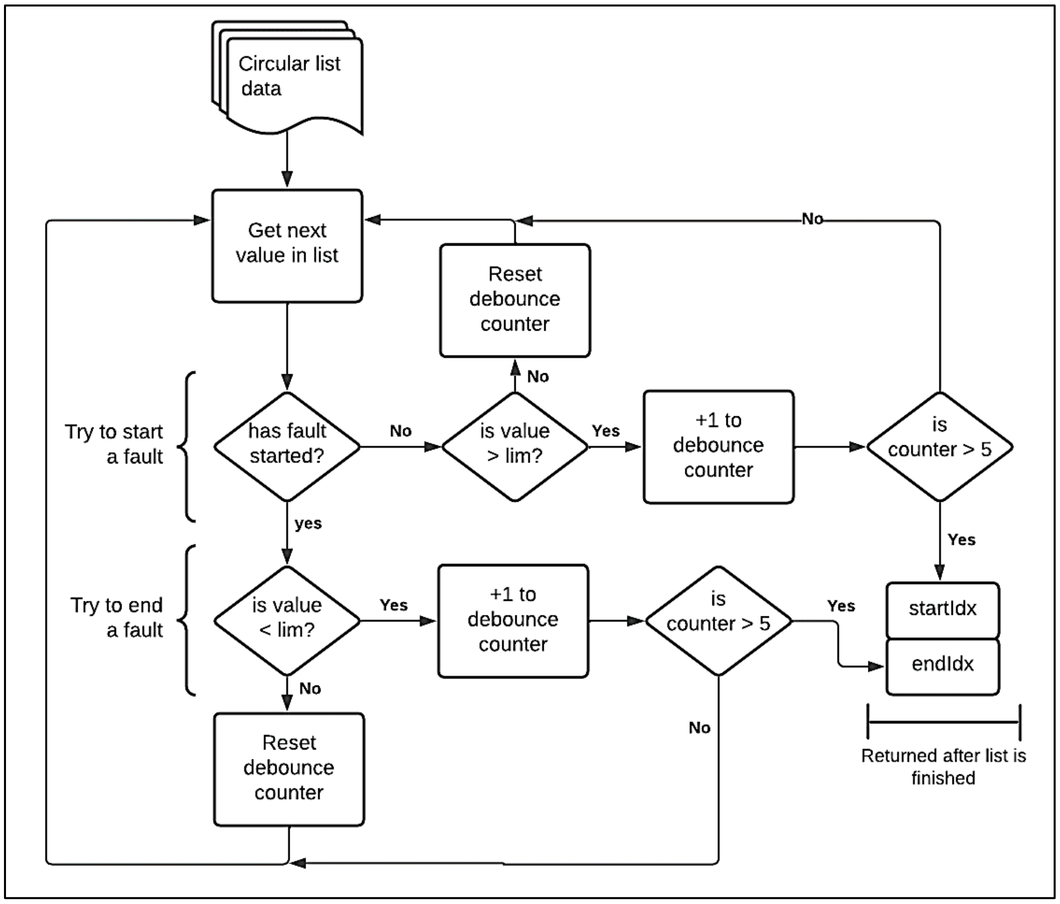


Figure 5.8: Periodic Fault Detection Algorithm

The fault detection algorithm operates on the entire length of the circular buffer trying to populate the indices of the start and/or end of the fault. If a fault is started and ended within the length of the circular buffer, then both indices will be populated and returned. However, in most cases the buffer ends without finishing the fault. If this occurs then the start Index is saved, the location and time data is collected then the algorithm waits until the next batch of data. Once the algorithm identifies the end index of the fault, the location and time data are collected and added to the array of fault data. Once a fault is defined by both a start and end index the fault data is ready to be transmitted to the user. A bit is toggled to indicate that a certain fault is ready to send.

In the case of a fault that spans multiple evaluations, the continuity of an ongoing fault must be preserved. When a fault is left uncompleted the check SMS state is blocked because communication functional block has the ability to change the parameters used to search for a fault. The device must ensure that if a fault is still being evaluated and that the limits that are used to evaluate the end of the fault are the same as the ones used to start the fault. The process of searching for indices, collecting fault data and or blocking the SMS module is shown in Figure 5.9.

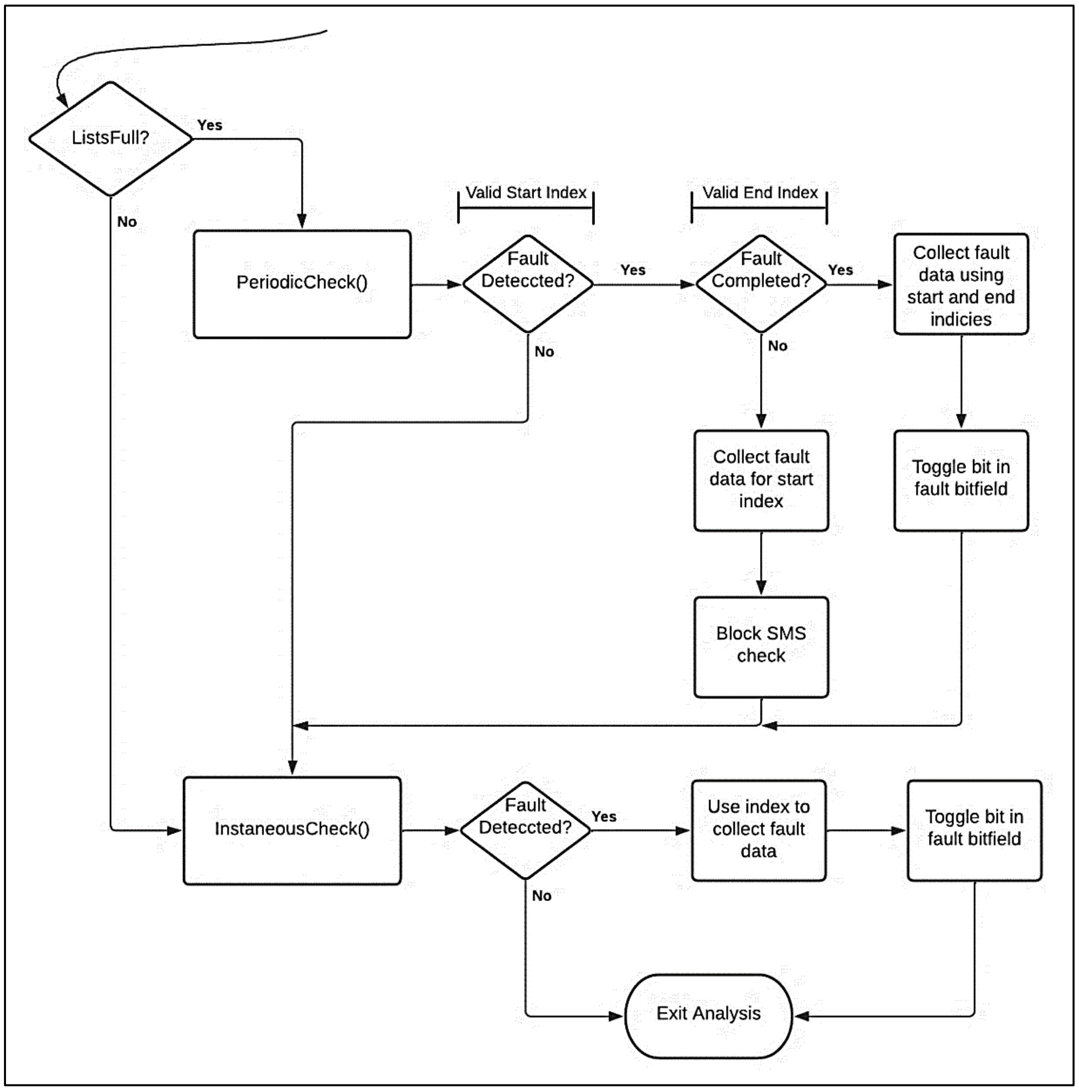


Figure 5.9: Fault Detection Post Processing Flowchart

5.5.2: Instantaneous Faults

The fault profile of acceleration faults greatly differ from the profile of periodic faults. Due to this a separate algorithm is needed to detect and capture the acceleration fault data. An acceleration fault occurs as a spike in the acceleration along one of the two axes. This results in four different faults: harsh acceleration, harsh brake, harsh corner left and harsh corner right. Since the device's accelerometer is parallel to the plane of the road and the positive Y axis is pointed forward, the faults exist on separate Cartesian axis. Figure 5.10 illustrates the accelerometer readings that classify each of the acceleration based driver faults.

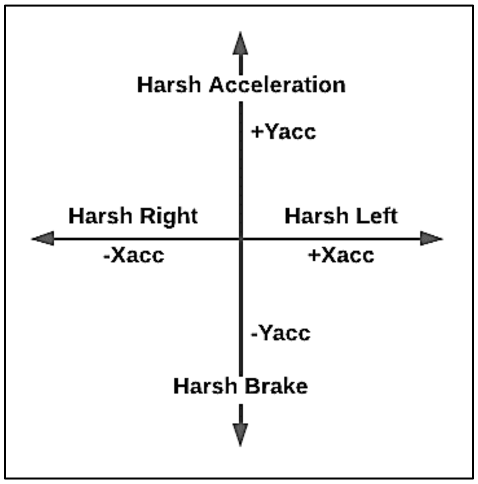


Figure 5.10: Accelerometer Fault Directions

The instantaneous driver fault algorithm simply checks the stream of accelerometer readings on the circular buffer. Starting at t = 0 and ending at t = -N. Where the number of previous values to check is defined as the accelerometer window length. The values within this window are averaged and checked against the limits for each fault. Limits defined in Table 5.5. If a fault is found, then the location and time of the fault is collected, and the acceleration average is recorded. The fault check along that axis is then skipped for the length of the window to ensure that multiple faults are not created from one occurrence.

Through experimentation, it was found that the limits for the instantaneous faults need to be lower than the peak value. This is due to in part because of the averaging of the window and because the acceleration is sampled at a 1hz rate. The peak of the fault can easily occur between two samples, shown in Figure 5.11. Due to this, a very rapid acceleration fault can register as a much smaller fault due to the loss of data between samples.

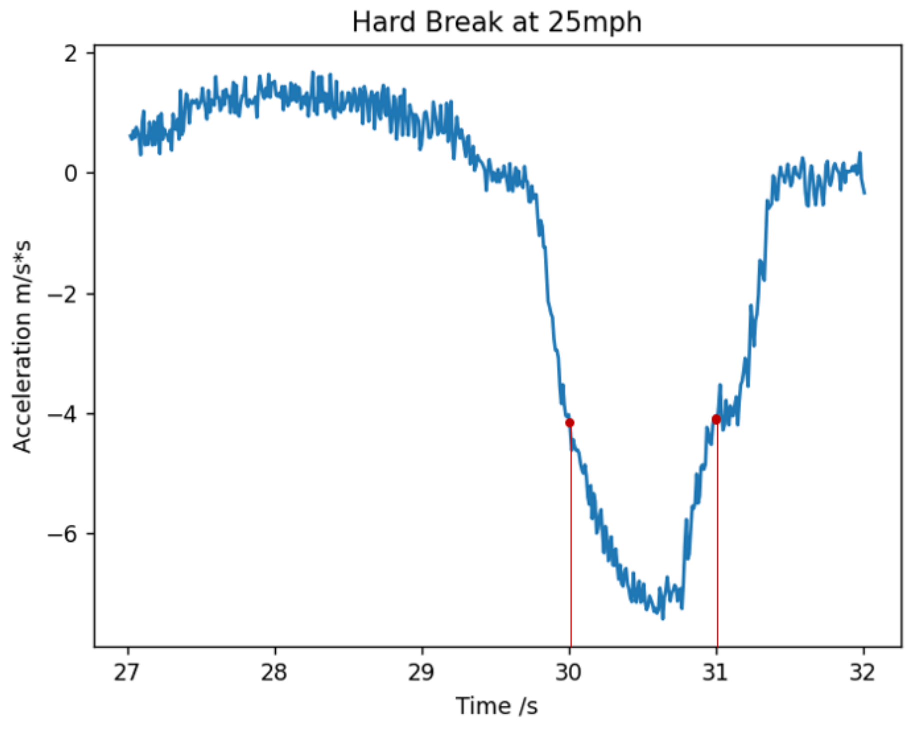


Figure 5.11: Data Loss in Acceleration Sampling

In the example above, the accelerometer is run at full speed while the vehicle is traveling at 25mph. The brakes are then quickly applied to cause a harsh brake. If the accelerometer window length is set to 2 and calculation starts at t=31 then the sampled average is -4.39m/s^2. However, when we take the true average acceleration between the two samples then the average is -6.08m/s^2, resulting in a 32 percent error. Due to this error, the limits to determine an instantaneous fault must be lower than the peak value so that the data lost is accounted for between samples.

5.6: Communication

In most automotive IoT devices on the market, the cellular module handles all communication to and from the device. The method of communication varies from device to device with most modules sending data over the cellular network. Depending on the type of data sent, the protocol used can vary. The most common protocols to use are MQTT and HTTP, where both use TCP/IP communication format. Whatever the protocol used, the data is sent from the device to a central server that hosts the backend data processing and the user interface that allows the user to visualize the data. In order to keep the scope of this thesis bounded, a backend server is not used. Instead, the device communicates over Short Message Service (SMS) to a user's phone where the data can be displayed. SMS is a Transmission Control Protocol (TCP) and is commonly used to send commands to devices. The advantage of using SMS to send commands is that if the device is in an area of low or no coverage, the command will sit on the network for a period of time before it expires. Depending on the provider, this expiration time can range from 24hrs to seven days and gives the device a higher probability of receiving the command.

A further advantage of using SMS is that it requires no backend server. The cellular module and network provider handle nearly all aspects of receiving and transmitting data, allowing for a simple and effective method of communication to and from the device. It is due to these advantages that SMS based reporting is used within this thesis. Shown in Figure 5.7, the SMS module is executed in almost every path of the main system state machine with the exception of a pending fault. This is because the communication module handles all data transfers over the cellular network.

5.6.1: SMS Data Reporting

All reports sent from the device utilize the same AT-command sequence to generate and send data to the user. The set of AT-commands used are shown below in Table 5.6

Table 5.6: SMS AT-commands

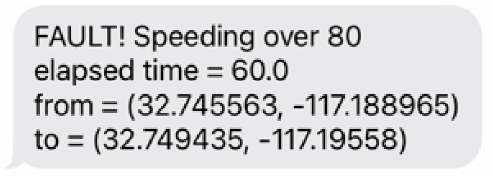
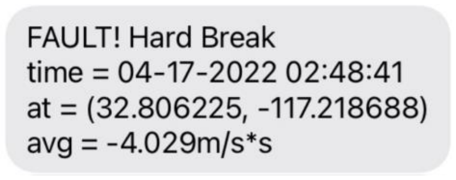
Command	Action
ATE1	Echo On
AT+CMGF=1	Set message format
AT+CSMP=17,167,0,0	Set text mode param.
AT+CSCS=”GSM”	Define character set
AT+CMGS=”[phone#]”	Set recipient
0x1A	Send message

The commands for sending messages tell the cellular module how, where and what to send. The AT+CMGF command sets the device into numeric mode where it accepts numbers as command parameters. The AT+CSMP sets the network transmission parameters and upon sending the AT+CMGS message the device will begin to build a SMS message. The body of the message is generated by collecting data over UART after the AT+CMGS command is given. The collection of data will stop and the message will send when the 0x1A code is sent. It is important to note that the character limit for an SMS is 160 characters. If the length exceeds this limit, then the message will be split into separate segments and sent individually.

