Custom Data Acquisition System for the Cal Poly Racing Baja Team

A Senior Project Report

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Joe Keenan

June 2025

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND 3

2.1 CAN 3

2.1.1 CAN 2.0 4

2.1.2 CAN FD 4

2.2 AEM 5

2.2.1 AEM Loggers 6

2.2.2 AEM Add-Ons 7

2.2.3 DBC Files 8

2.3 AiM 8

2.3.1 AiM Loggers 9

2.3.2 AiM Add-Ons 10

2.4 MoTeC 12

2.4.1 MoTeC Loggers 12

2.4.2 MoTeC Add-Ons 13

2.5 Conclusion 14

3. FORMAL PROJECT DEFINITION 15

3.1 Customer Requirements 15

3.2 Engineering Requirements 17

3.3 Use Cases 18
3.4 Conclusion 19
4. SYSTEM DESIGN AND IMPLEMENTATION 20
4.1 System Level Design 20
4.2 Hardware Design 25
4.2.1 Component Selection 25
4.2.2 Schematic Design 27
4.2.3 Layout and Routing 31
4.3 Software Design 35
4.3.1 High-Level Design and Implementation 36
4.3.2 Storage Task 39
4.3.3 Communication Task 43
4.3.4 Radio Communication Task 46
4.3.5 Sensors Task 49
4.3.6 Health Monitoring Task 53
5. SYSTEM TESTING AND ANALYSIS 55
5.1 SD Testing 55
5.2 Checksum Testing 56
5.3 Metadata Testing 57
5.4 CAN Testing 57
5.5 Radio Testing 59
5.6 System Testing 60
6. CONCLUSION 61
6.1 Design and Testing Conclusion 61
6.2 Future Works 62

7. REFLECTIONS 64

REFERENCES 66

APPENDICES

A. Code Implementations 68

LIST OF TABLES

Table		Page
3.1	*Engineering Requirements Table*	17
5.1	Checksum Unit Testing Results	56
5.2	CAN Testing Results	58
5.3	Radio Distance Testing Results	60

LIST OF FIGURES

Figure		Page
2.1	Example CAN 2.0 Frame [3]	4
2.2	AEM CD5L Dashboard	6
2.3	AEM thermocouple expansion unit	7
2.4	AIM MXS 1.3 Dashboard	9
2.5	AiM GPS09C GPS Module	11
4.1	CAN Bus Loading Analysis	21
4.2	File Metadata Byte Diagram	22
4.3	System Level Block Diagram	24
4.4	STM32 Pinout	28
4.5	USB Transceiver Wiring	29
4.6	CAN Transceiver Wiring	29
4.7	GPS I2C Wiring	30
4.8	Micro SD Reader Wiring	30
4.9	4 Layer Stackup based on JLCPCB Stackup	31
4.10	120Ω impedance profile for a 4-layer stackup	32
4.11	90Ω impedance profile for a 4-layer stackup	32
4.12	2D Layout of the top PCB	33
4.13	2D layout of the base PCB	34
4.14	3D assembly of the DAQ PCBs	35
4.15	Software tasks loop diagram	37
4.16	Shared Data Section Diagram	39
4.17	Storage State Diagram	40

4.18 SD Task Write State Block Diagram 43
4.19 Communication State Diagram 44
4.20 CAN Message Processing Block Diagram 46
4.21 Radio Communication State Diagram 48
4.22 Sensors State Diagram 51
4.23 Health Monitoring State Diagram 54
5.1 Dummy CSV file created on SD card 56

Chapter 1

INTRODUCTION

Baja SAE is an international collegiate competition run by the Society of Automotive Engineers (SAE) [1] where teams design, build, test, and compete with offroad baja-style vehicles. In the United States, there are three competitions each year across the country for teams to compete in. There are three categories of events for which teams are scored: static events, dynamic events, and the endurance event. Static events include challenges to test a team's ability to effectively communicate design choices and business aspects of making a vehicle. Dynamic events test the abilities of the vehicle to perform in different conditions, including an acceleration event, a maneuverability event, a suspension event, and a traction event. Finally, the endurance event tests the vehicle's and driver's ability to withstand rough terrain while racing against other teams' cars for four hours. At the end of the competition, all the event scores are added together to determine the top-three overall teams at the competition. To win the competition, it is crucial to perform well in all three categories of events.

Cal Poly Racing [2] competes in the Baja SAE series of competitions. Over the last several years, Cal Poly Racing has had moderate success, with several top three trophies in dynamic events. The Baja vehicle runs the series' only dually actuated electronic CVT, an electronically controlled hydraulic clutch for 4WD, and a custom dashboard. Additionally, the car runs with freewheels in the front, an aero package, and Genesis Racing shocks with a custom anti-roll bar.

To design a vehicle capable of withstanding the harsh events in a Baja SAE competition, it is critical to fully understand how every system of the car is behaving.

Understanding the load cases, such as impacts, is essential to making informed design choices and considerations. To understand these, a Data Acquisition System (DAQ) is needed to log and process sensor data.

In this senior project, a DAQ was created on the same platform as the rest of the vehicle that can be used to log data on the car from all control systems as well as sensor aggregation units. Additionally, the DAQ is able to transmit data wirelessly to monitor the vehicle state remotely. With a DAQ, design engineers for the team will be able to collect data to influence designs for future years and diagnose issues. Additionally, electrical engineers on the team will be able to design new sensor units that can be used in the future to collect even more data.

Chapter 2

BACKGROUND

To make informed decisions about what this DAQ should look like, it is important to investigate available commercial solutions. This can help to inform design choices and what features to add. It can also be helpful to determine the shortcomings of different commercial solutions so that this project does not repeat the same mistakes.

2.1 CAN

Controller Area Network (CAN) [3] busses, developed by Bosch in the 1990s, are commonly used in automotive applications to connect different control or instrumentation nodes. They allow for any node to communicate with any other node on the bus. CAN utilizes a two-wire asynchronous differential twisted pair signal to transmit across the bus. The asynchronous nature of this protocol reduces the number of wires required to transmit data. By utilizing a differential twisted pair, noise and interference are reduced, improving reliability and robustness of the network. However, only utilizing a single differential pair means that a node can only transmit or only receive at any given time, reducing throughput.

CAN is an addressed-based communication protocol. An address can correspond to a specific node or to a specific message. Since CAN only utilizes a single differential signal, it must negotiate to determine which node is transmitting and which nodes are receiving. The lowest address trying to be transmitted wins the negotiation, meaning that priority can be assigned to messages by assigning a lower value for an address to the message. This addressing and need to negotiate also adds overhead to

the transmission, reducing overall data throughput. Additionally, there are control bits, cyclical redundancy (CRC) bits, and end of frame (EoF) bits that contribute to overhead.

2.1.1 CAN 2.0

CAN 2.0 [4] is the most commonly used CAN protocol in the automotive industry. This version of CAN uses an 11-bit identification or address section with a maximum of 8 bytes of data transmitted. The bitrate for this version of CAN can be up to 1 Megabit per second (Mbps). With an assumption of 1 byte of data transmission, the overhead can be computed as $\frac{48\text{bits}}{56\text{bits}}$. With the assumption of 8 bytes of data transmission, the overhead can be computed as $\frac{48\text{bits}}{112\text{bits}}$. The more data that is transmitted per frame, the less overhead impacts the total data throughput. An example data frame of CAN 2.0 can be seen in fig. 2.1.

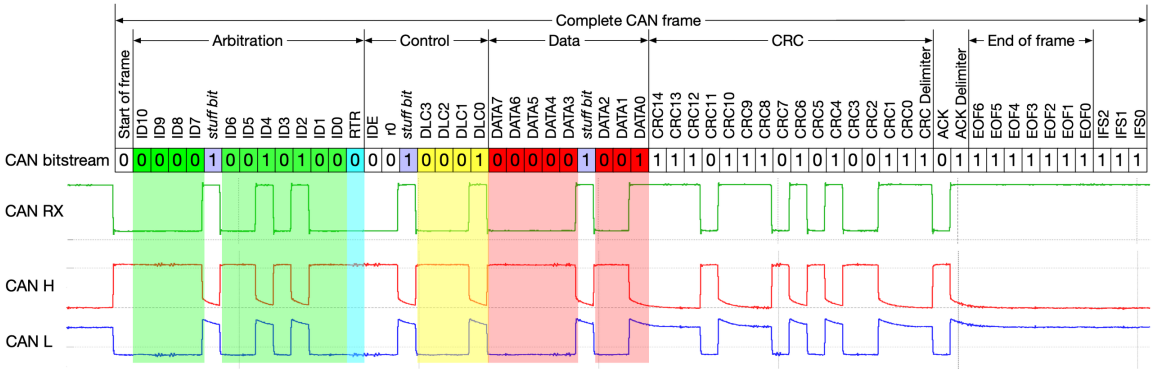


Figure 2.1: *Example CAN 2.0 Frame [3]*

2.1.2 CAN FD

In 2012, Bosch released a new version of CAN with a flexible data rate called CAN FD[5]. This new version of CAN has several improvements including faster data rates and allowing for more data to be transmitted in each frame. The flexible data rate

comes from CAN FD's ability to increase the data rate during the data section of the transmission. The protocol supports up to 8 Mbps of throughput during the data section of transmission and up to 1 Mbps of throughput during the beginning and ending parts of the frame, allowing for significantly more data throughput overall. In addition to the flexible data rate used by CAN FD, the maximum size of the data section increased from 8 bytes to 64 bytes. This decreases total overhead significantly, making the transmission more efficient. These two characteristics combined provide for a much more efficient transfer of data when transmitting large amounts of data very frequently.

To use CAN FD, it must be supported by the hardware, including both the CAN controller and the CAN transceiver. Additionally, to rely solely on CAN FD, each node on the bus must be using CAN FD. Otherwise, conflicts may occur, impacting frames being sent or received. If there are both a CAN FD node and a CAN node on the same network, the system would operate without too many issues, but the CAN node would not be able to read any CAN FD messages.

2.2 AEM

AEM [6] designs data loggers and dashboards for testing purposes for cars and other vehicles. These data loggers connect to different modules that connect to various CAN sensors and breakouts. Some of these modules also come with an inertial measurement unit (IMU) and a global positioning system (GPS) which are useful for understanding the entire vehicle state.

2.2.1 AEM Loggers

There are several different AEM loggers available for purchase. The two major loggers available for purchase are the AEM CD-5L and the AEM CD-7L. The CD-5L costs $1,574.95 without the IMU or GPS and the CD-7L costs $2,081.95 without the IMU or GPS. This cost provides access to data logging features as well as a dashboard display as seen in fig. 2.2. These dashboards can be configured to display different information to a driver or engineer to get a snapshot of the current state of the vehicle. A computer can be easily connected to these devices to upload different configurations as well as to download the logged files.

Figure 2.2: *AEM CD5L Dashboard*

The loggers log files in a comma separated file (csv) format for ease of use with tools such as Matlab or Python. These files can be visualized on the AEM dashboard as well to give a quick glimpse at explaining what is happening.

2.2.2 AEM Add-Ons

To connect more sensors or instrumentation to the AEM loggers, additional add-ons must be purchased and configured. Some of these add-ons include a CAN module which allows the AEM logger to connect to and interface with a CAN bus, and a thermocouple unit shown in fig. 2.3 for measuring temperature. These expansion units, before the cost of any sensors, are several hundred dollars. To hook up any sensors, at least one of these expansion units must be purchased.

Figure 2.3: *AEM thermocouple expansion unit*

AEM also sells some sensors that are simple to integrate into their ecosystem. These sensors include thermocouples, air and water temperature sensors, and several styles of pressure transducers. Each of these sensors is somewhat expensive on their own, costing around $100 per individual sensor. The AEM loggers can also interface with traditional analog or digital sensors so long as the appropriate add-ons are purchased.

In addition to wired sensors, AEM is also compatible with some wireless sensors for things such as tire pressure monitoring systems (TPMS) or fuel sensors. To communicate with these wireless sensors, a wireless module or a logger with wireless

capabilities is needed. AEM utilizes a proprietary system called X-Wifi to connect to these wireless sensors, that are already a part of their CD5 and CD7 loggers. AEM's sensors that can utilize the X-Wifi capabilities of the logger are priced similarly to their wired sensors.

2.2.3 DBC Files

In addition to AEM specific sensors, the AEM data loggers can interface with any custom hardware that is outputting its data over CAN. To configure the AEM to read data from a CAN bus, including its own CAN-capable sensors, a configuration file must be made to tell the logger which value corresponds to which bytes of a specific CAN frame. A CAN Database file (.dbc) is a standardized file format for describing what information is contained within a CAN frame. An example .dbc file can be seen in listing A.1 in the appendix. AEM utilizes this file format for its data loggers to interface with a CAN network so long as the proper hardware add-ons are purchased.

2.3 AiM

Another company that makes data loggers to attach to a vehicle is AiM Sports [7]. AiM makes data loggers that are specifically intended for motorsport applications as well as other products such as dashboards, steering wheels, and ECUs. Similarly to AEM, AiM also makes combined dashes and loggers, as well as different accessories and sensors for their dashboards and loggers.

2.3.1 AiM Loggers

AiM develops several loggers with similar specs to the AEM loggers with some additional features. One such logger, the MXS 1.3 is shown in fig. 2.4. The MX series and MXM series loggers are comparable to the CD5 and CD7 logger dashes that are sold by AEM. These loggers also act as dash displays, with the MX series having a nicer display than the MXM series. The MX series and the MXM series both support many different ECU communication protocols over CAN or RS-232 as well as several different direct sensor inputs.

Figure 2.4: *AIM MXS 1.3 Dashboard*

AiM also develops loggers that do not serve as a dash, only containing the functionality for aggregating and logging data. These loggers offer most of the same features that the dash loggers offer, with CAN connectivity, wireless connectivity, and different direct sensor inputs. There are two main products offered in this line, the ECULog and the XLog. The ECULog is the cheaper version with fewer features,

only supporting external sensor inputs and communication. The XLog includes all of this and an onboard IMU and GPS, making it easier to monitor vehicle position and behavior during a lap.

2.3.2 AiM Add-Ons

To communicate with sensors, there are a few required accessories needed. These are a data hub that can be used to communicate with different accessories and a channel expansion to provide more ports for connecting different sensors to the CAN network. The channel expansion ports provide power and ground signals, as well as an analog sensor input for reading the sensor data. The data hub just acts as a way to reduce the number of devices directly connected to the logger itself. These devices cannot be purchased directly from AiM and instead must be purchased from a retailer like Summit. On Summit's website, the data hub is $77, and the channel expansions are $288, making them somewhat expensive requirements for using sensors.

Besides these essential add-ons, there are some other useful add-ons. AiM makes a GPS add-on shown in fig. 2.5 that can provide location data, as well as a thermocouple hub that can be used similarly to the thermocouple add-on that AEM makes. They also make a storage expansion option, to increase the total amount of data that can be stored on board. None of these add-ons are required to utilize the system and to log data, but they are useful tools that improve the system and provide some useful functionality.

Figure 2.5: *AiM GPS09C GPS Module*

In addition to the hardware accessories to improve the experience and connectivity of AiM data loggers, AiM also sells different kinds of sensors for automotive applications. These include temperature sensors, pressure sensors, combined temperature and pressure sensors, position sensors, speed sensors, and rpm sensors. Most of these sensors have analog voltage outputs, meaning that to communicate with the data logger, these sensors must be connected to the logger directly or to a unit that can transmit the data over CAN or other supported communication protocols. These sensors, like the other add-ons and loggers, cannot be purchased directly from AiM and must be purchased from a third-party retailer. The pricing on sensors is similar to the pricing of similar sensors sold by AEM.

