Boston University
Electrical & Computer Engineering
EC464 Capstone Senior Design Project

User's Manual

Torque Vectoring

Submitted to

Izzy Nguyen
Boston University Terrier Motorsport
110 Cummington Mall, Boston, MA 02215
+1 (978) 868-4804
izzyn@bu.edu

by

Team 9
FSAE Torque Vectoring

Team Members

Will Krska wkrska@bu.edu
Giacomo Coraluppi gcor@bu.edu
Alex Zhou alexzhou@bu.edu
Jonathan Ye jonye@bu.edu
Nick Marchuk nmarchuk@bu.edu

Submitted: April 27, 2023

Torque Vectoring User Manual

Table of Contents

Executive Summary

As motorsports categories turn towards electric vehicles, teams are developing new strategies to reduce lap times. One such strategy is to develop a torque vectoring system to independently control two rear wheels of the vehicle to improve cornering performance and driver control. Our project will deliver a fully functional torque vectoring system that can be integrated with the tractive system of the car as well as user-friendly documentation to assist clients with implementing the torque vectoring system. The proposed technical approach is to design a proportional–integral–derivative (PID) controller to track the yaw rate of the rear-wheel driven vehicle. To do so, a 2-degree of freedom (DOF) linear model and a 7-DOF nonlinear model will be designed. The developed controller will be embedded into an onboard microcontroller on the vehicle to control the independent motor inverters. To maintain the robustness of the hardware, a Controller Area Network (CAN) interface will be developed so that the motors can maintain a given torque difference from the controller. The main innovative feature of the project is to independently control how much power goes to each of the rear wheels rather than giving the same power to both to improve speed and control. Additionally, we focus on user serviceability so the system can easily be adapted to any future vehicle. The developed system will be easily implemented on existing vehicles given our properly adaptable software codebase, and an utilization/implementation guide will be communicated through comprehensive documentation.

1 Introduction

Torque vectoring is utilized in automotive vehicles to control the amount of torque and power distributed to each individual wheel. Using this type of efficient technology allows the vehicle to have more traction around tight corner-style turns while maintaining greater acceleration. Torque vectoring also aims to bolster the steering response and handling based on the driver's inputs under certain driving conditions.

As part of Terrier Motorsport's new formula-style electric race vehicle, our team designed and constructed a fail-safe torque vectoring system to best utilize the performance of each rear wheels' independent drives. Our goal was to develop a torque vectoring system that accounts for the dynamics, controls, sensors, and electrical components of a formula race car in an efficient manner. Current conventional solutions only apply one brake to create a torque differential, which wastes a lot of energy; our system drastically reduces the amount of wasted energy output in comparison. Given the compact nature of our physical design and the availability of our software and hardware components, we hope that our torque vectoring system prototype can be deployed not only to Terrier Motorsport's new electric race car, but that it can also serve as a customizable framework for configuration and implementation on other styles of Formula Society of Automotive Engineers (FSAE)-based race vehicles.

In addition to the simplicity and modularity of system deployment, our team focused on affordability of the entire system. While numerous torque vectoring systems already exist out there, these solutions are generally commercial-based and often proprietary for very specific configurations. As a result, these systems are generally more expensive to purchase, in addition to being not as modular/simple. In contrast, we strived to limit our prototype to a reasonable cost of under $500 (within our allocated budget) so that even organizations with some form of resource/financial scarcity may have a better opportunity to afford our solution.

In the next several sections, we describe the setup and installation of the necessary components needed to run our project. We then detail the exact procedure to operate our system, including accounting for scenarios of abnormal operations. In addition, we give a technical overview about our project background and decision-making process, including a bill of materials and the engineering standards we aimed to abide by. Supporting diagrams and charts are attached to the end of this report under the appendices section.

