THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

"THE CUBER"

Self-Drive Design Report

2022 IGVC Competition

Team Lead
Daniel Mezhiborsky (daniel.mezhiborsky@cooper.edu)

Team Members

Jonathan Lu (jonathan.lu@cooper.edu)
Alex Koldy (koldy@cooper.edu)
Neil Sawhney (neil.sawhney@cooper.edu)
Derrick Yu (derrick.yu@cooper.edu)
Ridwan Hussain (ridwan.hussain@cooper.edu)
Ilona Lameka (ilona.lameka@cooper.edu)
Azra Rangwala (azra.rangwala@cooper.edu)
Giuseppe Quaratino (giuseppe.quaratino@cooper.edu)
Joshua Yoon (yoon11@cooper.edu)
David Yang (david.yang@cooper.edu)
Jacob Koziej (jacob.koziej@cooper.edu)
Lizelle Ocfemia (lizelle.ocfemia@cooper.edu)
Seokwoo Chung (seokwoo.chung@cooper.edu)
Henry Son (son3@cooper.edu)
Jaewon Cho (jaewon.cho@cooper.edu)
Hannah Rajarao (rajarao@cooper.edu)
Seong-Jun Yea (seongjun.yea@cooper.edu)
Isabella Flynn (isabella.flynn@cooper.edu)
Lorina Ilkkan (lorina.ilkkan@cooper.edu)
Amelia Roopnarine (amelia.roopnarine@cooper.edu)
Ani Vardanyan (ani.vardanyan@cooper.edu)
Isaiah Rivera (isaiah.rivera@cooper.edu)

Supervised by
Neveen Shlayan (neveen.shlayan@cooper.edu)
Michael Giglia (michael.giglia@cooper.edu)

Statement of Integrity: I certify that the design and engineering of the vehicle by the current student team has been significant and equivalent to what might be awarded credit in a senior design course.

Faculty Signature:____________________ **Date:**____________

Contents

1 Conduct of design process, team identification and team organization 1
- 1.1 Introduction 1
- 1.2 Organization 1
- 1.3 Design assumptions and design process 1

2 Effective innovations in your vehicle design 1
- 2.1 Innovative technology applied to your vehicle 1
  - 2.1.1 DBW Nodes 1
  - 2.1.2 DBW Firmware: `node_fw` 2
  - 2.1.3 Unified CAN 2
  - 2.1.4 Open Source and Free Software/Hardware Principles 3
- 2.2 Innovative concept(s) from other vehicles designed into your vehicle 3

3 Description of mechanical design 3
- 3.1 Overview 3
  - 3.1.1 Brake by wire 3
  - 3.1.2 Mechanical Emergency Stop 4
  - 3.1.3 Front Encoders 4
  - 3.1.4 Rear Encoders 4
- 3.2 Decision on frame structure, housing, structure design 5
- 3.3 Suspension 5
- 3.4 Weatherproofing 5

4 Description of electronic and power design 5
- 4.1 Overview 5
- 4.2 Power distribution system (capacity, max. run time, recharge rate, additional innovative concepts) 6
- 4.3 DBW Expansion Boards - Electronics Suite concepts 6
  - 4.3.1 Expansion Boards 6
- 4.4 Safety devices and their integration into your system 7

5 Description of software strategy and mapping techniques 7
- 5.1 Overview 7
- 5.2 Obstacle detection and avoidance 7
- 5.3 Software strategy and path planning - `move_base`, `teb_local_planner` 7
- 5.4 Map generation 8
- 5.5 Goal selection and path generation 9
- 5.6 Diagrams 9

6 Description of failure modes, failure points and resolutions 10
- 6.1 Vehicle failure modes (software, mapping, etc) and resolutions 10
- 6.2 Vehicle failure points (electronic, electrical, mechanical, structural, etc) and resolutions 10
- 6.3 All failure prevention strategy 10
- 6.4 Testing (mechanical, electronic, simulations, in lab, real world, etc.) 10
- 6.5 Vehicle safety design concepts 11

7 Simulations employed 11
- 7.1 Simulations in virtual environment 11
- 7.2 Theoretical concepts in simulations 11

8 Performance Testing to Date 13
- 8.1 Component testing, system and subsystem testing, etc 13

9 Initial Performance Assessments 13
- 9.1 How is your vehicle performing to date 13

10 Mandatory Unit Test Data To Date 14
- 10.1 Unit test 3: Speed limit test 14

Figure 1: Our first test day :)

1 Conduct of design process, team identification and team organization

1.1 Introduction

The Cooper IGVC Team is a subset of the Cooper Union Autonomy Lab. As a team and organization, we focus on building work that is generalizable to projects beyond our own. We have been building the Cooper IGVC car with the forefront focus of creating transferrable knowledge and design heritage.

For these reasons, we chose to build our car full-stack. All core electronic, software, and mechanical systems are produced by the Cooper IGVC team. This includes our power system, drive-by-wire suite, and technical autonomy stack.

The design process has been guided by numerous industry standard practices.

1.2 Organization

Each system of the car has been assigned to a specific subteam for design and test - Tech, Controls, EE (Electrical) Hardware, ME (Mechanical) Hardware, and Drive-By-Wire (DBW). The Mechanical Hardware (ME HW) team mounts the equipment on the car and create enclosures for more sensitive components. The Electrical (EE HW) designs the electrical system through PCB and power system design. The Drive-By-Wire subteam programs the electronics and develop the CAN architecture. Controls analyze data from the sensors to create velocity and motor commands. Tech collects data from sensors to handle lane-detection, objection detection, and localization.

1.3 Design assumptions and design process

We have built, over time, a design process for the car that includes a number of core utilities and practices. For communication, we use team chat software similar to Slack (Mattermost). Subteam communication is extremely important, and team members are encouraged to and regularly visit group chats of subteams other than their own to collaborate on tasks. To track our engineering tasks, we use Atlassian's Jira. For documentation, we use Outline Wiki. All work is kept in Git repositories, and all production car code is in a single monorepo.

We have used several different software and communication solutions, and we've found that this combination is a good compromise between organic, unhindered collaboration, safety and design review integration, and accurate tracking of work and changes.

Typically, we describe our team structure as a stack: (ME HW)-(EE HW)-(DBW)-(CONTROLS)-(TECH). Each subteam typically interfaces with the teams above and below it, if they exist. For example, for accelerator control, ME HW may provide mounting locations, EE HW may provide an electrical interface board to the accelerator pedal, DBW may write the firmware, and controls may utilize CAN messages exposed by DBW to fulfill a command received from the technical stack.

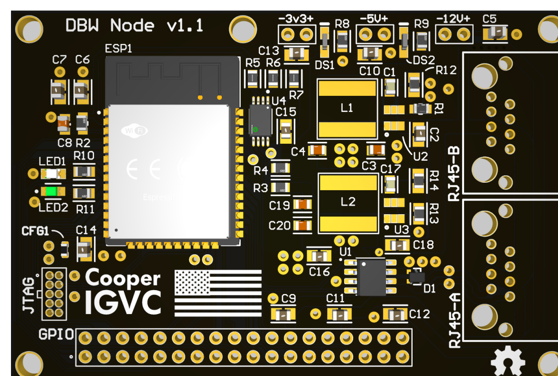
2 Effective innovations in your vehicle design

2.1 Innovative technology applied to your vehicle

We've strived to make a solid base for future autonomy research, so much of the team's focus is placed on the drive-by-wire systems.

