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 FIGURES v

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 5

  2.2.2 AEM Add-Ons 6

  2.2.3 DBC Files 8

 2.3 AiM 8

  2.3.1 AiM Loggers 8

  2.3.2 AiM Add-Ons 10

 2.4 MoTeC 12

  2.4.1 MoTeC Loggers 12

  2.4.2 MoTeC Add-Ons 13

3. FORMAL PROJECT DEFINITION 15

 3.1 Customer Requirements 15

 3.2 Engineering Requirements 17

 3.3 Use Cases 18

4. SYSTEM DESIGN AND IMPLEMENTATION 20

4.1 System Level Design 20
4.2 Hardware Design 21
4.3 Software Design 21
4.3.1 High Level Design and Implementation 21
4.3.2 Storage Task 24
5. SYSTEM TESTING AND ANALYSIS 28
6. CONCLUSION 29
6.1 FutureWorks 29
7. REFLECTIONS 30
REFERENCES 31
APPENDICES
A. Code Implementations 32

LIST OF FIGURES

Figure		Page
2.1	Example CAN 2.0 Frame [2]	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	System Level Block Diagram	20
4.2	Software tasks loop diagram	22
4.3	Storage State Diagram	25

Chapter 1

INTRODUCTION

Baja SAE is an international collegiate competition run by the Society of Automotive Engineers (SAE) where teams design, build, test, and compete with offroad baja style vehicles. In the United States, there are three competitions held 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 different challenges to test a team's ability to effectively communicate design choices and business aspects of making a vehcile. Dynamic events test the abilities of the vehicle to perform in different conditions. These events include an acceleration event, a manueverability event, a suspension event, and a traction event. Finally, the endurance event tests the vehicles and drivers ability to withstand rough terrain and wheel to wheel racing for a full four hours. At the end of the competition, all the event scores are added together to determine who the top three overall teams at the competition are. To win the competition, it is crucial to perform well in all three styles of events.

Cal Poly Racing competes in the Baja SAE series of competitions. Over the last several years, Cal Poly Racing has had a moderate amount of success, with several place tropies in dynamic events. The Baja vehcile 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.

In order to design a vehcile capable of withstanding all the harsh events in a Baja SAE competition, it is critical to fully understand how every system of the car is behaving. Additionally, undertstanding the load cases, such as impacts, is also essential to making informed design choices and considerations. To understand the the vehicle and the load cases, a data collection system is needed to log and process sensor data. Data Acquisition Systems (DAQs) are tools that allow for the logging and processing of data.

Chapter 2

BACKGROUND

2.1 CAN

Controller Area Network (CAN) busses are commonly used in automotive applications to connect different control or instrumentation nodes together developed by Bosch in the 1990s. This allows 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 signle 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 all also contribute to overhead.

2.1.1 CAN 2.0

CAN 2.0 is the most commonly used CAN protocol in the automotive. This version of CAN uses an 11 bit identification or address section with maximum of 8 bytes of data transmitted. The bitrate for this version of CAN can be up to 1 Mega bit per second (Mbps) [4]. 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.