2 System Overview and Installation

2.1 Overview Block Diagram

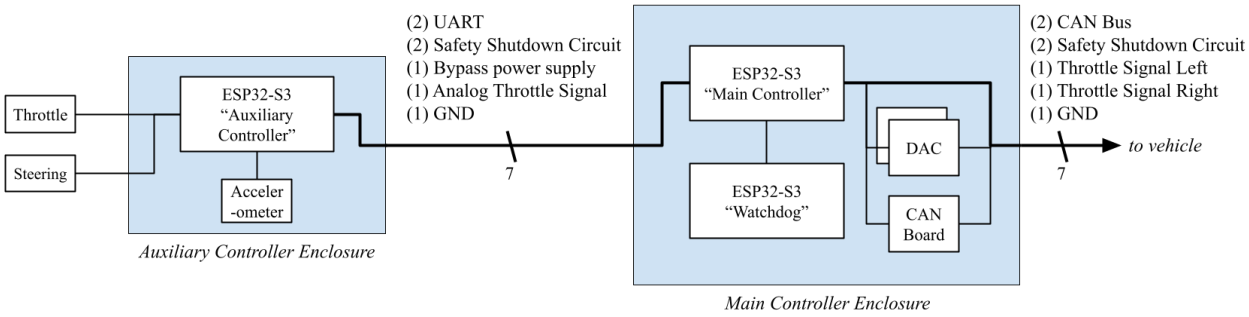


Figure 1: Block diagram of the entire system. Data transmission flows from the left to the right side.

The system consists of two enclosures, connected to each other by a custom wiring harness, and to the vehicle with a similar wiring harness.

The auxiliary controller enclosure houses the auxiliary controller, which relays input sensor data to the main controller via the Universal Asynchronous Receiver-Transmitter (UART) communication protocol. The enclosure includes our accelerometer sensor as one of our data inputs; the other two sensor inputs come from the steering wheel and the throttle pedal, which are all communicating to the auxiliary controller via Inter-Integrated Circuit (I2C).

The main controller enclosure houses several primary components: the main controller, which is running the torque vectoring algorithm; the watchdog, which monitors system behavior for anomalies; two Digital to Analog Converters (DACs), which output throttle signals; and a CAN Bus controller, which communicates telemetry data with the vehicle.

The output of the overall system is our 3D-printed vehicle demonstration model. Assembled with acrylic boards, printed wheels, and drone motors, the car will respond to the user's inputs coming from the steering, throttle, and accelerometer by taking in the output values produced from the main controller and feeding it through the motor controller attached to the vehicle. In turn, the wheels will spin and/or turn accordingly in real-time.