The body of these messages falls within three categories. Fault reports, state reports and periodic location reports. The largest of all are the faut reports that contain data on the vehicles motion while a fault was generated. These fault reports can be defined in two parts. Faults generated from a periodic check (RPM/speed) or faults generated from the instantaneous check (acceleration). As with how the search algorithms work, the periodic faults contain data on the start and end of the fault while the instantaneous faults contain data only at the creation of the fault. The information needed to classify each fault may contain the latitude, longitude, time and max value of the fault. Below in Table 5.7 the data within each fault and an example SMS message is shown.

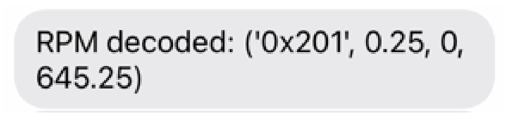
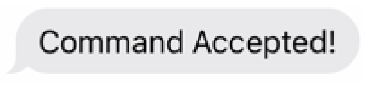
Table 5.7: Data Used Within Fault Reporting

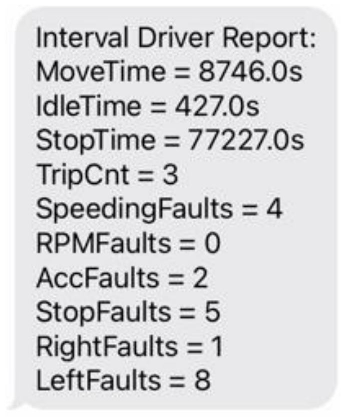
Periodic Fault (RPM/Speed)	Instantaneous Fault (Acceleration)
• Start Latitude, Longitude, Time • End Latitude, Longitude, Time	• Start Latitude, Longitude, Time • Type of Acceleration • Max acceleration reached

 FAULT! Speeding over 80
elapsed time = 60.0
from = (32.745563, -117.188965)
to = (32.749435, -117.19558)
	 FAULT! Hard Break
	time = 04-17-2022 02:48:41
	at = (32.806225, -117.218688)
	avg = -4.029m/s*s

The next type of SMS sent to the user are state reports. These reports are used to notify the user on key milestones that the device has reached. These reports consist of the success or failure of CAN decoding, acceptance of a user command and daily driver reports. The first of these state reports are sent once the device has attempted to decode the CAN bus. If RPM or speed have been decoded then the decoding of said parameter is reported to the user. If speed is not decoded then the device reports that it is falling back to GPS speed. The device will also notify the user that a command has been successfully received. Finally, every day the device will send a summary report that lists the time spent in each state as well as the number of faults detected. Below in Table 5.8 the data within each state report and an example SMS message of each is shown.

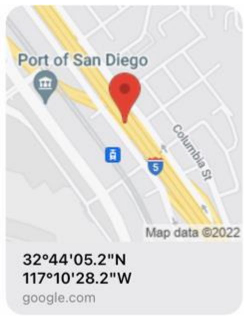
Table 5.8: Data Used Within State Reports

CAN Decoding	Command Acceptance
• Parameter decoded • PID, scaling factor, Byte location, current value	• Text notifying user that a command has been accepted
 RPM decoded: ('0x201', 0.25, 0, 645.25)	 Command Accepted!
Daily Driver Report	
• Number of trips within 24hr period • Count of each fault type sent • Total move, idle, sleep time	

 Interval Driver Report:
MoveTime = 8746.0s
IdleTime = 427.0s
StopTime = 77227.0s
TripCnt = 3
SpeedingFaults = 4
RPMFaults = 0
AccFaults = 2
StopFaults = 5
RightFaults = 1
LeftFaults = 8	

The final type of SMS sent to the user is a periodic location report. These reports are used to keep the user or users updated on the current location of the device. The period of time between these reports are configurable and the method to do so is covered in the next chapter. By default the periodic location report is sent every four hours while stopped and every ten minutes while moving. Unlike the other reports, these are plottable by using a Google Maps link. This is possible because only the current latitude and longitude are sent to the user. The report is shown in Table 5.9.

Table 5.9: Data Used Within Periodic Location Reports

Periodic Location Report
• Current Latitude and Longitude sent as google maps link
 32°44'05.2"N 117°10'28.2"W google.com

5.6.2: SMS Commands

As mentioned, the device is user configurable. This is done by reading SMS messages coming from the user to the device and parsing the message. The reception of commands is far more simple as the cellular module receives and stores messages into memory automatically. The commands needed to read messages off of the cellular module are shown in Table 5.10

Table 5.10: Receive SMS AT-commands

Receiving	
AT+CMGR=0	Receive all unread messages
AT+CMGD=1,4	Delete all messages

The AT+CMGR command is used to define what kind of messages are read from memory. When a message is received it is marked as "UNREAD" then once the message is read the module marks it as "READ". Therefore, when sending the AT+CMGR=0 command the cellular module transmits all unread messages over UART to the device for processing. After a message has been read, the AT+CMGD command is given to erase all messages from memory, allowing the module to only hold unread messages.

In order to correctly parse the data from the received message, the command must be sent with a start of command sequence (CMD) and end of command sequence (CMD). This allows the device to easily parse the contents of the command as well as filter out spam messages that may be sent to the device. The format of the message can be seen below in Table 5.11. To further distinguish what command is sent. The commands must be sent with a comma separating the command type and the data value or values associated with said command. Table 5.11 shows all commands that can be sent to the device.

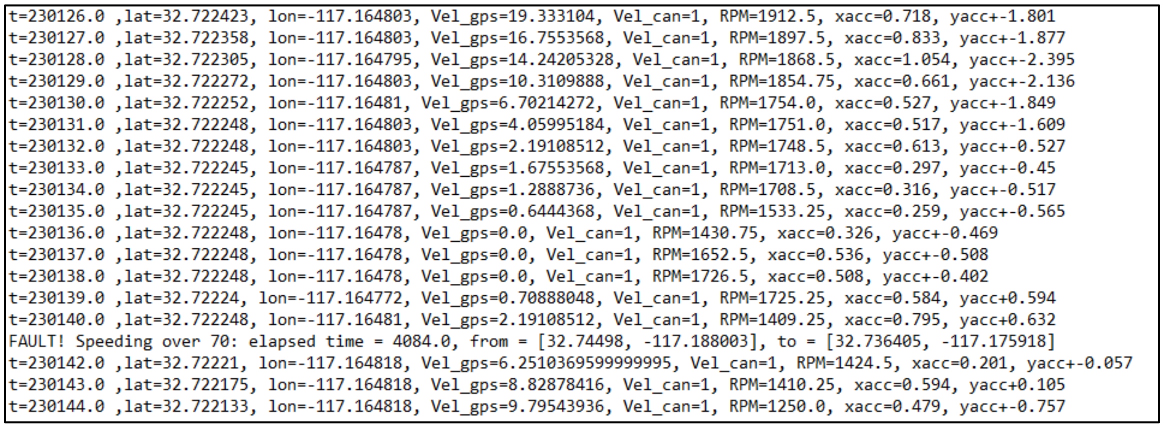
Table 5.11: Table of Supported commands

CMD <comma separated command>> CMD			
Command Type	Command Format	Data	Effect
RPM Limit	limit,rpm,data[0]	RPM	Set RPM limit
Speed Limit	limit,speed,data[0]	mi/hr	Set speed limit
Brake Limit	limit,stop,data[0]	- m/s^2	Set brake limit
Acceleration Limit	limit,go,data[0]	m/s^2	Set acceleration limit
Left Corner Limit	limit,left,data[0]	m/s^2	Set left corner limit
Right Corner Limit	limit,right,data[0]	- m/s^2	Set right corner limit
Window Length	window,data[0]	# samples	Set circular buffer length
Loc Report Period	hb,data[0]	seconds	Set loc period
Driver Report Period	driver,data[0]	seconds	Set driver rpt period
Get current location	loc	N/A	Google maps api

The table above shows all supported commands and the format in which to send them. As appropriate with a driver safety device, most commands pertain to the generation of driver faults and their limits. This is the reason why the SMS module is blocked when a periodic fault is left open. If the limit for the fault is changed mid fault, then the fault may end prematurely, or it may continue forever. Therefore, the SMS module is only checked when no faults are pending to be completed, allowing the device to read and implement the command in one pass of the SMS functional block.

5.7: Logging and Simulation

Another key functional block in the system is the one that pertains to the logging of all data collected by the device. As the device only reports periodic location reports and faults to the user, most of the data collected goes unused. Used normally in the test environment, the logging of data is imperative for the development and debugging of the device. The process of logging data begins at boot where the device creates an empty txt document with the current date and time. Once CAN decoding has finished, the found parameters are saved and logged. The device then continues normal operation until the circular buffers are full. Once full, the device dumps the contents of the buffers to the log file. This is shown in Figure 5.7 of the system state machine. Once a fault is sent, the fault data is logged. After a trip has ended and the device enters the Stopped state and the log file is closed and saved. The process begins again once the device moves and starts a new trip. This allows each trip to be encapsulated into individual log files. The logging is done in an effort to ease the debugging and development of the device, allowing the developer to review the operation of the device. Below in Figure 5.12 a section of a log file is shown.



```
t=230126.0 ,lat=32.722423, lon=-117.164803, Vel_gps=19.333104, Vel_can=1, RPM=1912.5, xacc=0.718, yacc+-1.801
t=230127.0 ,lat=32.722358, lon=-117.164803, Vel_gps=16.7553568, Vel_can=1, RPM=1897.5, xacc=0.833, yacc+-1.877
t=230128.0 ,lat=32.722305, lon=-117.164795, Vel_gps=14.24205328, Vel_can=1, RPM=1868.5, xacc=1.054, yacc+-2.395
t=230129.0 ,lat=32.722272, lon=-117.164803, Vel_gps=10.3109888, Vel_can=1, RPM=1854.75, xacc=0.661, yacc+-2.136
t=230130.0 ,lat=32.722252, lon=-117.16481, Vel_gps=6.70214272, Vel_can=1, RPM=1754.0, xacc=0.527, yacc+-1.849
t=230131.0 ,lat=32.722248, lon=-117.164803, Vel_gps=4.05995184, Vel_can=1, RPM=1751.0, xacc=0.517, yacc+-1.609
t=230132.0 ,lat=32.722248, lon=-117.164803, Vel_gps=2.19108512, Vel_can=1, RPM=1748.5, xacc=0.613, yacc+-0.527
t=230133.0 ,lat=32.722245, lon=-117.164787, Vel_gps=1.67553568, Vel_can=1, RPM=1713.0, xacc=0.297, yacc+-0.45
t=230134.0 ,lat=32.722245, lon=-117.164787, Vel_gps=1.2888736, Vel_can=1, RPM=1708.5, xacc=0.316, yacc+-0.517
t=230135.0 ,lat=32.722245, lon=-117.164787, Vel_gps=0.6444368, Vel_can=1, RPM=1533.25, xacc=0.259, yacc+-0.565
t=230136.0 ,lat=32.722248, lon=-117.16478, Vel_gps=0.0, Vel_can=1, RPM=1430.75, xacc=0.326, yacc+-0.469
t=230137.0 ,lat=32.722248, lon=-117.16478, Vel_gps=0.0, Vel_can=1, RPM=1652.5, xacc=0.536, yacc+-0.508
t=230138.0 ,lat=32.722248, lon=-117.16478, Vel_gps=0.0, Vel_can=1, RPM=1726.5, xacc=0.508, yacc+-0.402
t=230139.0 ,lat=32.72224, lon=-117.164772, Vel_gps=0.70888048, Vel_can=1, RPM=1725.25, xacc=0.584, yacc+0.594
t=230140.0 ,lat=32.722248, lon=-117.16481, Vel_gps=2.19108512, Vel_can=1, RPM=1409.25, xacc=0.795, yacc+0.632
FAULT! Speeding over 70: elapsed time = 4084.0, from = [32.74498, -117.188003], to = [32.736405, -117.175918]
t=230142.0 ,lat=32.72221, lon=-117.164818, Vel_gps=6.2510369599999995, Vel_can=1, RPM=1424.5, xacc=0.201, yacc+-0.057
t=230143.0 ,lat=32.722175, lon=-117.164818, Vel_gps=8.82878416, Vel_can=1, RPM=1410.25, xacc=0.594, yacc+0.105
t=230144.0 ,lat=32.722133, lon=-117.164818, Vel_gps=9.79543936, Vel_can=1, RPM=1250.0, xacc=0.479, yacc+-0.757
```

Figure 5.12: Example Device Log File

One of the largest obstacles found while building the device was in the testing of new features. This was due to the inability to view the device in runtime while driving the test vehicle. As new features were added the chance of a runtime error halting the test increased dramatically, a method of simulating the vehicle became necessary.

The need for simulation began at the start of development where the CAN bus algorithms were under test. The testing of these functional blocks require repeated analysis to tune and develop. To simulate the bus, log files are recorded while the vehicle is driven during short trips. Once collected and back at the bench, the secondary CAN port on the Raspberry Pi shield is connected to the primary port. By playing the log file over the secondary port, the device can be fooled into thinking it is connected to a vehicle.

As the device increased in complexity the need for simulation shifted to the other sensors as well. The edge cases within the automotive environment create many avenues for device errors to arise. A method evaluating the code from the bench became vital to the identification of errors in the code. To accommodate this, the generated log files are employed to source data for the simulation. The functions responsible for collecting data from the sensors are instead directed to parse the data from the log file, allowing the device to operate on the same data that was collected during a trip. Holes in the software could then be identified and the many edge cases within the vehicle could be covered.

Chapter 6: Device Evaluation

The device was constantly tested through all stages of both hardware and software development. A special focus on analysis was necessary to evaluate how effect the device performed while decoding the vehicle bus, tracking trips and detecting faults.

6.1: Evaluation: Decoding the Vehicle Bus

Over the course of development, the device was placed in many different vehicles to evaluate if the CAN decoding algorithm was effective. Logging all the parameters needed to classify both RPM and speed, a local database of parameters was created to track similarities and identify vehicles that cause failures. Due to the variation in how vehicles report the data, not every vehicle is able to be decoded. Below in Table 6.1 the list of decoded and partially decoded vehicles is shown.

Table 6.1: Vehicle Decoding Library

	RPM Parameters			Speed Parameters		
Vehicle	PID	Scale	Byte Loc	PID	Scale	Byte Loc
2007 F150	0x201	.25	0	-	-	-
2016 F150	0x201	.25	0	0x201	.006213	4
2012 Yukon	0x0C9	.25	2	-	-	-
2009 Civic	0x17C	1	4	0x309	.006213	0
2019 Civic	0x210	1	2	-	-	-
2017 Grand Cherokee	0x0D8	1	0	0x0D8	1	2
2020 Corolla	0x1E7	.25	4	-	-	-
2015 Tacoma	0x1E7	.25	4	-	-	-
2010 i135	-	-	-	-	-	-

The table above shows that the RPM parameter is far more common to decode than the speed parameter. The source of this discrepancy can be explained by a number or reasons. The first cause failure stems from the presence of data on the bus. In many of the cases where speed is not detected, it was found that the data was not broadcasted on the bus. Vehicles made by

BMW and AUDI are also known to limit the data on the bus for security reasons. The second cause of failure can be attributed to the number of scaling parameters for speed. The most common scaling parameters are used in decoding, but in some vehicles the scaling parameter is not common and a failure arises. The final source stems from error in the GPS speed used to decode the CAN bus. If the location fix accuracy is low, then the calculated speed carries this error as well. If the error is high enough then the value used in decoding will not match what is present on the bus.

