York College of Pennsylvania Capstone Design

FSAE

Spring 2024

Abstract

From year to year the main goal of a FSAE team should be iterative improvement on previous designs. One clear avenue of improvement was to make our data acquisition wireless to allow more immediate viewing of important vehicle KPIs. The more immediate access would allow our team to better tune and troubleshoot our car before and during competition. Other improvements include better electrical wiring practices and adherence to standard coding practices. This report will in detail cover design considerations, system requirements, fabrication, coding, and testing methodology.

Table of Contents

1. Introduction

1.1. About this Project

The FSAE design competition is an annual event held at the international speed way in Michigan. The event is sees engineering students design, fabricate, and compete against each other with formula style vehicles. The created car is run through a series of technical inspections and on track tests. Scores are determined through these tests and quality of engineering skills demonstrated throughout the event. At YCP our team is subdivided into engine, body, aero, data acquisition and electrical teams. As a computer engineering student, I played a crucial role as a middleman between electrical and data acquisition. My contribution consisted of making a new data acquisition system to allow for real time data, the cars dashboard and aiding the data acquisition team with sensors.

1.2. Background Information

1.2.1. DAQ System

Past year teams have utilized a mixture of the ECU's onboard data acquisition system and aftermarket DAQs. The clearest way to improve the system is provide more immediate feedback to the team for troubleshooting. As to use the ECU or DAQ the data would have to be retroactively examined after testing. Other possibilities for improvement are a more complete sensor suite such as adding GPS or more accurate instrumentation.

1.2.2. Dashboard

Prior teams used a combination of multistate indicator lights and a tachometer in the form of a LED stick. The dash's main purpose is to also allow the affixing of various switches for controlling ignition, power, and a required E-Stop.

2. Experimental Method

2.1. Development of Engineering Requirements

Development of requirements for this project is heavily dependent on professors' direction and team's needs. Once a direction was set it's up to the students to design and create an appropriate system for the cars overall functionality. The team's needs shift over the projects progression so being flexible is important to meet the FSAEs competition deadline.

2.1.1. Environmental and Safety Concerns

The DAQ system and associated dash electrical systems are 5V. Due to low voltage the system is inherently safe. The on-board microcontroller handles no safety critical control over any part of the car. As this would violate FSAE rules [1]. Environmentally all effort was used in reusing old cabling and other components to reduce waste.

2.1.2. Societal and Cultural Considerations

With the focus of the project being a design competition on creating a formula vehicle the project inherently has little impact on society. As for cultural considerations the FSAE competition itself heavily plays into car culture. The competition brings together large manufacturers and many young engineers who will work within the industry forever shaping car culture.

2.1.3. Sustainability

Generally, year to year the much of the prior team's parts are reused to cut costs and be more sustainable. The engine, ECU wheels, seat and more are consistently refused to this effect. Electrical connectors and some standard wiring harness pieces are also recycled into each year's car.

2.1.4. Constraints

To have a fully realized real time DAQ system the data would need to be read multiple times per second. The required data rate needed to achieve this is based off PE3 documentation regarding CAN communication [Appendix A, 1.0-1.1]. Each data point is sent as a big endian two-byte value. Therefore, the required data rate will be tied to the number of datapoints. This ended up being 22 data points so 44 bytes of data plus a 4-byte header for LoRa. Therefore, at achieve at a minimum of 4 packets per second a data rate of 1.5 Kbps. However, LoRa also uses CRC therefore the required data rate is at least double. In practice a higher data rate would be advantageous to allow for code execution time in packetizing, parsing, and saving data to a database. See appendix A figure 1.2 for the final listing of available data. The system must also be easy to use and require little user interaction to function properly.

As for physical constraints reusing the old PE3 8400A ECU was a requirement. The ECU calculates several important datapoints as the ECU uses said information to run the engine [3]. The ECU has several analog and digital inputs for sensor input as well as a CAN bus connection for transferring said data [Appendix A 1.3- 1.4, 3 & 4] This constrained us to 8 analog inputs and 7 digital inputs. This proved to sufficient for the sensors specced by the system integrations team. The second physical limitation concerned the cars dash. The overall area is quite small and most of the real estate was covered by the steering wheel. The effective area is allotted to the outer edges of the dash. I decided to advocate for as much space as possible and to work the design to what the frame team provided once built.

The budget proved to be the third most important constraint. The engine, electrical and body portions of the car are truly the only parts required to make a functional car for

competition. Therefore, all other parts including a DAQ system must be justified in usefulness and cost. While an upper limit of budget was not formally discussed. It was crucial that such a systems cost should be small compared to other parts of the car. The total cost target set by me was to not exceed $350 for such a system.

Constraints	Target	Actual
Cost	Under $350	$326.87**
Data Rate	3 Kbps	6.25 Kbps
Sensor Inputs	6 Analog - 4 Digital	8 Analog – 7 Digital

Table 2.1 System Constraints **see BOM Appendix B

2.1.5. Standards and Codes

The rules set by FSAE do not cover anything regarding a DAQ system [1]. Most standards observed involved electrical safety including proper covering for 12V connections and safety switch placement on the dash. Within the electrical team standards were set for wire coloring, wire labels and wire management. Typical coding practices were observed including white space, clear comments, and the use of a GitHub repository [5].

2.2. Subsystem Design and Fabrication

Considering the design requirements and timeline of completion by early May some of the overall design changed from the summer semester. Firstly, the ESP32 was replaced with an even smaller faster board. This replacement was the RP2040 RFM to handle both LoRa communication and to use a prebuilt CAN hat from Adafruit. The second largest change was shifting from a custom coded website for viewing data to an open-source solution. Grafana was chosen due to time constraints and already having the features desired for viewing data [6]. It was easy to implement and already has native support for running on a Raspberry Pi.

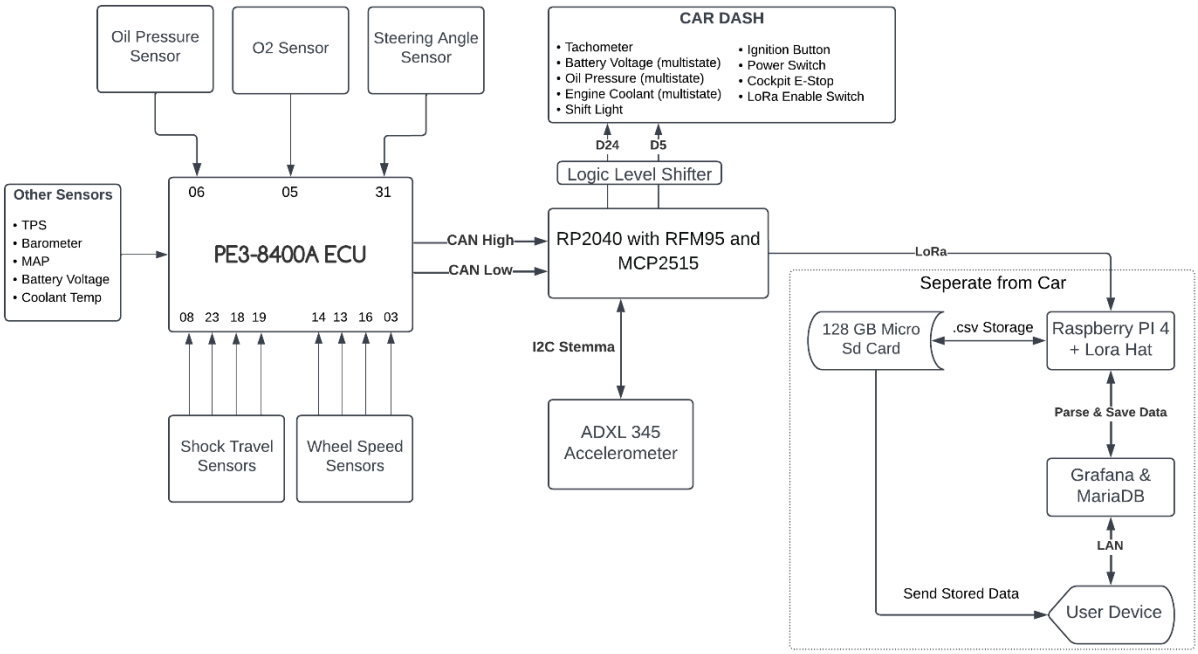


Figure 2.2 – Overview of DAQ System

The DAQ system consists of three main components to read and process data. The PE3 8400A ECU, RP2040 RFM with CAN controller and a Raspberry Pi 3B+. Sensors are polled by the ECU and sent to the RP2040 via its CAN bus on regular intervals. Said data is packetized and sent off car to the Raspberry Pi via LoRa at 915 MHz. The Raspberry Pi receives the data and reparses, scales, and converts the data to appropriate units. Finally, the data is saved to a MariaDB on the Raspberry Pi. To view or download data a user can connect to a LAN network with their laptop. Login to Grafana and view data dashboards and download data at their convenience. Each packet of data is timestamped for graphing data overtime.

2.2.1. RP2040

The RP2040 serves two purposes, it parses and sends CAN data to the Raspberry Pi and controls the cars dashboard. To parse CAN data an add on MCP2515 can controller was added that has a I2C interface and a very easy to use library. CAN packets are sent on regular intervals outlined in appendix A figures 2.0-2.1. Each data point is a 2-byte value formatted in big endian. To keep coherence the data is packetized for LoRa in the exact

same format as CAN. Looking at the below figure 1.2 it shows in totality of how the data is packetized. Note when sending a LoRa packet in RadioHead the used library, a 4 byte header is added. Aside from CAN data a ADXL345 accelerometer was also interfaced with the RP2040. This provides 3-axis accelerometer data via an I2C connection and is included in the LoRa packet. See Appendix C for full source code.