AiM utilizes a tool called CAN builder to set up the logger to properly handle and process CAN inputs. This tool is essentially a graphical interface for configuring the logger to properly read CAN messages. CAN builder is capable of interfacing with AiM's proprietary XC1 file types, the standard DBC file types, or building a custom driver for reading CAN messages. The CAN builder tool can only be used on AiM devices that are supported by their Race Studio 3 software.

2.4 MoTeC

A third company that makes data logging solutions for automotive application is MoTeC [8]. MoTeC is an automotive aftermarket company owned by Bosch that focuses on making electronic equipment for race sport uses. They make Electronic Control Units (ECUs), data loggers, dashboards, Power Distribution Modules (PDMs), and several other products that can be used to monitor or improve the performance of vehicles. These products are similar in spec to the products made by AEM and AiM.

2.4.1 MoTeC Loggers

Similarly to AEM and AiM, MoTeC offers both independent logger units as well as combined dash and logger units. These loggers have similar specs to the AEM and AiM loggers, with the basic functionality of CAN channels, RS-232 channels, analog sensor inputs, digital sensor inputs, and the ability to use expansion units to increase the total amount of available connectivity. The dashboard options are very similar to the dashboards available from AEM and AiM.

There are three available options that support the full range of connectivity for a combined dash and logger and three additional options for just the logger. The cheapest of the dash and logger combined units, the C125, costs approximately $2,200 from third part retailers before any additional add-ons are acquired. The cost of the cheapest logging only option, the L120, costs approximately $1,800 from third party retailers. These prices are higher than the previously discussed AiM and AEM options.

One of the main differences between this option when compared to the AEM or AiM loggers is that none of the available loggers or combined dash and loggers have telemetry available. To add telemetry data, an additional fee or upgrade is required to add the MoTeC T2 Telemetry features.

2.4.2 MoTeC Add-Ons

Like the AEM and AiM, MoTeC also makes different devices and sensors to compliment their data loggers to improve the overall functionality and compatibility. MoTeC makes an expander unit, similar to the CAN expander made by AiM called the SVIM that can collect several different analog or digital sensors and connect to the CAN network for the MoTeC. Additionally, MoTeC makes an expansion unit called the E888 that has similar features the SVIM but includes thermocouple specific inputs. This unit additionally is more limited on which loggers it can connect to, only being compatible with the SDL3 logger and the ACL logger.

In addition to the expansion units, MoTeC also makes various sensors that can be connected to the loggers or expansion units. Similarly to AEM and AiM, they make thermocouples, various temperature sensors, pressure sensors, and speed sensors. Additionally, they also make linear and rotary potentiometers for applications

such as steering angle measurements or suspension travel measurements as well as an IMU. Just like AiM, these sensors are not sold directly by MoTeC and must instead be purchased from a third-party retailer. These sensors are somewhat expensive but comparable in price to the sensors made by AEM and AiM.

2.5 Conclusion

When comparing CAN and CAN FD, CAN FD is superior in almost every way. The main drawback to using CAN FD is that many sensors that can be purchased that are designed to integrate with CAN data loggers use CAN 2.0, not CAN FD. A CAN FD node should be able to read a CAN 2.0 frame if the data bitrates are the same. This is often not the case, however, as one of the major benefits of CAN FD is the higher data bitrate, this is often not the case.

The biggest issues that will come up with any of the OEM loggers is the throughput of data on the CAN bus. All three of the OEM loggers use CAN 2.0, limiting the frame data section to 8 bytes and the data bitrate to 1 Mbps. Additionally, none of the OEM solutions investigated included onboard sensors for vehicle dynamics. They all support add-ons for an IMU and GPS, but it is an additional cost on an already expensive investment. On a similar vein, expanding the OEM loggers to support more sensors is expensive.

Since CAN FD is preferred and the cost of the OEM loggers and their add-ons are very high, a custom solution is the desired approach. With a custom DAQ, onboard sensors can be added, and long-range wireless functionality that would not exist in an OEM logger without add-ons can also be added.

Chapter 3

FORMAL PROJECT DEFINITION

The aim of this project is to make a fully functional and simple to use data acquisition system for the Cal Poly Racing Baja SAE team for monitoring controls information and logging sensor data on the CPx25 vehicle and for use in future vehicles. This includes the hardware unit, the software and firmware required to run the system, and tools for interfacing with the system.

3.1 Customer Requirements

The customer for this project is the Cal Poly Racing Baja Team. As the Electrical Technical Director of this team for this year, I can also be considered the customer for this product and am responsible for defining the customer requirements. These requirements come from a combination of the existing systems on the car, desired data, the ability to expand the desired sensors, the programming and data collection experience of the team, and reusability.

The customer requirements are as follows:

- A data storage rate of at least 400 Hz
- Enough storage to be able to store at least 5 hours' worth of data
- The ability to download files
- A long-range wireless radio to monitor data periodically
- The ability to add or remove different sensors or data modules

- An IMU and GPS to monitor vehicle dynamics
- The ability to directly wire some sensors to the DAQ unit
- The data must be in a format that can be easily viewed
- Less than 7.5W

The storage rate comes from the control loop rate of the fastest controller on the vehicle. Since there are no current plans to increase the controller frequencies or add any sensors that require a faster sampling rate, this is the minimum required collection rate. The endurance event at a Baja event lasts for four hours. Adjusting for time spent waiting for the race to start and any final checks just before the event, five hours should accommodate a full endurance days' worth of data. To quickly diagnose issues on the vehicle, knowing what is happening before the car arrives back in the pits is useful. Downloading files directly off the DAQ has been an issue for several years and significantly delays the processing of data. Additionally, it was often the case that the pit crew would forget to add the storage device back onto the system, and no data would be collected until the system was reopened to analyze data again. Being able to download a file over USB or another interface would remove the need to take the SD card out of the system. The ability for the system to be modular and add other sensors is also important. It is often the case that there is a sensor that is wanted for a single test but should not be left on the vehicle permanently. For a similar reason, being able to directly wire a sensor to the DAQ is useful. Having an onboard GPS and IMU are critical sensors for monitoring vehicle dynamics. Finally, since most of the team do not have experience working with custom electronics, it is important that all data is easily accessible and shareable.

3.2 Engineering Requirements

From the customer requirements, engineering requirements can be derived. These requirements reflect the technical specifications that the rest of the system will be designed for.

Table 3.1: *Engineering Requirements Table*

Spec. Number	**Metric**	**Requirement**	**Tolerance**	**Risk**	**Compliance**
1	Data Storage Rate	400 Hz	Min	M	A, T
2	Storage Capacity	4 GB	Min	H	I
3	IMU Sampling Rate	200 Hz	Min	M	T, S
4	GPS Sampling Rate	10 Hz	Min	M	T, S
5	Radio Distance	2 miles	Min	L	T
6	Power Draw	5 W	Max	H	A, T
7	CAN Busses	2	Min	L	I
8	ADC Resolution	10 bits	Min	L	I

These engineering requirements are mostly tied to the customer requirements regarding the data storage rate and the desire to understand the behavior of the car's dynamics. Additionally, the length of an endurance event was used to derive several engineering requirements as well. The rest of the customer requirements come down to design choices and less down to numbers.

3.3 Use Cases

There are three main cases of use for this system:

(1) To validate design models

(2) To monitor vehicle health during drives at testing and competition

(3) To diagnose unexpected failures of a system

An example of the first case would be an engineer attempting to validate a brakes model for the car. The model has 9 degrees of freedom model using brake pressure, shock heights, vehicle acceleration, and other parameters to determine when the wheels will "lock" or break traction. To validate the model for our design, these sensors' data is needed. This is the case for several different vehicle dynamics models used to design the baja car.

An example of the second case would be when the car is driving in the endurance event of a competition. During an endurance event, the car will be driving continuously for several hours without the ability to regularly inspect parts or systems for failure. To maintain an understanding of the vehicle's health and critical systems during this event, we can view wirelessly transmitted data. This helps the pit crew know if or when to expect the vehicle is coming back to the pits for a repair.

An example of the third case would be we are at testing or at competition, and something breaks or starts performing poorly. To properly diagnose the root cause of an issue, knowing what was happening immediately preceding a failure can be useful or even critical. In many instances, the car or the component may not have been visible to the tester, and the driver may not be able to diagnose the issue on their own. In this case, having access to sensor data that can be used to extrapolate what

the vehicle was doing prior to failure can help an engineer fix or improve upon their design to reduce the risk of failure in the future.

3.4 Conclusion

Based on the wants and needs of the team, the most important criteria is that data can be logged at the frequencies needed to evaluate models. Following this, the data must be easily accessible to as many people as possible. This includes the ability to easily remove data from the DAQ. After this, the data needs to be accessible, even when the car is not, to monitor health and diagnose issues more quickly during testing and competition. Lastly, having access to IMU and GPS data onboard is useful for many different models. With the primary constraint being time, this priority list was able to help inform choices about what features to focus on and what features to delay.

Chapter 4

SYSTEM DESIGN AND IMPLEMENTATION

The bulk of this project was in the system design and implementation. This section details considerations, design choices, issues encountered, and how each sub component interacts with other subcomponents.

4.1 System Level Design

To accomplish all the tasks necessary for this DAQ, several design choices were made at the system level. These include decisions about what microcontroller was being used, the type of data being stored, the media device that data is being stored on, how other sensor or control nodes will communicate with the DAQ, what onboard sensors are on the DAQ, how or if the DAQ can communicate wirelessly, how a user can interface with the DAQ, and how firmware updates can be applied.

The first thing considered was how nodes can communicate with the DAQ. The previous DAQ solution used CAN communication with a 1 Mbps data rate to collect data from all nodes on the car. This became a throughput bottleneck, and lower priority CAN IDs would occasionally be blocked by high throughput, higher priority data. The robustness of the CAN bus was very nice and allowed for reliable communication between all nodes on the car, so newer CAN interfaces were primarily explored. There are currently 20 sensors and over 100 bytes of internal controls information worth of desired data during competition and testing that are sent from three different nodes in seven different messages. The majority of this data is being transmitted at 400 hz. Additionally, during testing season, there are an additional 20 sensors that are used

to help validate various models and analyses. Worst-case CAN loading for different data rates was plotted for both CAN and CAN FD in fig. 4.1 to determine what data rates are acceptable. To keep the robustness of CAN and to improve issues related to throughput, CAN FD with an arbitration rate of 1 Mbps and a data rate of 5 Mbps was selected to leave room for more future sensor or controller additions.

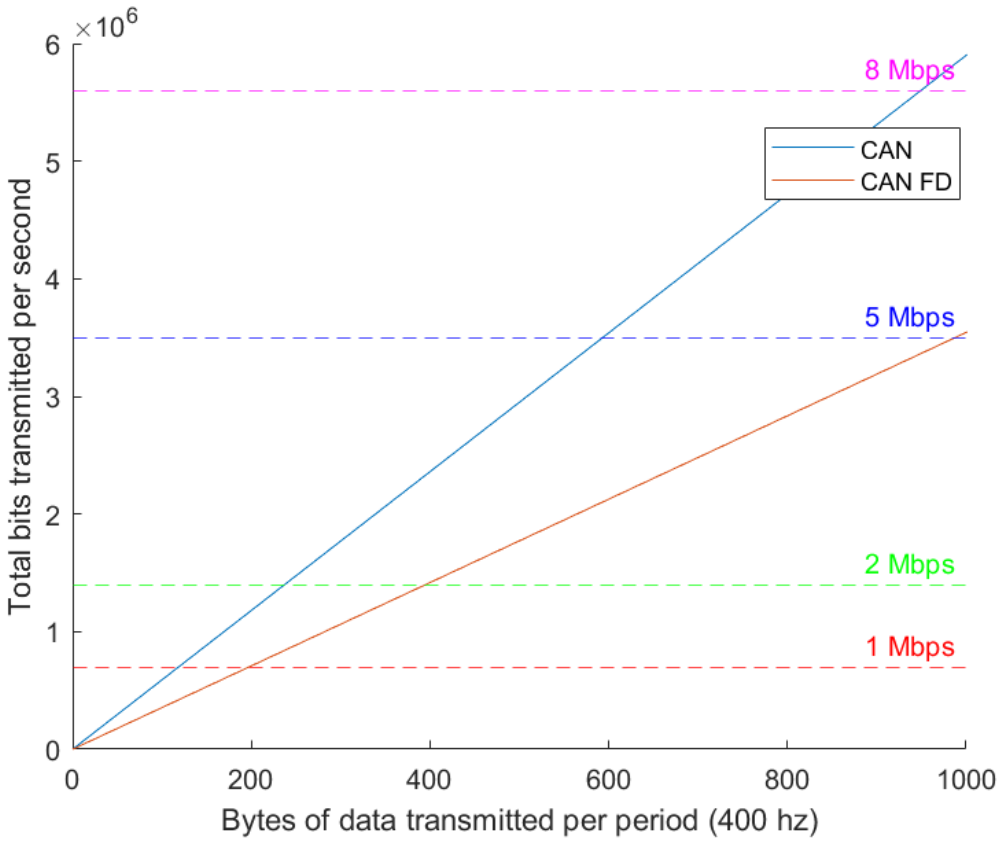


Figure 4.1: *CAN Bus Loading Analysis*

The next consideration was how data is stored on the DAQ. The previous DAQ solution utilized the onboard micro-SD card reader on the Teensy 3.5 [9] and Teensy 4.1 [10] microcontrollers and stored data in a CSV format. The micro-SD card worked well with the small form factor, large number of resources available, common communication support with SPI or SDIO, and sufficient storage for 10 hours of data collection. An alternative option would be to use an SSD. This would allow for larger

storage volumes but would add significant cost to implementation complexity, mass, physical volume, and power consumption. As a result, a micro-SD card was selected as the storage medium.

With an SD card as the storage medium, the maximum amount of data stored is the largest constraint. In a CSV file, each digit of a number is represented as a character, meaning that in order to represent the number "100", three bytes are required in addition to the comma delineator. This same number can be represented with just a single byte if the data is stored in a binary format. Knowing that the total amount of data stored during competition is nearly 200 bytes at 400 Hz, and the worst-case values for 1-byte values being three characters, 2-byte values being 5 characters, and 4-byte numbers being ten characters, the amount of data stored over 10 hours would be 6.6 GB. As a result, the data is stored in a binary format described in fig. 4.2.