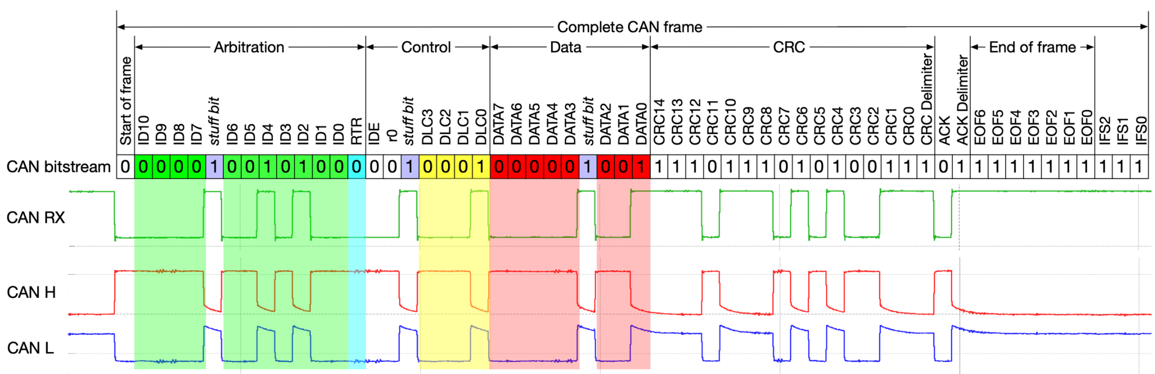

Figure 2.1: *Example CAN 2.0 Frame [2]*

2.1.2 CAN FD

In 2012, Bosch released a new version of CAN with a flexible data rate called CAN FD. 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 [3]. This allows for significantly more data throughput overall. In addition to the flexible data rate used by CAN FD, the maximum size of the data section was increased from 8 bytes to 64 bytes [3]. This decreases total overhead significantly making the transmission more efficient. These two new characteristics combined provide for a much more efficient transfer of data, particularly when transmitting high amounts of data very frequently.

In order 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. The system would theoretically operate without too many issue, but the control section of the frame would be different sizes to accomondate the additional control bits needed for the CAN FD frame.

2.2 AEM

AEM designs data loggers and dashboards for testing purposed for cars and other vehicles. These dataloggers connect to their different modules that connnect to various can sensors and breakouts for different types of sensors. Some of these modules also come with an inertial measurement units (IMU) and 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 purcahse. 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 [1]. This cost provides access to data logging features as well as a dashbaord display. 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 seperated file (csv) format for ease of use with tools such as Matlab or Python for visualizing the data. These files can also be visualized on the dashboard as well to give a quick glimpse at explaining what is happening.

2.2.2 AEM Add-Ons

In order 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 for measuring temperature. These expansion units, before the cost of any sensors, are several hundred dollars on their own. In order 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. In order to communicate with these wireless sensors, a wireless module in needed or a logger with wireless capabilities is needed. AEM utilies a propietary system call X-Wifi to connect to these wireless sensors that is 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 more traditional wired sensors.

2.2.3 DBC Files

In addition to AEM specific sensors, the AEM dataloggers can interface with any custom hardware that is outputting it's data over CAN. In order 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 standarized file format for configuring which is able to describe 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 dataloggers to interface with a CAN network so long as the proper hardware add-ons are purchased.

2.3 AiM

Another company that makes dataloggers to attach to a vehicle is AiM Sports. AiM makes dataloggers that are specifically intended for motorsport applications as well 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 senors 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. The MX series and MXM series loggers are comparable to th 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 also includes and on board IMU and GPS, making it easier to monitor vehicle position and behaviour during a lap.

2.3.2 AiM Add-Ons

In order to communicate with sensors or other pieces of information, there are a few required accessories needed. These are a datahub 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 expenive requirements for using sensors.

Besides these essential add-ons, there are some other useful add-ons. AiM makes a GPS add-on that can be added provide real 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 all of the hardware accessories to improve the experience and connectivity of AiM data loggers, AiM also sells many different kinds of sensors for automotive applications. These include temperture sensors, pressure sensors, combined temperature and pressure sensors, position sensors, speed sneors, and rpm sensors. Most of these sensors are analog voltage outputs, meaning that in order to communicate with the data logger, these sensors must be conncted to a the logger directly or to a unit that can transmit the data over CAN or other supported communication protoocls. These sensors, similar to 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 that they call CAN builder to setup 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 propietary 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 datalogging solutions for automotive application is MoTeC. MoTeC is an automotive aftermarket company owned by Bosch that focues on making electronic equipment for race sport uses. They make ECUs, data loggers, dashboards, 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 functionailty 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 approximatel $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. In order 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 compatability. MoTeC makes an expander unit, similiar 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 the SVIM, but includes thermocouple specific inputs. This unit additionally is more limited on which loggers it can connect to, only being compatabile 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. Similiarly 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 purchased from a third party retailer. These sensors are somewhat expesnive but comparable in price to the sensors made by AEM and AiM.