CAN ID	Name	Poisition in CAN Packet	Position in Lora Packet	Type	Resolution per Bit	Range	Position in Raspi code
0CFFF048	RPM	0-1	0-1	Unsigned Int	1	0 -30000	4-5
	TPS	2-3	2-3	Signed Int	0.1	0 -100	6-7
	Fuel Open Time	4-5	4-5	Signed Int	0.1	0-30	8-9
	Ignition Angle	6-7	REMOVED	Signed Int	0.1	-20-100	REMOVED
0CFFF148	Barometer	0-1	6-7	Signed Int	0.01	0-300	10-11
	MAP	2-3	8-9	Signed Int	0.01	0-300	12-13
	Lambda	4-5	10-11	Signed Int	0.01	0-10	14-15
	Pressure Type	6-7	REMOVED	Int	N/A	0 - psi, 1 -kPa	REMOVED
0CFFF248	Analog 1	0-1	REMOVED	Signed Int	0.001	0-5	REMOVED
	Analog 2	2-3	12-13	Signed Int	0.001	0-5	16-17
	Analog 3	4-5	14-15	Signed Int	0.001	0-5	18-19
	Analog 4	6-7	16-17	Signed Int	0.001	0-5	20-21
0CFFF348	Analog 5	0-1	18-19	Signed Int	0.001	0-5	22-23
	Analog 6	2-3	20-21	Signed Int	0.001	0-5	24-25
	Analog 7	4-5	22-23	Signed Int	0.001	0-5	26-27
	Analog 8	6-7	REMOVED	Signed Int	0.001	0-5	REMOVED
0CFFF448	Freq 1	0-1	24-25	Signed Int	0.2	0-6000	28-29
	Freq 2	2-3	26-27	Signed Int	0.2	0-6000	30-31
	Freq 3	4-5	28-29	Signed Int	0.2	0-6000	32-33
	Freq 4	6-7	30-31	Signed Int	0.2	0-6000	34-35
0CFFF548	Battery Voltage	0-1	32-33	Signed Int	0.01	0-22	36-37
	Air Temp	2-3	34-35	Signed Int	0.1	-1000-1000	38-39
	Coolant Temp	4-5	36-37	Signed Int	0.1	-1000-1000	40-41
	Temp Type	6-7	REMOVED	Int	N/A	0 - F, 1 - C	REMOVED
ADXL345	Acceleromter X	N/A	38-39	Signed Int	0.01	-16 - 16	42-43
	Acceleromter Y	N/A	40-41	Signed Int	0.01	-16 - 16	44-45
	Acceleromter Z	N/A	42-43	Signed Int	0.01	-16 - 16	46-47

Table 2.2.1 – Data available to DAQ system see GitHub for most up to date version [5]

2.2.2. Dashboard

The cars dashboard as mentioned is controlled by the RP2040. It uses several neopxiels behind plastic diffusers for displaying the state of coolant temperature, oil pressure and battery voltage. Alongside these indicators are a neutral light indicating if the cars transmission is in neutral. This is accomplished by utilizing the neutral switch that comes preinstalled with the Honda CBR 600rr engine. By simply setting a GPIO pin to be a pullup Input the state can be easily checked. If pulled to ground or LOW by the neutral switch, then the car is in neutral. Below in figures 2.3 – 2.4 the ranges for the multistate and colors are shown. As well as the RPM ranges for the neopixel tach light. These ranges are set based on last years vehicle and may change before competition.

Value	Range 1	Range 2	Range 3
Battery Voltage	Less than 11.8 V	Between 11.8 and 12 V	Greater Than 12V
Oil Pressure	Less than 10 PSI	Greater than 10PSI	N/a
Coolant Temp	Less than 150 °F	Between 150 °F and 190 °F	Greater Than 190°F
Neutral Light	off	On	N/a

Table 2.2.2 – Dash light ranges

RPM	L1	L2	L3	L4	L5	L6	L7	L8
0 < RPM < 1800	off	off	off	off	off	off	off	off
1800 < RPM < 3100		off	off	off	off	off	off	off
1800 < RPM < 3100			off	off	off	off	off	off
3100 < RPM < 4400				off	off	off	off	off
4400 < RPM < 5700					off	off	off	off
5700 < RPM < 7000						off	off	off
7000 < RPM < 8300							off	off
8300 < RPM < 9600								off
9600 < RPM < 10900								

Table 2.2.3 – Tachometer Ranges

Figures 2.5 – 2.9 show the construction process and finished dashboard on the car. The included switches from left to right are the LoRa enable, main power, ignition, and E-Stop. The only missing requirement is a symbol, marking the E-Stop as a shutoff switch. This is required per FSAE rules and will be added before competition [1]. The construction of the dash was done at my home woodshop. A template was laser cut based on the cars CAD model. Then the template was used to cutout a walnut board to shape. It was flush trimmed with a router and given a ¼" round over on its edges. Finally, a finish of oil and carnauba wax was applied. The mounting holes were also drilled to accommodate the lights and switches.

Figure 2.2.4 – Attached Template

Figure 2.2.5 – Oil and Wax Finish

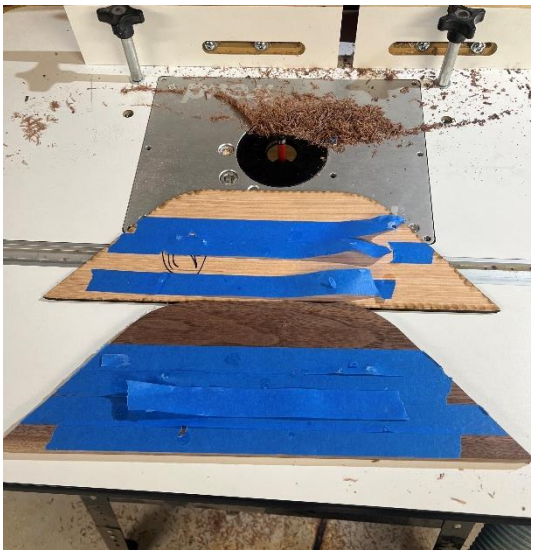


Figure 2.2.6 – Flush Trimmed to Shape

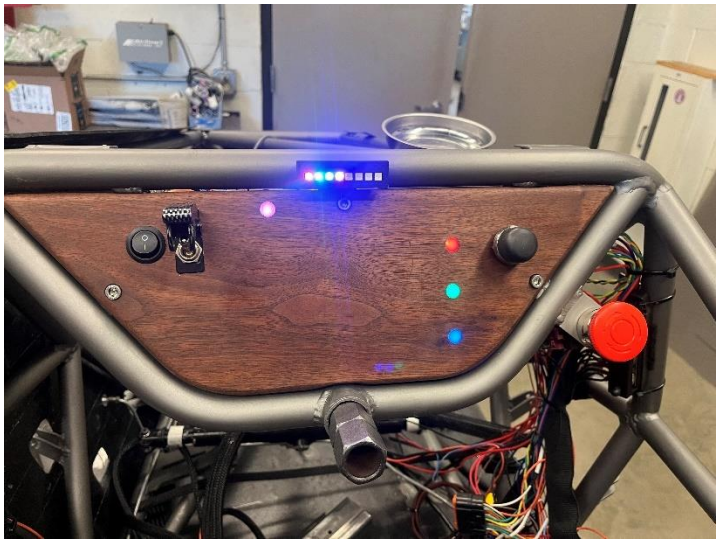


Figure 2.2.7 – Dash During Testing Process

2.2.3. Raspberry PI / Database

The Raspberry time soaked up a large portion of my spent time on this project. Up until this project I've never used a Linux based computer. It made troubleshooting tedious due to my unfamiliarity with the OS. The python code on the other hand, was quite trivial. It simply waits for a LoRa packet to be received with a specific header. When received it parses and converts the data to the correct units and resolution. It then saves the data in a Maria database. MariaDB is a fork of MYSQL and functions very similarly to MYSQL and is available in the Pi OS repository. Below is the layout of the main sensor data table used to store data. I picked a SQL based language as it's something I'm familiar with and was supported in Grafana. Initially I attempted to use SQLite however it wasn't properly supported in Grafana. I didn't leverage the relational aspect of SQL and just of easily used a NoSQL based language instead. However, for sake of simplicity and trying to keep the code easy to modify for later students I opted for MariaDB.

Name	SQLite Column Name	Data Type
RPM	RPM	Integer
TPS	TPS	Real
Fuel Open Time	Fuel_Open_Time	Real
Barometer	Barometer	Real
MAP	MAP	Real
Oil Pressure / Analog 2	Oil_Pressure	Real
Shock Pot FR / Analog 3	Shock_Pot_FR	Real
Shock Pot FL / Analog 4	Shock_Pot_FL	Real
Shock Pot BR / Analog 5	Shock_Pot_BR	Real
Shock Pot BL / Analog 6	Shock_Pot_BL	Real
Steering angle / Analog 7	Steering_Angle	Real
Wheel Speed FR / Freq 1	Wheel_Speed_FR	Real
Wheel Speed FL / Freq 2	Wheel_Speed_FL	Real
Wheel Speed BR / Freq 3	Wheel_Speed_BR	Real
Wheel Speed BL / Freq 4	Wheel_Speed_BL	Real
Battery Voltage	Battery_Voltage	Real
Air Temp	Air_Temp	Real
Coolant Temp	Coolant_Temp	Real
Acceleromter X	Accelerometer_X	Real
Acceleromter Y	Accelerometer_Y	Real
Acceleromter Z	Accelerometer_Z	Real
Timestamp	Timestamp	Timestamp

Table 2.2.8 SQL table layout

Grafana as previously mentioned is an easy-to-use open-source data visualization software. It allows user to view previous and current data with various graphs and graphics. Grafana has decent Raspberry Pi support and is very easy to install and learn how to use. To allow users to access in the field I opted to use the Raspberry Pi's wireless LAN hosting capabilities. This can be enabled either by installing and configuring several packages or using RaspAP. RaspAP is an opensource software for configuring the Raspberry Pi's wireless capabilities. This allows users to easy access Grafana and download data from it directly to their laptop. The data is formatted as CSV data and then can easily be used for retroactive troubleshooting or analysis. See appendix C for full code.

3. System Validation, Testing and Integration

3.1. System Integration

Integrating this three-part system with the sensor suite is the most crucial aspect of having a functional system. Doing a thorough checkout ensures everything is functioning as expected and is common practice in industry. However, doing a full checkout is something only possible once the is drivable.