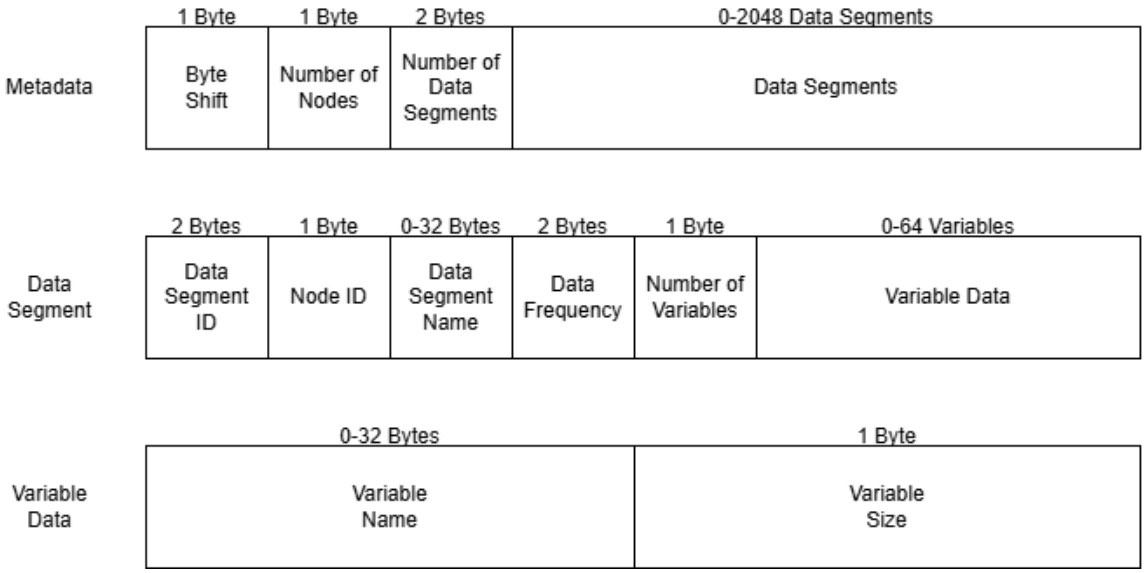


Figure 4.2: *File Metadata Byte Diagram*

To be able to parse this data, the data contained in the file is outlined at the beginning of the file, including information about data group IDs, the number of variables per ID, the size of each variable, and the name of the data associated with

each variable as described in fig. 4.2. This header is generated dynamically at runtime, so that if new data is added to the list of logged values, DAQ firmware does not need to be modified to accommodate new data. Each data segment is defined by the CAN message that will transmit the data. Due to this, each data segment is a maximum of 64 bytes in size.

The selected microcontroller was the STM32H750VBT6 from ST [11]. This microcontroller was selected for several reasons, including a vehicle-wide platform switch to STM32 microcontrollers, internal CAN FD controllers, SDIO connectivity, and a 480 MHz core frequency. The platform switch is driven by three main criteria:

- The ability to lay out the chip on a PCB
- Having access to the debug pins for programming and debugging
- Large amounts of community support and open-source libraries

Teensy development boards utilize NXP chips that have most of the same features as STM32 chips, but have slightly less community support. Additionally, STM32 microcontrollers are used in EE 329 and CPE 316, providing experience on the platform for students who have taken those courses.

Something that all off-the-shelf data loggers do is provide the ability to connect some analog or digital sensors directly to the logger. This functionality is desirable as it allows for a sensor to be added without external electronics, so long as the sensor output is supported by the logger. To include similar functionality, 4 analog sensor channels are available to be directly plugged into the DAQ. This includes a 5V source and a GND for each. Additionally, there is an onboard IMU and GPS to be used for the common measurements of acceleration and velocity. These sensors are on the PCB itself and require no additional wiring by a user.

The final major piece of functionality provided by the DAQ is the ability to wirelessly transmit data across a range of up to 2 miles. This is to provide live diagnostics during testing or competition to the crew in the pits to help diagnose issues and reduce time debugging if issues occur. To do this, a radio module was added. To comply with FCC and competition regulations, the main options for frequency ranges were 900 MHz, 2.4 GHz, and 5 GHz. Since competition and testing sites are often hilly and full of trees, higher frequency nodes struggle to transmit over long distances reliably. This drove the desire to utilize a 900 MHz radio module. The XBee 3 Pro was selected due to its wide community support and ease of integration. Figure 4.3 describes the basic way in which these components all interface together.

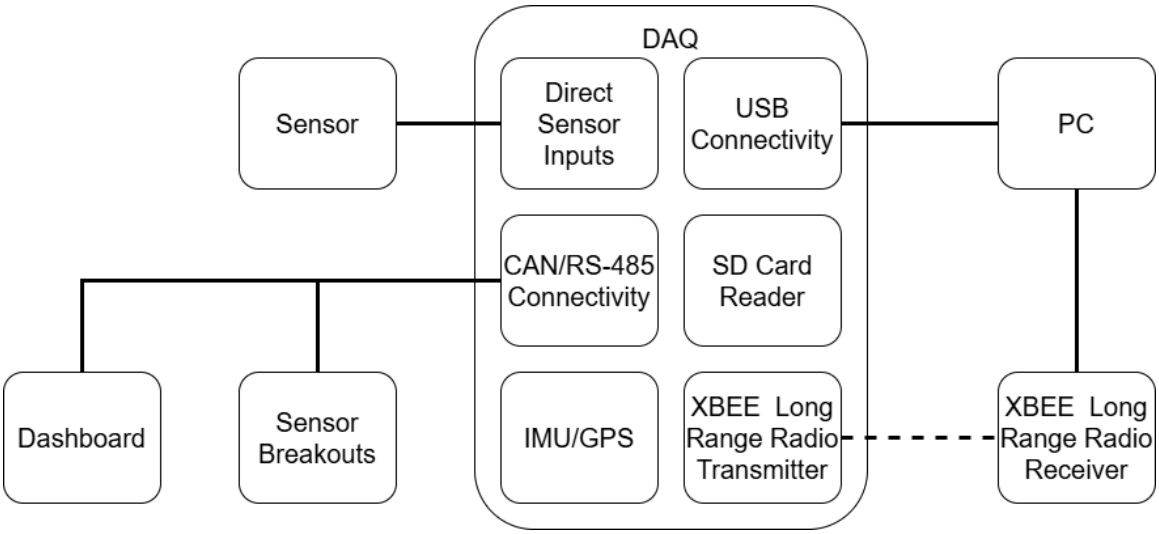


Figure 4.3: *System Level Block Diagram*

The last major design choice was to utilize the Arduino framework for writing the firmware for the DAQ. This was chosen to reduce the time required to write and test firmware for interfacing with the hardware. Specifically, configuring the peripherals for CAN and SDIO is complex and would have taken development time away from higher level software features. To use the Arduino framework, the PlatformIO [12]

extension from VSCode [13] was used to setup the project. STM32 microcontrollers utilize STM32Duino [14] as the source code for the Arduino integration.

4.2 Hardware Design

The hardware design, except component selection for the microcontroller, onboard IMU and GPS, and CAN transceiver, was largely outside the scope of this project and was completed by other members of the competition team. As the most senior member of the electronics team and the Electrical Technical Director, though, part of my responsibilities to the project included design reviews and feedback on schematic design, layout, and routing.

As with any embedded systems project, the code for the project depends on the hardware design. Different peripherals of the microcontroller being used and sensors connected drive different parts of the firmware implementation. The specific peripherals that are heavily utilized are CAN FD, UART, I2C, SDIO, and ADC.

4.2.1 Component Selection

The first major component selected was the STM32H750VBT6 which was discussed in the system design section. With this microcontroller, there was access to all the desired peripherals for all the necessary communication. This microcontroller does come in a somewhat large package, being a QFP100 chip. The pins are all 3.3V tolerant, meaning that 5V analog sensors need to be lowered to have a range that does not exceed 3.3V, and all digital IO of peripherals need to operate on 3.3V. There are more pins available than needed, leaving the potential for adding additional functionality in future years as new features are desired.

To communicate with the CAN bus, a CAN FD transceiver is required. The CAN controller exists internally to the microcontroller, but it does not include a CAN transceiver internally. The CAN transceiver needs to support at least a 5 Mbps data rate to support the CAN throughput discussed in the system design section, and needs to support the 3.3V IO voltage of the microcontroller. The TCAN1043ADYYRQ1 chip from Texas Instruments [15] was selected. This chip can support up to a 8 Mbps data rate and can be configured to use 3.3V for all the digital IO.

Similarly, a USB-UART transceiver was selected to support a USB connection from a user to the DAQ. Just like the CAN transceiver, the digital IO that physically connects to the microcontroller must support a 3.3V voltage. The FT231XS-R chip from FTDI [16] was selected due to its popularity and large amount of documentation. Additionally, this chip includes an EEPROM that contains information about the device. This can be programmed to include identifying information about the device.

The GPS selected was the NEO-M9N-00B module from u-blox [17]. This module was selected because it was utilized in a prior senior project and supports several different communication protocols [18]. That senior project focused on creating an all-in-one solution for tracking vehicle orientation and heading using an IMU, GPS, and magnetometer. This also included code for interfacing with these sensors. This senior project was also a useful reference for good practices for routing the RF signal from the GPS to the antenna.

A different IMU than the one from the prior senior project was selected. The main criteria for the IMU were that it had community support for interfacing with it, could operate with IO voltages of 3.3V, and could measure ± 3 Gs of acceleration. The peak possible acceleration is approximately 10 Gs and occurs when the vehicle stops immediately due to a crash or a large, unmovable obstacle. According to the

mechanical team, the vehicle should not experience accelerations of more than 3 Gs under typical operation. The LSM6DSLTR IMU from ST [19] fits all these criteria.

The final major component selected was the micro-SD card reader. A push-in/push-out reader was desired as it helps apply mechanical retention to the SD card. The vibrational load on the system during operation is non-negligible, so it is important to mechanically retain the SD card to improve the overall reliability of the system. The 2201778-1 micro-SD card reader from TE Connectivity [20] was selected due to TE Connectivity's high amount of reliability and meeting all the other needs of the component.

4.2.2 Schematic Design

As mentioned above, the schematic design was largely out of the scope of this project. Despite this, there are a few important details regarding the schematic design that are important to cover due to how they impact the firmware design and reliability of the system.

The first important aspect of the schematic design is the pinout used for the STM microcontroller. This pinout can be seen in fig. 4.4 and was generated and validated by utilizing the STM32CubeIDE IOC tool and double-checking with the data sheet for the chip. Throughout the testing process, there were a few pinout issues which were discovered that delayed software testing of some key components, such as the SD card reader. These issues were eventually resolved with a second revision of the board.

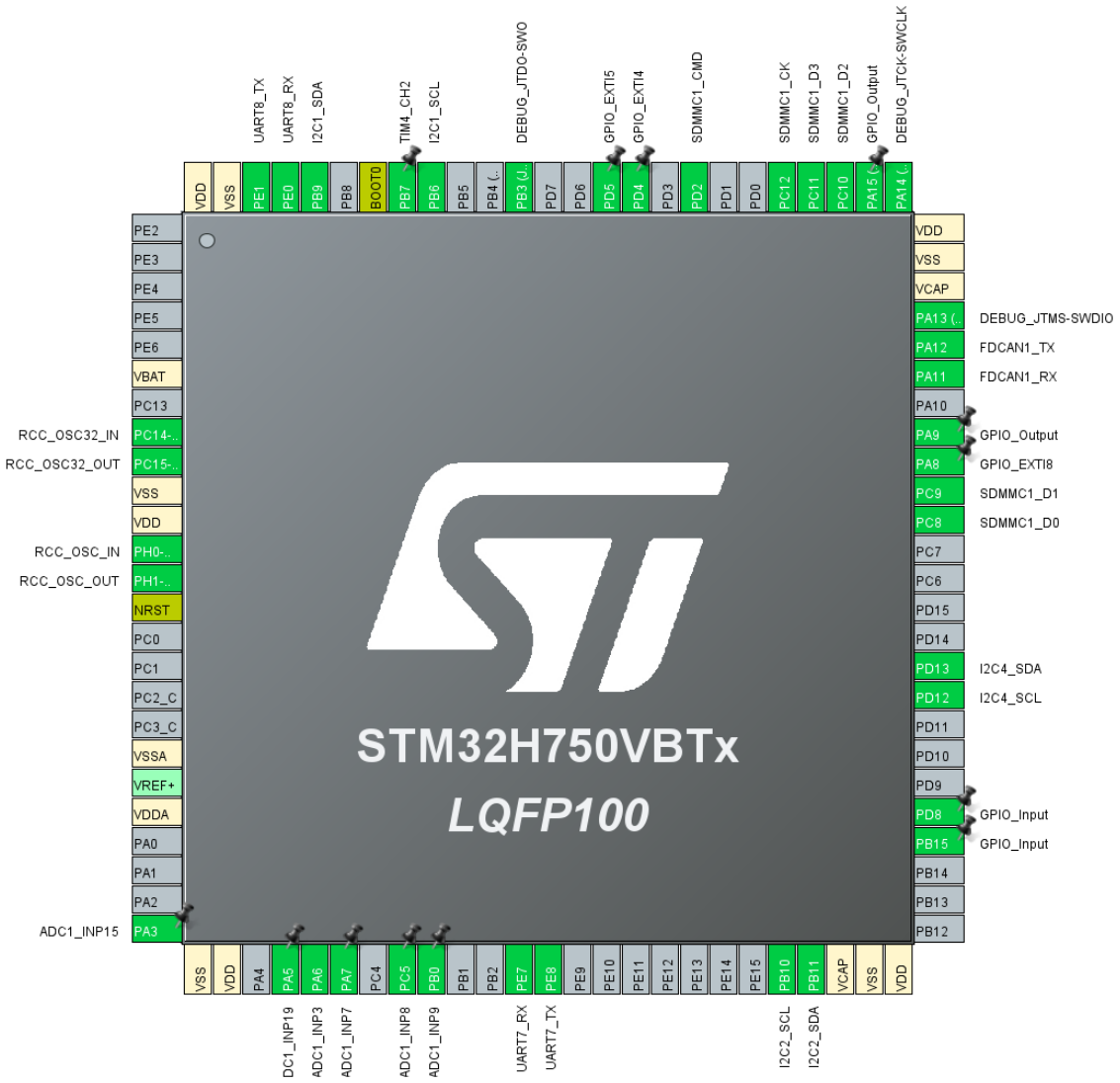


Figure 4.4: *STM32 Pinout*

The next major aspect of the schematic design was to ensure that all directional communication signals were connected to the appropriate pins between the microcontroller and the peripherals. Specifically, it was important to ensure that for both UART connections, the TX pin of the microcontroller was connected to the RX pin of the appropriate component. On the CAN controller to CAN transceiver connection, though, the opposite needs to occur. The CAN TX pin on the microcontroller for the

CAN controller must be connected to the CAN TX pin of the CAN transceiver and vice versa for RX. This can be seen in fig. 4.5, fig. 4.6, and fig. 4.4.