2.1.1 DBW Nodes

We created a modular DBW system. This system is based on our own IGVC DBW Node:



The DBW Node uses an ESP32 microcontroller, which we found both relatively easy to source in the silicon parts shortage, and much more highly featured than 8-bit microcontrollers like those found on Arduinos. The DBW Node board has CAN hardware, JTAG debugging, an EEPROM, and power rails created with switching regulators for high efficiency. The large GPIO header is used to host expansion boards, which extend each DBW Node with functionality specific to a task. To date, we have created brake, accelerator, shift, blinker, and encoder expansion boards. We are planning additional boards as well, and we also have a protoboard expansion board that can be used for quick prototyping.

These are also highly cost-effective, so we can comfortably experiment with them and train students on how to use them.

2.1.2 DBW Firmware: `node_fw`

The firmware for the DBW Nodes is highly shared between the different expansion board uses. All firmware is written in C. The base Node board has a suite of system firmware modules, such as EEPROM, CAN, tasking, watchdog, and base behavior modules. This code is then expanded and the firmware differentiated using expansion module code. As such, the vast majority of the code is shared, with only one copy existing in the repository. Behavioral validation (such as ESTOP management) for the base system need only be conducted on the majority of the code once. Generally, we have extremely fast iteration and easily understood firmware behavior, and strongly recommend others pursue unified firmware bases like this.

We've written a suite of software to support this, including integrated build systems, simulation tools, monitoring software, etc.

2.1.3 Unified CAN

CAN bus is an industry standard vehicular communications bus. We find that CAN bus is an extremely useful, excellent tool for vehicle communications, but can often be difficult to manage because of the complexity of maintaining CAN bus definitions, code, and interfacing.

Several industry-standard tools exist, like those made by PEAK-System GmbH and Vector Informatik GmbH, but these are expensive and we wanted to avoid closed-source software. Instead, we have modified available open-source software, particularly `cantools`, and have written our own software tools to assist:

- `gen_dbc.py`, `can.yml`: We define CAN messages and templates in a human-readable file, `can.yml`. `gen_dbc.py` translates this into a DBC file, using `cantools`.
- C code generation: we modified `cantools` to automatically generate C code for our DBW Nodes based on the messages in the DBC file.
- `node_fw` CAN system: The CAN module automatically receives and decodes desired messages. It also automatically encodes and sends desired messages. It is highly coupled with the generated C code.
- `igvc_bl`: Bootloader that allows the DBW Nodes to receive firmware updates over CAN.
- `cand` (*"candy"*): `cand` is a highly performant Python CAN daemon that automatically decodes incoming messages and encodes outgoing messages using a central Redis database.
- Test runners and data parsers (`igvcutils`): We use scripts to automatically run tests, dumping and saving bus data, and converting it for graphed viewing later.

The result is a usable, flexible, and pleasant CAN system. Code in Python to send a command to the accelerator node and read encoder data is as simple as:

```
1   from cand.client import Bus
2
3   bus = Bus()
4   bus.send('dbwNode_Accel_Cmd', {'TargetVel': 0.5})
5
6   speeds = bus.get('dbwNode_Encoder_Data')
```

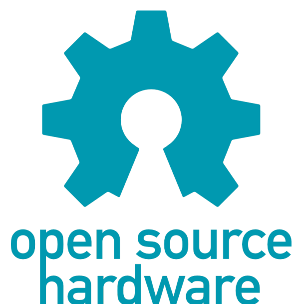
Figure 2: Interacting with DBW through `cand`

2.1.4 Open Source and Free Software/Hardware Principles

We commit to open-sourcing all work performed by our team by July 18, 2022.

Our project, especially without a DBW kit or RTK, would not have been possible without a great number of other open-source projects.

We will release our mechanical parts designs, documentation, firmware, software, electrical design, and PCBs in source format, to the extent possible.



2.2 Innovative concept(s) from other vehicles designed into your vehicle

We would like to acknowledge the other Self-Drive teams, without which we would have little to work on. Specific thanks to Detroit-Mercy for your paper on your DBW system, and to Embry-Riddle for their writing on their emergency brake design, upon which ours is based.

3 Description of mechanical design

3.1 Overview

3.1.1 Brake by wire

The vehicle's braking system is split up into two parts; the front disc brakes are controlled via a hydraulic trailer brake actuator, while the rear drum brakes remain connected to the master cylinder (Figure 2). The resulting empty output of the master cylinder is plugged with a custom-made bolt and crush washer. The Hydrastar, the hydraulic brake actuator of choice, is connected via soft brake lines to a custom-made tee-fitting. A pressure sensor is connected to the one of the tee's outputs, while the other output is connected to a M10x1.25 double banjo bolt, which allows fluid to flow to both front brakes. The Hydrastar is controlled through DBW.

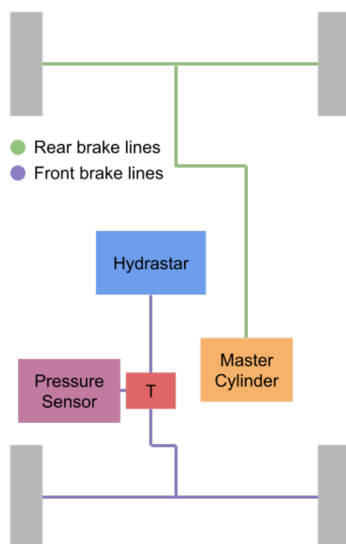


Figure 3: Brake-by-Wire Hydraulic Diagram

3.1.2 Mechanical Emergency Stop

A gas compression spring rated for 20 lbs of force has been mounted to the built in parking brake (Figure 4). The equilibrium position of this piston pushes up on the parking brake, and acts to stop the car. An electromagnet, powered by the main power supply for our electronics, which is rated for 130 lbs of force attracts a steel plate which prevents the gas spring from going to equilibrium. This system ensures that if there is a power failure, the electromagnet will detach from the steel plate and cause the parking brake to be pushed upwards in a quick, but safely damped motion.

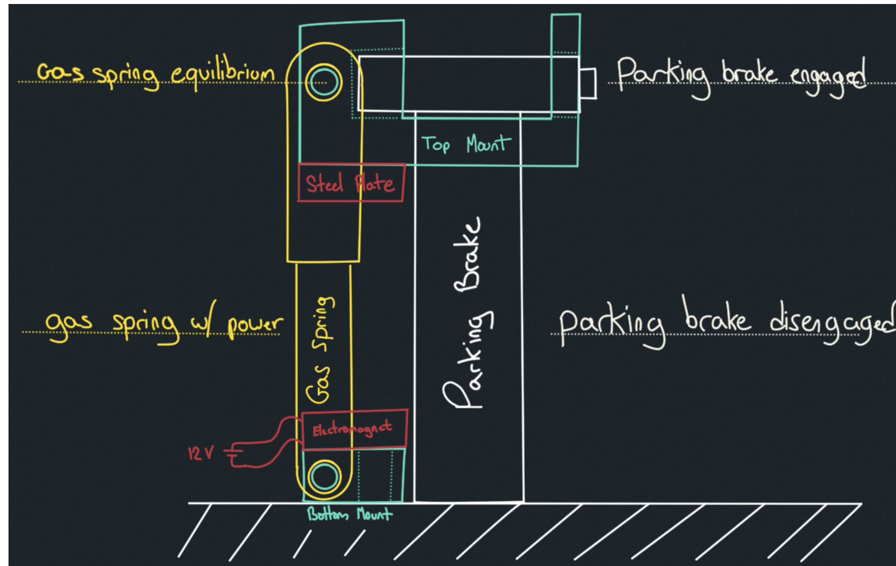


Figure 4: Mechanical Estop Diagram

3.1.3 Front Encoders

Gears have been placed around the front driveshaft, prior to the differentials, on both the passenger and driver sides. This gear meshes with a secondary gear on the encoder shaft. The system ensures accurate and reliable sensing of both front wheels to be used for odometry.