6.2: Evaluation: Trip Tracking and Fault Detection

The evaluation of trip tracking and fault detection go hand in hand as both functionalities require the proper collection of data to operate correctly. As the device only reports through SMS, a separate method of evaluation is needed to plot individual trips and the generated faults. Fortunately, all data collected during each trip is logged into separate files for simulation. These logs also contain the SMS data used in reporting. By pulling the logs off the device and processing them, the trips can be reconstructed and the faults shown on a map.

In the first trip example, the device was driven from La Jolla, South on I-5 to downtown San Diego, then back North on I-15 to Claremont, where surface streets were taken West towards the starting point. While on the route, the vehicle was driven in a way that would create multiple faults to best evaluate the performance of the device. Figure 6.1 shows the route of the trip as well as the faults generated. Each fault is shown with a unique marker. Periodic faults are shown with a maker placed at the start and end to show the road segment the fault occurred on. Instantaneous faults are marked at the location of creation. A legend of each fault and their associated marker is depicted in Table 6.2.

Table: 6.2: Legend of Fault Icons

Icon	Fault Type	Icon	Fault Type
🔴	Speeding (start and stop)	⚫	RPM (start and stop)
🔵	Hard Brake	🔵	Hard acceleration
🟣	Hard Left Corner	🟣	Hard Right Corner

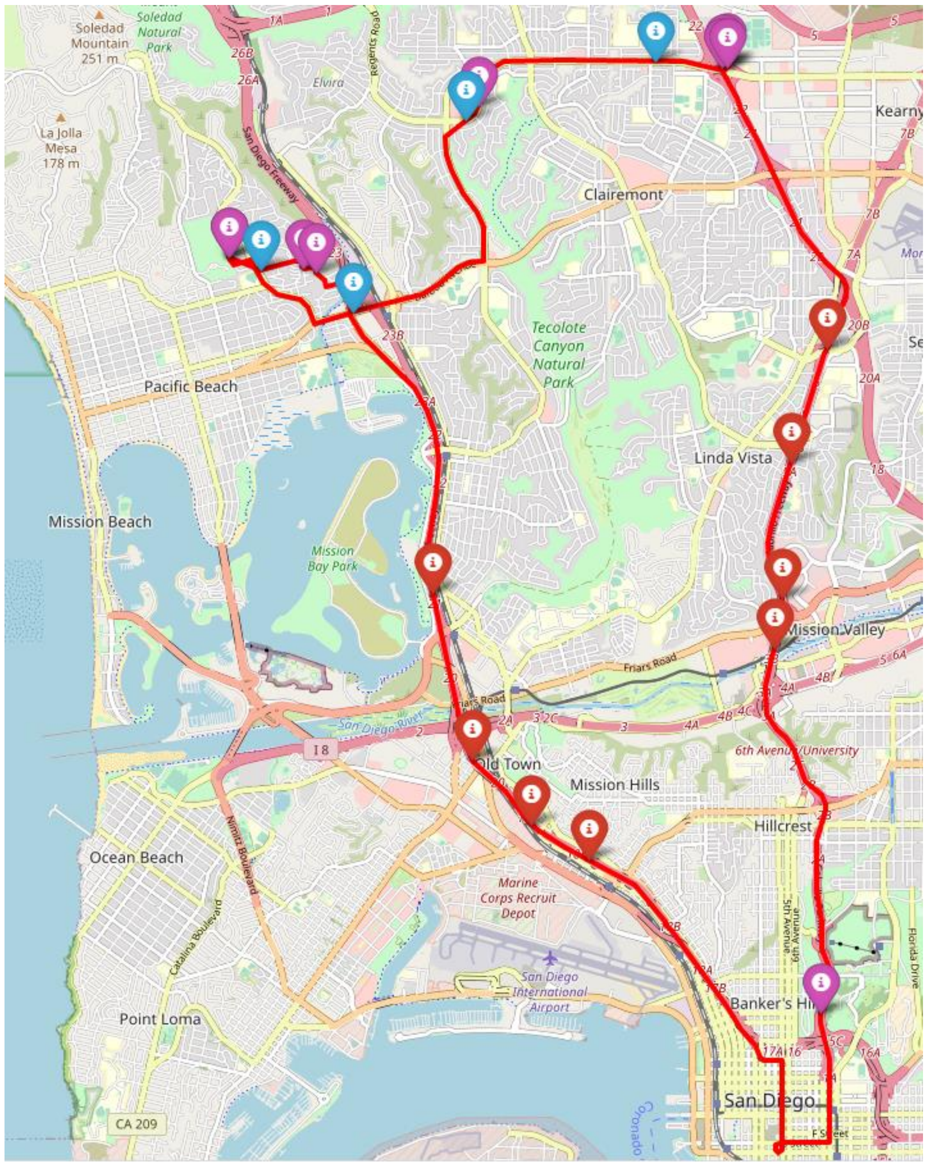


Figure 6.1: Example 1: Trip and Fault Overview

The figure above shows that the device correctly detected the entirety of the trip by starting and stopping at the correct locations. A failure in trip tracking is identified by the device ending a trip while waiting for a stoplight or in the abrupt ending of a trip while the vehicle is in motion. Figure 6.1 shows that the vehicle state machine is effective in detecting the status of the vehicle and therefore identifying the start and end of trips.

The trip above covers many different types of roadways a vehicle may travel. Residential roads, city streets, downtown streets and highways are all traversed. The red line on the map shows the GPS locations collected once a second. Most cases show the position of the vehicle with an accuracy of 3m or less. The location accuracy degrades as the vehicle travels through the downtown streets. The presence of tall buildings creates an urban canyon and results in multipath errors. The effect of these multipath errors can be seen in the plot as a deviation from the traveled roadway.

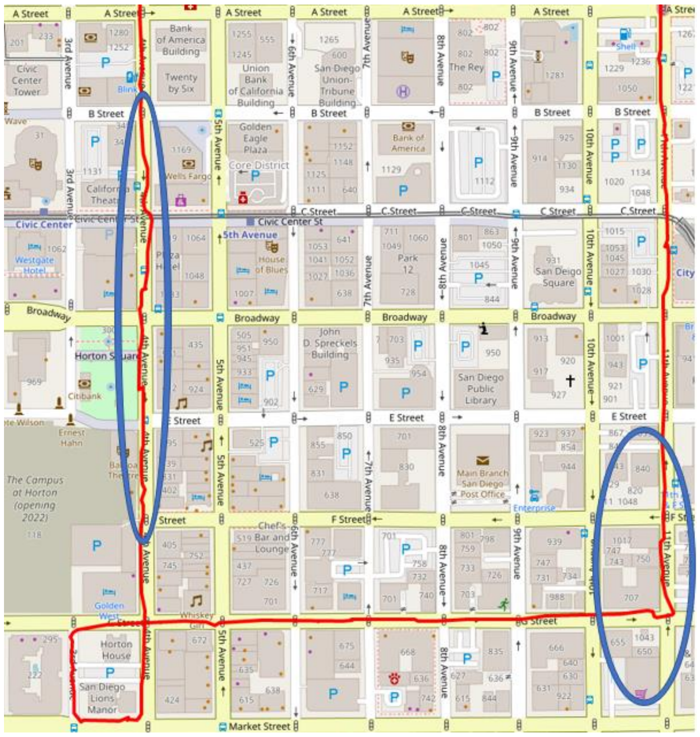


Figure 6.2: Example 1: GPS Deviation of Urban Canyons

It is due to these deviations that the GPS is not the primary source for speed data. As the accuracy of the location fix decreases, the error in the GPS speed rises. In some cases, the accuracy degrades to a point where a fix can be lost. Although the GPS module may perform correctly for 99 percent of the time. It is the 1 percent chance of collecting incorrect or unavailable data from the GPS that results in its usage as a fallback data source.

Figure 6.1 also shows the occurrence of faults during the trip and Table 6.2 gives the legend of faults and their markers. During the trip four instances of speeding over 80mph were detected, nine harsh left turns and four harsh braking faults were detected. Figure 6.4 zooms in on the speeding faults and where they occurred.

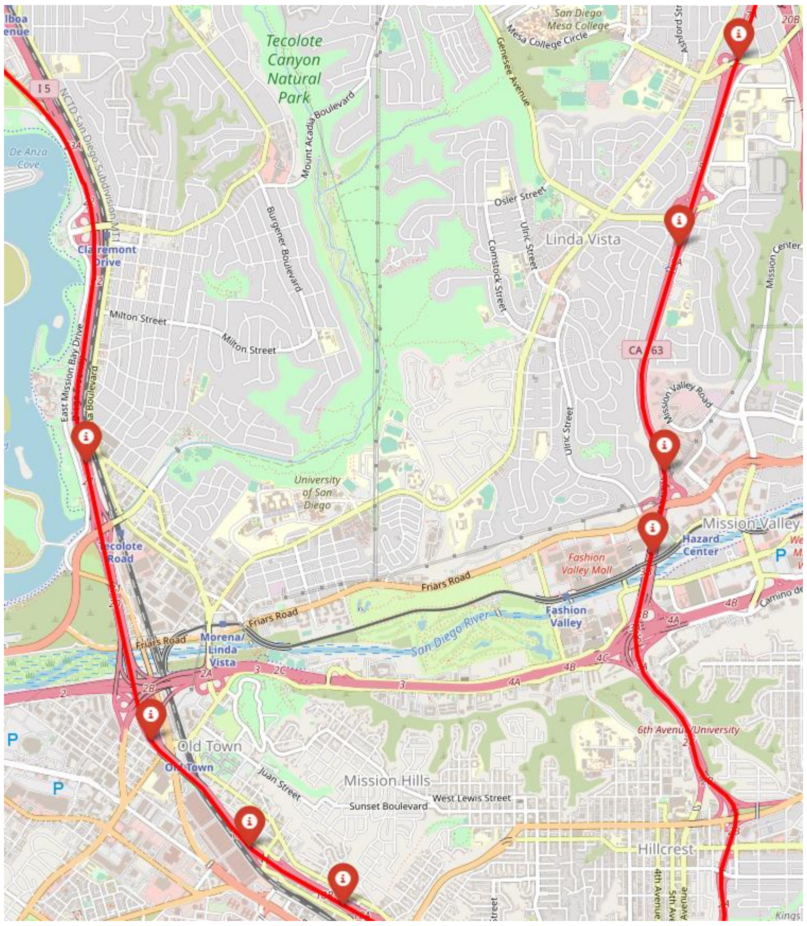


Figure 6.3: Example 1: Speeding Fault Locations

Each speeding fault is characterized by the start and end of a fault. Figure 6.3 shows each of the four speeding faults and where they started and ended. By analyzing the raw logs of the file, the start and end of the speeding is correctly identified. It is also shown that each of the faults only occur while traveling on a highway. Speeding on other road types are missed because the speed limit set statically to 80mph. It is very rare that a vehicle would travel above the limit while traveling on any roadway other than a highway. Therefore, the scope of the speeding fault only contains roadways where high speeds are allowed.

A similar analysis can be done on the occurrence of the harsh braking and cornering faults. Multiple instances of each were recorded and are identified by three accelerometer samples where their average exceeds +/- 3m/s^2 depending on the direction of the fault. As with all fault limits, the values are hard coded and this results in the faults occurring on specific roadways. For the instantaneous faults, they tend to occur while on city streets because the speed is great enough to result in longer periods of harsh acceleration. The concentration of these faults on city streets can be seen in Figure 6.4.

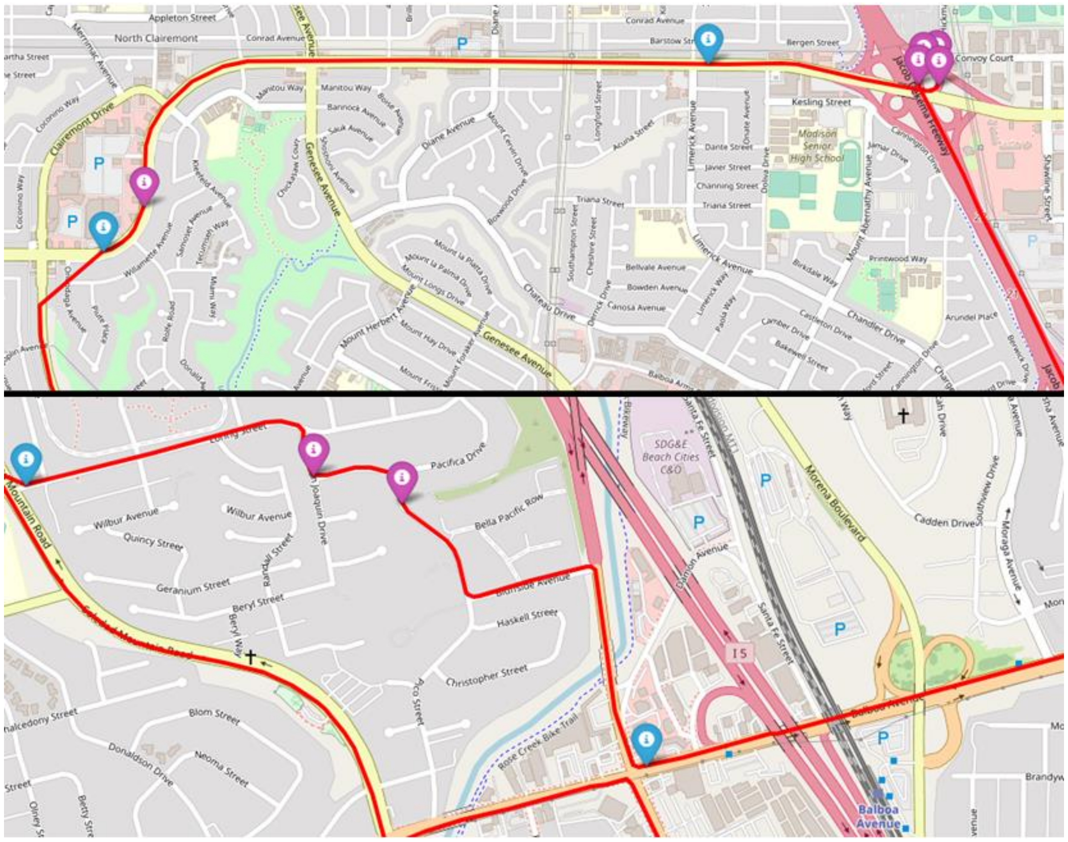


Figure 6.4: Example 1: Acceleration Fault Locations

The device performs exceedingly well according to its design. All peripherals are initialized correctly. The vehicle bus is decoded and fallbacks are used when needed. The device correctly identifies vehicle states, starts trips accordingly and collects data over the entirety of the trip. This data is then processed and faults are sent over the cellular network to the user. Functionally, the device follows the same standards as most automotive IoT devices on the market. However, during the analysis of the device’s operation, weaknesses are shown in the topic of fault detection. As the device was driven through the trip, faults were intentionally created in each type of road traveled. The device fails to detect multiple faults while traveling on the highway, downtown and across city streets. As shown in Figure 6.5 road segments where faults should have been detected are highlighted.