3.1.1. Rp2040

The RP2040 was utilized in combination with the Raspberry Pi as the primary testbed for LoRa. LoRa has several important parameters that change how it functions. These parameters are Spreading Factor, Coding Rate, and Bandwidth. Each affects the functionality of LoRa differently between data rate, distance, and reliability in a noisy environment. Specifics can be found on "thethingsnetwork.org" [7]. However, generally a higher spreading factor slows data rate but adds more distance. Coding rate adds more

error correction bits effectively slowing data rate but making it more resistance to noise. Finally, a lower bandwidth will allow for greater range again at the cost of a lower data rate. To pick the LoRa settings I experimented with sending chunks of 1000 packets with various LoRa settings. This was tested in an open area with the approximate distance of 1200ft between the RP2040 and Raspberry Pi. Below in figures 3.0 – 3.3 show the resulting graphs and data from testing. TBP is time between packets measured in milliseconds and the x-axis is SF and Coding Rate settings.

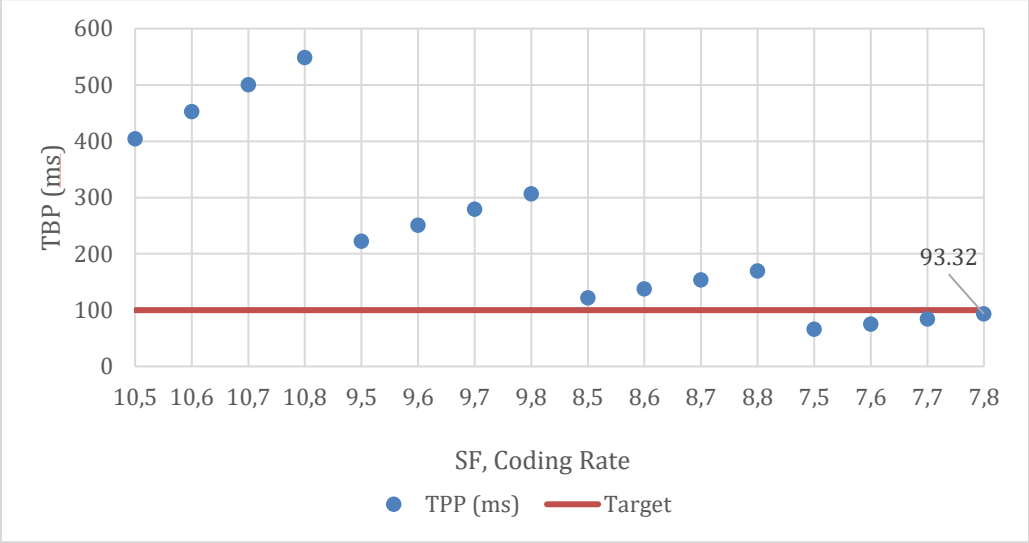


Figure 3.1.1 – 125kHz bandwidth

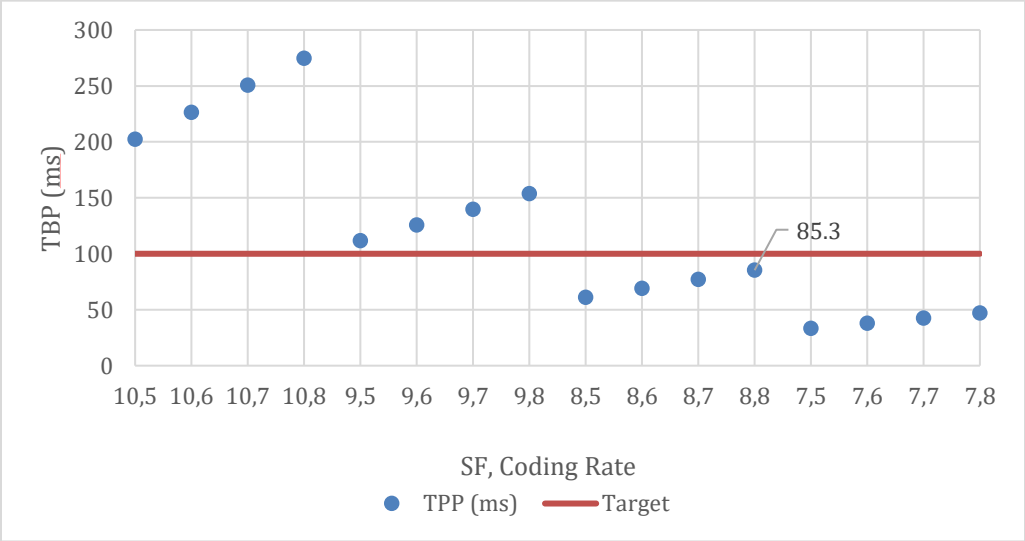


Figure 3.1.2 – 250kHz bandwidth

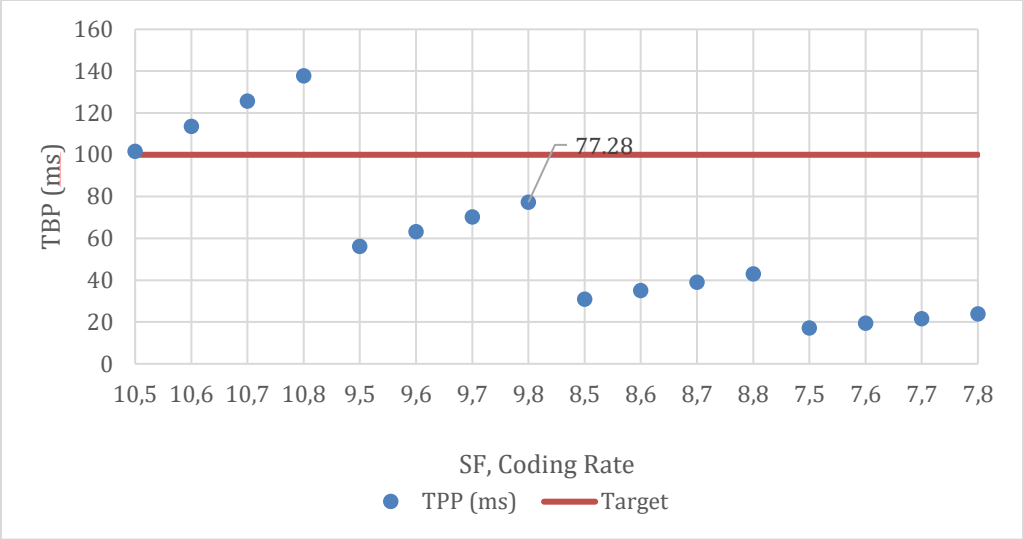


Figure 3.1.3 – 500kHz bandwidth

Bandwidth	Coding Rate	SF	Time to Send a Packet (ms)	Packet Loss	Average RSSI	Average SNR
500kHz	8	9	77.3	2.16%	-96.0	-2.7
250kHz	8	8	85.3	0.96%	-95.9	2.1
125kHz	8	7	93.3	0.88%	-95.6	3.7

Table 3.1.4 – Additional relevant test data

From the above figures I selected the closet data point below the 100ms mark. This allows padding for removing or adding more data points to the system. This also allows plenty of time for the data to be parsed saved and viewed on Grafana for live viewing of data. While also selecting a data point with good noise resistance and range. I Settled on using 250kHz bandwidth, coding rate of 8 and a spreading factor of 8. This setting was a middle ground and can easily be adjusted as needed for more range or higher data rate. As for evaluating the rest of the code, it was verified through examining data on the Raspberry Pi. Using a known input and tracking it through the system to verify that all data was managed correctly. The RP2040's writing was also verified and checked out as sensors on the car became available to test.

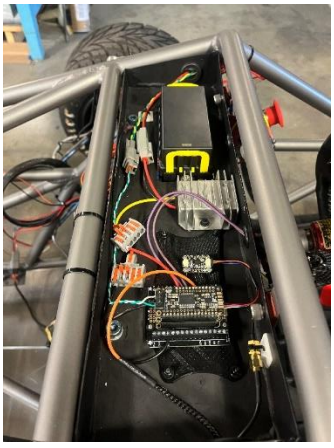


Figure 3.4 – RP2040 installed

3.1.2. Dashboard

The Neopixels lights and ranges were tested using pseudo data. This testing was done off car and the lights preformed as expected. The switches were verified with a multimeter and through first test run of car's engine. When initially running the car the Neopixels would display incorrect colors and sometimes freeze up. According to the Neopixels datasheet they are 3.3V logic level tolerant. When doing initial testing this was found to be true but in practice on the car this was not the case. Adding a logic level shifter resolved this issue. The other possibility is to power the Neopixels off 3.3V from the RP2040. While this would also solve the issue it would reduce the max brightness of lights. For an indoor application this would suffice. However, for the car the full range of brightness is a necessity when outside on a sunny day.

3.1.3. Raspberry PI / Database

Similarly to the RP2040 the majority of checkout involved using pseudo data. This ensures data is passed along and converted correctly. This also allows checkout from ECU all the way to the users end on Grafana. In terms of integration the only major change made to the Raspberry Pi was removing some of the conversions. This is because the PE3 ECU can have set conversions in its tuning software. This saves processing time when parsing data into the database. Making it as it is as simple as an insert statement for processing data. Another key step is the creation of a sort of user guide. While the system requires little user intervention, there's still needs to be a cleared outline process on its use. The documentation should outline a step-by-step startup guide, troubleshooting steps and basic information on the system. This will be to both aide later teams who may want to use this system and its use at competition.

The only large issue found in this entire DAQ system was a timestamp issue. The Raspberry Pi lacks the circuitry to keep its own time. This is something common in computers but the Raspberry Pi due to its barebones nature. It relies solely on using an NTP to reconcile its hardware clock. This resulted in timestamps in Grafana being in the past. While the user's laptop was up to date when viewing data. To combat this a simply RTC hat was added. This Real Time Clock uses a crystal oscillator to track time much like

any other computer. A minor issue faced is the SQL server's reliability and LoRa hat's reliability. The hat sometimes throws an incorrect wiring exception on startup. The hat is press fit into the Raspberry Pi's header pins. Therefore, the only conclusion I can draw is that the hat's connectors are loose to the header pins. As for the SQL server it has on one occasion corrupted and refused all attempts to start up the service. A fresh reinstall can fix this buts it requires direct reinstall on the Raspberry Pi.