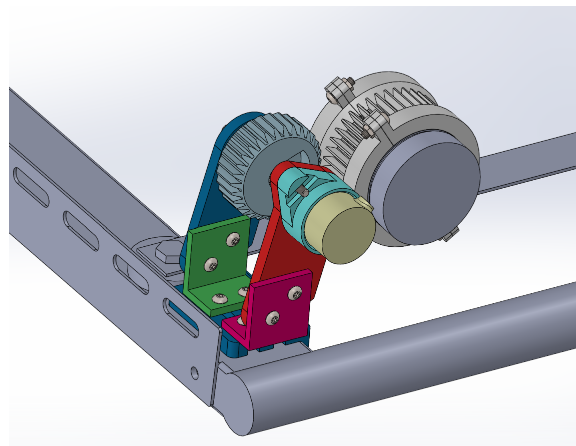


Figure 5: front encoder system

3.1.4 Rear Encoders

Since there is no drive shaft for the rear wheels, a circular gear is placed on the inside rim of the rear tires, and a smaller gear on the encoder meshes with it. This encoder is mounted using an aluminum bracket behind the mounting point where the frame is secured to the wheel hub.

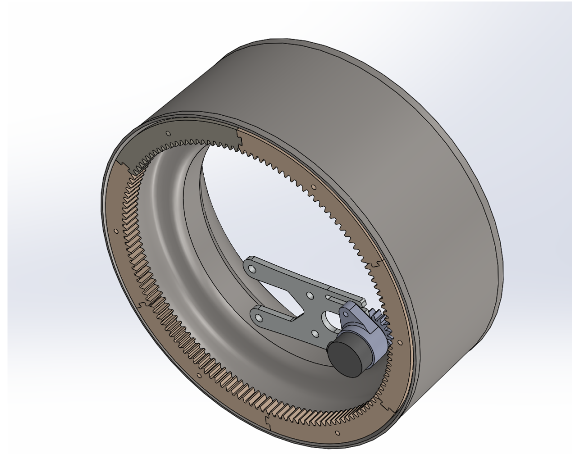


Figure 6: Rear Encoder system

3.2 Decision on frame structure, housing, structure design

To keep the driver and passenger safe from falling out, webbings were installed which are secured with paracord that is rated for 500 lbs, and carabiners that are rated for 500 lbs. The carabiners make it easy to clip the webbings on and off for entering and exiting the vehicle.
The electronics have been mounted in the back, on a shelf that has been installed above the battery. Holes were drilled in the chassis so that all the wires can be routed to the back in a safe and neat manner.

The DBW Nodes are distributed throughout the car and are mounted inside the dash, under the seats, and in the rear.

3.3 Suspension

The suspension of the car was left unaltered from its factory state.

3.4 Weatherproofing

Plexiglass has been installed on the back to redirect the water away from the electronics. All exposed wire connections on the roof of the car have been enclosed with 3d printed covers. This includes the exposed wires for the gps antenna, as well as for the 3d LIDAR.

4 Description of electronic and power design

4.1 Overview

Each DBW Node is an ESP32 microcontroller PCB that is connected to the CAN bus of the car. Most of the DBW Nodes have additional expansion boards mounted on top to add additional circuitry for a specific purpose (i.e. controlling the brake actuator). All node boards, computers, sensors, and additional PCBs are powered by the power distribution system in the rear of the car, consisting of the 48 VDC main pack, DC-DC converters, an inverter, and 12V bus rails, with fuses to systems and to individual components.

4.2 Power distribution system (capacity, max. run time, recharge rate, additional innovative concepts)

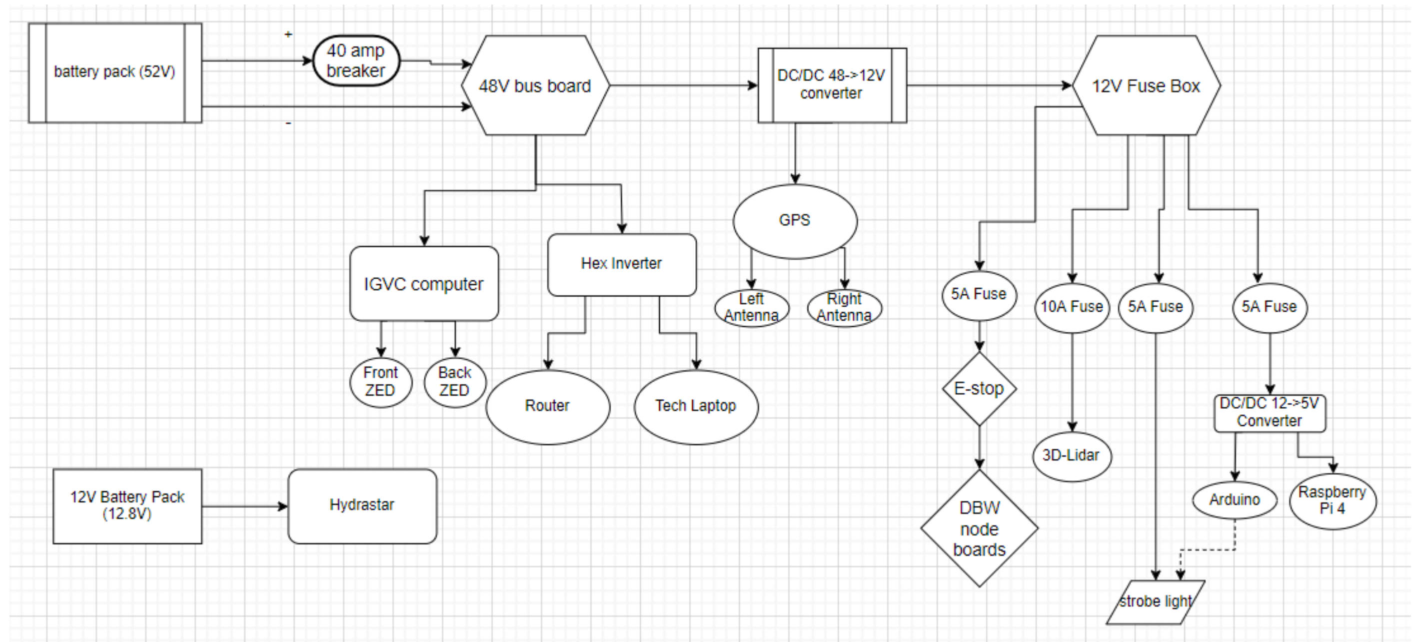


Figure 7: Power Connection Diagram

The power distribution system begins with tapping the vehicle's 48 VDC (nominal, usually actually 52VDC) battery pack (which consists of four 12 VDC AGM connected in series). The battery terminals are tapped and connected directly to a 40A circuit breaker - the maximum system power is about 1.8 kW. From the 40A circuit breaker, fused 48V lines are used to supply power to the rest of the electrical components on the car: technical computer (which has a DC ATX power supply), one DC-DC converter (48V to 12V), and an inverter (48 VDC to 100-120V AC). The DC-DC converter powers 12V components, including the GPS, DBW nodes, Ethernet router, and lighting. The inverter is used to power any engineering/development tools needed in the field, such as laptops and power tools. In addition to these components, a separate 12V battery pack is used to power the Hydrastar. This circuit still only runs if the circuit breaker is closed.

We have begun designs of redundant power systems, and hope to include fully redundant 12V buses on our car for DBW and the Hydrastar soon.

4.3 DBW Expansion Boards - Electronics Suite concepts

Six major aspects of the car are controlled autonomously by the DBW node boards: the accelerator/throttle, the brake actuator, the steering wheel, turn signals, and shift (Forward, Neutral, Reverse). Each aspect, with the exception of steering, has been assigned to a single DBW node board with an additional expansion board that serves to accomplish the hardware objectives needed.