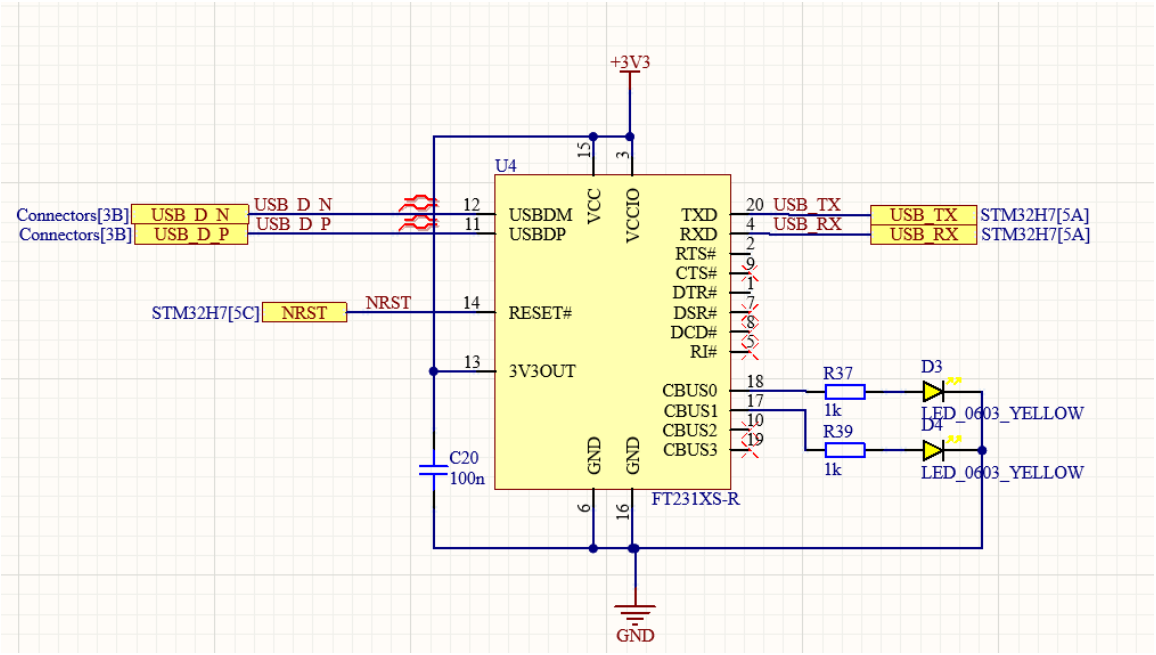


Figure 4.5: *USB Transceiver Wiring*

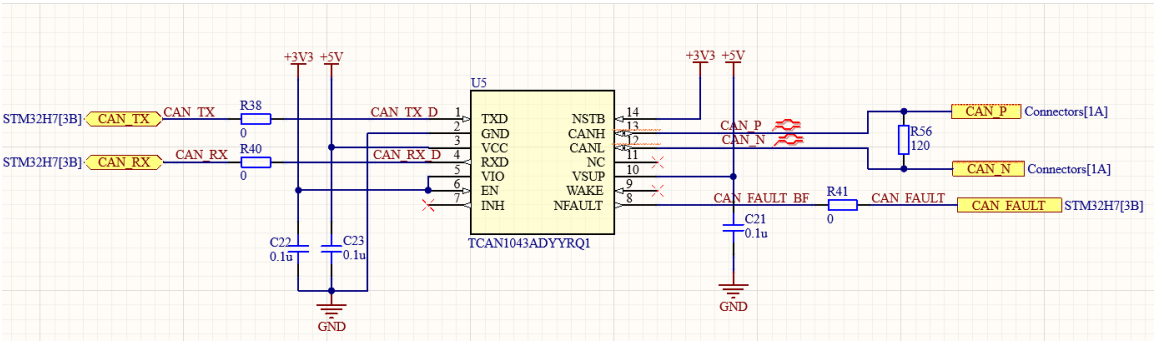


Figure 4.6: *CAN Transceiver Wiring*

It is also important that pull-up and pull-down resistors are connected to the appropriate signals for communication lines and control signals. Specifically, pull-up resistors needed to be added for the I2C communication lines as well as the SDIO data lines. Additionally, the DAQ acts as a terminating node in the CAN network, so the termination resistor for the CAN bus must be present.

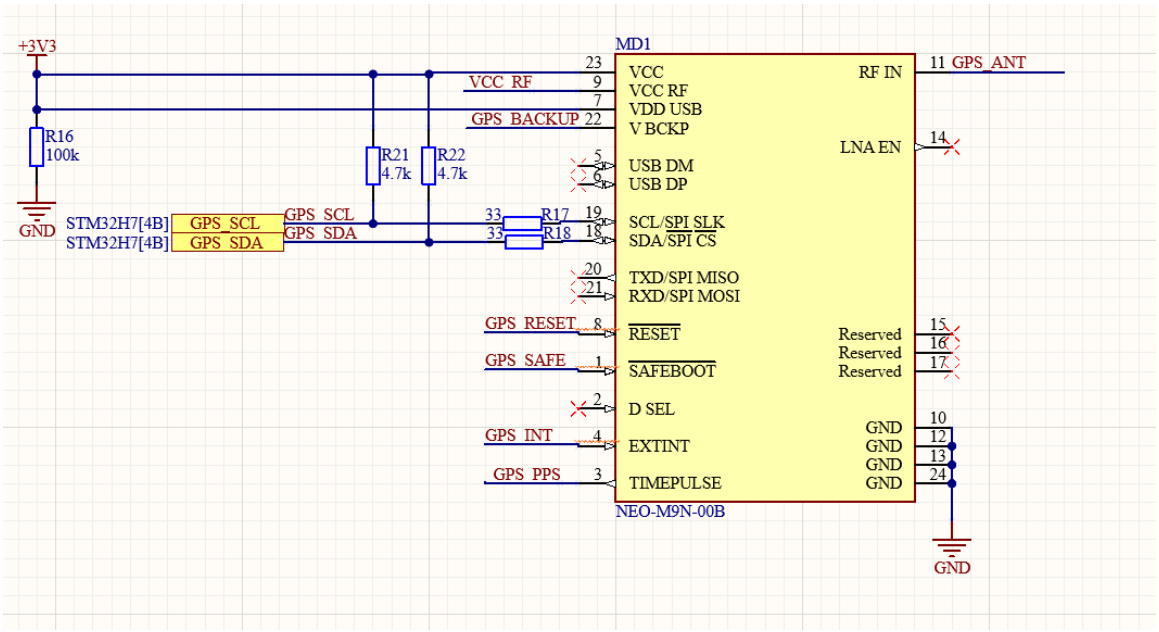


Figure 4.7: *GPS I2C Wiring*

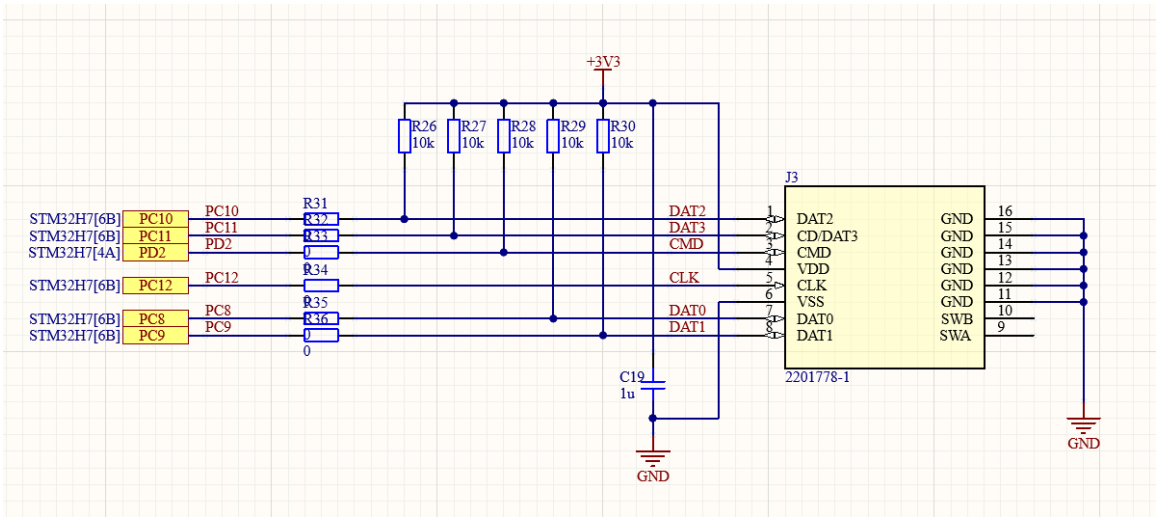


Figure 4.8: *Micro SD Reader Wiring*

The final important aspect for this at a system level was to ensure that the controlled impedances of signals were configured if they were needed. Specifically, the CAN spec and USB spec detail controlled differential impedances for their signals. As a result, these signals needed to be marked as differential pairs on the schematics as seen in fig. 4.5 and fig. 4.6.

4.2.3 Layout and Routing

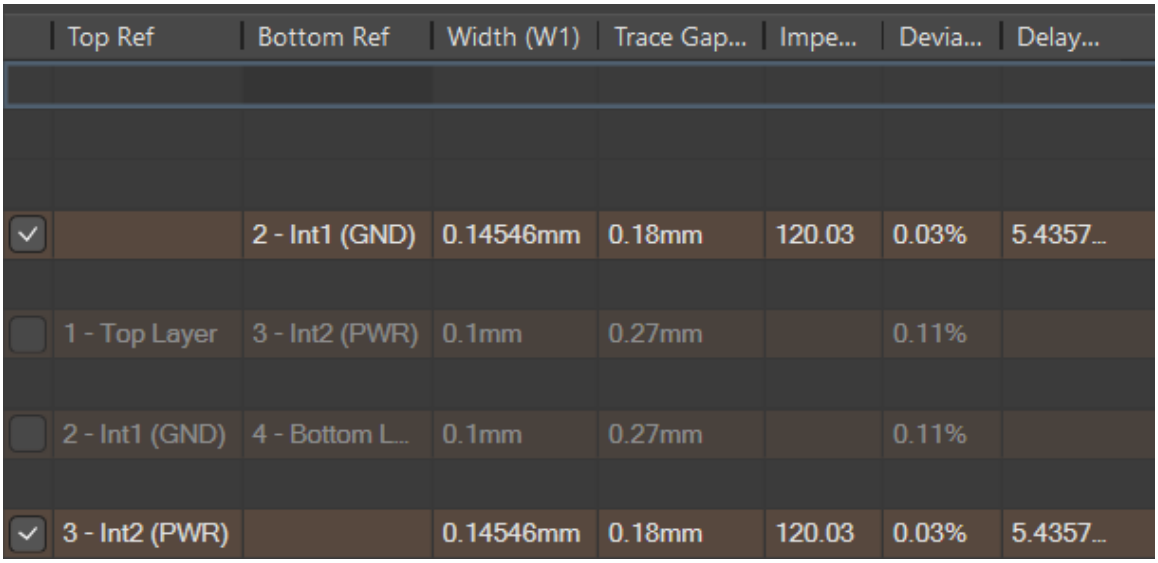
Similarly to the schematic design, the layout and routing were mostly outside the scope of this project and were done by another member of the competition team. However, the layout and routing were reviewed to ensure that a user can effectively interface with the hardware and that all specifications for items such as differential pair impedances were being met. Additionally, layer stackups, impedance profiles, and rules checks were reviewed to ensure that the PCB would comply with the manufacturer's requirements and tolerances. The PCBs were manufactured with JLCPCB. The four layer stackup used for the hardware can be seen in fig. 4.9. There are two signal layers on the top and bottom with two copper planes on the internal layers, one for 3.3V and one for GND. These internal layers make it easy to provide power and ground to many different places and provide consistent references for differential pairs that exist on the signal layers.

#	Name	Material	Type	Weight	Thickness	Dk	Df
	Top Overlay		Overlay				
	Top Solder	SM-001	Solder Mask		0.0254mm	4	0.03
	Top Surface Finish	PbSn	Surface Finish		0.02mm		
1	Top Layer	CF-004	Signal	1oz	0.035mm		
	Dielectric 1	PP-017	Prepreg		0.2mm	4.6	0.02
2	Int1 (GND)	CF-004	Signal	1/2oz	0.0175mm		
	Dielectric 3	Core-039	Core		1.065mm	4.6	0.02
3	Int2 (PWR)	CF-004	Signal	1/2oz	0.0175mm		
	Dielectric 5	PP-017	Prepreg		0.2mm	4.6	0.02
4	Bottom Layer	CF-004	Signal	1oz	0.035mm		
	Bottom Surface...	PbSn	Surface Finish		0.02mm		
	Bottom Solder	SM-001	Solder Mask		0.0254mm	4	0.03
	Bottom Overlay		Overlay				

Figure 4.9: *4 Layer Stackup based on JLCPCB Stackup*

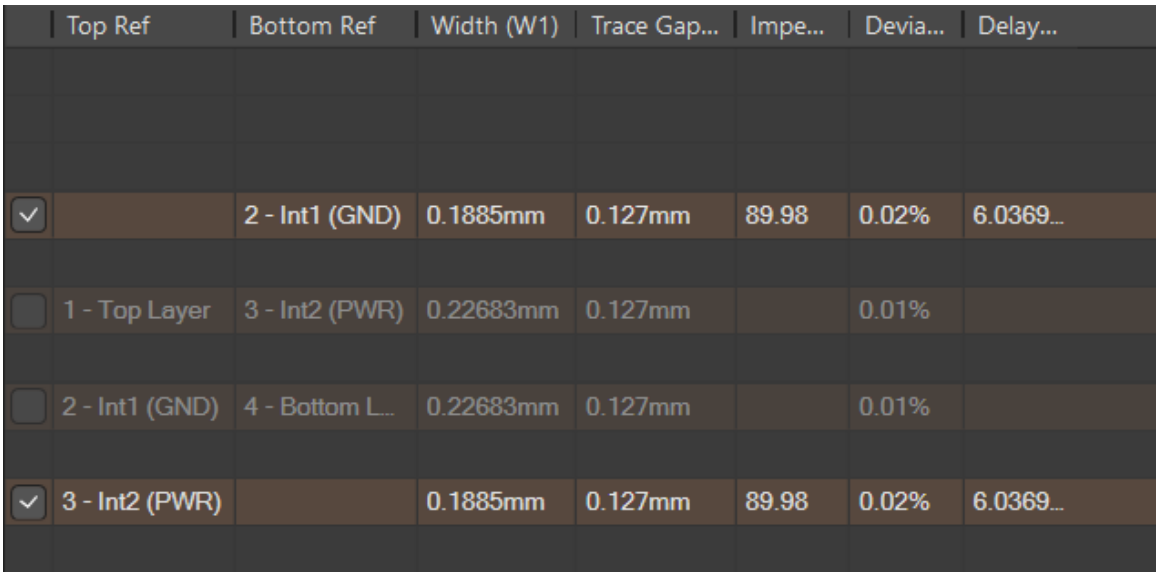
The CAN spec states that the differential impedance of a CAN bus should be 120Ω. As can be seen in fig. 4.10, the impedance of the differential pair will be 120.03Ω, or a less than 0.03% deviation. Similarly, the USB spec includes a 90Ω differential

impedance. As can be seen in fig. 4.11, the impedance of the differential pair with this profile will be 89.8Ω, or just above a 0.02% deviation.



	Top Ref	Bottom Ref	Width (W1)	Trace Gap...	Impe...	Devia...	Delay...
✓		2 - Int1 (GND)	0.14546mm	0.18mm	120.03	0.03%	5.4357...
	1 - Top Layer	3 - Int2 (PWR)	0.1mm	0.27mm		0.11%	
	2 - Int1 (GND)	4 - Bottom L...	0.1mm	0.27mm		0.11%	
✓	3 - Int2 (PWR)		0.14546mm	0.18mm	120.03	0.03%	5.4357...

Figure 4.10: 120Ω *impedance profile for a 4-layer stackup*



	Top Ref	Bottom Ref	Width (W1)	Trace Gap...	Impe...	Devia...	Delay...
✓		2 - Int1 (GND)	0.1885mm	0.127mm	89.98	0.02%	6.0369...
	1 - Top Layer	3 - Int2 (PWR)	0.22683mm	0.127mm		0.01%	
	2 - Int1 (GND)	4 - Bottom L...	0.22683mm	0.127mm		0.01%	
✓	3 - Int2 (PWR)		0.1885mm	0.127mm	89.98	0.02%	6.0369...

Figure 4.11: 90Ω *impedance profile for a 4-layer stackup*

The largest design choice for the layout was to split the hardware into two separate PCBs that could be stacked on top of each other. The purpose of this was to reduce the overall footprint of the design by utilizing the third dimension. Due to connector

sizes and some large components, the area of a single PCB would have needed to increase significantly, approximately 1750mm^2. By utilizing the third dimension, the total package size was able to decrease by a similar amount. This is critical for physically mounting the device onto the car, as there is a limited amount of space and few mounting points available. The way these PCBs attach together can be seen in fig. 4.14.