4. Summary & Conclusions

4.1. Fulfilment of Design Goals

The system in some ways more than surpassed its laid-out goals. With the current amount of data being sent it averages 4 -5 packets of data per second. This effectively over samples the available CAN data. Meaning that there is no data loss compared to just logging the data on an aftermarket DAQ. The only potential for packet loss is in the LoRa transmission itself. If line of sight is kept from the Raspberry Pi and car from testing it should not be an issue. As for visualization its highly customizable within Grafana and can be tailored by later teams if they wish. The dashboard is aesthetically pleasing and acts as a place for important driver controls. Finally, the overall cost summed up to just over $300 (see Appendix B).

4.2. Challenges

Through the FSAE project this semester the largest challenge was coordination between teams. As the largest capstone ensuring things getting done within a proper timeframe can be challenging. This proved especially true for the DAQ system as the entire car needs to be in working order to verify the entire system. With limited time at the end of the semester with a working car it really hurts the usability of a system such as this. Other challenges include my own unfamiliarity with the mechanical aspect of cars. While

I have learned quite a bit through spending time with the team a better base knowledge would've allowed me to make headway much faster.

4.3. Future Work

While the system did fulfil all expected goals it by no means isn't without flaws or room for improvement. I believe that the LoRa settings and data rate could be better dialled in to provide more reliability. This can be accomplished though more testing but wasn't due to time constraints. Secondly, some more in-depth documentation could be made on the code itself. Due to how I coded the Raspberry Pi and RP2040 its easily to modify the packet size and what data is being passed. This was to allow later teams to modify and use much of my work for later systems. However, I don't expect non computer engineering student to be able to figure out the code without documentation. The GitHub repository will be a great asset however nothing beats a step-by-step guide. For another future improvement, the Grafana dashboards could use some improvement. I'm by no means an expert at data visualization. To improve them a sit-down meeting with key team members would aide in refactoring and adding needed displays. An additional potential enhancement could be making a configuration wizard changing the needed coded parameters for different data.

5. References

[1] SAE International. (2023, September). FSAE_Rules_2024_V1.

[2] AN400 Rev C– application note can bus protocol for pe3 ... (n.d.-a). https://pe-ltd.com/assets/AN400_CAN_Protocol_C.pdf

[3] Pe3 series Engine controller manual. (n.d.-b). https://pe-ltd.com/assets/pe3-series-manual.pdf

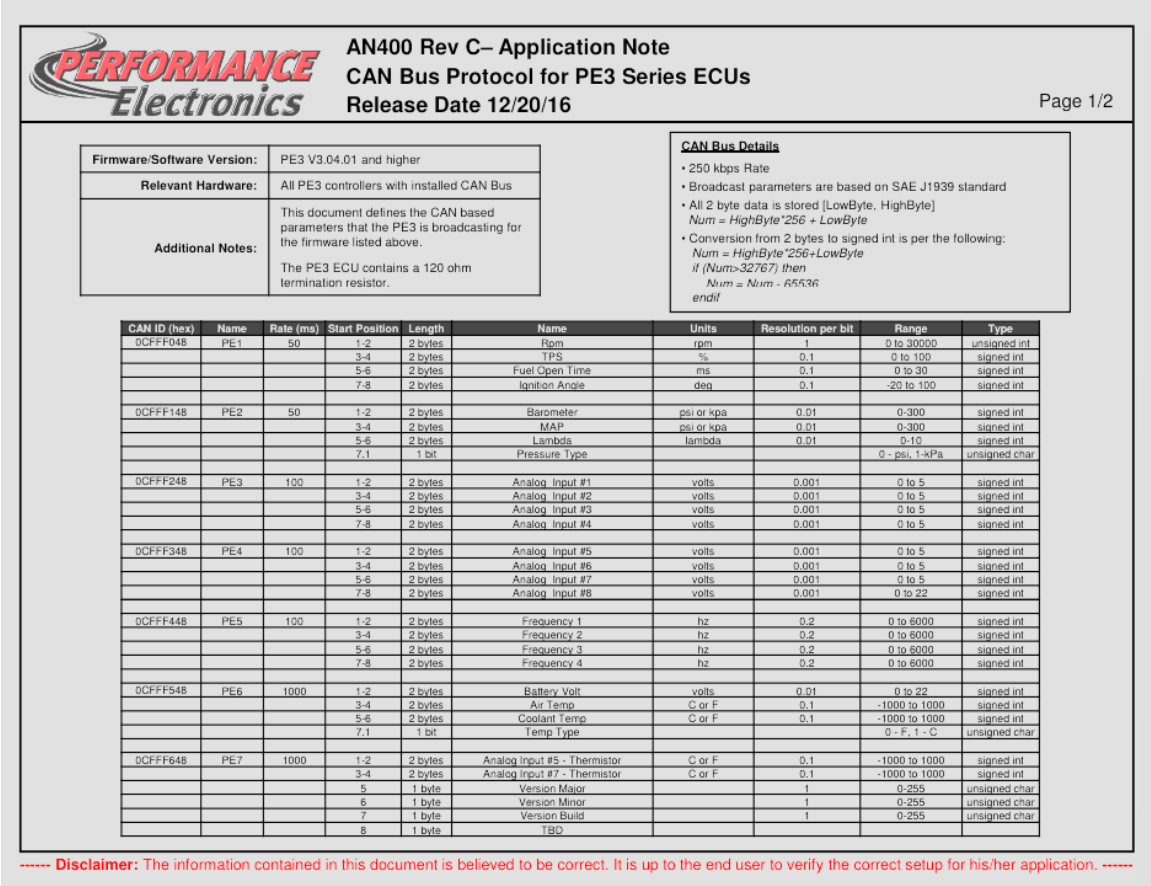
[4] Performance Electronics. (2011, October). PE3_Wire_Diagram_R4. https://pe-ltd.com/assets/PE3_Wire_Diagram_R4_10-29-11.pdf

[5] Kelller, J. (n.d.). *JKELLER11/FSAE2024*. GitHub. https://github.com/jkeller11/FSAE2024

[6] Grafana: The Open Observability Platform. Grafana Labs. (n.d.). https://grafana.com/?pg=community&plcmt=topnav

[7] *Lorawan*®. The Things Network. (n.d.). https://www.thethingsnetwork.org/docs/lorawan/

Appendix A:



PERFORMANCE Electronics

AN400 Rev C– Application Note
CAN Bus Protocol for PE3 Series ECUs
Release Date 12/20/16

Page 1/2

Firmware/Software Version:	PE3 V3.04.01 and higher
Relevant Hardware:	All PE3 controllers with installed CAN Bus
Additional Notes:	This document defines the CAN based parameters that the PE3 is broadcasting for the firmware listed above. The PE3 ECU contains a 120 ohm termination resistor.

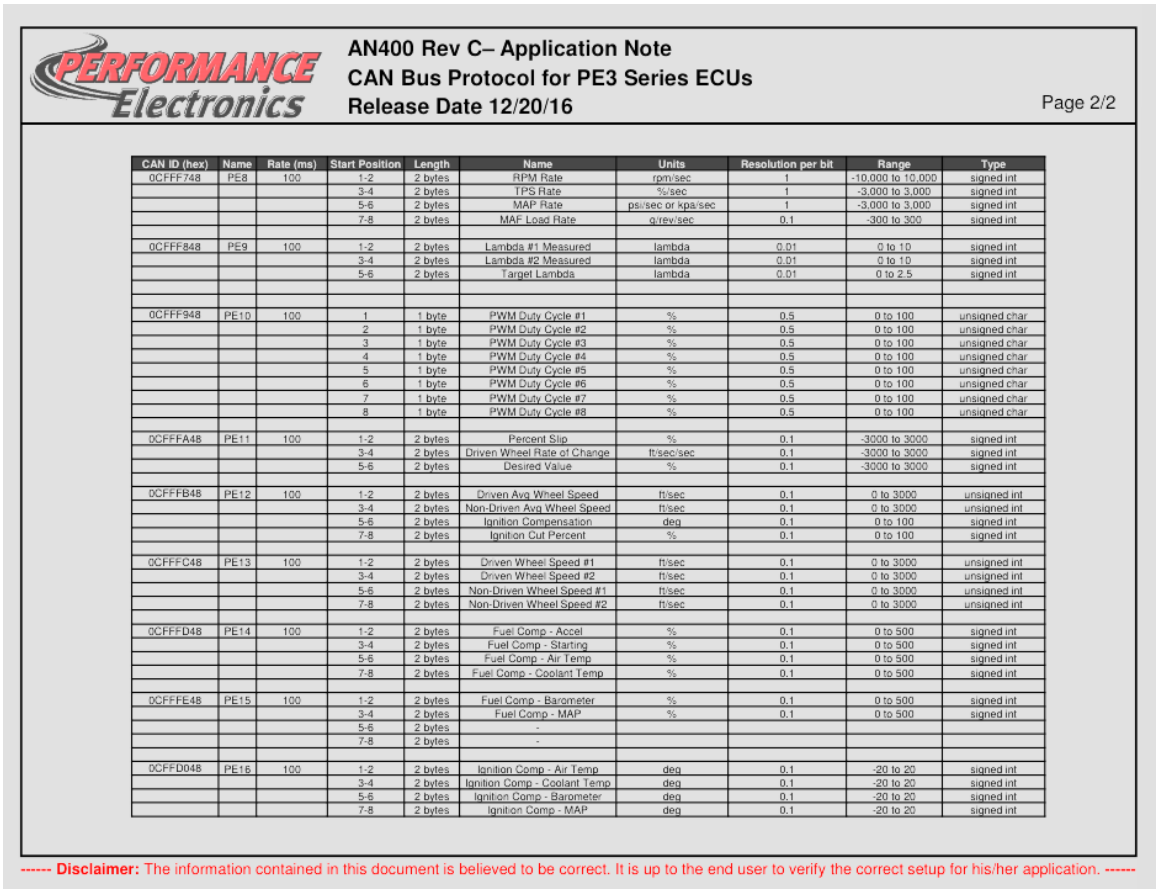
CAN Bus Details

- 250 kbps Rate
- Broadcast parameters are based on SAE J1939 standard
- All 2 byte data is stored [LowByte, HighByte]
  *Num = HighByte*256 + LowByte*
- Conversion from 2 bytes to signed int is per the following:
  *Num = HighByte*256+LowByte*
  if (Num>32767) then
  Num = Num - 65536
  endif