4.3.1 Expansion Boards

Here is a brief summary of the expansion boards to date:

- Brake: Optoisolated, MOSFET-based - for Hydrastar PWM drive.
- Accelerator: The most safety-critical board. Switchover DPDT relay control for tapping into the accelerator pedal. When power is lost or DBW is inactive, control is electromechanically restored to the driver's pedal. Onboard comparators also immediately override DBW and restore control to the driver if the pedal is pressed. These are electronic hardware features that cannot be overridden by the DBW Node. DAC directly from the DBW Node's ESP32 is used to emulate the pedal signals. Feedback sensing is provided to the Node, which can monitor and log both the physical pedal position and the actual output at the car terminals.
- Encoder: Simple voltage division for connecting two encoders to a DBW Node; input-only.

- Shift, Blinker: Similar architecture to the Accelerator board, with relay control. When power is lost or DBW is inactive, relays switch to physically return control to the car's native interfaces (stalks and buttons).
- Steering: Due to a lack of documentation from the manufacturer of the EPAS, Allied Motion, the stock motor driver was replaced with an ODrive, and absolute position sensing is accomplished with a standard automotive steering encoder. The ODrive is controlled in Python with a **simulated** DBW Node, which follows the same state sequences and CAN commands as a true hardware DBW Node would. This will be replaced in time with a new expansion board to control the ODrive over UART or private CAN.

4.4 Safety devices and their integration into your system

The vehicle has multiple mechanical emergency stop buttons that are separated and placed in different places on the vehicle: the rear of the car, left side of the car, right side of the car, on the dashboard, along with a wireless E-stop. These emergency stop buttons cut off power to DBW, which, because of the electronic design, results in a controlled, safe stop by triggering a gas spring which forces the physical emergency brake to be pulled, and disables (in hardware) active DBW input to the car. In addition, there are multiple fuses as well as a circuit breaker on the car to ensure that the car nor any components of the car pull too much power, and monitoring to detect failed components and assert an ESTOP.

5 Description of software strategy and mapping techniques

5.1 Overview

Our software distribution can be broken logically into three distinct categories: DBW, Controls, and Technical. The level of abstraction goes up with each category, and each category can only communicate through one another, (e.g., Tech does not directly communicate to DBW).

DBW serves as the lowest-level interface to our vehicle. Each DBW node is running a base firmware shared between all nodes along with additional module firmware meant to specialize each node to a specific task (e.g. controlling the throttle). Nodes broadcast their current state over CAN at set intervals (1Hz, 10Hz, 100Hz, 1kHz) and are controllable over CAN.

Controls serves as the middle ground between DBW and Tech. The primary role of controls is to process abstract velocity commands from the tech stack and create inputs to DBW to make them happen on the car. Controls plays a critical role in characterizing the way our car moves and reacts to various inputs. Controls code runs on both the technical computer for high-and mid-level controls, and directly on the DBW nodes for low-level controls (e.g., P controllers).

Technical serves as the highest-level interface to the vehicle. The goal of technical team is to handle environmental awareness and behavioral actions. The former includes mapping out the entire environment as a global map and localizing the car within the global map, while the latter involves dictating the state in which the car is in - i.e. whether it is moving forward, turning right, or stopping in front of a barrel. Tech handles sensor inputs and filtering for the various algorithms that are used, such as the stereoscopic ZED camera (and its integrated IMU), the Velodyne 3D Lidar, etc., by interfacing the sensors themselves with the Robot Operating System, or ROS.

5.2 Obstacle detection and avoidance

Obstacle detection is performed in the local planner and object identities are assigned live in RGBD space using computer vision.

Object avoidance was handled by building a Behavior Tree using the ROS Behavior Tree Library, BT++. Fallback nodes were used to handle the return processes each time the Behavior Tree is ticked. All conditional nodes in the BT++ library return either success or failure in order to determine if a corresponding action node will be executed. Every tick, the Fallback Node looks for the first child that does not fail. Once a condition node is triggered, the Fallback Node is allowed to halt ticking and execute the proper action assigned to that condition, which can be a costmap modification or direct velocity behavior intervention.

5.3 Software strategy and path planning - `move_base`, `teb_local_planner`

The ROS `move_base` package is the driving force behind path planning. The global planner receives a pose stamped message in addition to a global costmap in order to generate a path using Dijkstra's algorithm.

The newly generated path is optimized and locally followed by `teb_local_planner`. A pure pursuit controller takes in the optimized path in addition to an odometry message in order to generate a twist message. The pure pursuit controller additionally determines a goal point by calculating the distance between the current location of the vehicle and a look-ahead point on the path. The curvature is calculated and transformed into a steering angle using the car's wheelbase and other kinematic parameters, which is ultimately executed in DBW along with a forward velocity command.

5.4 Map generation

The Technical team uses a combination of RTAB-Map and the `move_base` package to generate a local and global costmap of the surrounding environment of the autonomous vehicle.

RTAB-Mapping, also referred to as Real-Time Appearance-Based Mapping, is used due to its flexibility to be applied to many different camera systems such as RGB-D, Stereo Camera, Lidar, etc. Whenever a loop-closure is detected, it adds new constraints to the system which optimizes the errors in the graph. It also uses smart memory management system to limit the number of loop-closure points. GPS information helps RTAB-Map avoid incorrect loop closures and aids in localization. RTAB-Map is integrated with the ZED Stereo Camera to create the following image:

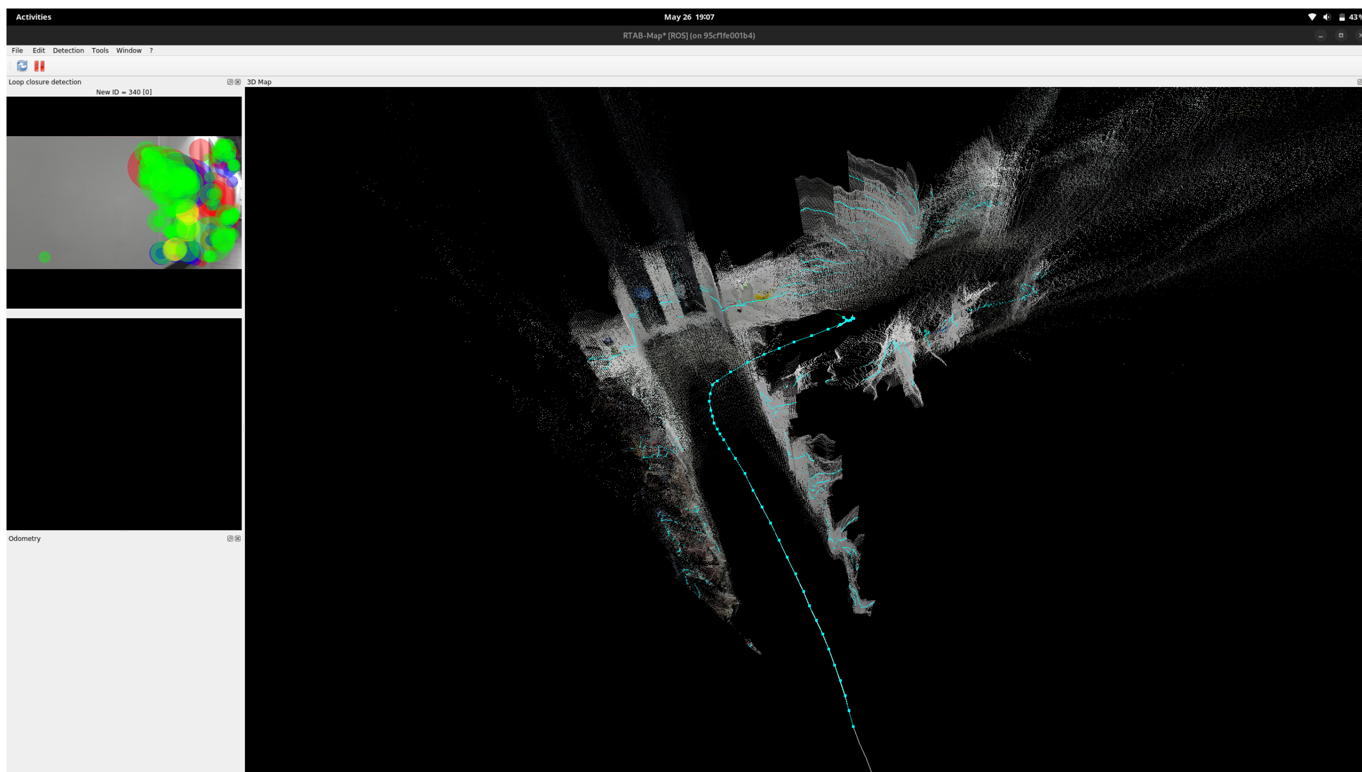


Figure 8: 3D Point Cloud from RTAB-Map with ZED

The move_base ROS package is also integrated with the odometry data output from the ZED node in order to build both a local and global costmap of the surrounding environment of the autonomous vehicle. The following costmap is visualised using the ROS Visualization Tool (RViz):

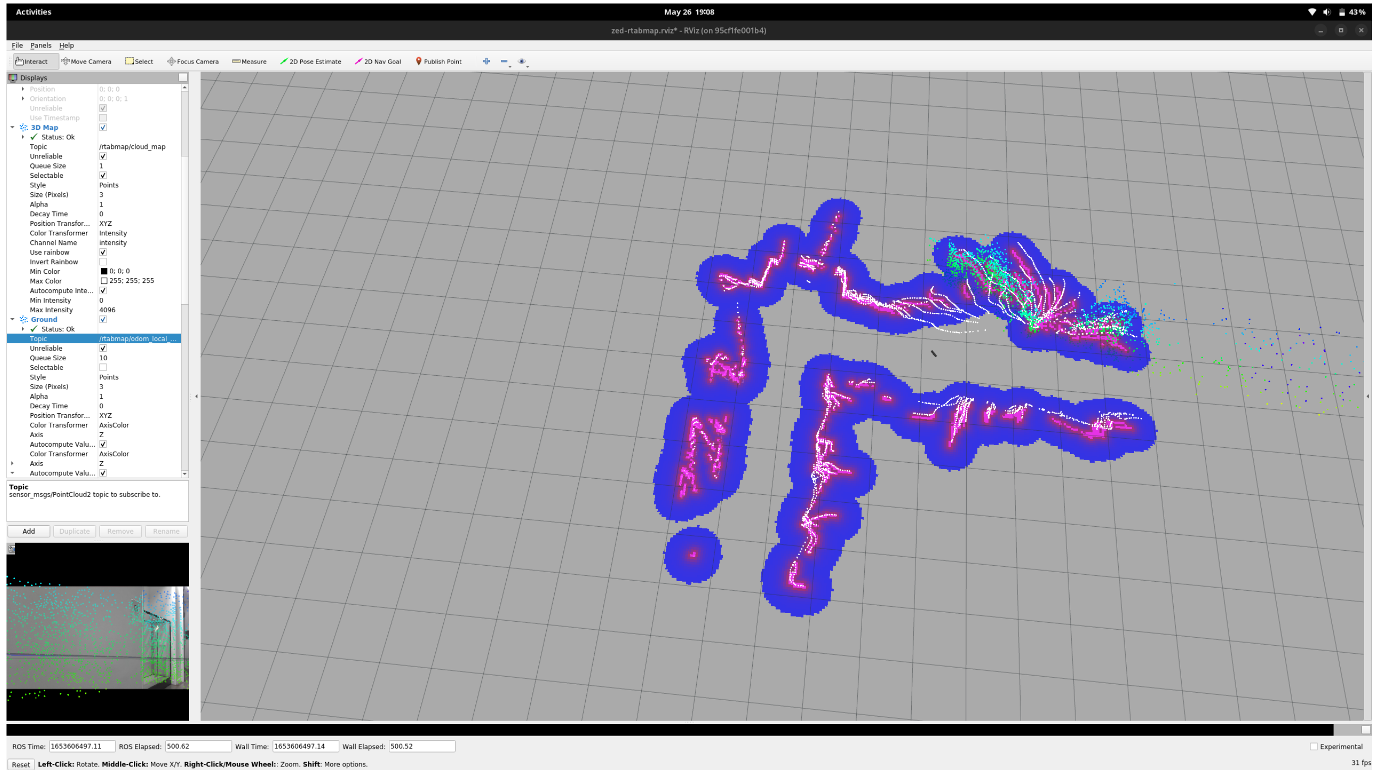


Figure 9: Corresponding Costmap Using move_base ROS Package

5.5 Goal selection and path generation

The goal can be selected by the user on the global map or by GPS coordinate. `move_base` computes a global path, which will be modified locally by behavioral modifiers with the local planner. Modifiers include lane keeping, sign detection and obstacle detection. Modification is either through modification of the local costmap or through direct behavioral intervention, e.g. commanding zero velocity at a stop sign for a set time period.