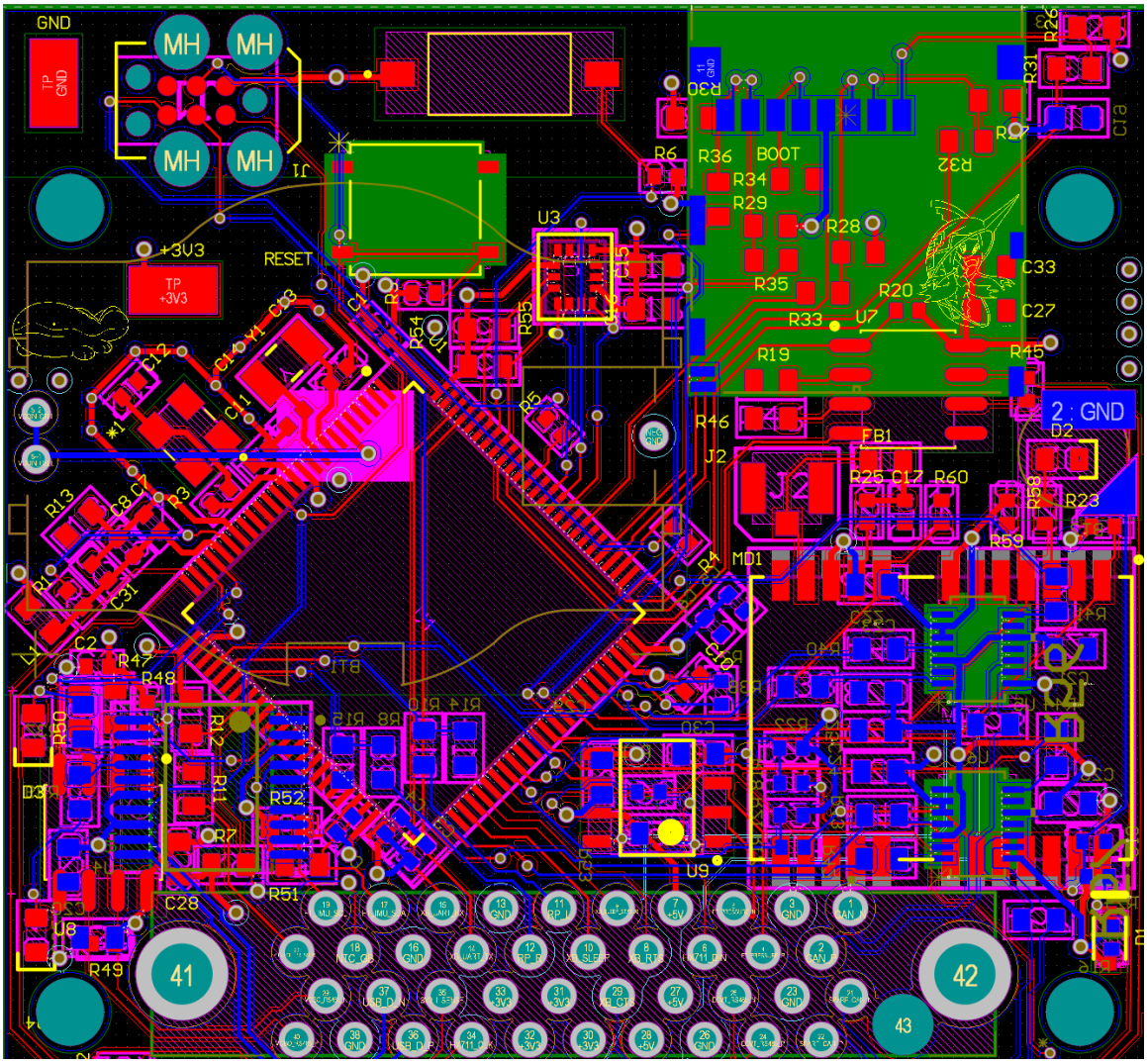


Figure 4.12: *2D Layout of the top PCB*

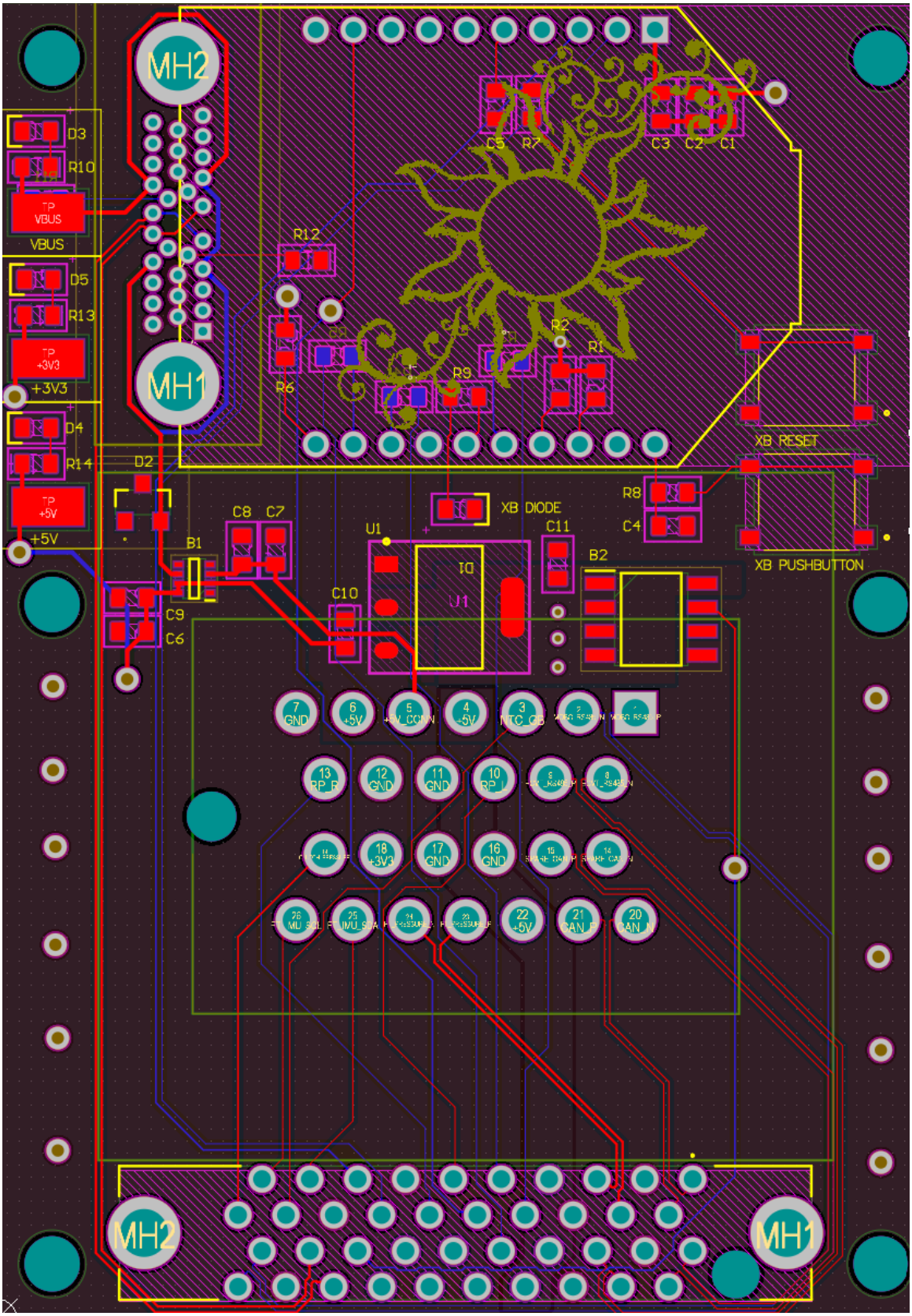


Figure 4.13: *2D layout of the base PCB*

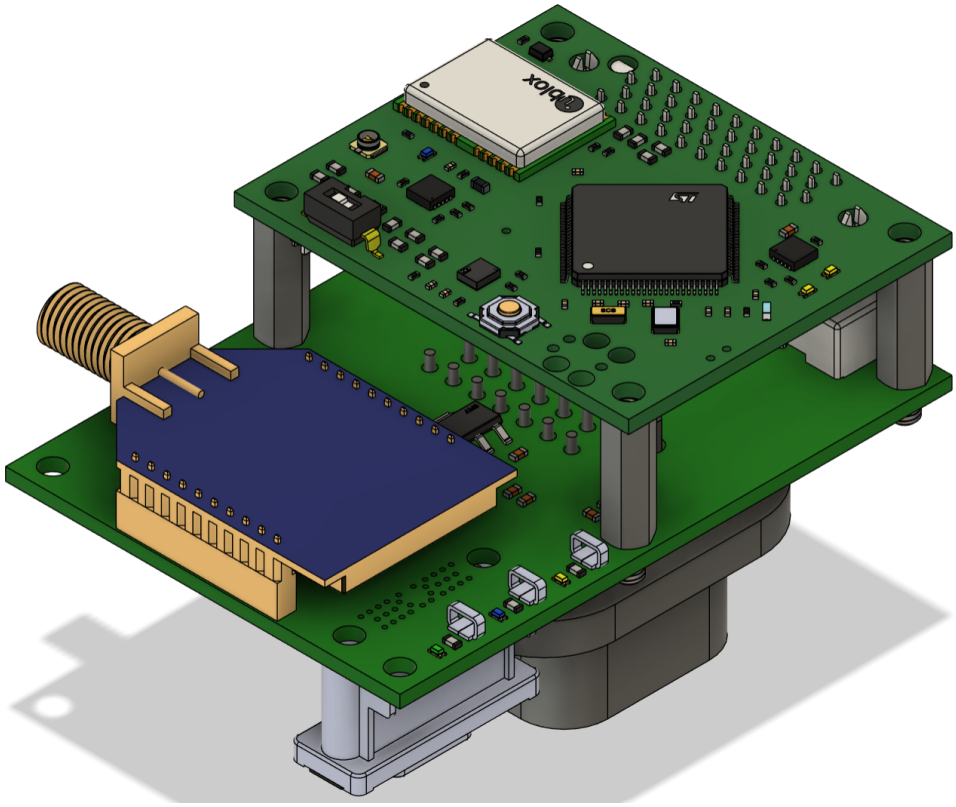


Figure 4.14: *3D assembly of the DAQ PCBs*

The biggest place where component placement is critical is the location of the micro-SD card reader. This component needs to be placed in a position that, when assembled into an enclosure, the SD card can be inserted into or removed from the reader. As can be seen in fig. 4.12, the SD card was placed on the edge of the board. Referencing fig. 4.14, this is at the edge next to the radio module, having enough clearance to be able to insert a finger for removal and insertion.

4.3 Software Design

Most of the scope of this project was related to the software and firmware design and implementation. This section details the design of the software, it's implementation,

as well as issues encountered and how they were addressed or what needs to happen to address the issue in the future.

4.3.1 High-Level Design and Implementation

The software for this DAQ is designed and implemented to be run in a pseudo-parallel manner. This means that several different tasks run sequentially, with each task running small chunks or sub-tasks during each cycle. Each task is then designed as a state machine to segment itself into these smaller sub-tasks. In order for each of these tasks to communicate with each other, there is one group of data that each task can reference. Since this is a sequential action still, conflicts will not occur, and there does not need to be any extra protection for accessing the same data or variable at the same time.

The DAQ runs four main tasks that are continuously cycled through:

(1) Storage

(2) Communication

(3) Radio Communication

(4) Sensors

(5) Health Monitoring

The storage task is responsible for the actual logging of data. The communication task is the task responsible for reading data off the CAN bus. The radio communication task is responsible for medium-long range telemetry. The sensors task is responsible for the reading of all on-board sensors. The health monitoring task is responsible for

updating any LED indicators for the health of the system. Together, these five main tasks handle the running of all parts of the system. As can be seen in fig. 4.15, these tasks all run sequentially before looping back to the start.

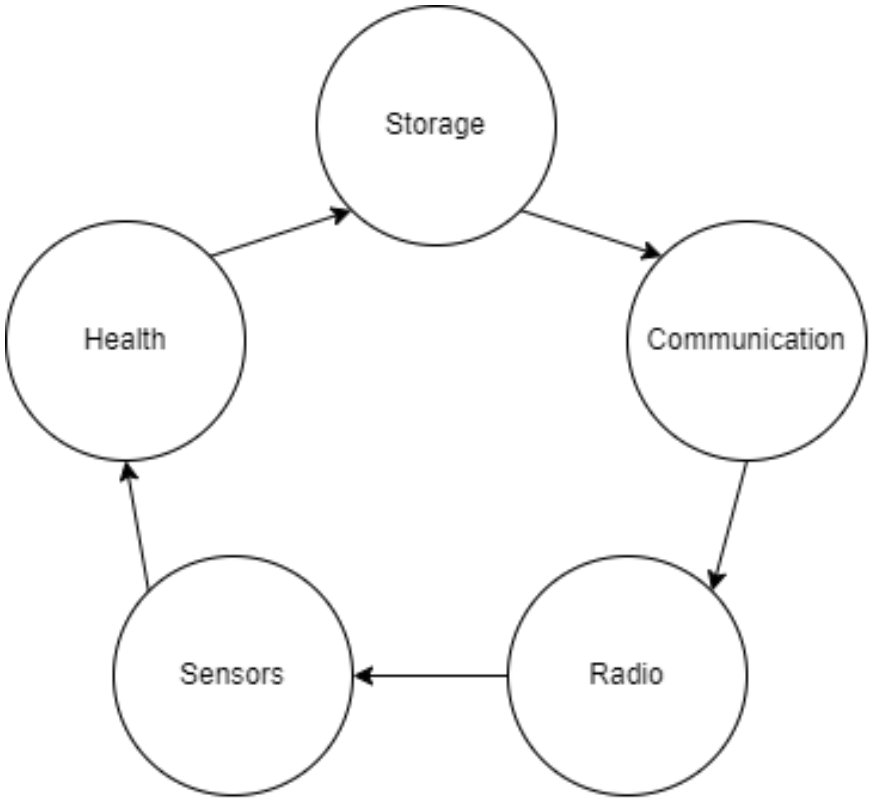


Figure 4.15: *Software tasks loop diagram*

Object-oriented programming (OOP) is utilized significantly for this design. Every task and sensor are a class that has individual constructors depending on the needs of the task. Additionally, each task gets its own enum classes to define the various states for each state machine. Two parent classes are utilized by everything. For different tasks, there is an abstract task class that forces an implementation for a function that runs the task and requires the intercommunication data to be passed in. For sensors, there is an abstract sensor class that forces an implementation for two functions: an initialization function and an update function. Additionally, this class implements a read function that returns the current sensor value.

Since this project is based on the Arduino platform, several of the basic tasks for interacting with the hardware are handled by the platform. This includes things such as setting the mode of a pin, reading and writing from data registers, configuring UART, SPI, and SDIO interfaces, and configuring the interrupt vector table. Even with the configuration of these peripherals being handled by Arduino, they still need to be configured with the proper settings to be used. Despite all falling under the Arduino umbrella, each platform is maintained independently of the others. The people maintaining the Arduino platform for Teensy and the people maintaining the Arduino platform for STM do not communicate with each other, and as a result, there are some implementation differences that exist between different hardware platforms using Arduino.

Additionally, not every family of microcontroller from STM receives the same support or even the same supported functions from the Arduino framework. One specific issue that arose was a versioning issue for STM32duino. In version 18.0.0 and onwards, the memory sizes of the H7 family of STM microcontrollers were updated incorrectly, causing code to not run and the debugger to be unusable. This issue was resolved by reverting the version of STM32duino back to 17.6.0. The PlatformIO team has been notified of this issue and has responded that they are working on a fix. Other issues that arose that were not issues on the Teensy platform were the signedness of pins on the STM32, multiple timers on the same timer channels, timer registers having different sizes, and the same interrupt being tied to multiple pins. One of the largest benefits of this change was having access to the debugger. Without access to a debugger, many of these issues would not have been solved or would have taken weeks or longer to debug.