CAN ID (hex)	Name	Rate (ms)	Start Position	Length	Name	Units	Resolution per bit	Range	Type
0CFFF048	PE1	50	1-2	2 bytes	Rpm	rpm	1	0 to 30000	unsigned int
			3-4	2 bytes	TPS	%	0.1	0 to 100	signed int
			5-6	2 bytes	Fuel Open Time	ms	0.1	0 to 30	signed int
			7-8	2 bytes	Ignition Angle	deg	0.1	-20 to 100	signed int
0CFFF148	PE2	50	1-2	2 bytes	Barometer	psi or kpa	0.01	0-300	signed int
			3-4	2 bytes	MAP	psi or kpa	0.01	0-300	signed int
			5-6	2 bytes	Lambda	lambda	0.01	0-10	signed int
			7.1	1 bit	Pressure Type			0 - psi, 1-kPa	unsigned char
0CFFF248	PE3	100	1-2	2 bytes	Analog Input #1	volts	0.001	0 to 5	signed int
			3-4	2 bytes	Analog Input #2	volts	0.001	0 to 5	signed int
			5-6	2 bytes	Analog Input #3	volts	0.001	0 to 5	signed int
			7-8	2 bytes	Analog Input #4	volts	0.001	0 to 5	signed int
0CFFF348	PE4	100	1-2	2 bytes	Analog Input #5	volts	0.001	0 to 5	signed int
			3-4	2 bytes	Analog Input #6	volts	0.001	0 to 5	signed int
			5-6	2 bytes	Analog Input #7	volts	0.001	0 to 5	signed int
			7-8	2 bytes	Analog Input #8	volts	0.001	0 to 22	signed int
0CFFF448	PE5	100	1-2	2 bytes	Frequency 1	hz	0.2	0 to 6000	signed int
			3-4	2 bytes	Frequency 2	hz	0.2	0 to 6000	signed int
			5-6	2 bytes	Frequency 3	hz	0.2	0 to 6000	signed int
			7-8	2 bytes	Frequency 4	hz	0.2	0 to 6000	signed int
0CFFF548	PE6	1000	1-2	2 bytes	Battery Volt	volts	0.01	0 to 22	signed int
			3-4	2 bytes	Air Temp	C or F	0.1	-1000 to 1000	signed int
			5-6	2 bytes	Coolant Temp	C or F	0.1	-1000 to 1000	signed int
			7.1	1 bit	Temp Type			0 - F, 1 - C	unsigned char
0CFFF648	PE7	1000	1-2	2 bytes	Analog Input #5 - Thermistor	C or F	0.1	-1000 to 1000	signed int
			3-4	2 bytes	Analog Input #7 - Thermistor	C or F	0.1	-1000 to 1000	signed int
			5	1 byte	Version Major		1	0-255	unsigned char
			6	1 byte	Version Minor		1	0-255	unsigned char
			7	1 byte	Version Build		1	0-255	unsigned char
			8	1 byte	TBD				

------ **Disclaimer:** The information contained in this document is believed to be correct. It is up to the end user to verify the correct setup for his/her application. ------

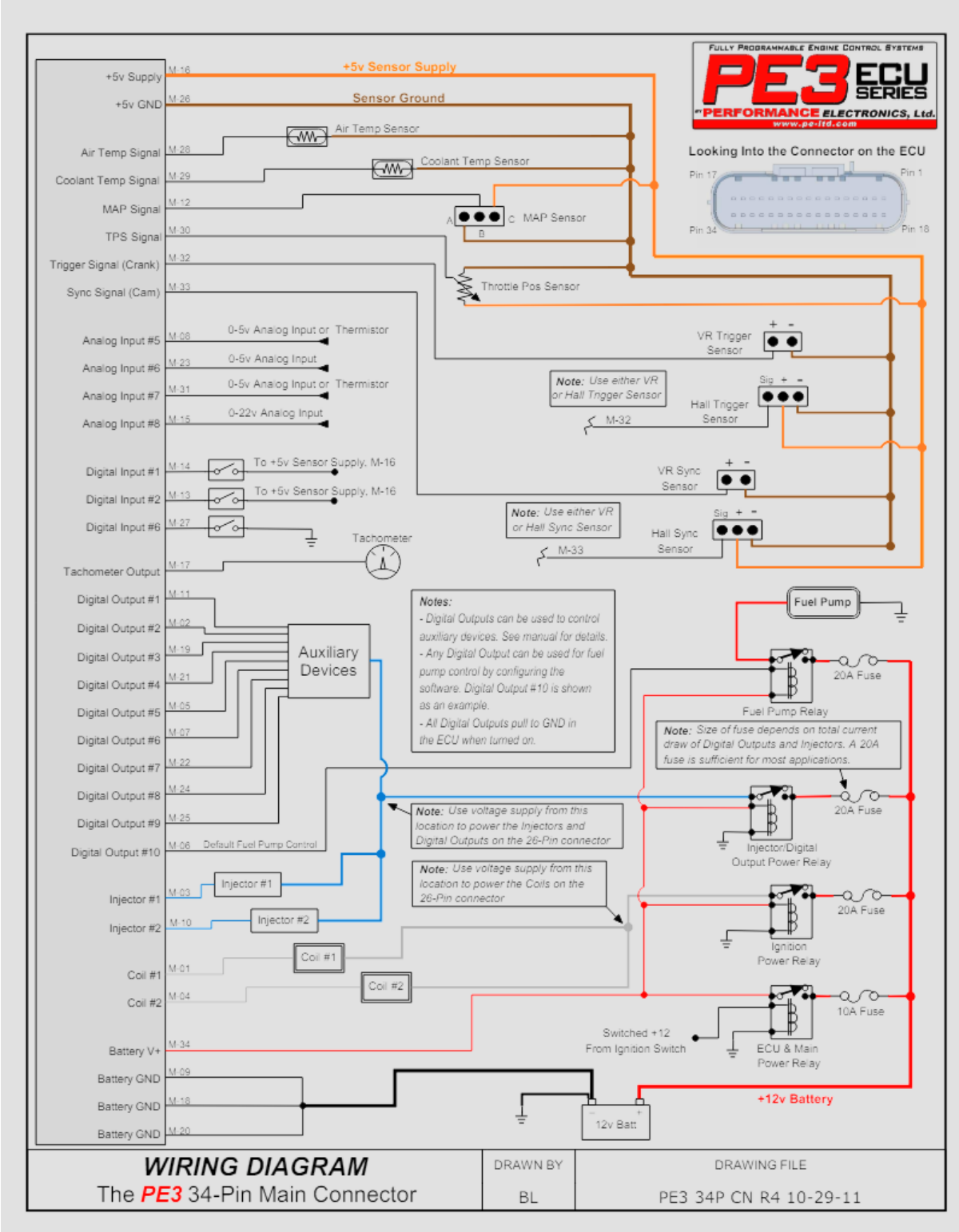
PE3 CAN Documentation 1 of 2



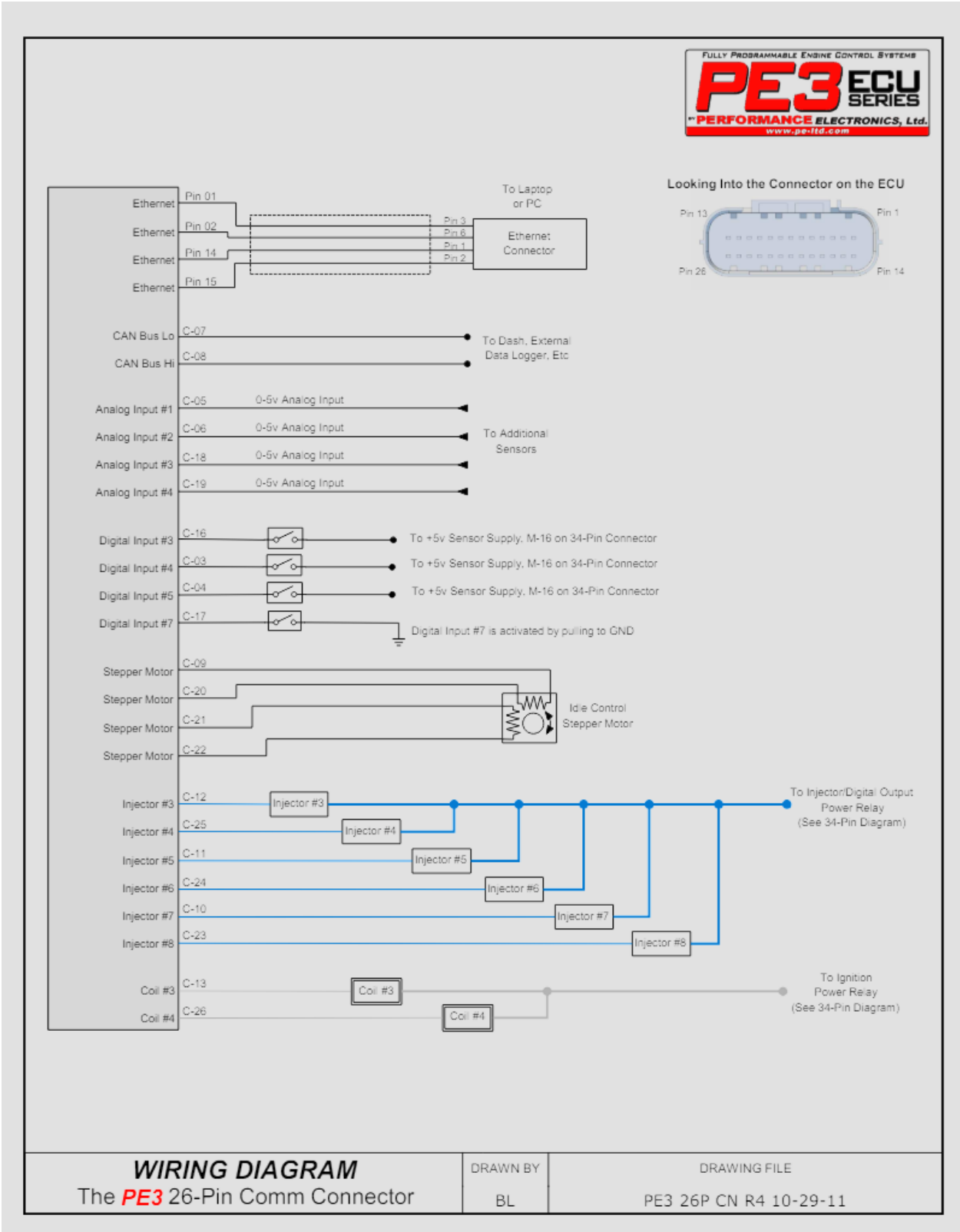
AN400 Rev C– Application Note
CAN Bus Protocol for PE3 Series ECUs
Release Date 12/20/16