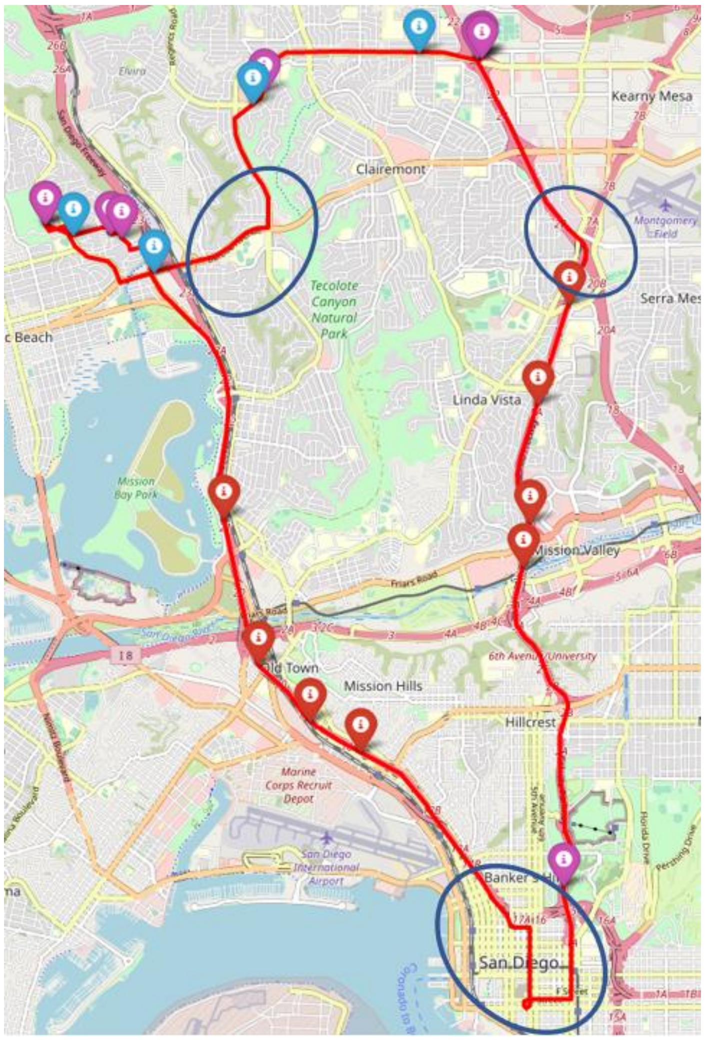


Figure 6.5: Example 1: Road Segments with Missed Faults

The inconsistency in fault detection stems from the static fault limits that are used to define individual faults. The figure above displays where each fault type occur and shows a distinct pattern where certain faults are only detectable on specific roadways. For example, speeding is only detected on a highway because speeding on other roadways occurs at a velocity lower than 80mph. The concentration of acceleration faults on city streets illustrates that the current model of fault detection is insufficient in detecting faults while traveling on the highway or downtown. This necessitates the need for an expansion in the fault detection functionality so that the scope of each fault can be extended to all road types.

Chapter 7: Road Speed Estimation

As explained in the previous chapter, there are weaknesses with the method of fault detection currently used within the industry. Most faults occurring in other driving scenarios are missed because fault limits are static. A method of adaptively changing fault limits is needed to catch these missed faults.

7.1: Classification of Road Types

Vehicles operate on many different types of roads with varying speed limits depending on the state or territory. Identifying the exact speed limit of the road without using a camera system or an expensive API is virtually impossible. Grouping similar roads together b use cases and speed limits allows for a general model that can be created to classify the type of road being traveled. Within the United States, roads are classified by their speed limit and these limits range from 25mph to 80mph. Identifying the type of road traveled, fault limits can be tuned to the current operation of the vehicle. Three classifications are used: highway, city street and traffic. The types of roads that fall within each classification are shown below in Table 7.1[32].

Table 7.1: Road Types and Their Speed Limits

Road Type	Speed Limit(mph)	Classification
Interstate(uncongested)	65-80	Highway
Highway(uncongested)	65-80	
Expressway	45-55	City Street
Arterial	35-45	
Local Streets	25-35	
Downtown Streets	25-35	Traffic
School Zones	25	
Congested Roadways	N/A	

The easiest classification to define is the highway designation. Vehicle operation while on a highway is similar regardless of the speed limit. Each of the road types within this category allow for the uninterrupted movement of vehicles on the road. While on a highway the lateral acceleration is minimal while the speed remains high and relatively constant if there is no congestion on the road.

Other road classifications become harder to determine as the types of roads that fall within their designation span wider use cases. For the city street designation this is especially true. The different types of roads traveled may be transporting cars through neighborhoods, city blocks or expressways. Each of these road types are defined by different speed limits and the vehicle's operation change. The defining factor of these road types are that stoplights are present, and the vehicle comes to a stop periodically. This factor is used in the following chapters to detect the road type.

The final road classification is traffic. This designation covers the operation of the vehicle at low speeds. This occurs while within a school zone, or in any of the road types where congestion is present. The presence of congestion allows any of the road types to appear as one with a low speed limit. The defining factor for this designation is the presence of frequent stops with a low average speed.

7.2: Road Type Detection

As mentioned above, each road classification has one or more distinctive features. It is the exploitation of these features that allows the device to determine what road class is being driven. As with the detection of faults, the detection of the road class is done by analyzing the contents of the circular buffers. The primary circular buffer used is either CAN speed or GPS speed depending on the decoding status of the vehicle bus. This buffer can be analyzed to find the average speed of the vehicle and whether the device is at rest for any period of time. The analysis of average speed, stop count and stop length are the focus of analysis for the detection of the road states.

The highway state is the most identifiable state as the operation of the vehicle on a highway includes high speeds with no stops. The absence of stops with the vehicle moving greater than 60mph is indicative of a highway. Conversely, while on a city street, the average speed and the presence of stoplights are the defining features. On average, the time spent waiting at a red light is 40 seconds[32]. Therefore, by searching the speed buffer for periods of rest allow for the detection of stops that indicate stoplights. The average speed can also define a city street as the speed limits for the roads contained within the classification fall within 25mph to 55mph. The final, and hardest to distinguish, is the traffic classification. As the road types within this class are defined by their low speed limits, they also contain all occurrences of heavy congestion. The need to encompass congestion within this class is due to the fact that the vehicle is operated at low speeds with frequent short stops. It is this operation that allows the traffic class to be detected. Below in Table 7.2 the road states, classification of features and the values to distinguish them are shown.

Table 7.2: Road Classes and Distinctive Features

Road Classification	Distinctive Features	Numeric Constraints
Highway	• No stops • High average speed	• Stop count = 0 • Average speed > 60
City Street	• Multiple long stops • Medium average speed	• 0 < Stop count < 3 • Average stop time > 15s • 25 < Average speed < 60
Traffic	• Frequent short stops • Low average speed	• Stop count >= 3 • Average speed < 25

The process to detect the road states leverages the speed data from the vehicle to ascertain the current road state. The average speed, number of stops and length of stops are used to determine the road state. As the operation of a vehicle can dramatically change at any point, multiple collection cycles must be analyzed and concatenated together to properly determine the correct road type. As each new section of data is analyzed, the average speed is calculated to assign an internal state. These internal states mirror that of the road states. To continue and process the stop metrics, the device must collect three consecutive sections of data that fit the same internal state. If an internal state is not the same as the previous internal or the same as the currently assigned road state, then the concatenation is cleared and assigned the new batch of data. The current internal state will then be used to check against the next batch collected. Once three consecutive internal road states have been collected and concatenated together, the concatenated buffer is searched for the occurrences of stops. The number of stops and the average time stopped are used to determine if the internal state is a valid transition. If the transition is valid, then then the current road state is changed to match the internal state, the fault limits are adjusted, and the concatenated buffer is cleared. The process of detecting and assigning new road states can be seen below in Figure 7.1

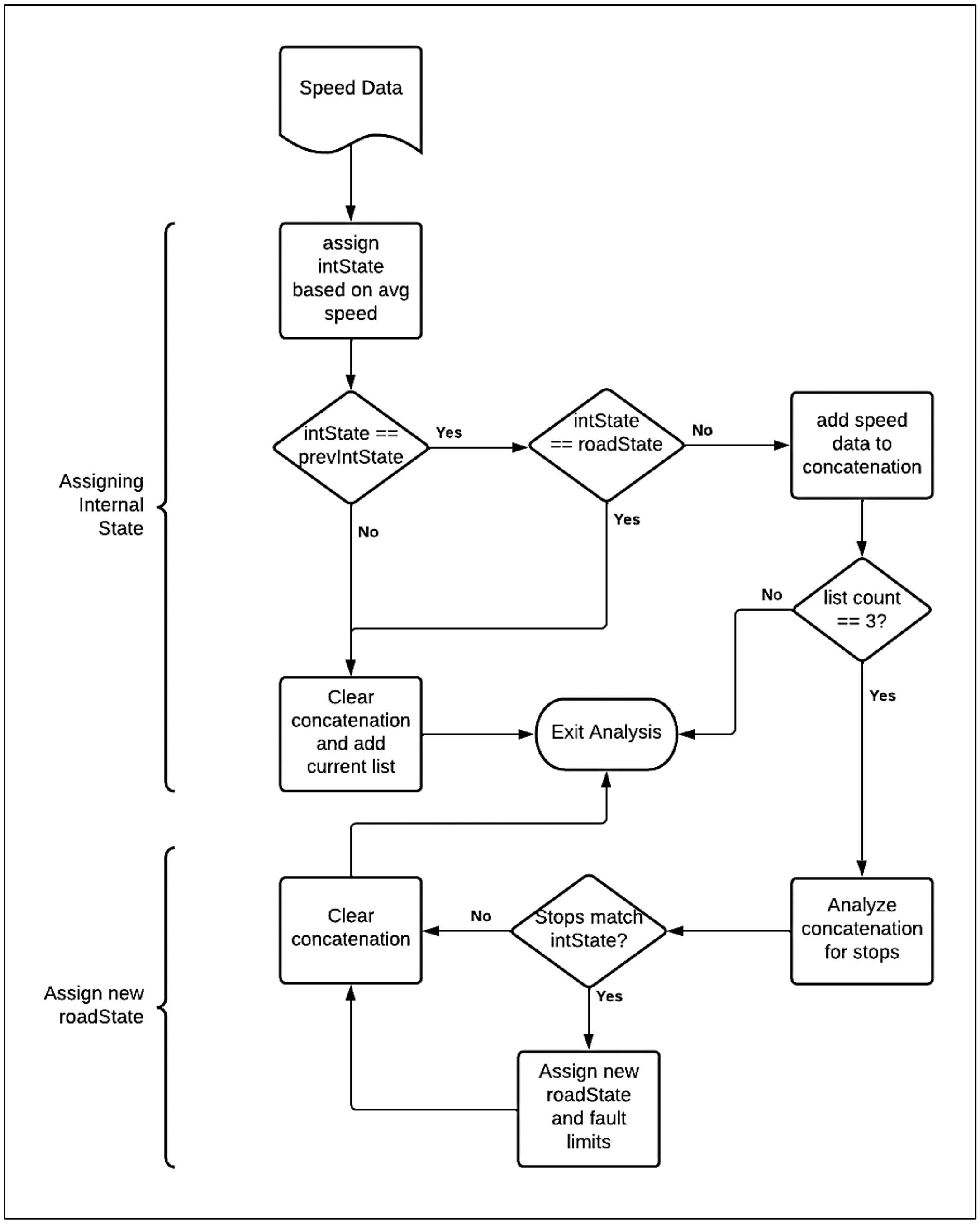


Figure 7.1: Road State Detection Flowchart

The functional block that is responsible for the detection of road states is placed directly up stream from the fault detection algorithm. This allows for the road state to be determined before the circular buffers are evaluated for faults. As with fault detection, once the buffers are full and the device in motion, the next iteration of the road state detection algorithm will begin. Figure 7.2 shows the updated main system state machine with the added functionality placed before faults are checked.

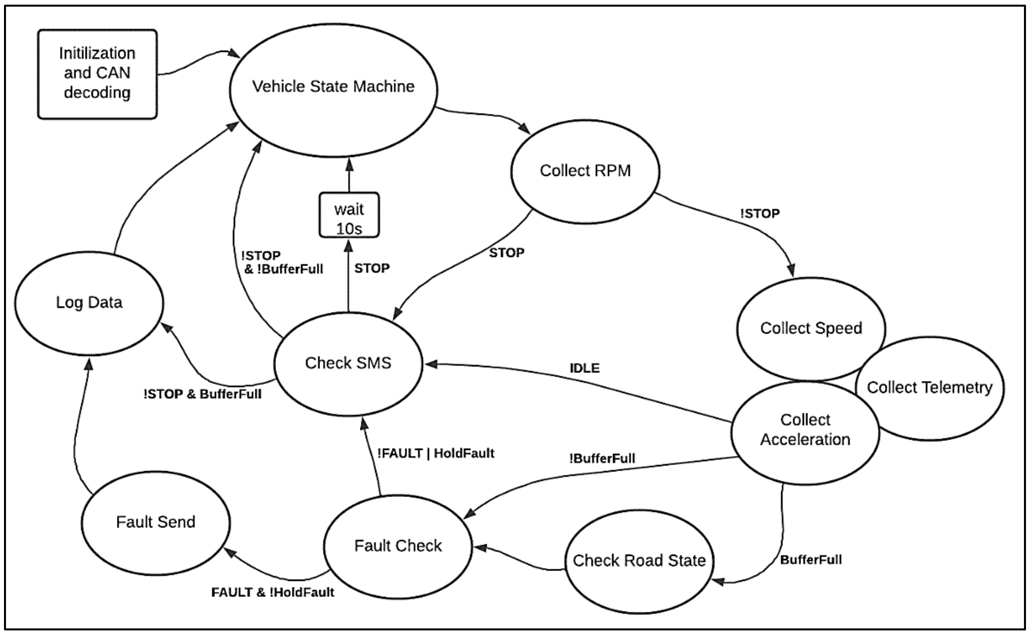


Figure 7.2: Main System State Machine with Road States

7.3: Adaptive Fault Limits

As laid out above, detecting the road states allows for greater visibility of the vehicle's operation. The fault limits need to change to best detect faults for each type of road class as the operation changes. Out of the list of possible driver faults, the most affected by the road state are harsh braking, harsh cornering and speeding. The speed limit parameter is best defined as ten miles per hour over the maximum speed limit within the class. This results in the highway speed limit set to 80mph and the city street speed limit set to 55mph. Due to the definition of the traffic class, the speed limit while traveling in traffic inherits the previous road class identified. Allowing the device, if leaving an area of congestion, to maintain the most probable speed limit when the congestion subsides.

The two parameters used to detect harsh accelerations are the acceleration window length and the average acceleration measured over the window. These parameters are more complex to set as the acceleration values during a harsh brake are not known. The profiles of harsh braking and cornering must be collected to set the proper limits. To accomplish this, a separate program is run on the device that collects the acceleration data at the maximum rate. The following figures were generated by performing the harsh maneuvers on a rural roadway to analyze the acceleration profiles of at different speeds.