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 will include 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 as such am reesponsible for defining the customer requirements for this project. These requirements come from a combination of the existing systems on the car, desired data that is wanted to be collected, 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 sensor 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
- Low enough powere draw to not impact the battery life of the vehcile

The storage rate comes from the controls loop rate of the fasted controller on the vechicle. Since there are no current plans to increase the controller frequencies or add any sensors that require a faster sampling rate, this can be the minimum achievable collection rate for a while. The endurance event at a baja event lasts for four hours. Accomondating for time spent waiting for race to start and any final checks just before the event, five hours should accomondate a full endurance days worth of data. In order 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 often was the case that the user 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. 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 would not want to be left on the vehicle permanantly. 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 and they are always wanted. Finally, since the majority of the team are mechanical engineers, it is important that all data is easily accessible and that everyone is able to view the data.

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 to.

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 behaviour 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 use cases for this system:

(1) To collect vehcile and subsystem data to validate design models

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

(3) To diagnose unexpected failures of a system

In the first case, an engineer is attempting to validate a braking model for the car. The model is a 9 degree of freedom model and uses brake pressure, shock heights, vehcile acceleration, and other paramaters to determine when the wheels will ”lock”, or break traction. In order to validate the model for our design for information for the next design cycle as well as the design event part of competition, these sensor's data is needed. This is the case for several different vehicle dynamics models used to design the baja car.

In the second case, 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 vehicles health and criticial systems during this event, we can view wirelessly transmitted data. This helps the pit crew know if or when to expect the vehcile coming back to the pits for a repair.

In the third case, we are at testing or at competition, and something breaks or starts performing worse unexpectedly. 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 a failure can help an engineer fix or improve upon their design to reduce risk of failure in the future.

Chapter 4

SYSTEM DESIGN AND IMPLEMENTATION

0–7	8–15	16–31
Byte Shift	Number of senders	Number of messages

0–7	8–15	16–23
Field2	Field3	Field4

4.1 System Level Design

In order to accomplish all of the tasks necessary for this DAQ, there were several design choices that