CAN ID (hex)	Name	Rate (ms)	Start Position	Length	Name	Units	Resolution per bit	Range	Type
0CFFF748	PE8	100	1-2	2 bytes	RPM Rate	rpm/sec	1	-10,000 to 10,000	signed int
			3-4	2 bytes	TPS Rate	%/sec	1	-3,000 to 3,000	signed int
			5-6	2 bytes	MAP Rate	psi/sec or kpa/sec	1	-3,000 to 3,000	signed int
			7-8	2 bytes	MAF Load Rate	g/rev/sec	0.1	-300 to 300	signed int
0CFFF848	PE9	100	1-2	2 bytes	Lambda #1 Measured	lambda	0.01	0 to 10	signed int
			3-4	2 bytes	Lambda #2 Measured	lambda	0.01	0 to 10	signed int
			5-6	2 bytes	Target Lambda	lambda	0.01	0 to 2.5	signed int
0CFFF948	PE10	100	1	1 byte	PWM Duty Cycle #1	%	0.5	0 to 100	unsigned char
			2	1 byte	PWM Duty Cycle #2	%	0.5	0 to 100	unsigned char
			3	1 byte	PWM Duty Cycle #3	%	0.5	0 to 100	unsigned char
			4	1 byte	PWM Duty Cycle #4	%	0.5	0 to 100	unsigned char
			5	1 byte	PWM Duty Cycle #5	%	0.5	0 to 100	unsigned char
			6	1 byte	PWM Duty Cycle #6	%	0.5	0 to 100	unsigned char
			7	1 byte	PWM Duty Cycle #7	%	0.5	0 to 100	unsigned char
			8	1 byte	PWM Duty Cycle #8	%	0.5	0 to 100	unsigned char
0CFFFA48	PE11	100	1-2	2 bytes	Percent Slip	%	0.1	-3000 to 3000	signed int
			3-4	2 bytes	Driven Wheel Rate of Change	ft/sec/sec	0.1	-3000 to 3000	signed int
			5-6	2 bytes	Desired Value	%	0.1	-3000 to 3000	signed int
0CFFFB48	PE12	100	1-2	2 bytes	Driven Avg Wheel Speed	ft/sec	0.1	0 to 3000	unsigned int
			3-4	2 bytes	Non-Driven Avg Wheel Speed	ft/sec	0.1	0 to 3000	unsigned int
			5-6	2 bytes	Ignition Compensation	deg	0.1	0 to 100	signed int
			7-8	2 bytes	Ignition Cut Percent	%	0.1	0 to 100	signed int
0CFFFC48	PE13	100	1-2	2 bytes	Driven Wheel Speed #1	ft/sec	0.1	0 to 3000	unsigned int
			3-4	2 bytes	Driven Wheel Speed #2	ft/sec	0.1	0 to 3000	unsigned int
			5-6	2 bytes	Non-Driven Wheel Speed #1	ft/sec	0.1	0 to 3000	unsigned int
			7-8	2 bytes	Non-Driven Wheel Speed #2	ft/sec	0.1	0 to 3000	unsigned int
0CFFFD48	PE14	100	1-2	2 bytes	Fuel Comp - Accel	%	0.1	0 to 500	signed int
			3-4	2 bytes	Fuel Comp - Starting	%	0.1	0 to 500	signed int
			5-6	2 bytes	Fuel Comp - Air Temp	%	0.1	0 to 500	signed int
			7-8	2 bytes	Fuel Comp - Coolant Temp	%	0.1	0 to 500	signed int
0CFFFE48	PE15	100	1-2	2 bytes	Fuel Comp - Barometer	%	0.1	0 to 500	signed int
			3-4	2 bytes	Fuel Comp - MAP	%	0.1	0 to 500	signed int
			5-6	2 bytes	-				
			7-8	2 bytes	-				
0CFFD048	PE16	100	1-2	2 bytes	Ignition Comp - Air Temp	deg	0.1	-20 to 20	signed int
			3-4	2 bytes	Ignition Comp - Coolant Temp	deg	0.1	-20 to 20	signed int
			5-6	2 bytes	Ignition Comp - Barometer	deg	0.1	-20 to 20	signed int
			7-8	2 bytes	Ignition Comp - MAP	deg	0.1	-20 to 20	signed int

PE3 CAN Documentation 2 of 2



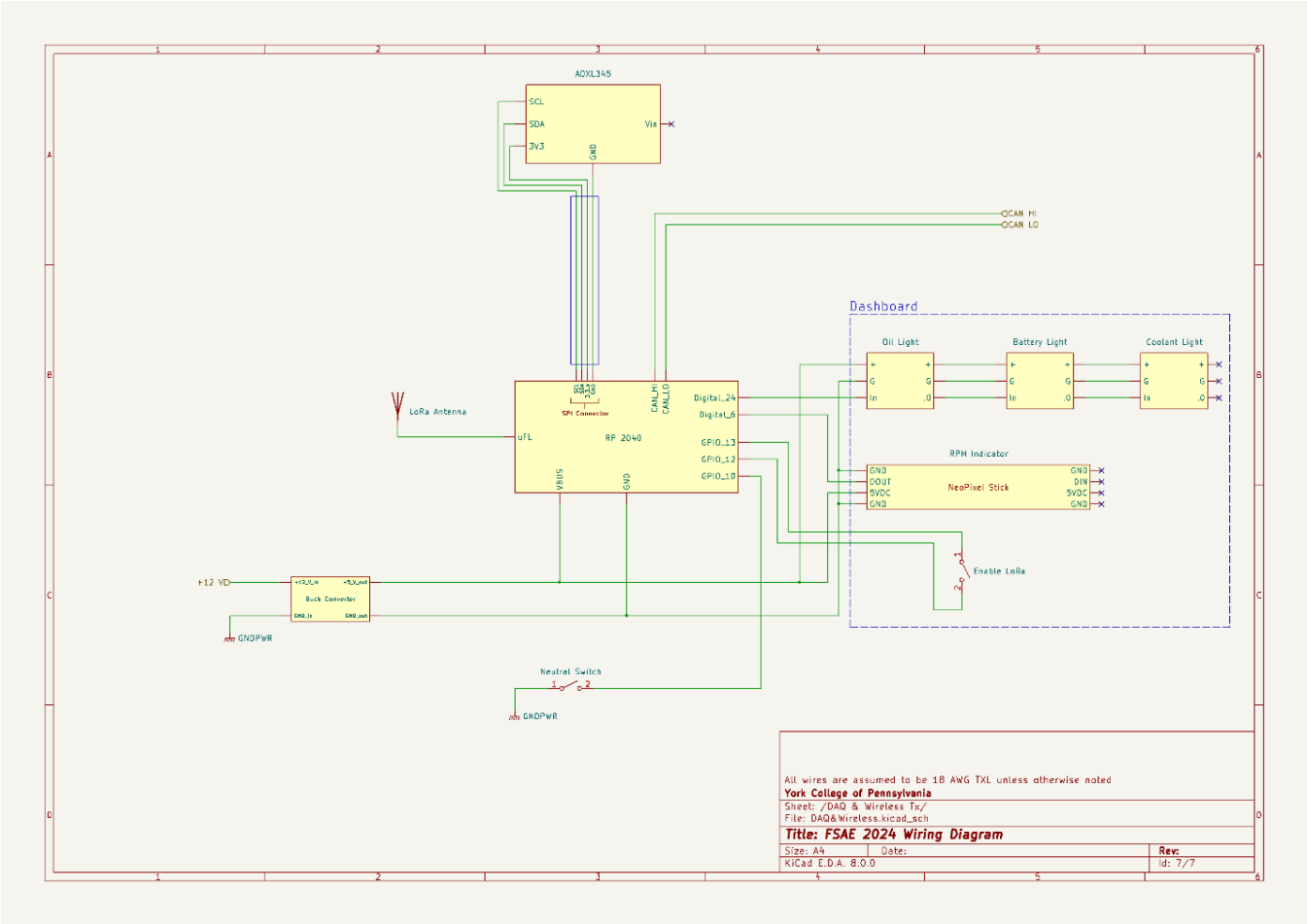
PE3 Suggested Wiring Diagram 1 of 2



PE3 Suggested Wiring Diagram 2 of 2

Appendix B: Parts List and Schematics

Schematics



Schematic of DAQ system and Dash courtesy of Reagan Ansel

Parts Lists / Bill-of-Materials

BOM							
Line	**Part (w/ Hyperlink)**	**Description**	**Supplier Part #**	**Supplier**	**Cost Per Unit**	**Quantity**	**Total Cost**
1	RP2040 RFM95	Main MCU for Dash and sending LoRA data	5714	Adafruit	29.95	1	$ 29.95
2	CAN Controller	CAN Controller for reading data from ECU	5709	Adafruit	12.5	1	$ 12.50
3	Power pack 10,000 mAh	Battery bank to power Raspberry PI	B07QXV6N1B	Amazon	21.99	1	$ 21.99
4	Raspberry Pi 4	MCU for Hosting Grafanna and storing data	B07TC2BK1X	Amazon	99	1	$ 99.00
5	Raspberry Pi Lora Hat	Hat to add LORA capability to Raspberry pi	4074	Adafruit	32.5	1	$ 32.50
8	Neopixel bar	Back up for Tachometer and other indicators	1426	Adafruit	5.95	2	$ 11.90
9	Neopixel lights	Used for other dash indicators	1312	Adafruit	7.95	2	$ 15.90
10	USB A to C cable	Power & Data cable for PI	4472	Adafruit	2.95	1	$ 2.95
12	Hdmi to micro hdmi	Display output for PI	1322	Adafruit	8.95	1	$ 8.95
13	12 to 5v Buck converter	To step down power for MCUs, other sensors and Neopixels	B01NALDSJ0	Amazon	9.99	1	$ 9.99
15	64gb Micro SD card	For Raspberry boot drive and data storage	B09X7BYSFG	Amazon	12.9	1	$ 12.90
16	RP2040 Terminal Block	For mounting to car as well as easier wiring	2926	adafruit	14.95	1	$ 14.95
15	Sundries	Various matierals, small lengths of wire, eletrical tape, zip ties etc	2927	adafruit	15	1	$ 15.00
17	ADXL335	Accelerometer compatable with chosen MCUs	1231	Adafruit	17.95	1	$ 17.50
18	Antenna	915Mhz Antennas for LoRa 2 pack	B0CTXL61LY	Amazon	12.99	1	$ 12.99
19	SMA extension cable	SMA extension cable for LoRa antennas 5 ft	Sm3ftSf100	Data - alliance	3.95	2	$ 7.90
Total Cost							$ 326.87