5.6 Diagrams

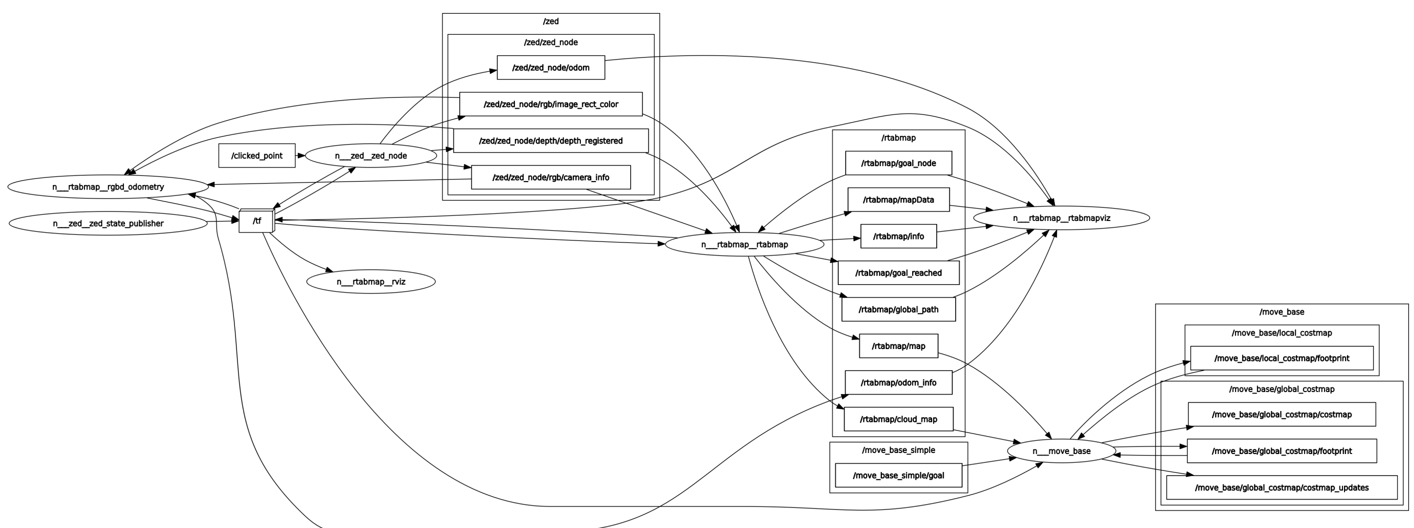


Figure 10: RQT Graph of Core Technical Stack

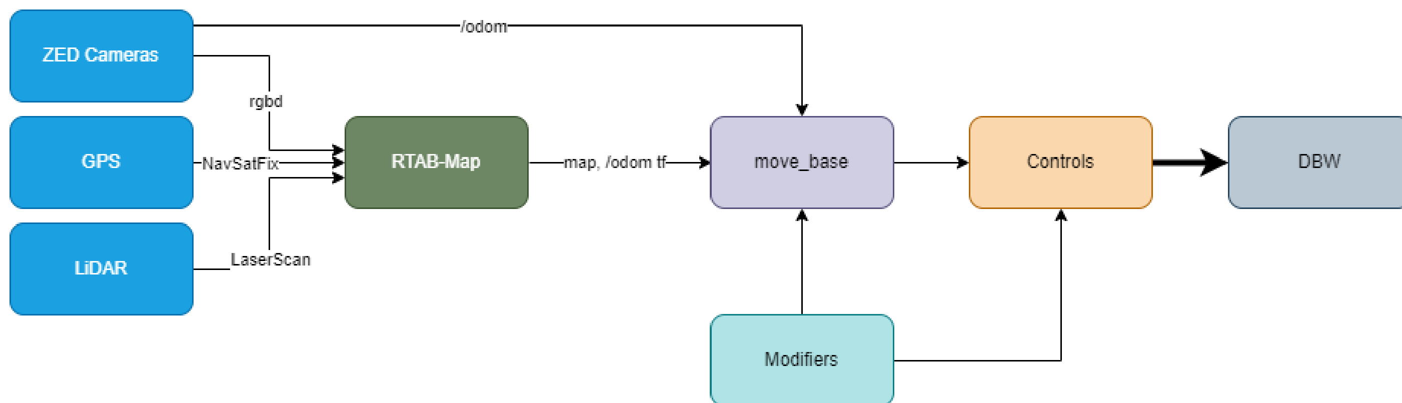


Figure 11: Simplified Tech Stack

6 Description of failure modes, failure points and resolutions

6.1 Vehicle failure modes (software, mapping, etc) and resolutions

Various components of `move_base` monitor the car's local and global situation, including ability to complete the desired local actions and global path. Various recovery behaviors are enabled, but are fairly limited because of the kinematics of a car (versus, for example, a smaller differential-drive robot). If navigation is unsuccessful, an ESTOP is triggered and the session is ended.

6.2 Vehicle failure points (electronic, electrical, mechanical, structural, etc) and resolutions

Our systems are designed to fail safely. Complete electrical failure results in control being handed back to the driver - core control switching is done with relays with their normal positions in the driver-control state, which works even in the total absence of power.

All systems on the car have some degree of self-monitoring and are capable of asserting an ESTOP, including the DBW Nodes, multiple pieces of software on the supervisor computer, controls software, and the technical stack. This is discussed in more detail below.

6.3 All failure prevention strategy

Proactive failure prevention primarily comes in the form of watchdog timers and supervisor daemons throughout our software stack. A watchdog trip or supervisor catching an error will result in an ESTOP being asserted.

For the watchdog located on our DBW Nodes, which monitors task timing, a trip results in a hard-reset of the node. Such a trip will immediately be visible to any supervisor monitoring our CAN bus, as the counter present in the node status message will have differed from it's previous update (i.e. its counter signal will differ), and the node will broadcast the startup reason as `WATCHDOG_TRIP`. Detection of such an event is cause for an ESTOP to be triggered by multiple supervisor daemons, as well as the faulted node itself, if it is capable.

This then moves us to our two supervisor daemons: `nodesd` and `safetyd`. The purpose of `nodesd` is to monitor the health of all our DBW nodes. If a node health message triggers a timeout, or the node health message itself indicates a failure, `nodesd` will issue an ESTOP to all nodes on the CAN bus. Like `nodesd`, `safetyd` is also monitoring DBW system health, but rather in relation to the current commands and reported output through simple rationality checks. For example, if our controls computer is requesting for the vehicle to be moving at a speed and yet the car is not in motion after a set time, `safetyd` will raise an ESTOP.

6.4 Testing (mechanical, electronic, simulations, in lab, real world, etc.)

We use lab equipment and test setups to validate that the electrical hardware on the car is behaving as intended. We verify basic electrical function as well as functionality once integrated with firmware and software.

Extensive checks of the DBW firmware were preformed in our lab before ever taking the car outside. These tests were done by placing the car onto jack stands and sending out predefined CAN messages to the system through an external computer. Performing these tests helped us develop the DBW system safely. Some critical safety tests we preformed before taking the car onto the street included testing if the car would trigger an ESTOP when going over the speed limit,

and testing if the supervisor computer would trigger an E-STOP if one of the nodes went down. Many safety checks, like maximum speed, are redundant and are triggered from multiple independent sources.

To test the Tech stack and the controls algorithms, we take the car outside and collect data. The tech team collects rosbags, which can be used to replay the data collected by the sensors on the car to test algorithms in a repeatable way. To test controls, we run test scripts that encode CAN messages with engineer input to characterize the actuators on the car. The data collected during our outdoor test days is then processed in the lab to refine the control algorithms on the car.

6.5 Vehicle safety design concepts

Our safety design primarily revolves around numerous software layers meant to monitor for failures from the DBW stack upwards.

At the lowest level, each DBW node monitors for hardware failures, if such a failure is encountered, an ESTOP will be raised. For nodes that rely on CAN messages, time deltas for each individual message are kept, if this time delta is exceeded, an ESTOP is raised. Additionally, we have a node supervisor daemon on an eternal computer connected to our CAN bus which monitors node health. If a node's health degrades, or an external ESTOP is triggered, an ESTOP will be issued to all nodes on the CAN bus and hardware ESTOP (power cut to DBW Nodes) tripped if necessary.

Building upon the DBW stack, our controls stack and tech stack must respect both the message rates of the nodes. For the DBW stack to be active, a periodic message must be sent over CAN for a node to maintain the state, failure to do so will result in a node returning to a neutral state, disabling DBW in a safe manner. By doing so, we can guarantee that nodes will only stay in the DBW active state if every other component responsible for controlling our vehicle in autonomous has not failed.

Finally, if all software is to fail, our vehicle has four physical E-STOP buttons and a wireless E-STOP button that cut power to all nodes. All hardware components on the vehicle have been designed in such a manner that a loss in power both restores control to the driver, but also brings the vehicle to a complete and safe stop.

7 Simulations employed

7.1 Simulations in virtual environment

The technical stack was initially tested in a virtual Gazebo environment that replicated the map of the IGVC self-driving course. To further simulate the vehicle, the technical and controls team utilized prerecorded, or "bagged", camera data which captured the vehicle's view of its surroundings and data published from the ROS nodes during the recording. By testing the controllers on this bagged data, both the controls and technical team can debug the software and tune it until its outputs closely represent those expected from a skilled human driver.

7.2 Theoretical concepts in simulations

Autonomous control systems usually rely on proportional-integral-derivative controllers (PID), which are feedback control loops that allow a system to gradually approach a specific value with minimal overshoot when properly tuned. Because the PID controller outputs an acceleration while the system takes in a percentage, an intermediate mapping function is added such that the output of the PID could be converted to a suitable input for the system. In order to determine this mapping function, the system is characterized using a series of step inputs ranging from 1% to 20% in 1% intervals. The characterization tests were done for both acceleration and braking, in which the system's response to a specific percent throttle or percent brake is collected over CAN, converted to a `csv` format, then processed in Excel. During the percent throttle processing, encoder data is used to determine velocity then velocity is graphed to determine the acceleration. These acceleration values are then graphed with their respective percent throttle to develop a mapping that takes in acceleration, multiplies it by some constant, then outputs percent-throttle. The same is done for the brakes, where the encoder data is collected, processed, and used to determine a deceleration-to-brake-percentage function.