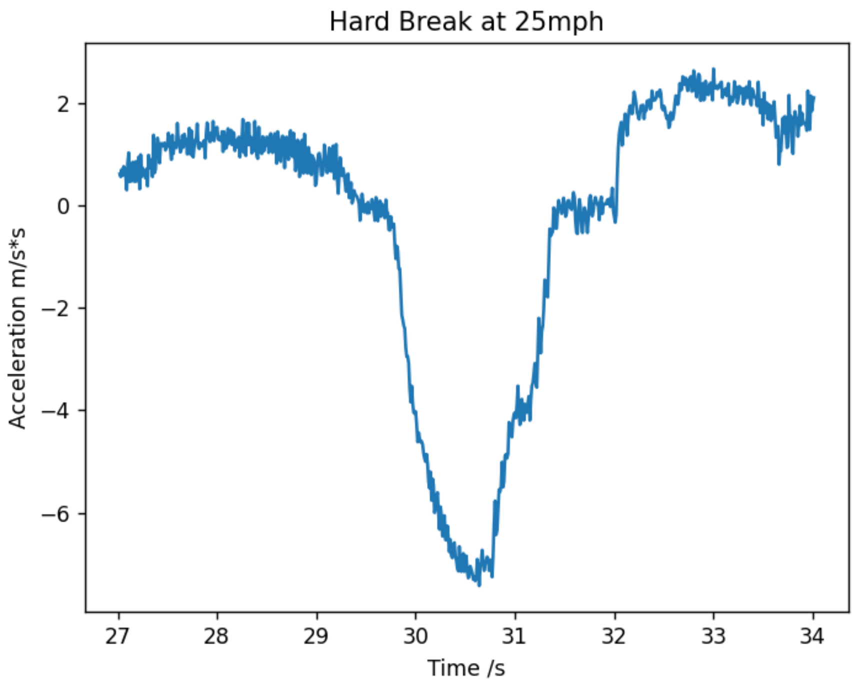


Figure 7.3: Acceleration of a Harsh Brake at 25mph

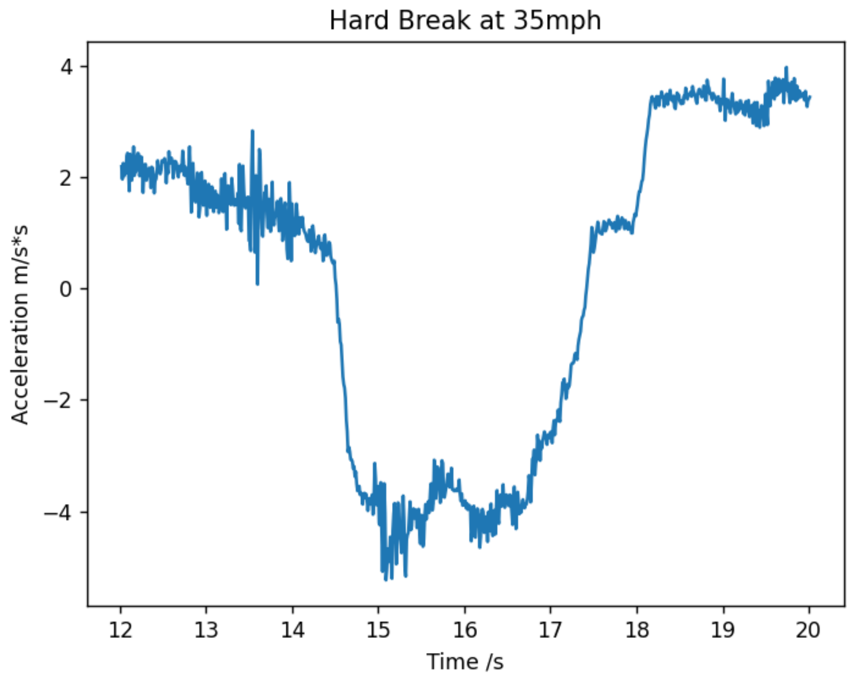


Figure 7.4: Acceleration of a Harsh Brake at 35mph

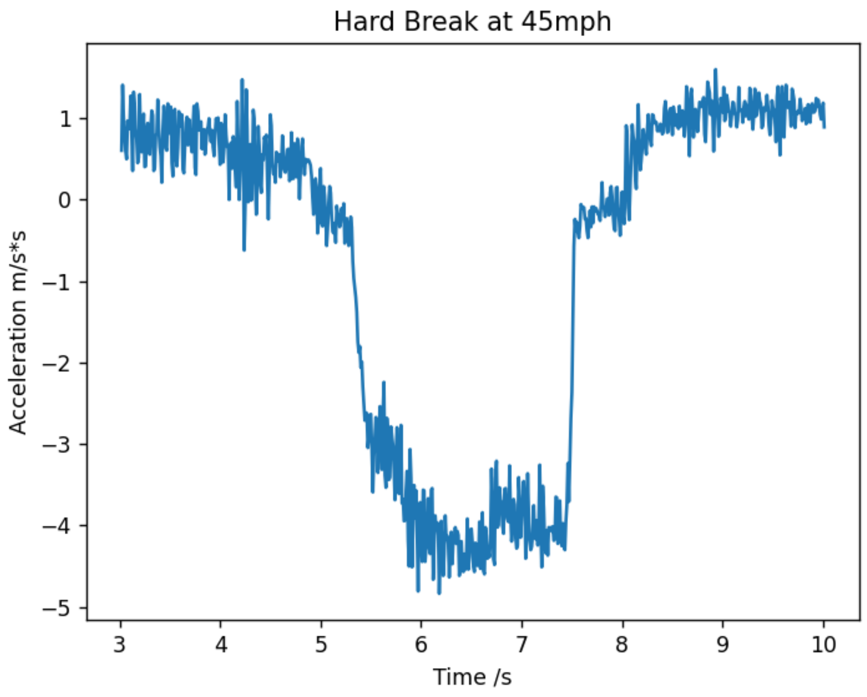


Figure 7.5: Acceleration of a Harsh Brake at 45mph

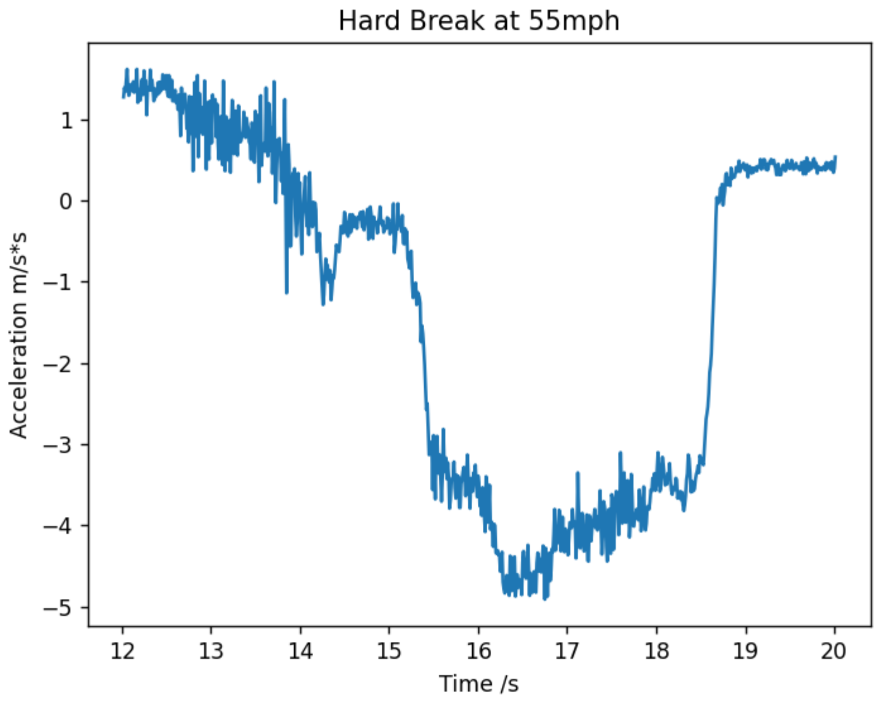


Figure 7.6: Acceleration of a Harsh Brake at 55mph

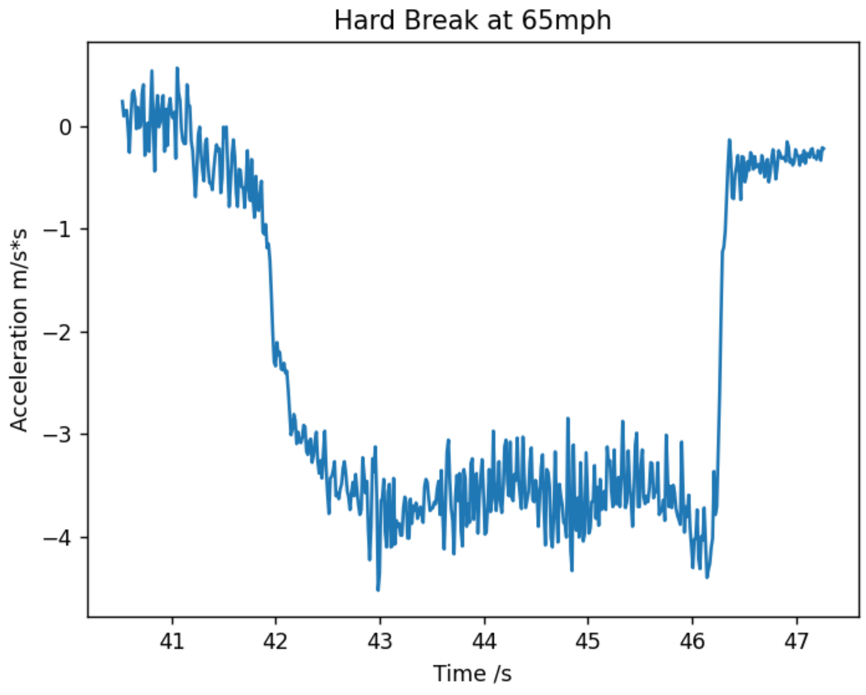


Figure 7.7: Acceleration of a Harsh Brake at 65mph

Figures 7.3 to 7.7 depict the acceleration along the Y-axis of the accelerometer while the vehicle is coming to a near stop from varying speeds. In each situation the brake pedal is depressed until the harsh brake begins to feel unsafe. Doing so best simulates the real world application of a harsh brake event. At low speeds the brake pedal is pressed to its maximum as faults at this speed often occur within an instant. The waveform in Figure 7.3 depicts this quick deceleration. Conversely, in Figure 7.7 when the vehicle is traveling at 65mph, the brake pedal is not at its maximum and the waveform shows the longer, less abrupt deceleration. The waveforms at each speed reveal that at lower speeds the harsh brake profile is characterized by a large and quick spike in acceleration. The profile can be characterized by a long period of less severe acceleration while at high speeds. To best detect harsh braking, the acceleration limits for different road states must reflect this trend. Table 7.3 lists the limits, derived from the above figures, to detect harsh braking in each road state.

Table 7.3: Harsh Braking Limits for Road States

Road State	Acc. Window Length	Average Acc.
Traffic	1	- 5
City Street	3	- 3.5
Highway	4	-3

A similar analysis is done for harsh cornering. In the city street and traffic road states, harsh cornering normally occurs when leaving an intersection and making at least a 90^o turn. A harsh corner on the highway state occurs if the vehicle is traveling along a ramp or interchange with a high rate of speed. This results in a high centripetal force that is read on the X-axis of the accelerometer. The following figures show the harsh cornering waveforms while traveling on different road states.