The last main component that impacts each part of this system is a shared data segment. This shared data segment is what allows all the tasks to communicate with

each other. Specifically, this includes variables for storing all the data that needs to be logged, flags for indicating when data has been updated, and fault flags to indicate when a task is encountering problems. Since this code runs sequentially, there are no issues of different tasks attempting to read from or write to the same chunk of the shared data section at the same time. As a result, there are no added protections for this scenario.

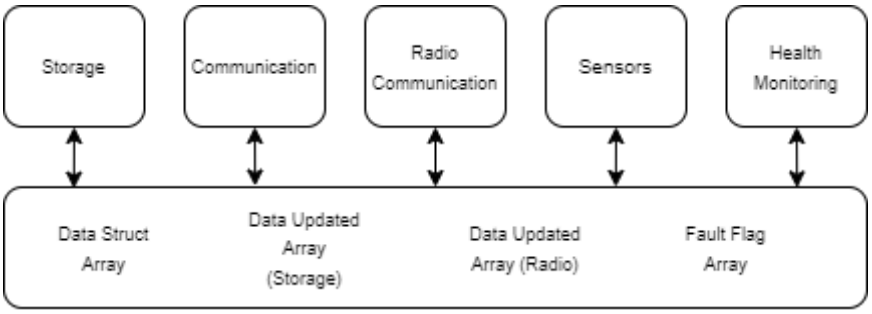


Figure 4.16: *Shared Data Section Diagram*

4.3.2 Storage Task

The first and foremost task that the DAQ runs is the storage task. This task is responsible for managing the SD interface as well as managing the filesystem. This includes things like initializing the communication interface with the SD card, setting up the filesystem, and writing data to the SD card.

As with all tasks, this was implemented via a state machine. A diagram for this state machine can be seen in fig. 4.17. This machine consists of seven states, with each state handling a single part of the task. These states are:

- Initialize
- Wait to Open
- Open

- Wait to Write
- Write
- Close
- Error

In this state machine, there are two types of states. These are action states and waiting states. The action states are all capable of entering the error state if their action fails unexpectedly. The waiting states are incapable of entering the error state, since their sole purpose is to wait for certain conditions needed for their corresponding action state.

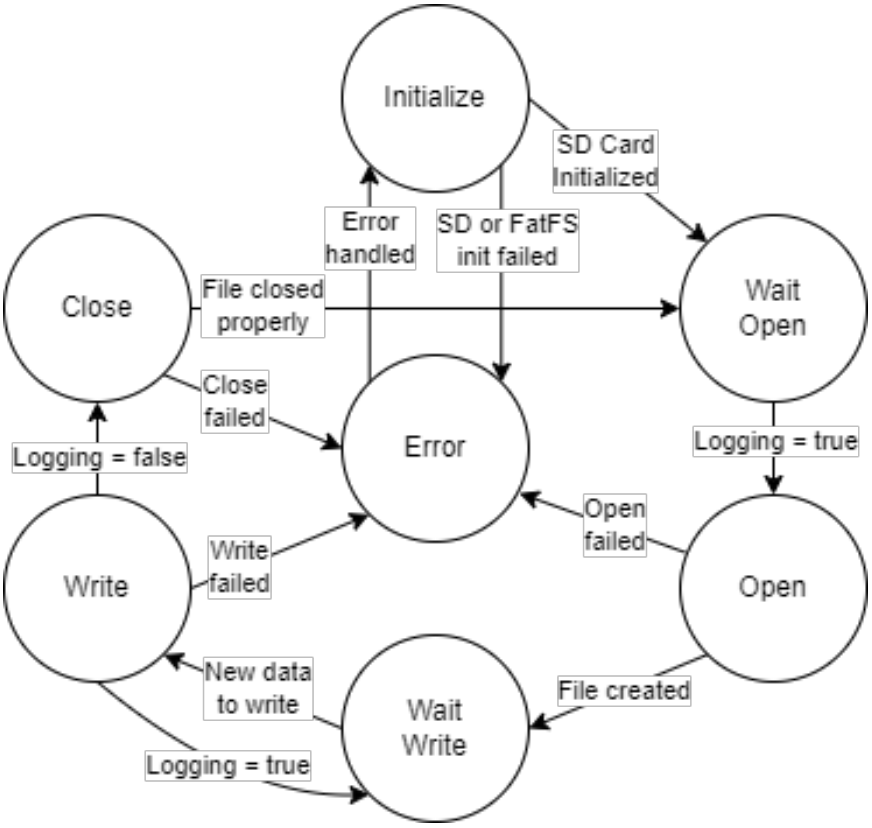


Figure 4.17: *Storage State Diagram*

The Initialize state is responsible for creating the SD instance. This includes configuring the SDIO communication interface that physically connects the SD card to the microcontroller as well as mounting the FatFS file system that is used by the SD card. The creation of the SD and File objects relies on STM32SD library [21]. This library provides the functions for initializing the SD interface and for mounting the file system. One major issue is present in this library that needed to be fixed. The library expects the FatFS implementation to have a function f_unmount, which can be used to unmount the filesystem. This function is not present in the FatFS implementation provided by STM32duino. To resolve this issue, the function call can be replaced with a f_mount function call with the appropriate arguments, or a wrapper function f_unmount can be created that calls f_mount with the appropriate arguments. The former was used in this project. If the returns of both the SD initialization and the FatFS initialization are success codes, the next state is Wait to Open. If any other codes are returned, the next state is the Error state.

The wait to Open state is a waiting state that holds the storage task until the logging flag is high. If the logging flag goes high, the next state is Open. The task can be held in this state indefinitely if the logging flag stays low. This can occur if the logging switch on the dashboard is disabled or removed.

The Open state is responsible for the creation of files. Once the task enters this state, it searches for the next available file name and attempts to create the file. The file is created and opened properly if the return of the open function, which is a wrapper around the FatFS f_open function, returns a non-null value. This state will attempt to open a file three times before deciding it is unable to. If the file is opened properly, the next state is the Wait to Write state. If the state is unable to open a file, the next state is the Error state.

The Wait to Write state is another waiting state that holds the storage task until new data is received. Once a new piece of data is received from the CAN bus or from reading the onboard sensors, this state transitions to the Write state. If there are no nodes on the CAN network or a CAN error and all the onboard sensors are disabled, the storage task can be held indefinitely in this state. It is determined that a piece of data is new by checking a boolean value for each data segment. These booleans are set to true in the communication task or the sensors task.

The Write state is responsible for writing the most recent data to the SD card. Once the task enters this state, it iterates through all possible pieces of data to see if they have been updated. If a piece of data has been updated, a data packet is created with the data ID, the data, and then a checksum to improve resiliency.

```
uint32_t calcChecksum(uint8_t *payload, uint16_t payloadSize) {
    uint32_t checksum = 0;
    for (int8_t i = 0; i < payloadSize; i++) {
        checksum += (i + 1) * payload[i];
    }
    return checksum;
}
```

This packet is then written to the SD write buffer. Once all updated messages are written to the SD write buffer, the buffer is then flushed to the SD card. This is done on each iteration as opposed to waiting for the buffer to fill up to reduce the blocking when writing to the disk, to reduce the potential risk of missing data updates. This is outlined in fig. 4.18. If the writes and flush are completed successfully and logging is true, the next state is Wait to Write. If the writes and flush are completed successfully and logging is false, the next state is Close. If the writes or flush fail, the next state is Error. Once the writes are complete, the booleans that are true for a data segment are reset to false.

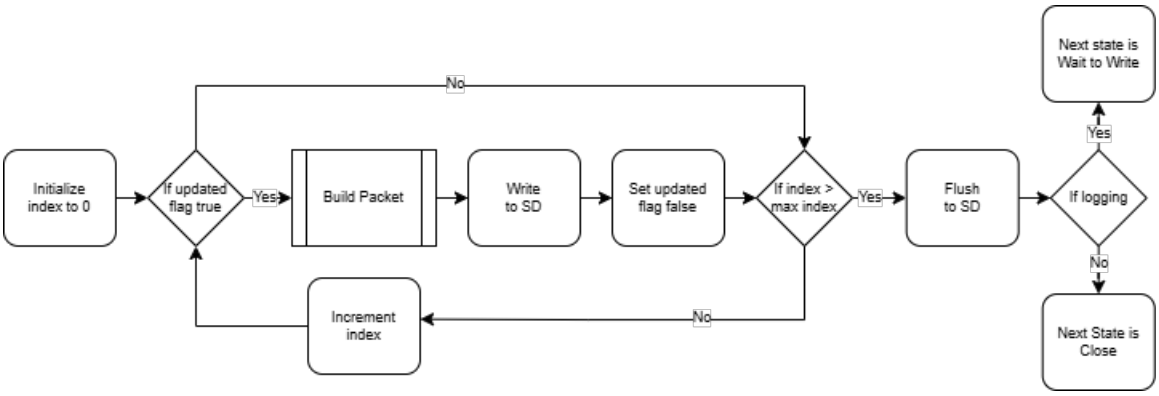


Figure 4.18: *SD Task Write State Block Diagram*

The final normal state is the Close state. This state is responsible for safely closing the current open file. If the file fails to close properly, the next state is the Error state. To do this, any data remaining in the buffer is flushed to the SD card, and the File object is closed. If the file closes properly, the next state is the Wait to Open state. If the final flush fails or the file fails to close, the next state is set to the Error state.

4.3.3 Communication Task

The next task is the communication task. This task is responsible for configuring and managing communication on the CAN bus. To do this, the ACANFD_STM32 library [22] was modified to support the operations needed to configure the CAN controller into CANFD mode and to read and write from the bus. One requirement of this library is that the CAN IRQ handler functions need to be defined by the user. These functions are defined in the main file and must be named properly to use the proper CAN controller, such as adding a 1 or 2 to distinguish between CAN1 and CAN2 hardware.

The state machine for the communication task contains four states. These states are:

- Initialize
- Wait for Message
- Process Message
- Error

As can be seen in fig. 4.19, the two main states which will be looped between are the Wait for Message and Process Message states that handle reading and processing messages.

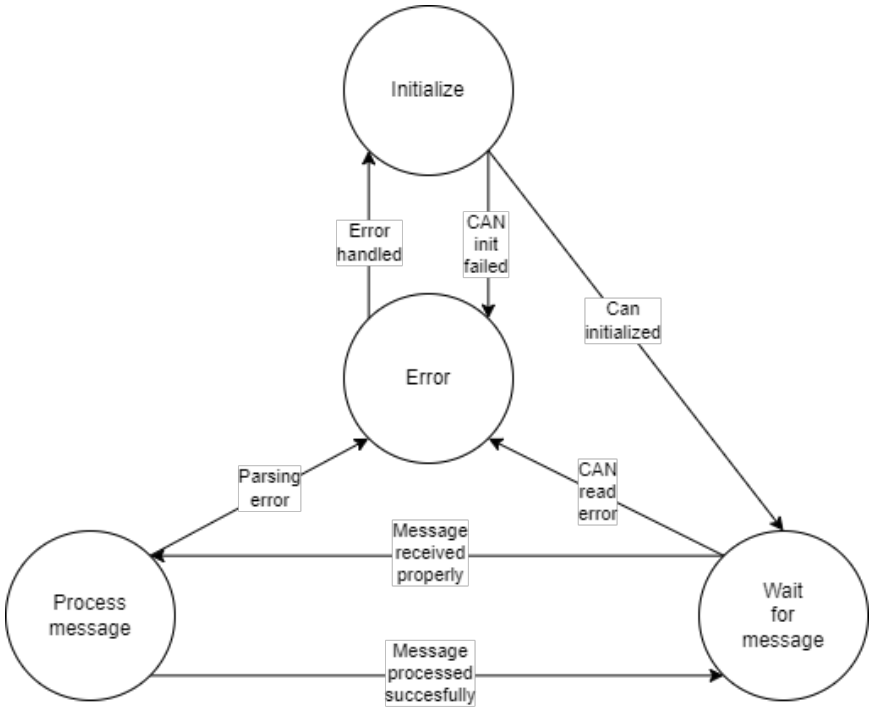


Figure 4.19: *Communication State Diagram*

The Initialize state is responsible for configuring the settings and initializing the CAN controller into CAN FD mode. To configure the hardware into CAN FD

mode, the pins that connect from the internal CAN controller to the external CAN transceiver need to be set to their alternate functions, and the bits in the configuration register need to be set to Normal FD mode. Additionally, the desired arbitration clock rate and data clock rate need to be set as well, so that the bus is transmitting at the desired speed. If there are no issues when configuring the CAN controller, the next state is set to Wait for Message. If there are any issues, the next state is set to Error.

The Wait for Message state is the state that is directly responsible for reading data off the CAN bus. If the CAN bus has a message available to be read, it reads the contents into a message variable. If no messages are ever received, it is possible to become stuck in this state indefinitely. If a message is received, the next state is set to Process Message. If there are errors reading the message, the next state is set to Error. Only one message will be read in this state, so if there are multiple messages ready to be read, the system will need to enter this state multiple times.

The Process Message state is responsible for taking the data from a received CAN message and moving it into the section of data shared by all tasks. Additionally, to indicate to the storage task that a piece of data is new, an array of booleans that is equal in size to the number of data blocks exists. When a message is received, the boolean that corresponds to the same integer value as the index of the received data block is updated to true. This boolean will be reset to false in the storage task. As is detailed in fig. 4.20, this state first checks if the CAN ID of a message is valid, something expected or able to be handled. It then copies the contents of the CAN message into the corresponding data segment in the shared data. Since it is expected that the CAN messages will fully correspond to the data segment and that the data segment will be byte aligned with no gaps, a simple memcpy operation can be done to move the data from the received message into the shared data. If this processing

is completed without error, the next state is set back to Wait for Message. If there are any issues with the CAN message or the copying of the data, the next state is set to Error.

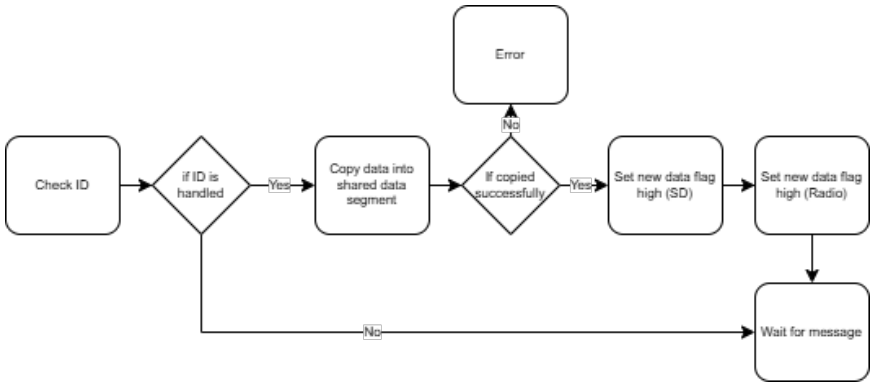


Figure 4.20: *CAN Message Processing Block Diagram*

The final state of the communication task is the Error state. This state is responsible for handling any errors that occur because of initialization, reading CAN messages, or processing the CAN messages. The most common errors that can occur are the bus failing to initialize properly, and an invalid or unhandled CAN message IDs are received. The most likely cause of the CAN hardware not initializing properly is improper wiring on the PCB or invalid configuration options or settings. In either of these scenarios, recovery is not possible, and the task must be held in the Error state. In the event of an invalid CAN ID received, the system can continue to operate successfully, and the task can be recovered by returning to the Wait for Message state. In the shared data, a flag is set hig,h indicating a fault in the communication task.