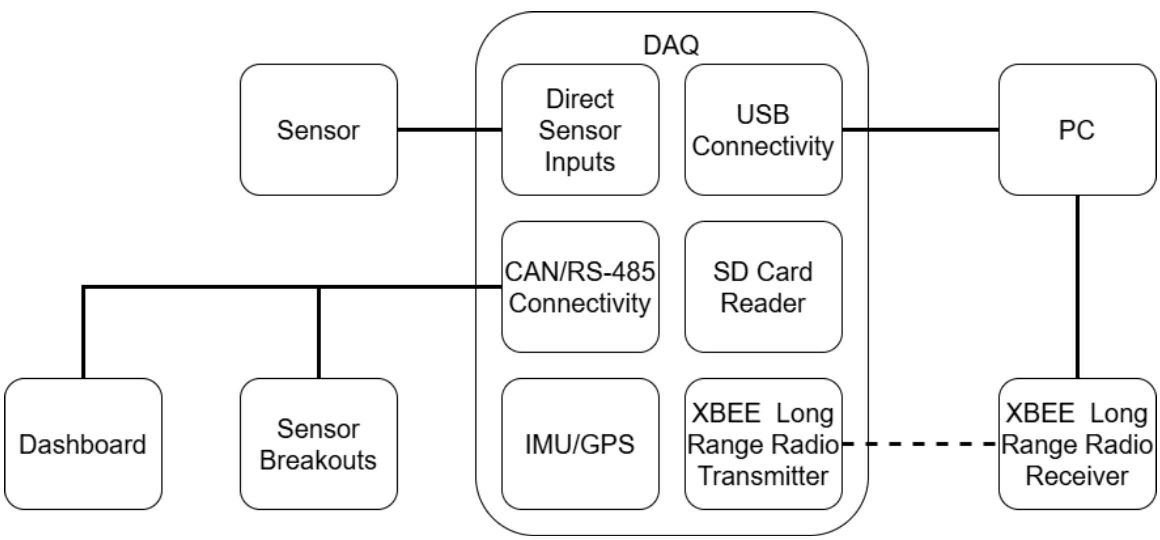

Figure 4.1: *System Level Block Diagram*

4.2 Hardware Design

4.3 Software Design

The majority 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 the ways in which they were addressed or what needs to happen to address the issue.

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 there are several different tasks that are running 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 ther does not need to be any extra protections 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. 4.2, these tasks all run sequentially before looping back to the start.

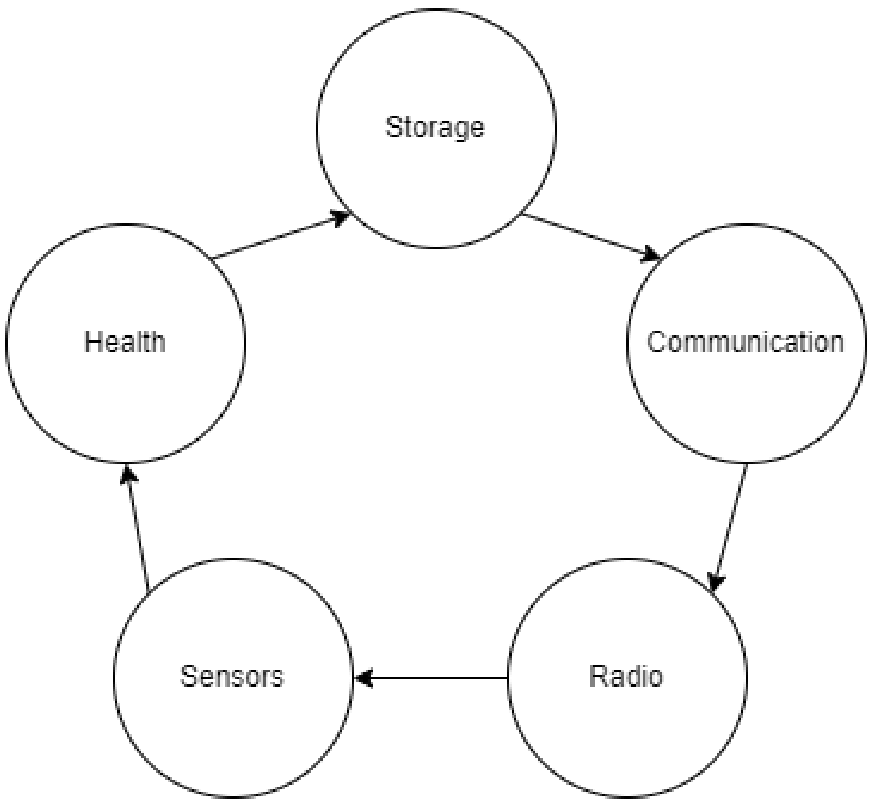

Figure 4.2: *Software tasks loop diagram*

Object-oriented programming (OOP) is utilized significantly for this design. Every task and sensor is a class that have individiual 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. There are two parent classes that 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 periphreals 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 each other. 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 microcontrollers from STM receives the same support or even the sample supported functions from the Arduino framework. One specific issue that arose was a versioning issue for STM32duino. In versions 18.0.0 onwards, the memory sizes of the H7 family of STM microcontrollers was 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. Other issues that arose that were not issus 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 deebugger, many of these issues would not have been solved or would have taken weeks or longer to debug.

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. 4.3. This machine consists of seven states, with each state handling a single part of the task. These state 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 unexpectandtly. 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.