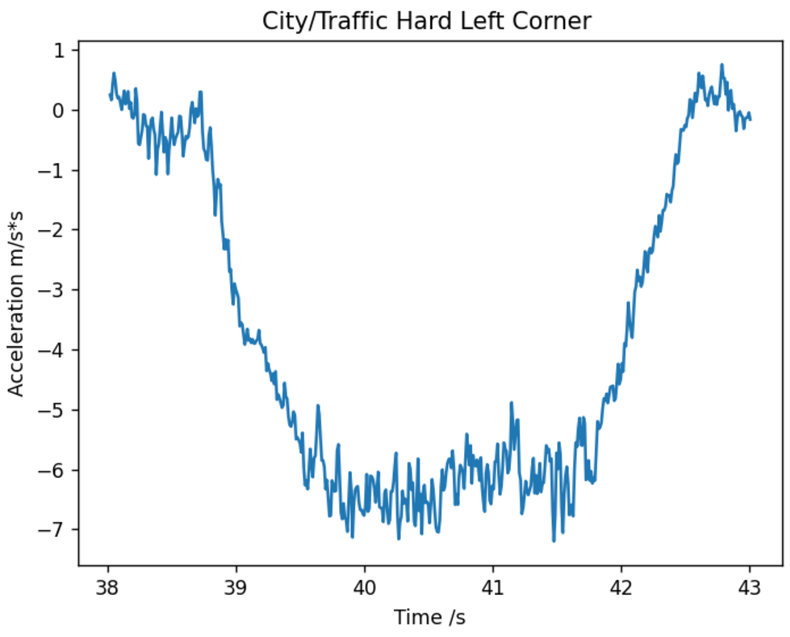


Figure 7.8: Hard Left Corner on City Street

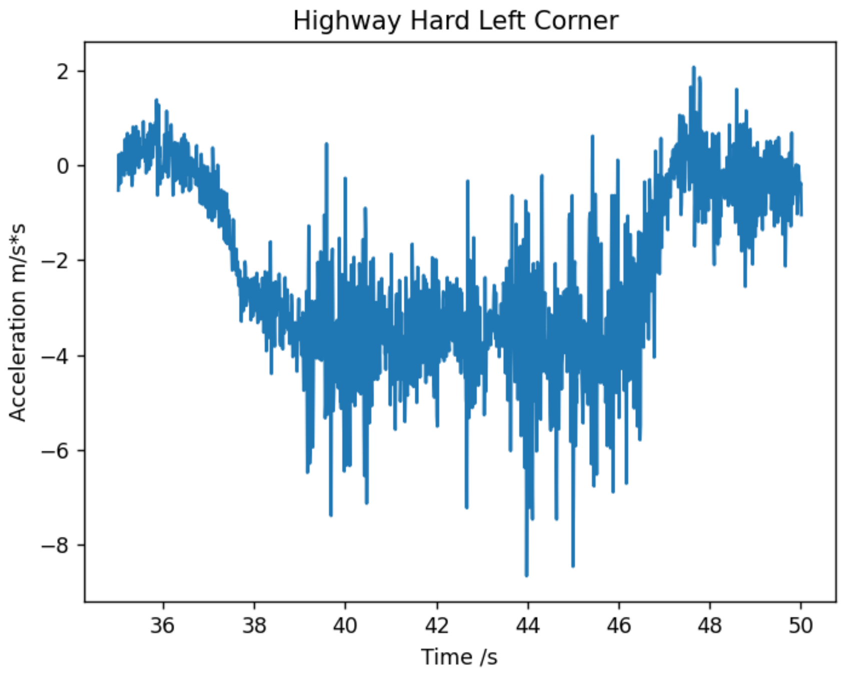


Figure 7.9: Hard Left Corner on Highway

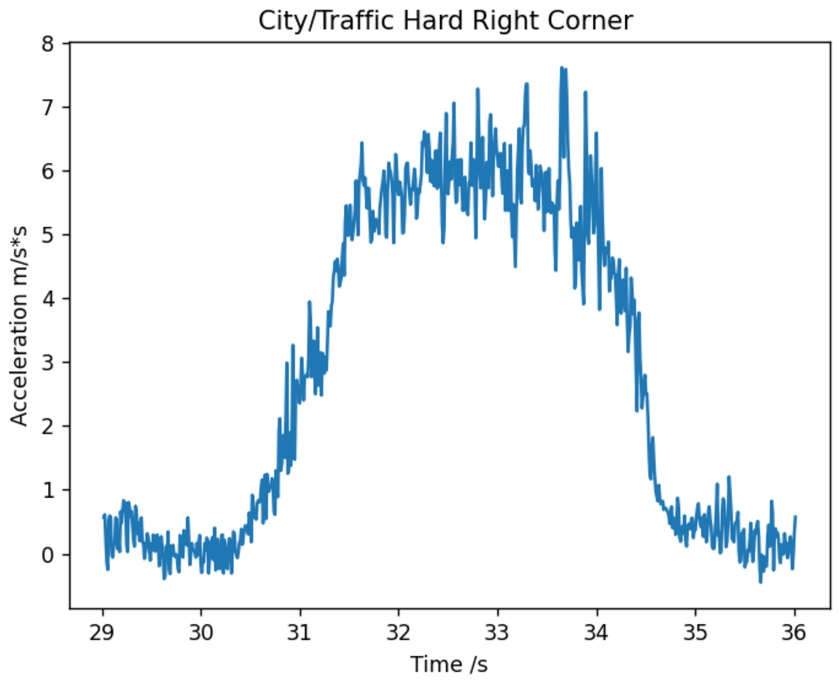


Figure 7.10: Hard Right Corner on City Street

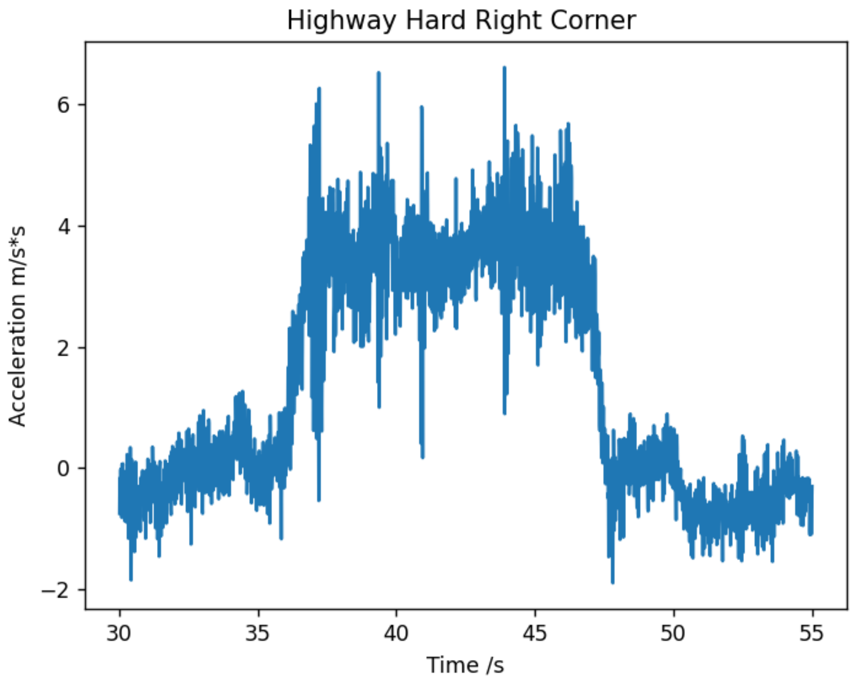


Figure 7.11 Hard Right Corner on Highway

The harsh cornering shown in Figure 7.8 to 7.11 show the acceleration waveforms produced on different road states. The trend revealed in the harsh brake analysis remains true with the analysis of the harsh cornering. As the speed of the vehicle increases, the peak acceleration during a fault decreases while the duration of the fault increases. During the collection of the data in the traffic and city street case, a U turn is performed at a high rate of speed. The window length for a harsh corner in these states should be set to roughly half of the acceleration duration shown in the plots. This is done to catch harsh cornering while making at least a 90deg turn. As was done with the harsh brake limits, the cornering faults can be tuned to the specific road state traveled. Below in Table 7.4 the limits for harsh cornering can be seen for each road state.

Table 7.4: Harsh Braking Limits for Road States

Road State	Acc. Window Length	Average Acc.
Traffic	3	+/- 5
City Street	3	+/- 5
Highway	5	+/- 3

By assigning fault limits based on the class of road traveled, the scope of each fault is expanded from one type or roadway to all roadways the vehicle can travel. The burden of detection then gets shifted to the identification road states instead of the individual faults themselves.

7.4: Fault Buffering

The final portion of the road state detection functionality is the process of buffering faults. As with SMS commands, if a fault limit is changed while the fault is in progress, the collection of that fault may terminate immediately or continue forever. Since the road state dictates the fault limits, when to apply those limits necessitates special consideration.

As described previously, the device sets both a current road state and an internal road state. It is the repeated setting of the internal state that confirms the change to a new current road state. Because of this, there is a delay between the true transition point and the detected transition point. This in some cases creates invalid faults. If the device is in a higher state like the highway state and moves to the city street state, no fault will be generated as the device is lowering its average speed. But for a transition up in state, the vehicle will exceed the speed limit in its transition. For example, if the device is in the city street state and merges onto the highway, then the device will take three collection cycles to reflect the highway state. During this delay an invalid fault will be created when the vehicle gets up to speed on the highway. A method of buffering faults is needed to store the faults while the device is evaluating a change from a lower state to a higher state. Figure 7.12 depicts the road state transitions and when faults are buffered.

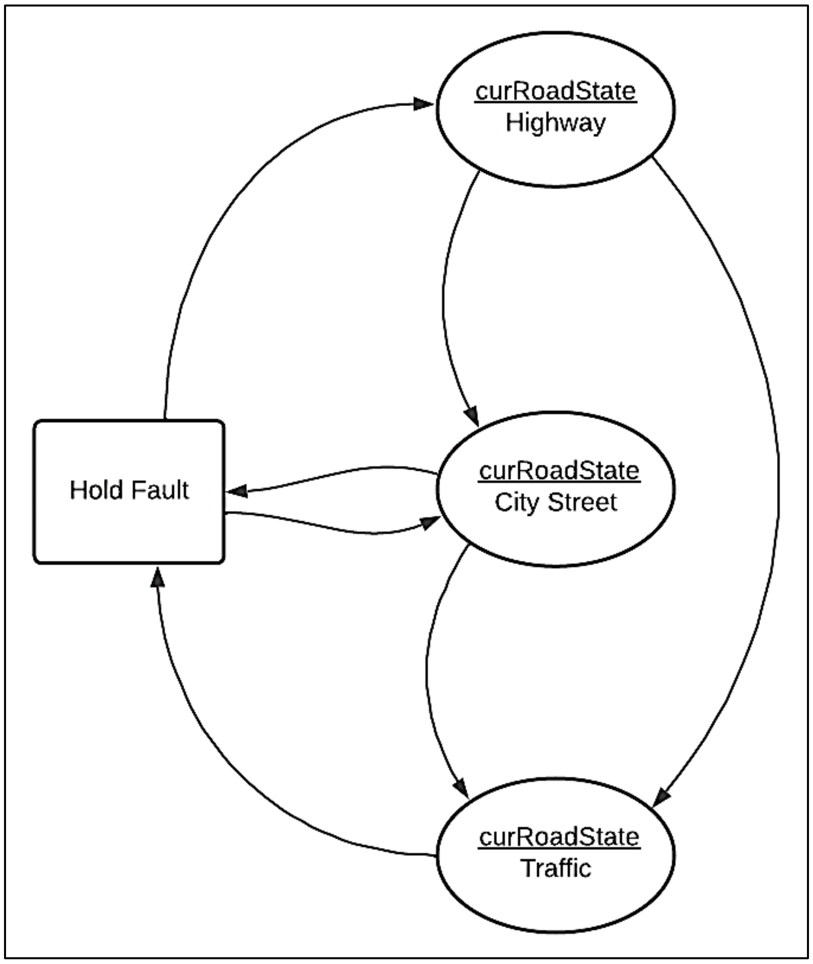


Figure 7.12: Road State Transition Diagram

Fault buffering does not occur when moving down in states as shown in Figure 7.12. When moving up in states, any fault that is generated gets placed into the fault buffer. If the road state change passes, then the faults in the buffer are proven to be invalid and they are thrown out. If the state change does not go into effect, then the data in the buffer is sent over the cellular network via SMS to the user. The information is also logged and marked as a delayed fault due to state change.

Chapter 8: Device Evaluation with Road Speed Estimation

With the second design iteration, the road state algorithm was integrated and the fault detection parameters were expanded to reflect the different road states. To properly depict the change in functionality, the trip used in Example 1 is performed again with the functionality improvements. This is done by loading the simulation file that was generated while the previous iteration was evaluated.

As the road state estimation process defines how faults are detected, it must first be evaluated to confirm that the correct road state is assigned to each segment of the trip. To best distinguish the multiple faults and road states, a legend of icons can be seen in Table 8.1.

Table 8.1: Legend of Fault Icons and Road States

Icon	Fault Type	.....	Icon	Road Type
🔴	Speeding (start and stop)		🟥	Highway
⚫	RPM (start and stop)		🟧	City Street
🔵	Hard acceleration		🟩	Traffic
🔵	Hard Brake			
🟣	Hard Left Corner			
🟣	Hard Right Corner			

Using the simulation file from the previous example and excluding the faults, the following map was created to show the detection of the road states during the trip. As with Example 1, the trip begins in La Jolla, goes through downtown San Diego and loops back to the start through I-15.

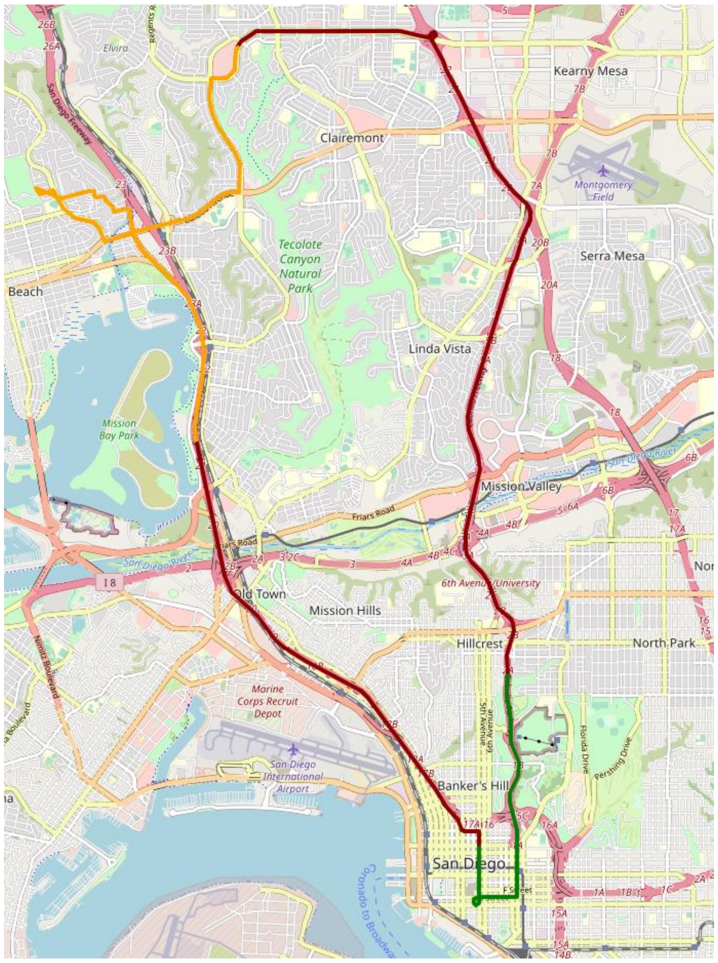


Figure 8.1: Example 2: Road State Estimation

As Example 2 shows, the device assigns each of the road states to one or more segments of the trip. It can be seen that the device correctly identifies the road state being traveled by looking at the roadways that the vehicle traveled during each segment and where the transition occurred. Each of the highway segments are correctly identified and the device properly differentiates the traffic and city street states. The delay in transition mentioned in the previous chapter can also be observed as the transition point from one state to another occurs after the true transition occurs. This is best seen as the device leaves downtown San Diego and merges onto the highway. The transition point occurs well past the freeway onramp because the device takes three collection cycles to confirm a new road state.

The detection of road states allows the device to expand the scope of each fault and allow each to be identified on all three of the road states. Plotting the trip again and overlaying the faults onto the road map, most faults from the previous iteration remain. The undetected faults are also identified in the missing sections highlighted in Figure 6.6.

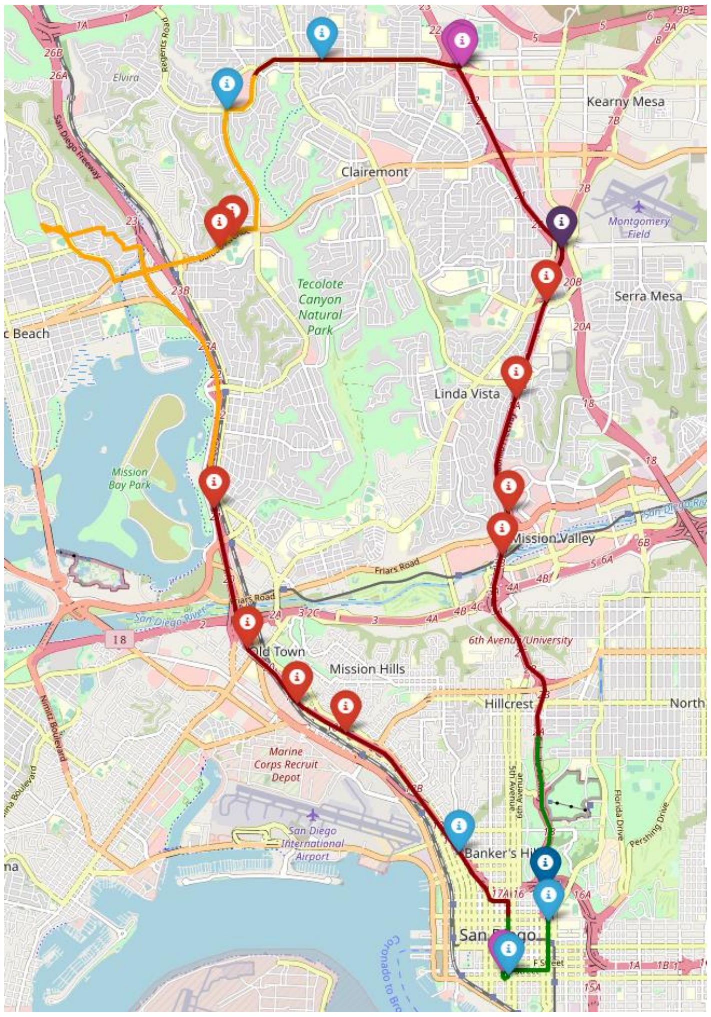


Figure 8.2: Example 2: Fault Detection with Road Class Estimation

By zooming into each of the different road states, the faults are shown with an increased granularity and the validity of each fault can be confirmed. Figure 8.4 depicts the two segments where the vehicle traveled along the highway. The instances of speeding remain unchanged as the limit for speeding along a highway was not changed from the first iteration. However, the limits for instantaneous faults were changed and faults were correctly identified. In the lower left corner of the map, a harsh brake was detected as the vehicle left the freeway. On the right side of the map a harsh right turn was detected as the vehicle traveled over an interchange between highways.