4.3.4 Radio Communication Task

The radio communication task handles communication with the radio module and the transmission from the vehicle to the pit area. The Xbee module handles the over-the-air transmission of data. The Xbee module is configured separately, before

being installed into the DAQ, accepting a UART communication line and relaying that data over the error. The module is not configured to handle lost packets, invalid packets, or any issues related to the data transmission, as dropping packets is not a catastrophic failure.

The radio communication task has five states:

- Initialize
- Wait
- Build
- Send
- Error

The state machine describing how to transition between these states can be seen in fig. 4.21. Similarly to the storage task, there is a wait state whose only function is to hold the task until a certain condition is met. This is the only state that cannot enter the Error state.

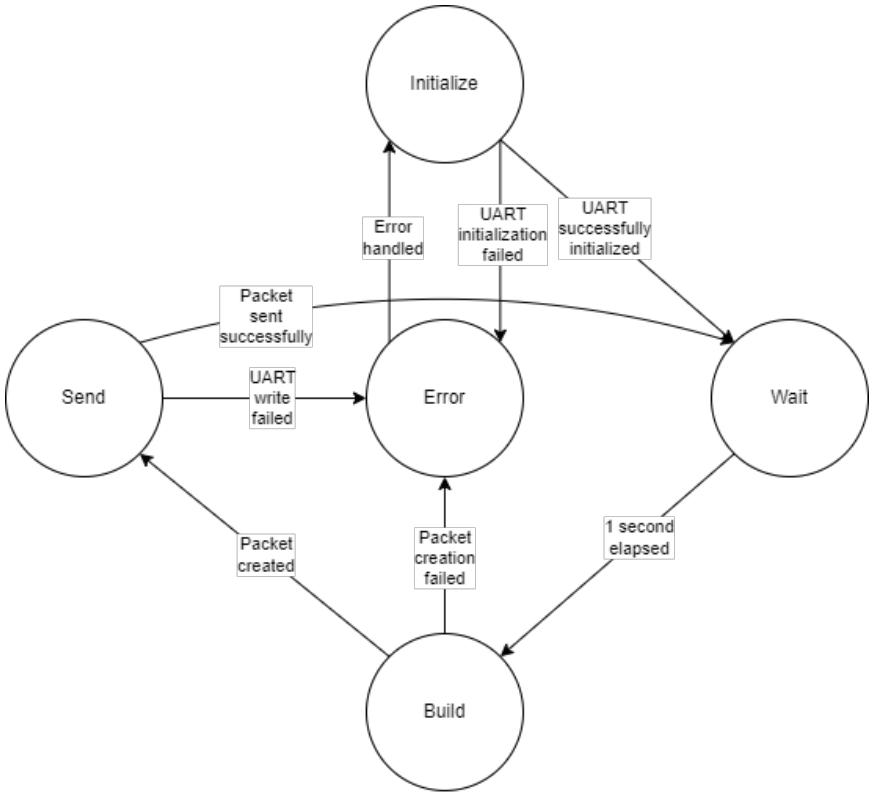


Figure 4.21: *Radio Communication State Diagram*

The Initialize state is responsible for configuring the UART communication interface with the Xbee radio. The Arduino HardwareSerial library handles configuring the registers to their alternate functions, so long as the proper tx and rx pins are set. The baud rate is configured to 57600 bits per second to comply with the Xbee's configuration. If UART initializes successfully, the next state is the Wait state. If the UART communication is not initialized successfully, the next state is the Error state.

The Wait state holds the radio communication task until it is time to send a packet over-the-air. If a one second timer has triggered, the next state is the Build state.

The Build state is responsible for creating a single packet that will then be transmitted over-the-air. Similarly to the storage task, this state checks which data segments have been updated and adds each segment that has been updated to the packet. The

data ID and the length of data are added before each data segment, and a checksum is computed and added to the end of the data segment. Differently from the storage task, each updated data segment is added to a single packet. If the packet is created successfully, the next state is the Send state. If there are issues creating the packet, the next state is the Error state.

The Send state writes the created packet over UART to the Xbee module. To reduce issues related to filling up the hardware buffer, the UART writes just a single byte at a time. Additionally, there is a check to ensure that the hardware buffer is not full and can accept the byte. The reason for this is that the write function that puts a byte or array of bytes onto the hardware buffer does not check if the buffer is full, and it will hold the program until the byte or bytes in the array are written to the buffer. This makes it very likely to cause interruptions in the code, harming the running of the other tasks. Once all bytes are written to the buffer, the next state is set back to the Wait state. If there are any issues with writing a byte, the next state is the Error state.

The last state in the radio communication task is the Error state. Similarly to the communication state, if there are any issues setting up the UART communication line, it is likely a result of faulty wiring or invalid parameters. In this event, the error is typically not recoverable. If the error occurs within the Build or Send state, the error can be recovered.

4.3.5 Sensors Task

The sensors task is responsible for reading the DAQ's onboard sensors. These sensors include the onboard IMU and GPS, as well as a few analog sensor inputs. The IMU and GPS both communicate over I2C, and the analog sensors utilize the

internal ADC of the STM32. These sensor readings are then added to the shared data section so that they can be written to the SD card.

This task has five states:

- Initialize
- Wait
- Read
- Send
- Error

The state machine describing the transitions can be seen in fig. 4.22. Similarly to the radio tasks, there is a time-based waiting task and four additional tasks for functionality. Different from the radio communication task is that there will be two states that cannot enter the error state, the Wait state and the Send state.

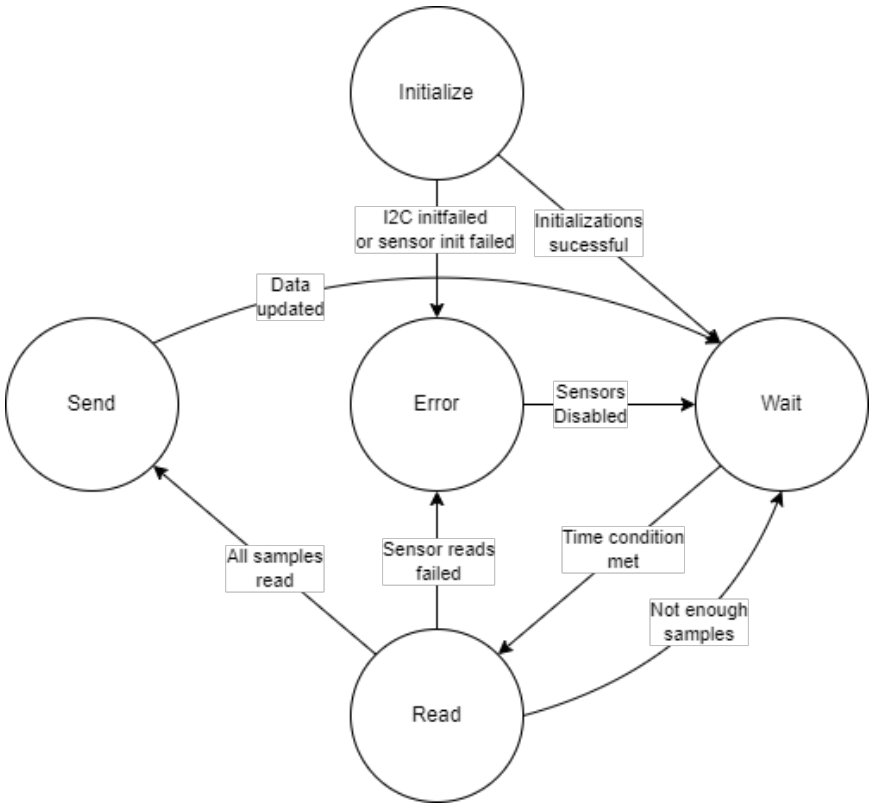


Figure 4.22: *Sensors State Diagram*

The Initialize state is responsible for configuring the peripherals needed to communicate with sensors and for configuring any sensors when it is applicable. Since both the IMU and GPS communicate over I2C, two I2C buses need to be configured. Additionally, each of these sensors need to be configured, including items such as setting up the sampling rate, any onboard filtering, or changing the sensor range. For the basic analog sensors, the pins need to be configured to use their ADC alternate function. The Arduino pinMode function takes care of this. Additionally, the bit resolution also needs to be configured. To comply with the requirements of this project, this must be at least 10 bits. If everything is configured properly, the next state is the Wait state. If there are any issues, the next state is the Error state.

The Wait state holds the task for a specified amount of time. The purpose of this is to ensure that sensors are sampled at fixed intervals to support digital filters. As a

result, the amount of time the task is held in this state is determined by the desired sampling rate of the sensors which pre-processing filtering is done on. Since the only thing this state does is wait for a time condition to be met, it is impossible to fault, and, as such, cannot enter the Error state. Once the time condition is met, the next state is the Read state. If no pre-processing of the data is done, the time held in this state is 0, and the next state is immediately set to be the Read state.

The Read state is responsible for reading the data from sensors. To do this, the analog channels need to be sampled, and samples need to be requested from the sensors communicating over I2C. Each sensor is only sampled if there were no issues with the configuration of the sensor. If any pre-processing is desired, it will occur in this state directly after reading a sensor's sample. At this point, there is no pre-processing, so there are no filter implementations directly on-board the DAQ. If all sensors are sampled without issue and the number of samples taken is equal to the number of samples needed for a filter, the next state is set to the Send state. If the sensors are all sampled without issue, but the number of samples taken is fewer than the number of samples needed for a filter, the next state is the Wait state. If there were any issues sampling the sensors, the next state is set to be the Error state. Issues can arise if there is no response to a request from an I2C sensor, or if an analog sensor returns a value outside of its expected range. If there is no pre-processing of data, each time the sensors are sampled, the next state will become the Send state.

The Send state is a simple state that updates the data in the shared data section and sets the flag that this data is new to true for the storage task and radio communication task. Since the code is sequential, there are no risks of conflicts arising from different tasks reading or writing at the same time. As a result, it is not possible for this state to fault, and as a result cannot enter the Error state. Once the shared data section

is updated and the flags indicating that the data is new are set to be true, the next state is set to the Wait state.

The Error state handles any issues that arise from configuring the communication with sensors, the configuration of the sensors themselves, or sampling of the sensors. If the communication interface fails to initialize properly, it is unlikely to be resolved. A flag is set in the shared data section to indicate this. However, if any of the communication interfaces initialize properly, the task is recoverable, and receiving some of the data is better than receiving none of the data. The same is true of the sensors themselves. If any of the sensors fail to initialize, a flag is set in the shared data section to indicate this, but the task can continue without this sensor. Reading the sensors behaves similarly. If a sensor fails to read properly, a flag is set in the shared data section, and the sensor is disabled in the Read state. The state machine then recovers into the Wait state.

4.3.6 Health Monitoring Task

The health monitoring task detects and reports issues with the system. This is done by toggling on-board LEDs to act as visual indicators in the event of issues. These issues include things such as CAN failures, SD failures, and a heartbeat.

This task is the simplest state machine with just two states and no Error state. These are the initialization state and the update state. The diagram for this state machine can be seen in fig. 4.23.

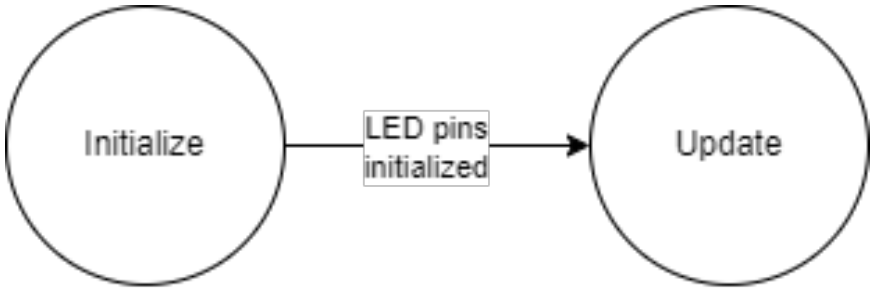


Figure 4.23: *Health Monitoring State Diagram*

The initialization state is responsible for configuring the pins for LEDs. Since all GPIO pins on an STM32 can be configured to be outputs, it is not possible to fail to initialize. As a result, the next state is always set to be the Update state.

The update state checks the status of the other tasks and toggles LEDs as needed. Additionally, the heartbeat is updated in this state. Similarly to the initialization state, since this state is only performing digital writes, or updating the output data register, it is not able to cause a fault. The heartbeat LED is toggled once every half second, for a total frequency of 1 Hz. The indicator LEDs directly use booleans from the shared data section to write a value to the LED. This means that the output of those LEDs is determined by the other tasks. The health monitoring task remains in the update state for the entire time the system is running.

Chapter 5

SYSTEM TESTING AND ANALYSIS

With the system fully designed, the next step is to confirm that all systems are operating correctly. There were several methods used to confirm all these results, including unit tests, inspection via the debugger, and simple test programs to confirm functionality on hardware.

5.1 SD Testing

The first part of this project that needed to be tested was the creation and writing of files on the SD card. To do this, a simple test program creating a csv file with dummy data was made. During this testing, there were several issues discovered. The first was that the STM32SD library does not define the function f_unmount even though it is used in the library. This function is typically a wrapper around f_mount and passing in 0's for the filesystem and option. By updating the f_unmount function call to a f_mount with the added parameters, the library worked fine.

After the issue with the library was resolved, the next issues were related to the hardware. The pinout for the SDIO interface had some conflicts, resulting in the SD card being connected to non-SDIO pins. This issue required a re-spin of the hardware to fix, delaying testing by several weeks. During this time, attempts were made to rework the PCB to connect the SD card to the correct microcontroller pins. It was discovered while attempting to run the test program that there were stability problems with the reworked connections. There were a few instances where the SD object would initialize properly, indicating that the library was working successfully.

Once the re-spun board arrived, it was confirmed that the SD library was functional and the test program successfully created a dummy csv file shown in fig. 5.1.

Test CSV File							
Created April 4th	2025						
var1	var2	var3	var4	var5	var6	var7	var8
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11

Figure 5.1: *Dummy CSV file created on SD card*

5.2 Checksum Testing

To confirm that the checksum calculation used in logging and radio communication was functioning as expected, a unit test was created. There were 5 different values tested, and the checksum was computed by hand and then compared to what was calculated by the DAQ. As can be seen in section 5.2, the checksum was computed correctly in all tests.

Table 5.1: *Checksum Unit Testing Results*

Test	Value	Hand Computed	DAQ Computed
1	0x0	0	0
2	0xAFAFAFAFAF	2625	2625
3	0x123456789ABCDEF	4432	4432
4	0xF000	240	240
5	0xFFFFFFFF	2550	2550

5.3 Metadata Testing

Metadata generation for files is a critical feature for being able to properly decode the data into something useful for design engineers to use. To confirm that the metadata is being generated properly, the file metadata must first be computed by hand. Then, since the SD card functionality has been confirmed to work previously, the file metadata can be written to a file on the SD card. This file can be viewed in a hex editor to see each byte of the file, where it was confirmed to match the metadata generated by hand.

After confirming that the metadata section of the file is being written correctly, it needed to be confirmed that data was also being written to the file correctly. To confirm this, dummy data can be written for a chunk of data that corresponds to each data section ID. By inspecting the data written to the file, it can be confirmed that the data was written correctly and that the correct number of bytes was written. This can also be done by inspecting a file in a hex editor.