However this process accounted only for the the linear portion of the graph. For a more accurate representation of the data, MATLAB/Simulink is used to recreate the mapping function previously determined in Excel. Data from the characterization testing is converted to velocity, then sent through a MATLAB Butterworth low pass filter. Similarly, the derivative of velocity is calculated, then filtered and plotted. The following figures show the determination of the mapping function using a first order differential model, as well as a sample of actual data plotted with the simulated response.

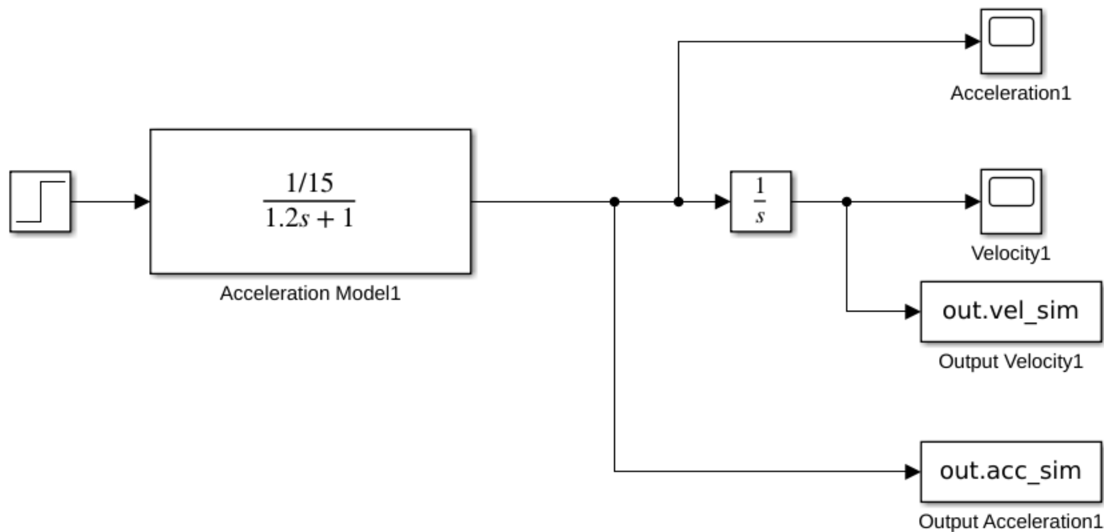


Figure 12: Determination of Acceleration Mapping using First Order Differential

The simulated response closely resembles the actual response, meaning this simulation is a good representation of the system.

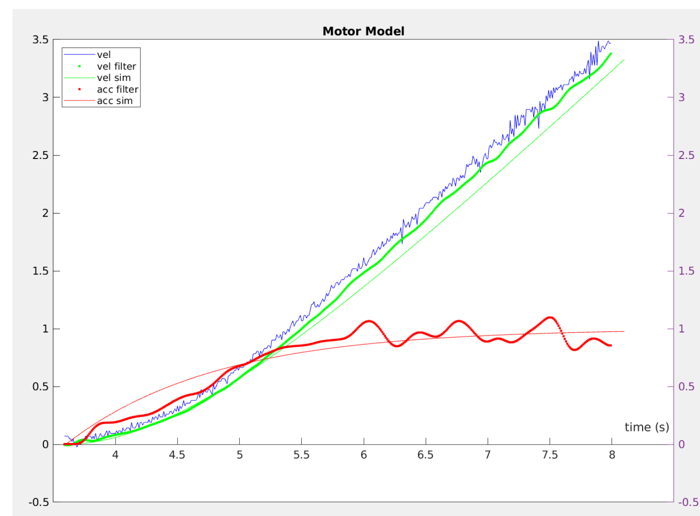


Figure 13: Sample of Actual Data and Simulated Data

Using this simulation of the vehicle's response to different inputs, MATLAB is then used to create a simulation of the ideal system response to various PID gains. Using MATLAB's preinstalled tuning function, a set of PID gains were determined such that they produced a gradual approach to the desired velocity. By maximizing the "robust" value on the MATLAB tuner, the gradual response minimizes possible overshoot of the desired velocity. This way, the vehicle is able to smoothly accelerate and decelerate, preventing erratic motion. The following figure shows the simulated PID controller, which resembles the PID controller used on the car.

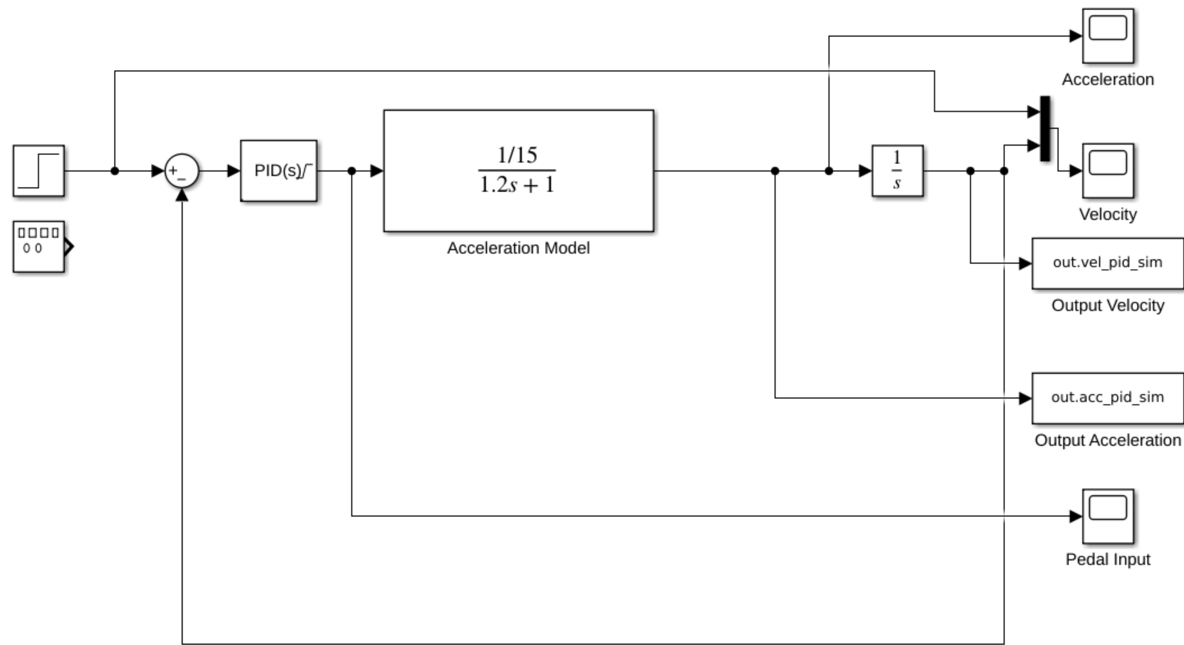


Figure 14: Simulated PID Controller

8 Performance Testing to Date

Extensive testing on the work of all of the subteams in the lab before taking the car out onto the road.

8.1 Component testing, system and subsystem testing, etc

The testing of the technical stack is primarily focused on testing individual sensors such as the Zed Camera, 3D LiDAR, and GPS module. Initial tests were done to connect the Technical computer to each of these individual sensors to collect data from them. This then developed into connecting the outputs from these sensors to move_base in ROS to generate a cost map for the high level controls team.