Appendix C: Source Code Listing

Most up to date source code will be on the Github repository as well as documentation and reference material [5]. This material will aide in modify code as needed for later years teams.

CAN_Sniffer (Main RP2040 program):

```
//Author: Duncan Keller
//Creation Date: 1/18/24
//Purpose of Code: Parse CAN Data from PE3 ECU packitize data and send over LoRa
to another Node. Also Handles setting sevearl
//Neopixels for cars dashboard

//See Packet_Layout_SQLite_Sensor_Pinout.xlsx for more info on LoRa Packet
specifics as well as tech report
#include <Adafruit_MCP2515.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL345_U.h>
#include <Adafruit_NeoPixel.h>
#include <SPI.h>
#include <Wire.h>
#include <RH_RF95.h>
#include "functions.h"   //custom functions to clean up code

//MCP2515 Parameters
//See Adafruit_MCP2515 library for more info
#define CAN_BAUDRATE (250000)  // Set CAN bus baud rate
#define CAN_CS 5               //Set CS pin

//RFM95 Setup Parameters
//See RadioHead library for more info
#define RFM95_CS 16
#define RFM95_INT 21
#define RFM95_RST 17
#define RFM95_FREQ 915.0
```

```
#define RFM95_CODINGRATE 8
#define RFM95_BANDWIDTH 250000
#define RFM95_HeaderID 0x22     //ID that Raspi Will check for when reading
packet
#define RFM95_SPREADFACTOR 8    //Spreading Factor Maxx is 12 any higher than 10
does not seem to function
#define RFM95_TXPOWER 23        //23 is max any higher than 13 can cause serial
connection to not work properly
#define buffSize 42             //Size of LoRa packet -- Array of bytes

//Neopixel Parameters
//See Neopixel library for more info
#define STICK_NUM 8
#define STICK_PIN 5
#define NEO_NUM 4
#define NEO_PIN 24
#define neutral 3

//Global Variables for lora array and Dash Values
byte LoRaBuff[buffSize]; //LoRa Packets
float RPM, BattVoltage, OilPressure, EngineCoolant = 0;
bool neutralOn = false;

//Create MCP object
Adafruit_MCP2515 MCP(CAN_CS);

//Create RFM95 object
RH_RF95 RF95(RFM95_CS, RFM95_INT);

//Create ADXL345 object and assign dummy ID (unused)
Adafruit_ADXL345_Unified ACCEL = Adafruit_ADXL345_Unified(12345);

//Create Neopixel Objects
Adafruit_NeoPixel NEO(NEO_NUM, NEO_PIN, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel STICK(STICK_NUM, STICK_PIN, NEO_GRB + NEO_KHZ800);

void setup() {
  //Pins for checking if data collection is 'ON' and neutral light
  pinMode(13,OUTPUT);
  pinMode(12,INPUT_PULLUP);
  pinMode(10,INPUT_PULLUP);
  digitalWrite(13, LOW);

  // INITIALIZE NeoPixel objects
  STICK.begin();
```

```
  NEO.begin();
  //Set Neopixels Max Brightness
  STICK.setBrightness(35);
  NEO.setBrightness(90);

  //Start Serial, MCP2515 (CAN), ADXL345 (Accelerometer), RFM95 (LoRa)
  //Serial.begin(9600);
  while (!startCAN(CAN_CS, CAN_BAUDRATE, MCP)) {while (1);}
  while (!startLoRa(RFM95_FREQ, RF95, RFM95_SPREADFACTOR, RFM95_TXPOWER,
RFM95_CODINGRATE, RFM95_BANDWIDTH, RFM95_HeaderID)) {while (1);}
  while (!startADXL345(ACCEL)) {while (1);}
}

//Begin main Program
void loop() {

  //Turns on Neutral Light if Pin 10 is pulled to ground
  if(digitalRead(10) == LOW){
    NEO.setPixelColor(neutral,0,0,255);
    NEO.show();
    neutralOn = true;
  }
  else if(neutralOn){
    NEO.setPixelColor(neutral,0, 0, 0);
    NEO.show();
    neutralOn == false;
  }

  // Check for new CAN packet
  int packetSize = MCP.parsePacket();

  if(packetSize){
    // received a packet read and add to LoRaBuff as well as set Neopixels
    readCAN(LoRaBuff, MCP, RPM, BattVoltage, OilPressure, EngineCoolant,
packetSize, NEO, STICK);
  }

  //For Data collection switch on dashboard
  if(digitalRead(12) == LOW){
    //Reads Accelerometer data and store it in the array
    readADXL345(LoRaBuff, ACCEL);

    //Sends LoRaBuff Array to raspberry PI
    sendLoRa(LoRaBuff, RF95, buffSize);
  }
```

```
}
```