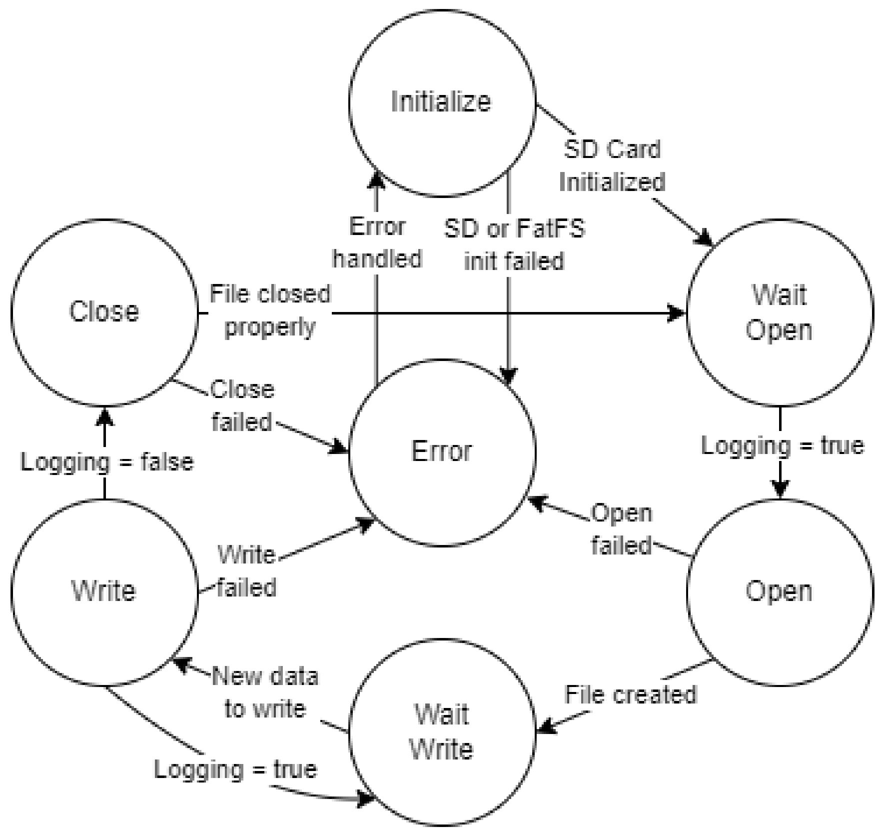

Figure 4.3: *Storage State Diagram*

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 filesystem that is used by the SD card. The creation of the SD and File objects relies on STM32SD library. This library provides the functions for initializing the SD interface and for mounting the filesystem. 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 approriate arguments, or a wrapper function f_unmount can be created that calls f_mount with the approriate 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. This is outlined in the decision tree in INSERT FIGURE.

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 indefinately 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. This is outlined in INSERT FIGURE. 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 indefinately in this state.

The Write state is responsible for actually writing the most recent data to the SD card.

Chapter 5

SYSTEM TESTING AND ANALYSIS

Chapter 6

CONCLUSION

6.1 FutureWorks

To improve upon this project for the future, there are a few additional features and a few design choices that could be imporved upon. From a hardware design perspective, the storage device could be improved upon to use a file system that supports larger file formats. This would allow for more files to be stored on the DAQ without the need to clear its storage as frequently. Additionally, two additional design choices could be made to improve the users ability to connect to and interface with the DAQ. These would be to replace the UART serial to USB interface with an ethernet connection and to add a short range wireless connection option. The wireless option would allow for an easier way to connect to the DAQ for more people. This would require the addition of some sort of wireless wifi or bluetooth transceiver or to use a microcontroller that supports a wireless interface. By replacing the USB interface with an ethernet based interface, faster file downloads can be achieved. Since the download speed via USB is limited by the stablity of the baud rates for UART, a more stable interface would allow for much faster transfers to occur, removing one of the larger bottlenecks in the system.

Chapter 7

REFLECTIONS

REFERENCES

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

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

[3] 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.

[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.

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";
```