DBW testing was closely tied with both the ME and EE hardware teams as we worked to connect to nodes to each of the actuators on the car. Testing began with trying to directly interface with each of the actuators (throttle, brake, and steer) without the use of nodes through circuits designed on breadboards or protoboard. These test circuits would eventually become the framework for designs of the break out boards designed to augment the nodes so the node could interface with a particular actuator. After connecting nodes to each of the actuators and testing them individually, we tested our firmware and CAN infrastructure. Unique firmware was tested isolated from the car's CAN bus initially to ensure that it was behaving properly. After that testing was done, the nodes were all connected to test the safety checks of the DBW system. We also ran test scripts of the initial control algorithms in the lab with car elevated.

After these safety tests occurred and we could ensure that the car was safe, we began testing outside. These performance tests outside served two main functions. The first was to collect data that the Technical team could use to test their lane and object detection algorithms. Additionally, these tests were done to characterize the actuators on the car to improve the controls algorithms.

9 Initial Performance Assessments

9.1 How is your vehicle performing to date

Our vehicle was successful in reproducing the safety checks that we had made in the lab out on the road. We manufactured conditions in which the DBW system should fail and the car executed an E-STOP in all cases. While in autonomous

mode, the car would trigger an E-STOP if it exceeded the speed limit of the competition as well as if any of the DBW nodes went offline. Testing showed that our tech stack was responsive and we were able to collect data in real time and publish twist messages that the controls team used to govern the commands sent to DBW. Our throttle controller proved to be quite responsive and capable of maintaining a relatively constant speed after several stages of characterization.

10 Mandatory Unit Test Data To Date

We are including the test for which we can provide data now:

10.1 Unit test 3: Speed limit test

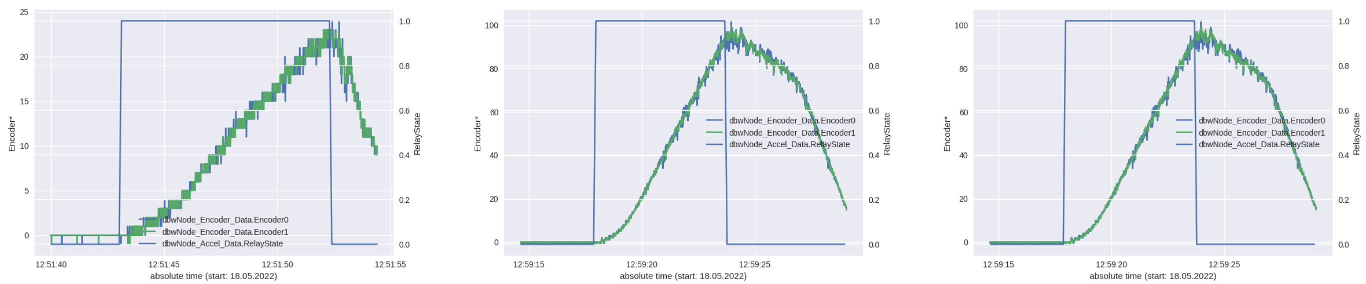


Figure 15: From left to right: Throttle at 5%, 10%, & 15%

While testing and characterizing our vehicle, we set our E-STOP limit to be at 10mph, given that our wheels are 27in in circumference and our quadrature encoder provides us with 4000 ticks per rotation, that leaves us with a value of about 94 ticks every 10ms. In our tests, as seen above, we applied the throttle at various percentages for five seconds, if the throttle node detected that our encoder tick delta exceeded our speed limit, an E-STOP was triggered and subsequently the relay that controls our throttle was disabled. As seen in Figure 15, even though our throttle was applied at 15% for the five second test, the relay was disabled as soon as our speed was determined to be above 10mph (94 ticks/10ms), cutting the test short.

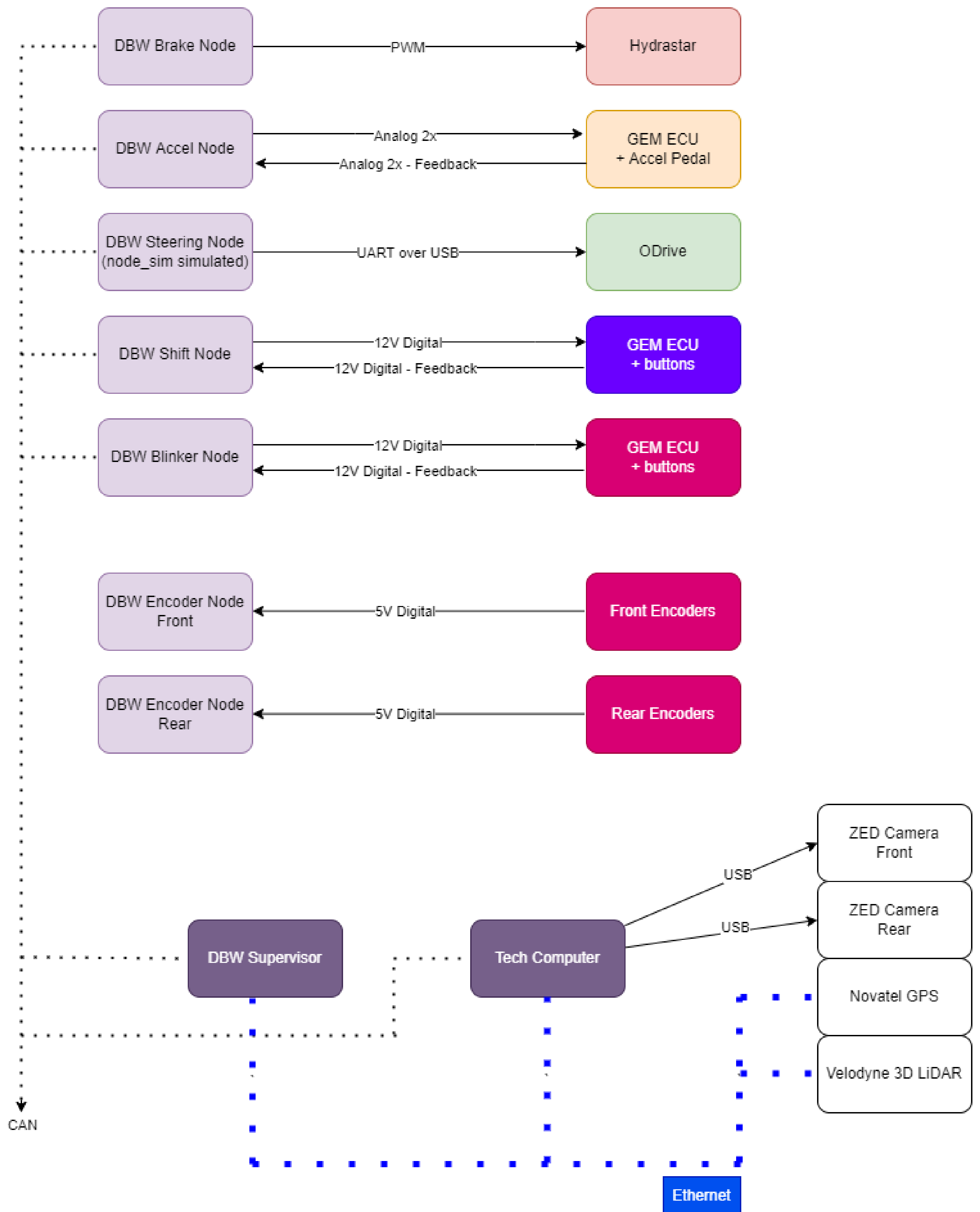


Figure 16: Appendix - Primary Communications Diagram - All-Car