5.4 CAN Testing

Two parts of CAN that needed to be tested. The first was that two or more CAN nodes were able to communicate, or that the DAQ was able to communicate with any other node. To accomplish this, another system designed for the Baja car was used. These two boards were wired together, and a CAN test program was written to confirm that both transmitting and receiving were fully functional. To start, an internal loopback test was performed, followed by an external loopback test, and finally completing a normal test between two nodes. This test was completed by using a fixed ID of 1, a data length of 32 bytes, and the data section was filled with

incrementing values starting from 0 and ending at 31, with each value corresponding to the index of the byte array of the message.

After the CAN bus was confirmed to work and data was transmitted, CAN loading also needed to be tested. This could be done empirically, knowing the total amount of data being transmitted in each message and the frequency at which each message was being sent. To confirm that the desired throughput was actually achieved, the total number of each CAN ID was tracked over 10 seconds to confirm that the correct number of CAN messages were being received.

Table 5.2: *CAN Testing Results*

Message	Frequency	Data	1 second	5 seconds	10 seconds
1	400 hz	36 Bytes	400	2000	4001
2	400 hz	48 Bytes	400	2000	4001
3	400 hz	48 Bytes	400	2000	4001
4	100 hz	8 Bytes	100	500	1000
5	10 hz	16 Bytes	10	50	100
6	10 hz	8 Bytes	10	50	100
7	100 hz	20 Bytes	100	500	1000

As can be seen in section 5.4, the CAN bus was not overloaded. This is the expected result and supports the previously computed CAN loading and confirms that there are no issues with the amount of data being transmitted. Additionally, this implies that there is room for additional nodes to send data, leaving room for different sensor expansions for permanent or temporary sensing units.

5.5 Radio Testing

The last system to test was radio communication. To test that this task is working as expected, the Digi XTCU tool can be used to read all the data received over the air. To confirm correct operation, a similar test used in the CAN testing was performed, where several bytes were written to the transmitting radio with incrementing values. On the receiving side, these bytes can be viewed in the XTCU tool, confirming that all bytes being sent were received.

After confirming that all bytes being transmitted were being sent and received properly, the packet construction needed to be confirmed. Since the checksum calculation was confirmed previously, the packet creation can be confirmed in the debugger. On one iteration, a packet was constructed by hand and then confirmed by utilizing the debugger and looking at the bytes of the created packet. It was confirmed that packets were being constructed correctly.

Finally, the radio's line of sight distance was tested. This was done by driving the DAQ further and further away from the base antenna until no packets were received at the base side. The base side remained at the aerospace hangar on campus (Bldg. 004) while the car was driven further down Highland Dr, and performance was measured at varying distances. As can be seen from section 5.5, as the distance increased, the number of dropped packets also increased. Once the car got 2.5 miles away, the number of packets being dropped became unacceptable. This testing was done with as close to line of sight as was possible, but there were no fully clear lines of sight going as far as needed, so there are some interferences due to trees and elevation change.

Table 5.3: *Radio Distance Testing Results*

Distance (mi)	Packets Dropped (%)	Acceptable (Y/N)
1 mile	12%	Y
1.5 miles	20 %	Y
2 miles	33 %	Y
2.5 miles	85 %	N
3 miles	100 %	N

5.6 System Testing

Finally, everything was tested together to confirm that the system was behaving as expected. This testing was performed on the car with two other nodes on the CAN bus, with the car idling for 10 minutes. The reason for performing system testing on the car was to have real data that could be collected and compared back to what the system was truly doing to confirm that data was being logged correctly and transmitted over the radio without issue. During this time, all data was logged with no known issues. After testing the system in the aerospace hangar, the DAQ system was tested at a vehicle testing day to confirm that during normal vehicle operation, no new issues arose. This test was done while the car was driving over the course of 4 hours, mimicking a competition endurance worth of testing. During this test, there were no issues with logging data or radio communication. From all of this testing, it is clear that the system is performing adequately and will be fully capable of logging data throughout testing and competitions.

Chapter 6

CONCLUSION

6.1 Design and Testing Conclusion

The custom DAQ has been used in testing going back for the past two months. In addition to this, it was used at the 2025 Arizona Baja SAE competition [23] during the first week of May. During these instances, the DAQ has performed extremely well, having logged over 30 hours' worth of data to aggregate over 10 GB worth of data. This data has been very useful in debugging issues related to hardware failures and software failures within the transmission. One specific issue that was found at the Arizona competition was a wrong configuration for a motor controller. This was discovered by seeing the motor saturate much earlier than it should have. With this discovery, we were able to update the motor controller configuration and proceeded to improve our acceleration performance by 0.3 seconds to place 3rd in the event.

The choice to use CAN FD instead of CAN was a smart choice, as there have been no throughput issues. Additionally, it reduced the number of IDs that the DAQ must support, making it easier to validate that all messages are being sent and received properly. This has come at the cost of all nodes that communicate with the DAQ needing to support CAN FD instead of CAN 2.0, limiting options for OEM CAN sensors. However, since this project is for an engineering club, different sensing nodes can be created as projects for new members. Additionally, microcontrollers on nodes must include a CAN FD controller, or an external CAN FD controller must be added to the hardware to support communication.

Wireless communication was a neat feature that has not been present on other OEM data logger solutions. For the team, it is a useful feature, but if this project were to be a professional product, this feature would likely be removed due to its niche use case. This is due to the additional costs of adding a radio module and an antenna. If there was demand for this feature, it could be added as a separate unit or as an extra, paid feature. Despite this, when it is working, this feature has provided enough value that it is a worthwhile inclusion.

The largest failure of this project was the failure to add support for downloading files via USB. Without this, the SD card must be removed from the system to recover files. In most data loggers, files can be downloaded over USB, Ethernet, or other interfaces that do not require opening the hardware. This was one of the features requested by the customer and if the project were to be continued in the future, this is the first area for improvement.

6.2 Future Works

To improve this project for future years, some design choices could be changed and additional features that could be implemented. The biggest missing feature is the ability to download files and interact with the device over a USB interface. The hardware supports this, but due to timeline constraints and other difficulties faced throughout the project, this feature did not get implemented in time. The purpose of being able to download files is to reduce the need to open the system to remove the SD card. For a similar reason, being able to interact over a USB interface would allow for files to be cleared.

The next change that I would make would be to utilize an EEPROM to store the information about what data is being collected, different modes, and different features.

Since everything is part of the compiled binary flashed to the microcontroller, if anything changes, such as adding new data IDs or an analog sensor behavior change, the code must be re-flashed to the microcontroller. If this information could be read from or written to an EEPROM at runtime, the need to re-flash the DAQ would decrease very significantly, improving reliability and decreasing the need to open the hardware even further. This would go together with needing a USB interface available to tell the system to update the EEPROM with new values, and would likely also need to have some validation implemented, such as computing a checksum from the new data in the EEPROM and comparing it to a pre-computed checksum.

A third piece of work for the future would be to investigate different radio options or different configurations for the XBee radio for transmitting data. As discussed in the testing section, the wireless communication did work, but the frequency with which packets were being dropped was very high. There are legal issues and rules concerning communicating on frequencies not in the 900 MHz, 2.4 GHz, or 5 GHz bands, so using there is not much wiggle room in terms of the frequency, but the transmit power could potentially be increased, or a directional antenna that moves to point towards the pits could be investigated as well.

The last thing that should be done for the future would be to increase the number of sensor nodes that can communicate with the DAQ. Creating generic analog sensor nodes, thermocouple nodes, a TPMS node, or other sensing nodes that can exist as standalone units that just need to connect to the CAN bus can be done with the only change on the DAQ's end being to update the information about data being stored to include these sensors. This was not within the scope of this project as adding sensor nodes was not feasible with the timeline constraints, but with a functioning DAQ unit, these nodes would be useful moving forward.

Chapter 7

REFLECTIONS

Throughout my senior project, I learned a lot about how to manage my time, how to properly set requirements, and how to design a system that interacts with pre-existing systems and goals. For myself, the largest challenge I faced as an engineer was my time management skills. It was difficult attempting to balance getting this project finished on time while balancing the rest of my courses and holding a leadership role in a time-demanding club. Something that helped me was finding gaps of time where I could treat working on senior project as a more structured class, having 1-hour or 2-hour periods of time where this was the only task I was working on several times per week.

Something else I learned was how to change scopes or change ideas when things are not working. This finished project was not the same topic as when I first started my senior project. Originally, I was working on integrating hardware accelerators with the Zybo-Z7 processor. Specifically, I was looking into adding an FFT hardware accelerator for processing audio inputs. I struggled to work on this project due to a host of issues with poor documentation on resources for the development board, a change of academic interests away from FPGA design, and growing interests in embedded systems and signal processing projects instead. As a result, the scope of this project changed into a more pure embedded systems project that had a more attainable scope for an individual senior project. As unfortunate as it was not to be able to follow through on the original design plans for my senior project, it helped me grow as an engineer.

Finally, I was able to become a better communicator while working on this project. Even though the senior project itself was individual, it was for a club project, and as a result, there was a significant amount of communication occurring between me and other members of the club to make sure that the features that were wanted could be added so that everyone's needs were met. Communication was also critical due to this project needing to be integrated into an existing electronics system. Without strong communication, integration issues could have become a major problem.

REFERENCES

[1] *SAE International*. 2025. URL: https://www.sae.org/.

[2] *Cal Poly Racing — Cal Poly SAE Collegiate Design Series Team*. 2025. URL: %5Curl%7Bhttps://www.calpolyracing.org/%7D.

[3] *CAN bus — Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/wiki/CAN_bus.

[4] R. B. GmbH. *CAN Specification Version 2.0*. Tech. rep. Robert Bosch GmbH, 1991. URL: https://web.archive.org/web/20221010170747/http://esd.cs.ucr.edu/webres/can20.pdf.

[5] R. B. GmbH. *CAN FD Specification*. Tech. rep. Robert Bosch GmbH, 2012. URL: https:/https://web.archive.org/web/20151211125301/http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can_fd_spec.pdf.

[6] *AEM Electronics Official Website*. 2025. URL: https://www.aemelectronics.com/.

[7] *AiM Sportline Official Website*. 2025. URL: https://www.aim-sportline.com/.

[8] *MoTeC Official Website*. 2025. URL: https://www.motec.com.au/.

[9] PJRC. *Teensy® 3.5 Development Board*. 2025. URL: https://www.pjrc.com/store/teensy35.html.

[10] PJRC. *Teensy® 4.1 Development Board*. 2025. URL: https://www.pjrc.com/store/teensy41.html.

[11] *STM32H750VB Product Page*. 2025. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32h750vb.html.

[12] *PlatformIO Homepage*. 2025. URL: https://platformio.org/.

[13] Microsoft. *Visual Studio Code*. 2025. URL: https://code.visualstudio.com/.

[14] *STM32 Arduino Core Github Repository*. 2025. URL: https://github.com/stm32duino/Arduino_Core_STM32.

[15] *TCAN1043ADYYRQ1 Product Page*. 2025. URL: https://www.ti.com/product/TCAN1043A-Q1/part-details/TCAN1043ADYYRQ1.

[16] *FTDIXS Product Page*. 2025. URL: https://ftdichip.com/products/ft231xs.

[17] *NEO-M9N Module Product Page*. 2025. URL: https://www.u-blox.com/en/product/neo-m9n-module.

[18] L. Chen and S. Tran. *GPS/IMU Board*. Undergraduate Senior Project. San Luis Obispo, CA: California Polytechnic State University, San Luis Obispo, June 2024. URL: https://digitalcommons.calpoly.edu/eesp/633/.

[19] *LSM6DS1 Product Page*. 2025. URL: https://www.st.com/en/mems-and-sensors/lsm6dsl.html.

[20] *2201778-1 Product Page*. 2025. URL: https://www.te.com/en/product-2201778-1.html.

[21] *STM32SD Library Github Repository*. 2025. URL: https://github.com/stm32duino/STM32SD.

[22] *ACANFD STM32 Library Github Repository*. 2025. URL: https://github.com/pierremolinaro/acanfd-stm32.

[23] *Baja SAE Arizona*. 2025. URL: https://www.sae.org/attend/student-events/baja-sae-arizona.

[24] *Digi XBee 3 Zigbee 3 RF Module*. 2025. URL: https://www.digi.com/products/embedded-systems/digi-xbee/rf-modules/2-4-ghz-rf-modules/xbee3-zigbee-3#specifications.

Appendix A

CODE IMPLEMENTATIONS

Code Listing A.1: *Example .dbc file for an IMU*

```
VERSION ""


NS_ :
        NS_DESC_
        CM_
        BA_DEF_
        BA_
        VAL_
        CAT_DEF_
        CAT_
        FILTER
        BA_DEF_DEF_
        EV_DATA_
        ENVVAR_DATA_
        SGTYPE_
        SGTYPE_VAL_
        BA_DEF_SGTYPE_
        BA_SGTYPE_
        SIG_TYPE_REF_
        VAL_TABLE_
        SIG_GROUP_
        SIG_VALTYPE_
        SIGTYPE_VALTYPE_
        BO_TX_BU_
        BA_DEF_REL_
        BA_REL_
        BA_DEF_DEF_REL_
        BU_SG_REL_
        BU_EV_REL_
        BU_BO_REL_
        SG_MUL_VAL_

BS_:

BU_: IMU_DBC
```

```
BO_ 1304 MESSAGE_4: 8 Vector__XXX
 SG_ IMU_TEMP : 48|16@1- (0.01,0) [-20|80] "C"  IMU_DBC
 SG_ EULER_Z : 32|16@1- (0.01,0) [-180|180] "deg/s"
    IMU_DBC
 SG_ EULER_Y : 16|16@1- (0.01,0) [-90|90] "deg/s"
    IMU_DBC
 SG_ EULER_X : 0|16@1- (0.01,0) [-180|180] "deg/s"
    IMU_DBC

BO_ 1303 MESSAGE_3: 8 Vector__XXX
 SG_ Raw_GyroII_Z : 48|16@1- (0.1,0) [-1000|1000] "deg/
    s"  IMU_DBC
 SG_ Raw_GyroII_Y : 32|16@1- (0.1,0) [-1000|1000] "deg/
    s"  IMU_DBC
 SG_ Raw_GyroII_X : 16|16@1- (0.1,0) [-1000|1000] "deg/
    s"  IMU_DBC
 SG_ ACC_RAW_Z : 0|16@1- (0.001,0) [-4|4] "g"  IMU_DBC

BO_ 1302 MESSAGE_2: 8 Vector__XXX
 SG_ ACC_RAW_Y : 48|16@1- (0.001,0) [-4|4] "g"  IMU_DBC
 SG_ ACC_RAW_X : 32|16@1- (0.001,0) [-4|4] "g"  IMU_DBC
 SG_ ANGULAR_VEL_Z : 16|16@1- (0.01,0) [-125|125] "deg/
    s"  IMU_DBC
 SG_ ANGULAR_VEL_Y : 0|16@1- (0.01,0) [-327.68|327.67]
    "deg/s"  IMU_DBC

BO_ 1301 MESSAGE_1: 8 Vector__XXX
 SG_ LIN_ACC_Z : 32|16@1- (0.001,0) [-4|4] "g"  IMU_DBC
 SG_ LIN_ACC_Y : 16|16@1- (0.001,0) [-4|4] "g"  IMU_DBC
 SG_ LIN_ACC_X : 0|16@1- (0.001,0) [-4|4] "g"  IMU_DBC
 SG_ ANGULAR_VEL_X : 48|16@1- (0.01,0) [-125|125] "deg/
    s"  IMU_DBC



BA_DEF_  "MultiplexExtEnabled" ENUM  "No","Yes";
BA_DEF_  "BusType" STRING ;
BA_DEF_DEF_  "MultiplexExtEnabled" "No";
BA_DEF_DEF_  "BusType" "CAN";
```