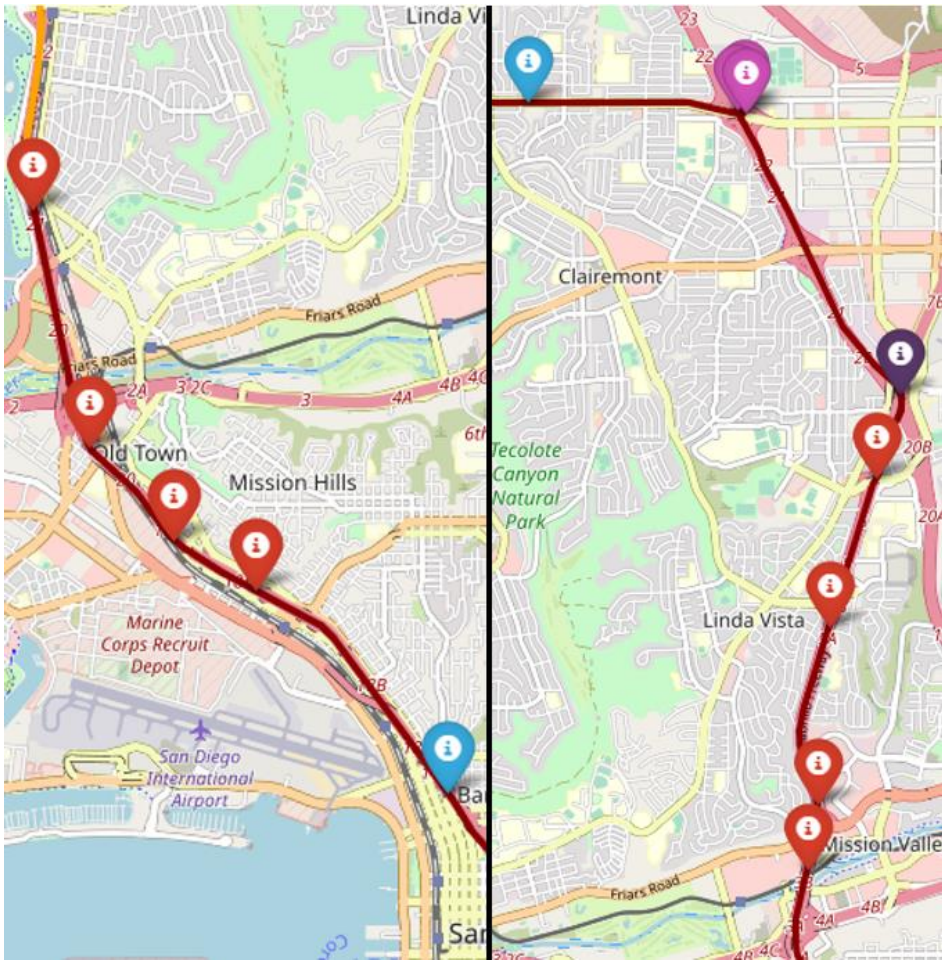

Linda Vi
Friars Road
Old Town
Mission Hills
Marine Corps Recruit Depot
San Diego International Airport
Clairemont
Tecolote Canyon Natural Park
Linda Vista
Mission Valle

Figure 8.3: Example 2: Highway Fault Detection

The city street segments occur at the start of the trip and at its end. During the start of the trip no faults were intentionally created. The lack of faults in this segment is expected. Conversely, at the end of the trip, multiple faults were created. The first being a hard brake that can be seen in blue on Figure 8.4. This fault was correctly identified in the first iteration. The next fault detected is a speeding fault that occurs while traveling over 55mph on a city street. With the second iteration, this fault is identifiable. The limits for the city street also caused the removal of the harsh braking and cornering faults that occurred at the very end of the trip. The faults that occurred here were not intentional and were a result of the static limits. By tuning the limits to match the profile of city street faults, the behavior of the trip proved to be within acceptable limits.

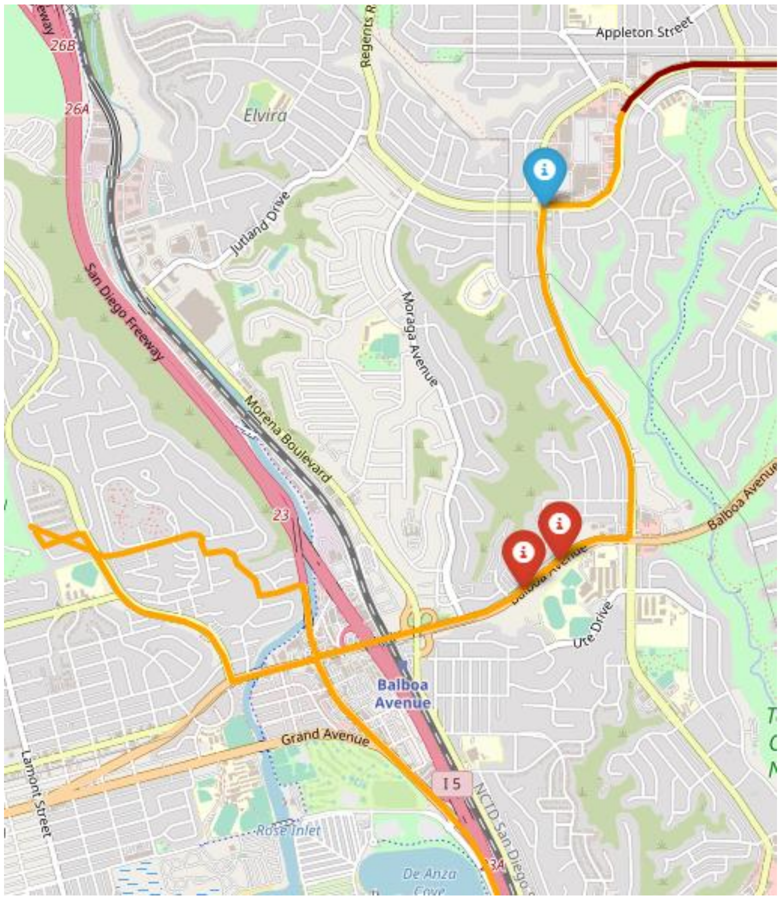


Figure 8.4: Example 2: City Street Fault Detection

The final roadway detected on the trip is the traffic state. As this state is triggered by low speeds and frequent stops. The detection of the state while traveling through downtown San Diego fits the designed requirements. As with the previous road states, the identification of the road reveals faults in the areas that were previously void. As this state is defined by low speeds, the faults that occur in the traffic state span short time periods and often have high magnitudes of acceleration. This allows for the detection of the two harsh brakes as well as the harsh left corner and the harsh acceleration as the vehicle enters the highway.

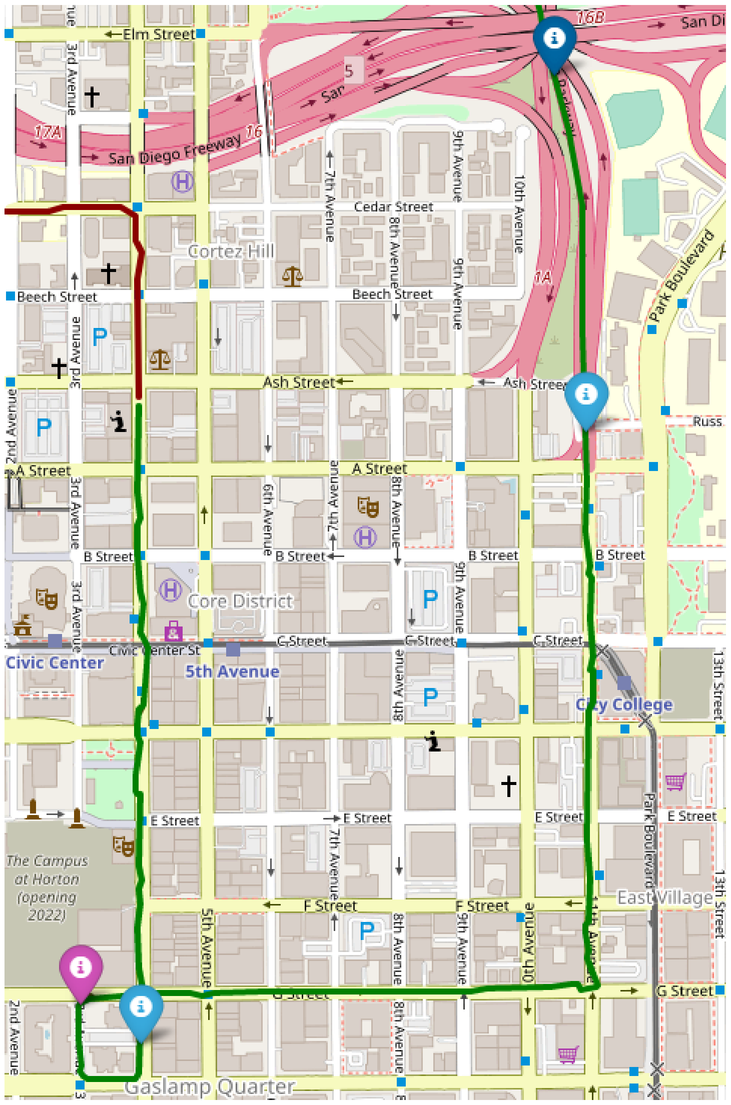


Figure 8.5: Example 2: Traffic Fault Detection

As shown in each of the three road states detected above. Faults are correctly detected in all road states. Tuning of the speeding and acceleration faults allow for the device to identify the correct fault profile within each road state. It is this addition of functionality that expands the current industry standard and allows the device to properly detect unsafe driving habits on the road. To provide further evidence that road state detection expands the scope of each fault, several trips are shown below. In each of the trips, faults are not intentionally created and are the result of the device being installed into an unaware driver's vehicle.