CAN_Sniffer Function.h file:

```
//Attempts to configure and start MCP2515
bool startCAN(int CS_PIN, long baudRate, Adafruit_MCP2515 &MCP){

  //Set MCP filter for the IDs used acording to PE3 Documentation on CAN Protocol
  // Checks if 12th bit is zero or allows values <= 7
  MCP.filterExtended(0x0000000,0x00000800);

  //Attempts start 10 times until timeout
  for(int x = 0; x < 10; x++){
    if(!MCP.begin(baudRate)){
      //Serial.println("Error initializing MCP2515.");
      delay(1000);
    }
    else{
      //Serial.println("MCP2515 chip found");
      return true;
    }
  }

  //Serial.println("MCP2515 Not Found");
  return false;
}

//Attempts to configure and start RFM95 LoRa Chip
bool startLoRa(long freq, RH_RF95 &RF95, uint8_t spreadFactor, uint8_t TxPower,
uint8_t codingRate, int bandwidth, int ID){


  for(int x = 0; x < 10; x++){
    if (!RF95.init()) {
      //Serial.println("LoRa module initialization failed");
      delay(1000);
    }
    else{
      //Serial.println("LoRa module initialization Successful");

      //Set Parameters of RFM95 module
      RF95.setFrequency(freq);
      RF95.setSpreadingFactor(spreadFactor);
      RF95.setTxPower(TxPower);
      RF95.setCodingRate4(codingRate);
      RF95.setSignalBandwidth(bandwidth);
```

```
      RF95.setHeaderFrom(ID);

      return true;
    }
  }
 // Serial.println("RFM95 Not Found");
  return false;
}

//Attempts to configure and start ADX345
bool startADXL345(Adafruit_ADXL345_Unified &ACCEL){
  for(int x = 0; x < 10; x++){
    if (!ACCEL.begin()) {
      //Serial.println("ADXL345 initialization failed");
      delay(1000);
    }
    else{
      //Serial.println("ADXL345 initialization Successful");

      //Set G Range for ADX345
      ACCEL.setRange(ADXL345_RANGE_8_G);
      return true;
    }
  }
  //Serial.println("ADXL345 Not Found");
  return false;
}

//Print ADXL345 data for debugging
void printADXL345(Adafruit_ADXL345_Unified &ACCEL){
   /* Get a new sensor event */
  sensors_event_t event;
  ACCEL.getEvent(&event);

  /* Display the results (acceleration is measured in m/s^2) */
  Serial.print("X: "); Serial.print(event.acceleration.x); Serial.print("  ");
  Serial.print("Y: "); Serial.print(event.acceleration.y); Serial.print("  ");
  Serial.print("Z: "); Serial.print(event.acceleration.z);
Serial.print("  ");Serial.println("m/s^2 ");
  //delay(500);
}

//Reads Accelerometer data and store it in the array parameter
void readADXL345(byte buff[], Adafruit_ADXL345_Unified &ACCEL){
   // Get a new sensor event //
```

```
  sensors_event_t event;
  ACCEL.getEvent(&event);

  //Convert and push accerlation values to LoRa Buffer Array
  buff[36] = byte(int(event.acceleration.x*100) & 0xFF); //LSB first
  buff[37] = byte(int(event.acceleration.x*100) >> 8); //MSB second
  buff[38] = byte(int(event.acceleration.y*100) & 0xFF); //LSB first
  buff[39] = byte(int(event.acceleration.y*100) >> 8); //MSB second
  buff[40] = byte(int(event.acceleration.z*100) & 0xFF); //LSB first
  buff[41] = byte(int(event.acceleration.z*100) >> 8); //MSB second
}

//Prints out last recevied CAN packet ID for debugging
void printCANID(Adafruit_MCP2515 &MCP){
  Serial.print("Received id 0x");
  Serial.println(MCP.packetId(), HEX);    //Print HEX ID
}

//Reads CAN data for dash data and store it in the array parameter
//Also Sets Neopixels when value is Set
void readCAN(byte buff[], Adafruit_MCP2515 &MCP, float &RPM, float &BattVoltage,
float &OilPressure, float &EngineCoolant, int packetSize, Adafruit_NeoPixel &NEO,
Adafruit_NeoPixel &STICK){
  byte MCPBuf[8]; //Temp buffer array

  MCP.readBytes(MCPBuf, packetSize);      // Parse packetsize of bytes into
MCPBuf STORED [LowByte, HighByte]
  int ID = MCP.packetId();

  //RPM
  if (ID == 0x0CFFF048) {
    RPM = (MCPBuf[1] << 8) + MCPBuf[0];  //Parse RPM for tachometer

    //Sets Neopixel Stick based on New RPM Value
    STICK.clear(); // Set all pixel colors to 'off'

    //Calculate number of LEDS to turn on
    float rpm = (RPM - 1800) / 1300;

    //Sets Neopixels on Stick based on RPM value
    for(int x = 0; x <= rpm; x++){
      STICK.setPixelColor(x,255,0,0);
    }

    // Send the updated pixel colors to the neopixel
```

```
    STICK.show();

    //Puts byte values into LoRa array
    for(int x = 0; x < 6; x++){
      buff[x] = MCPBuf[x];
    }
  }
  else if (ID == 0x0CFFF148) {
    //Puts byte values into LoRa array
    for(int x = 0; x < 4; x++){
      buff[x+6] = MCPBuf[x];
    }
  }

  else if (ID == 0x0CFFF248) {
    OilPressure = (MCPBuf[3] << 8) + MCPBuf[2];
    if(OilPressure > 32767){                    //signed int convserion
      OilPressure = OilPressure - 65536;
    }

    //Sets Neopixel based on new value
    int neoLED = 1; //Index of chained Neopixel
    float val = ((OilPressure/1000) * 25) - 12.5;  //Calc real value read from
CAN and convert to PSI

    if(val >= 10){
      NEO.setPixelColor(neoLED,0,255,0);
    }
    else{
      NEO.setPixelColor(neoLED,255,0,0);
    }
    NEO.show();

    //Puts byte values into LoRa array
    for(int x = 0; x < 6; x++){
      buff[x+10] = MCPBuf[x+2];
    }
  }

  else if (ID == 0x0CFFF348) {
    //Puts byte values into LoRa array
    for(int x = 0; x < 6; x++){
      buff[x+16] = MCPBuf[x];
    }
  }
```

```
else if (ID == 0x0CFFF448) {
  //Puts byte values into LoRa array
  for(int x = 0; x < 8; x++){
    buff[x+22] = MCPBuf[x];
  }
}

//Battery Voltage, Engine Coolant
else if (ID== 0x0CFFF548) {
  BattVoltage = (MCPBuf[1] << 8) + MCPBuf[0];
  if(BattVoltage > 32767){                    //signed int convserion
    BattVoltage = BattVoltage - 65536;
  }

  //Sets Neopixel based on new value
  int neoLED = 0; //Index of chained Neopixel
  float val = BattVoltage/100;  //Calc real value read from CAN

  if(val >= 12){
    NEO.setPixelColor(neoLED,0,255,0);
  }
  else if(val >= 11.8){
    NEO.setPixelColor(neoLED,255,172,28);
  }
  else{
    NEO.setPixelColor(neoLED,255,0,0);
  }

  EngineCoolant = (MCPBuf[5] << 8) + MCPBuf[4];
  if(EngineCoolant > 32767){                    //signed int convserion
    EngineCoolant = EngineCoolant - 65536;
  }

  //Sets Neopixel based on new value
  neoLED = 2; //Index of chained Neopixel
  val = EngineCoolant/10;  //Calc real value read from CAN

  if(val > 190){
    NEO.setPixelColor(neoLED,255,0,0);
  }
  else if(val >= 150){
    NEO.setPixelColor(neoLED,0,255,0);
  }
  else{
```

```
      NEO.setPixelColor(neoLED,0,0,255);
    }
    NEO.show();

    //Puts byte values into LoRa array
    for(int x = 0; x < 6; x++){
      buff[x+30] = MCPBuf[x];
    }
  }
}

//Sends LoRaBuff to raspberry PI
bool sendLoRa(byte buff[], RH_RF95 &RF95, int buffSize){
  return RF95.send((uint8_t*)buff, buffSize);

}

//Sends a packet of data with the value sent based on datas position in packet
//Mutlipler is for adjusting values
//RPM will always be zero
bool sendLoRaTestDataMulti(int multiplier, RH_RF95 &RF95, int buffSize){
  byte buff[buffSize];

  for (int x = 0; x <buffSize; x++){
    if(x % 2 == 0){
      buff[x] == byte((x*multiplier) & 0xFF); // grab low byte
    }
    else{
      buff[x] == byte((x*multiplier) >> 8); // grab high byte
    }
  }

  return RF95.send((uint8_t*)buff, buffSize);

}

//Sends a packet of empty data with only the specificed value being set
//Position must be even value!!
bool sendLoRaTestData(int position, int value, RH_RF95 &RF95, int buffSize){
  byte buff[buffSize];

  // define array as all zeros to avoid junk value
  for(int x = 0; x < buffSize; x++){
    buff[x] = 0;
  }
```

```
  // check if position is even and then set values
  if(position % 2 == 0){
    buff[position] = byte(value & 0xFF); // grab low byte
    buff[position+1] = byte(value >> 8); // grab high byte

    return RF95.send((uint8_t*)buff, buffSize);
  }
  else{
    return false;
  }
}
```

Raspberry Pi Code:

```
#Duncan Keller 2/21/24
#Grabs LoRa Data From RP2040 Running "CAN Sniffer"
#This code handles receiving LoRa data from the RP2040 on the car
#The data is then unpacked and sent to a MariaDB for use on Grafana

import mysql.connector
import time
import busio
from digitalio import DigitalInOut, Direction, Pull
import board
import adafruit_ssd1306
import adafruit_rfm9x
from datetime import datetime
import pytz
import shutil

# Button A
btnA = DigitalInOut(board.D5)
btnA.direction = Direction.INPUT
btnA.pull = Pull.UP

# Create the I2C interface.
i2c = busio.I2C(board.SCL, board.SDA)

# 128x32 OLED Display
reset_pin = DigitalInOut(board.D4)
display = adafruit_ssd1306.SSD1306_I2C(128, 32, i2c, reset=reset_pin)

# Clear the display.
display.fill(0)
display.show()
width = display.width
height = display.height

# Configure LoRa Radio
CS = DigitalInOut(board.CE1)
RESET = DigitalInOut(board.D25)
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, 915.0)
rfm9x.tx_power = 23
rfm9x.spreading_factor = 8
rfm9x.receive_timeout = 1.0
rfm9x.coding_rate = 8
```

```
rfm9x.signal_bandwidth = 250000
ID = 0x22 #Header ID or RP240

#Setup Variables
count = 0       #Count of Packets
packet = None   #Raw LoRa Packet
data = []     #Converted LoRa Data

#Reset Display
display.fill(0)
display.show()

config = {
    'user': 'fsae2024',
    'password': 'YCPfsae#123',
    'host': '127.0.0.1',
    'database': 'Sensor_Data',
}

#Create Datetime if data is aved before needing a timestamp
dateTime = datetime.now()
UTCdateTime = pytz.timezone('UTC')
dateTime = dateTime.astimezone(UTCdateTime)
dateTime = dateTime.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]

while True:
    # check for packet rx returned in bytes
    packet = rfm9x.receive(with_header=True)

    if not btnA.value:
        #Connection for MariaDB
        try:
            #connect with credentials
            conn = mysql.connector.connect(**config)

            cursor = conn.cursor()   #Create cursor object
            data_tuple=tuple(data)   #Format data

            #Ececute SQL command
            cursor.execute("INSERT INTO liveData VALUES " + str(data_tuple))

            conn.commit()  # Commit changes to databse

            #Reset Table
            if not btnA.value:
```

```
            cursor.execute("TRUNCATE TABLE liveData")

        conn.close()   # Close database conncetion

        #Display Cleared message and
        display.fill(0)
        display.text("Database Cleared", 15, 10, 1)
        display.show()
        time.sleep(3)
        display.fill(0)
        display.show()

    #Catch connection error and print
    except mysql.connector.Error as e:
        print(f"Error connecting to database: {e}")

if packet is not None and packet[1] == ID:

#Check if Packet has RP2040 ID

    # Display the packet number & RSSI
    display.fill(0)
    count += 1
    packetNum = "Packet Number: " + str(count)
    rssi = "Last RSSI: " + str(rfm9x.rssi)
    display.text(packetNum, 15, 10, 1)
    display.text(rssi,15,20,1)
    display.show()

    #Print Part of Packet To Terminal for Debugging
    for x in range(4,len(packet),2):
        val = packet[x] | (packet[x+1] << 8)

        #NOTE this is assuming all values are signed values or
        #That the unsigned dont go higher than 32766
        if val > 32767:
            val = val - 65536

        #TPS, Fuel Time, Air temp, coolant temp
        if (x==6) or (x==8) or (x==36) or (x==38):
            val = val / 10
        #Barometer, MAP, Battery Voltage, Accelerometer
        elif (x>=10 and x <=12) or (x== 34) or (x>= 40 and x<=44):
            val = val / 100
        #Freq / Wheel speed
```

```
            elif(x>= 26 and x <= 32):
                val = val / 20
            #Analog
            elif (x>= 14 and x<= 24):
                val = val /1000 #resolution conversion

            #Append new Value onto final list
            data.append(val)

        #Create UTC timestamp for Grafanna
        dateTime = datetime.now()
        UTCdateTime = pytz.timezone('UTC')
        dateTime = dateTime.astimezone(UTCdateTime)
        dateTime = dateTime.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]
        data.append(dateTime)
        print(data)

        try:
            #connect with credentials
            conn = mysql.connector.connect(**config)

            cursor = conn.cursor()   #Create cursor object
            data_tuple=tuple(data)   #Format data

            #Ececute SQL command
            cursor.execute("INSERT INTO liveData VALUES " + str(data_tuple))

            conn.commit()  # Commit changes to databse
        #Catch connection error and print
        except mysql.connector.Error as e:
            print(f"Error connecting to database: {e}")

        data = [] #Clear Data
```