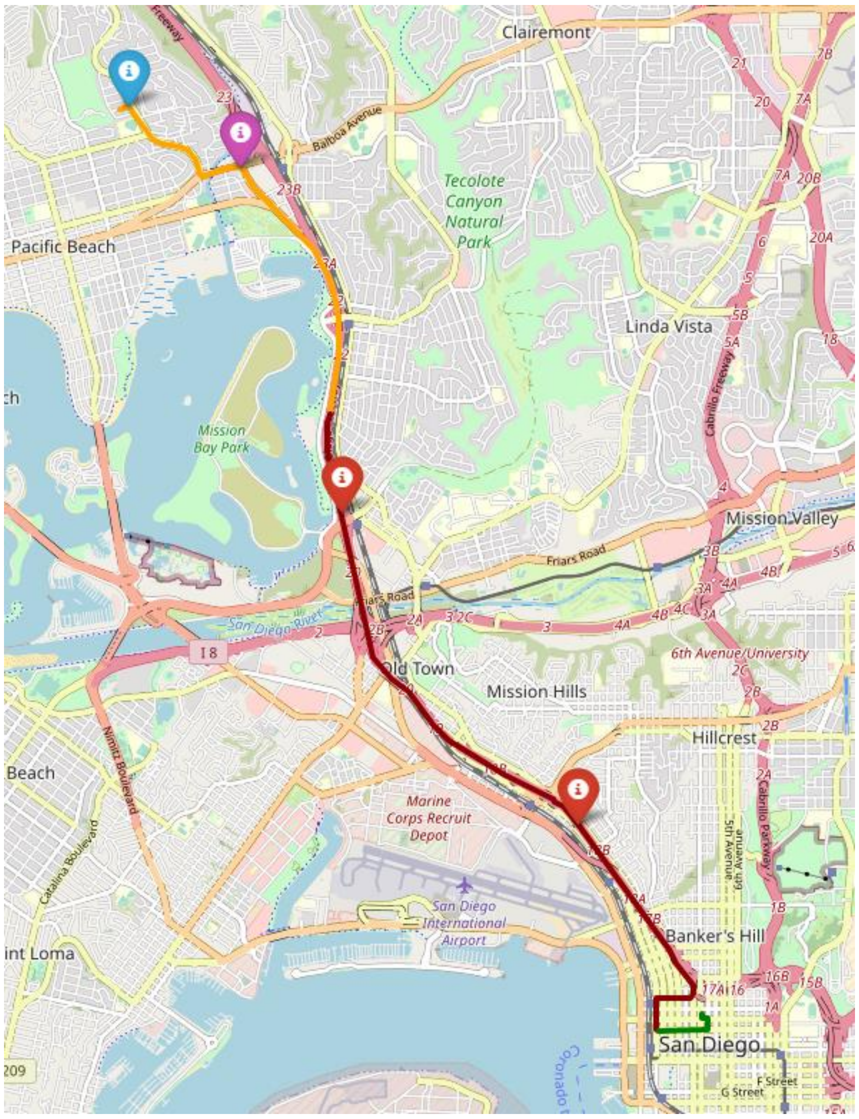


Figure 8.6: Example 3: Fault Detection with Road Class Estimation

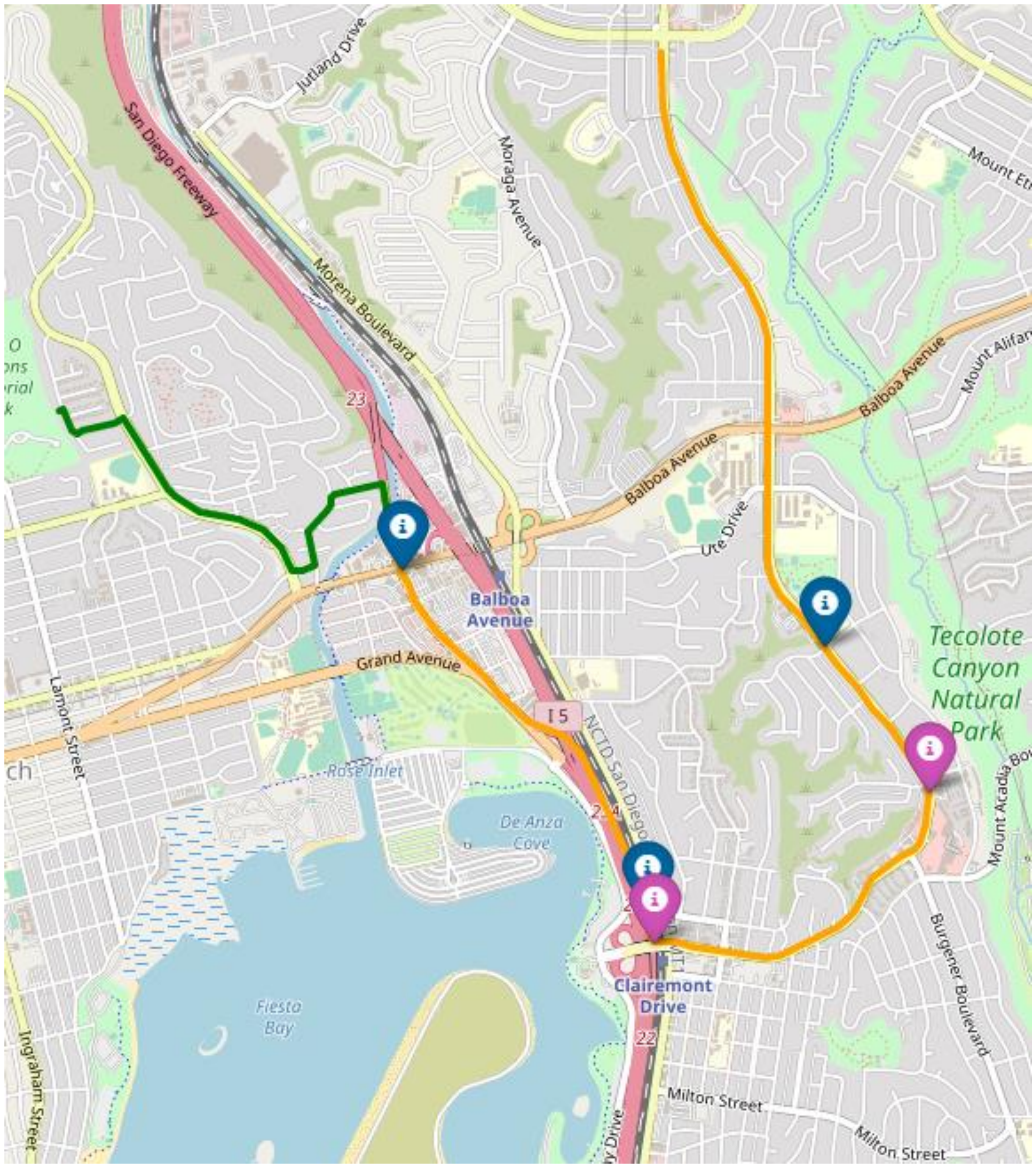


Figure 8.7: Example 4: Fault Detection with Road Class Estimation

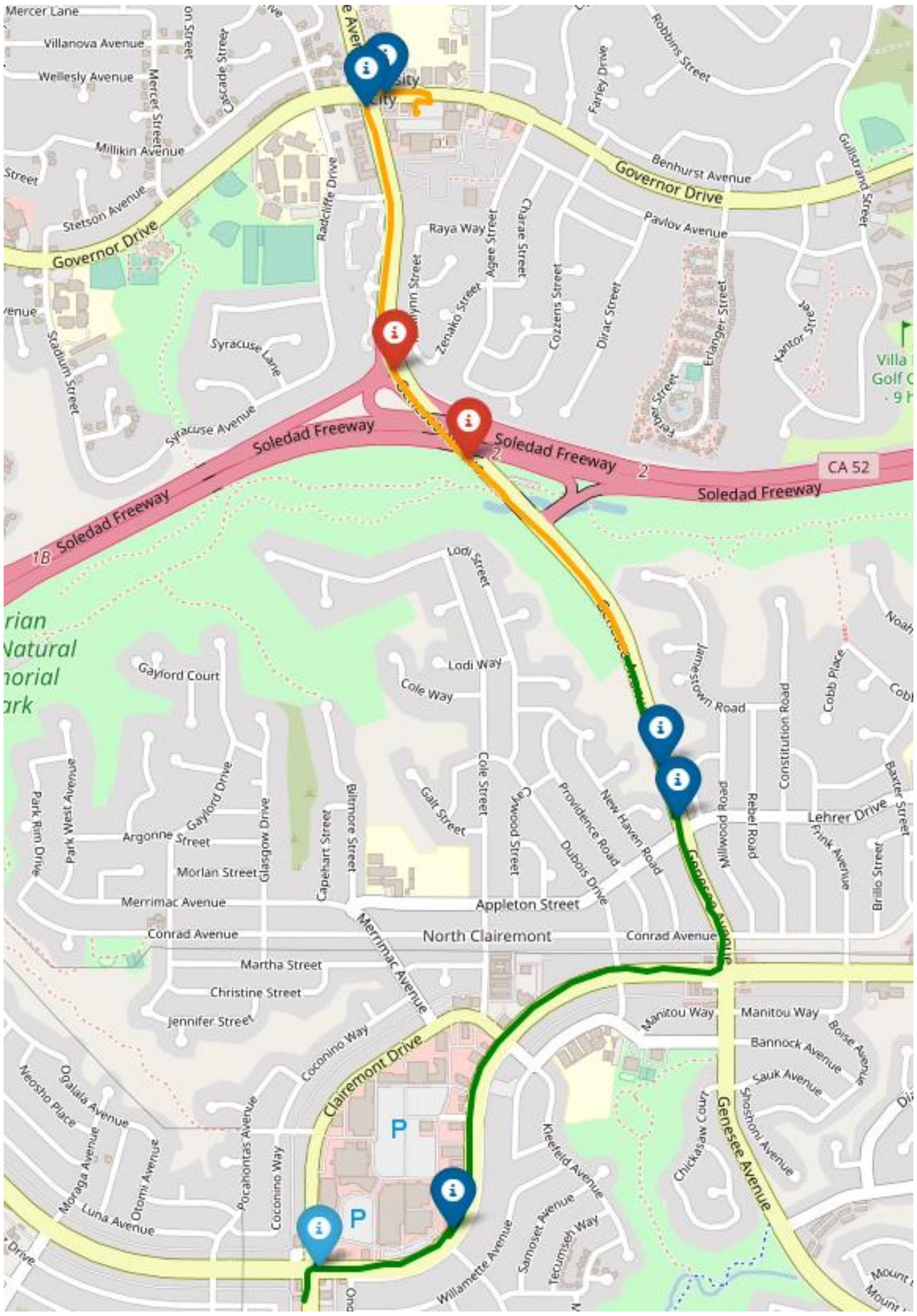


Figure 8.8: Example 5: Fault Detection with Road Class Estimation

Chapter 9: Conclusion and Future Work

9.1: Summary and Conclusion

A strong focus was placed on the design of a functional automotive tracking device for this thesis. The most commonly used sources of data were integrated to accomplish this goal. A device was built to monitor the movement and operation of a vehicle solely using only a GNSS chipset, accelerometer and two engine parameters. The data generated was leveraged to provide more information than initially intended.

Using just the information from the GPS location, the vehicle bus traffic is decoded for its RPM and speed parameters. The device was then able to detect the movement and speed of the vehicle. Using this method it was found that the decoding the vehicle bus was effective for the RPM parameter. The speed parameter was less reliable. As a result, the GPS was used as a fallback source for vehicle speed. The GPS speed is effective in measuring the velocity of the vehicle in most cases other than in an urban canyon.

The operation of the vehicle was able to be analyzed and unsafe driving could be detected by adding an accelerometer. By adopting the current industry standard for driver behavior, limits were set to identify unsafe driving practices. It was found that each driving fault was constrained to a specific type of roadway. Therefore, a method of automatically adjusting the fault limits for each roadway being traveled was created. The class of road being traveled could be determined by monitoring the length and number of stops as well as the average speed during each collection cycle. Defining the fault profiles while traveling along each road class, the fault limits could be tuned to each roadway, expanding upon the current method of fault detection used within the automotive industry today.

As faults were detected a method of transmitting the fault data was needed to circumvent the implementation and time needed to set up a backend server to receive faults. The fault data and any information on the device's operation was instead transmitted over SMS to an overseer's phone. The implementation of SMS also allowed for the device to accept commands from the user to request data or set key parameters that dictated the operation of the device.

With the successful initial design of the automotive tracking device, a method of monitoring basic vehicle operation was established. Then by expanding the awareness to the class of road being driven, a method of fault detection was created that exceeds the standard method of fault detection used on the market today.

9.2: Review of Research Goals

Table 9.1: Research Goal Outcomes

#	Research Goal	outcome
1	Design and build an IoT device that follows the current trends in the automotive IoT sector.	🟩

A device was successfully built to correctly perform all tasks related to initialization, data collection, vehicle state tracking, fault detection and communication. The hardware correctly tracks the current implementation used within the industry.

2	Develop a method of automatically decoding the vehicle bus for key parameters related to the operation of a vehicle.	🟨

An algorithm was built to decode the broadcasted messages. It was found that the RPM parameter was frequently decoded while the speed parameter was less successful. The source of error can be attributed to a lack of data on the bus or the larger number of scaling factors.

3	Evaluate the effectiveness of the industry method of fault detection	🟩

The standard method of fault detection was proven to be ineffective due to the use of static limits. While faults can occur on all roadways, speeding could only be found on highways and harsh braking and cornering were limited to surface streets.

4	Improve upon fault detection by detecting the type of road traveled and adaptively changing fault limits.	🟩

An algorithm for determining the category of road traveled was successfully created and new fault limits were determined by analyzing the fault profile generated at different speeds. Faults were then correctly detected along each category of road traveled.

9.3: Future Work

As the scope of this thesis was bounded to the development of an automotive safety device and the detection of driver faults, certain functional blocks that are commonly used within the industry had to be bypassed to ensure a timely completion. Therefore, several avenues of expansion are possible going forward.

The first area of future work is the conversion of the device from a Linux based prototype to an embedded system. Since the current implementation utilizes a Raspberry Pi to accomplish the goals of an IoT based device, the hardware used greatly overpowers the requirements. Rewriting the code in C and designing a custom PCB a more cost effective and streamlined device can be created.

The next are of expansion pertains to the usage of the cellular module. For this device to be functional in a commercial application, a system can be put in place so that the cellular module can be fully utilized. A backend server can be built or hosted so that the device can transmit data to a defined endpoint. By converting the fault data to binary, it can then be transmitted over the cellular network to the backend server for processing and storage. A user interface (UI) can then be developed to display this information in a human readable format for one or more device at a time.

The device can be further improved with the addition of other vehicle parameters. As the vehicle ECU contains data on every process within the vehicle, other parameters can be used to add functional blocks or expand existing ones. While this device used the broadcasted data to decode the information needed, the vehicle bus can be written using an existing command set. While not always supported, this method can be used to pull specific data from the vehicle when available.

The final area of improvement relates to the standardization of fault profiles. The limits used within this thesis serve as a first step in defining how faults change with the operation of the vehicle. As there is currently no published work on this topic, the profiles were collected by hand over many collection iterations. These profiles can be extended to include multiple data sources in an effort to standardize the many kinds of faults and how they can be detected.

REFRENCES

[1] Global status report on road safety 2018. Geneva: World Health Organization; 2018. License: CC BYNC-SA 3.0 IGO.

[2] "Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances." *Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances | Bureau of Transportation Statistics*, https://www.bts.gov/content/number-us-aircraft-vehicles-vessels-and-other-conveyances.

[3] Campbau, Todd. "An Automotive Minute, Average Age of Vehicles." IHS Markit, ihsmarkit.com/podcasts/automotive/average-age-vehicles-todd-campau.html.

[4] Thorton, Will. Selective Availability: A Bad Memory for GPS Developers and Users - Spirent. 24 Dec. 2019, www.spirent.com/blogs/selective-availability-a-bad-memory-for-gps-developers-and-users. Accessed 27 Apr. 2022.

[5] Dzhelekarski, P., & Alexiev, D. 2005. Initializing communication to vehicle OBDII system. ELECTRONICS, 5, pp. 21.

[6] On-Board Diagnostic (OBD) Regulations and Requirements: Questions and Answer. 2003. United States Environmental Protection Agency. License: EPA420-F-03-042

[7] "Controller Area Network (CAN) Overview - National Instruments." Www.ni.com, 1 Sept. 2020, www.ni.com/en-us/innovations/white-papers/06/controller-area-network--can--overview.html.

[8] Buscemi, Alessio, et al. A Data-Driven Minimal Approach for CAN Bus Reverse Engineering. 1 Feb. 2021.

[9] El Hakim, Ahmed. (2018). Internet of Things (IoT) System Architecture and Technologies, White Paper.. 10.13140/RG.2.2.17046.19521.

[10] Ltd, Arm. "What Are IoT Devices." Arm the Architecture for the Digital World, www.arm.com/glossary/iot-devices#:~:text=IoT%20devices%20are%20pieces%20of.

[11] Kasujee, Ibraheem, and Michele Mackenzie. "IoT Forecast: Connections, Revenue and Technology Trends 2021–2030." Analysys Mason, 25 Nov. 2021, www.analysysmason.com/research/content/regional-forecasts-/iot-worldwide-forecast-rdme0. Accessed 27 Apr. 2022.

[12] Elnashar, Ayman. IOT Evolution towards a Super-Connected World. www.arxiv.org/ftp/arxiv/papers/1907/1907.02589.pdf, 2019.

[13] Dorides, Carlo. GSA GNSS Market Report. issue 6 ed., Publications Office of the European Union, 2019.

[14] Federal Aviation Administration. "Satellite Navigation - GPS - How It Works." Faa.gov, 15 June 2015, www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/gps/howitworks/.

[15] Karaim, Malek, and Mohamed Elsheikh. "GNSS Error Sources." IntechOpen, 2018.

[16] Misra P, Enge P. Global Positioning System: Signals, Measurements and Performance. Massachusets: Ganga-Jamuna; 2010

[17] Klobuchar J. Ionospheric time-delay algorithm for single-frequency GPS users. IEEE Transactions on aerospace and electronic systems. 1987;(3):325-331

[18] Schuler T. On Ground-Based GPS Tropospheric Delay Estimation: Univ. der Bundeswehr Munchen; 2001.

[19] Borre K, Akos DM, Bertelsen N, Rinder P, Jensen SH. A Software-Defined GPS and Galileo Receiver a Single-Frequency Approach. Boston: Birkha¨user Boston; 2007

[20] Daneshmand S, Broumandan A, Sokhandan N, Lachapelle G. GNSS multipath mitigation with a moving antenna array. IEEE Transactions on Aerospace and Electronic Systems. 2013;49(1):693-698

[21] Xu G, Xu Y. GPS: Theory, Algorithms and Applications. Springer; 2016

[22] Wang, Zhi, et al. Multimedia Edge Computing. 2021.

[23] Raspberry Pi 3 Model B+ Specifications. www.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf

[24] "2 Channel CAN BUS FD Shield for Raspberry Pi." www.seeedstudio.com, 2020, wiki.seeedstudio.com/2-Channel-CAN-BUS-FD-Shield-for-Raspberry-Pi/

[25] "LPWA BG96 Cat M1/NB1/EGPRS." Quectel, 2021, www.quectel.com/product/lpwa-bg96-cat-m1-nb1-egprs/. Accessed 28 Apr. 2022.

[26] Raymond, Eric. "NMEA Sentences Manual." Opencpn.org, 2021, www.opencpn.org/wiki/dokuwiki/doku.php?id=opencpn:opencpn_user_manual:advanced_features:nmea_sentences.

[27] "DOT Analysis of Unsafe Driving." One.nhtsa.gov, 2017, one.nhtsa.gov/people/injury/research/aggdrivingenf/pages/introduction.html.

[28] Shinkle, Douglas. “Aggressive Driving and Speed.” Www.ncsl.org, 2017, www.ncsl.org/research/transportation/aggressive-driving-and-speed.aspx.

[29] National Highway Traffic Safety Administration. “NHTSA Speeding Overview.” NHTSA, 13 May 2019, www.nhtsa.gov/risky-driving/speeding.

[30] “Fatality Facts 2019: Yearly Snapshot.” IIHS-HLDI Crash Testing and Highway Safety, 2021, www.iihs.org/topics/fatality-statistics/detail/yearly-snapshot#when-they-died. Accessed 28 Apr. 2022.

[31] NWS Southern Region Headquarters. “What Is UTC or GMT Time?” Www.nhc.noaa.gov, www.nhc.noaa.gov/aboututc.shtml.

[32] “Signal Cycle Lengths.” National Association of City Transportation Officials, 2020, nacto.org/publication/urban-street-design-guide/intersection-design-elements/traffic-signals/signal-cycle-lengths/.