2.2 Physical Description

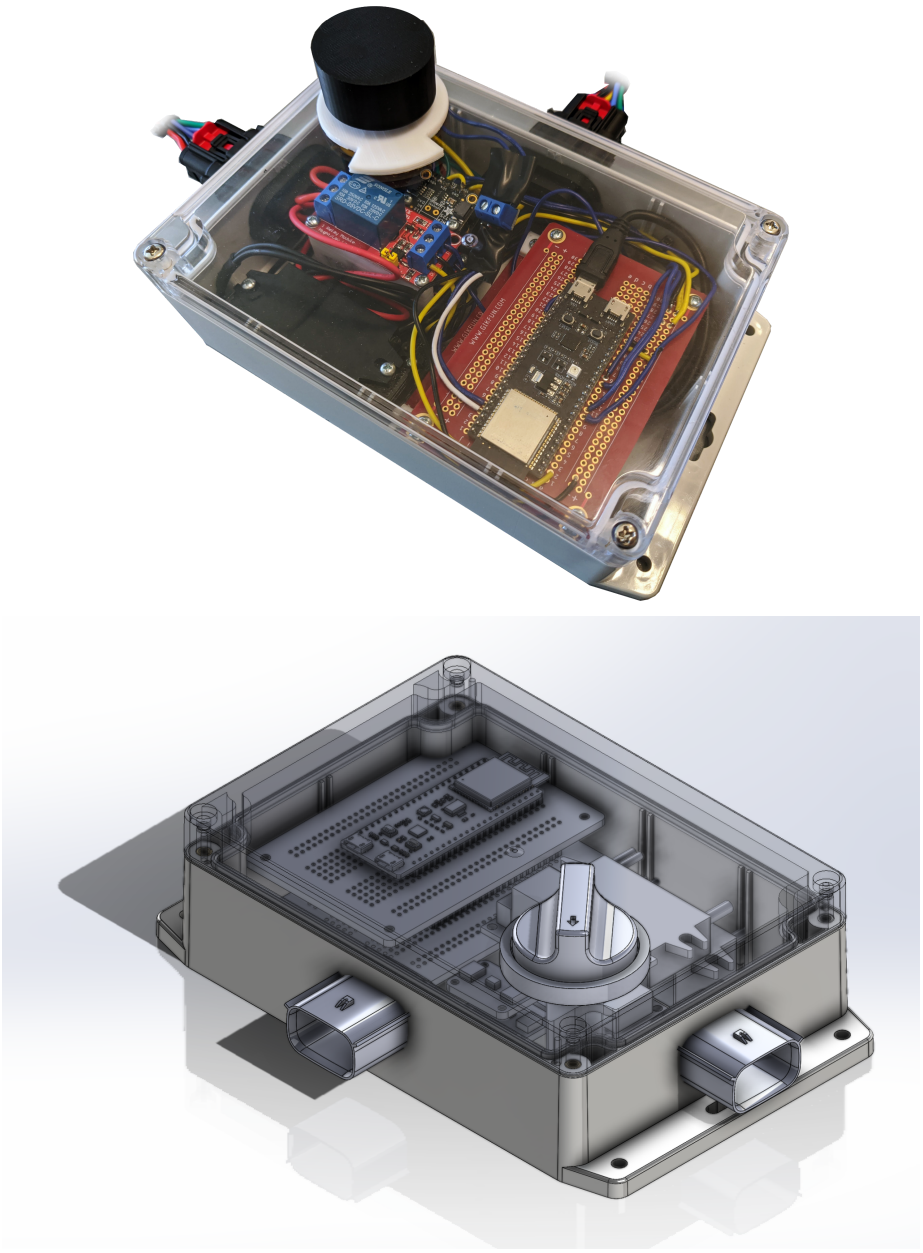


Figure 2: Image of main controller enclosure after assembly (Top); Computer Aided Design (CAD) representation of the main controller enclosure and its components (Bottom). The auxiliary controller enclosure is similar in design.

Each of two enclosures for the auxiliary and main controllers conforms to the requirements defined by our client. These include a compact 8 x 8 x 8in container for modular installation and weather-sealed packaging for all driving and environmental conditions. Our 3D-printed vehicle demonstration model serves as a visualization tool for users to see how manipulating the driving inputs affect the performance of the vehicle with regards to steering and motor control. Given that the client's electric race car has not been constructed yet, our demonstration model is a microcosm for the future vehicle's realistic behavior.

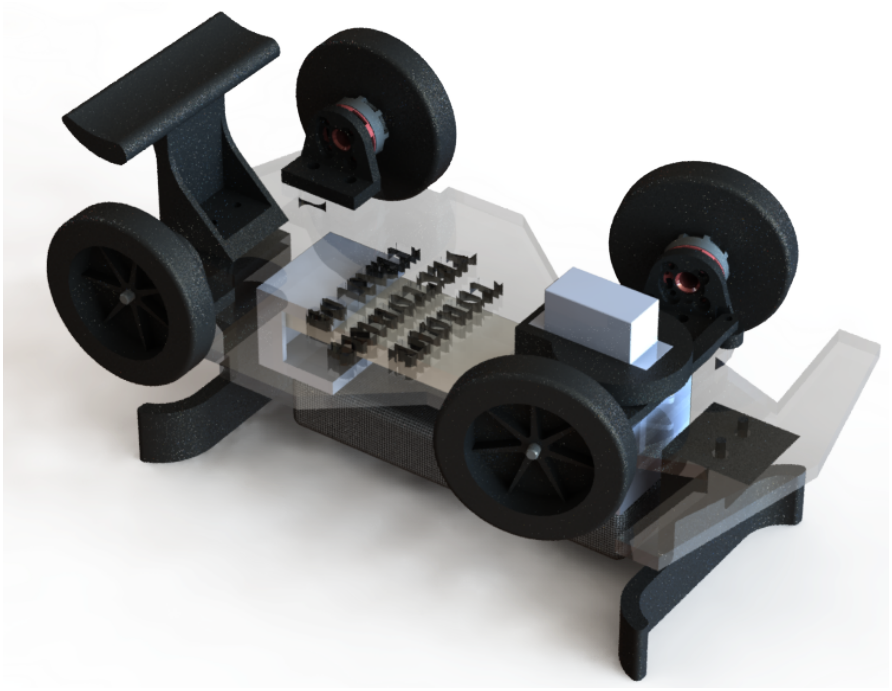


Figure 3: Render of the stationary demonstration model in CAD, featuring four wheels driven by hobbyist drone motors and a rudimentary steering rack

2.3 Installation, Setup, and Support

2.3.1 Physical installation and setup

When installing the main controller and auxiliary controller enclosures for the first time, ensure a rigid connection is made to the vehicle to prevent damage to the enclosure and electrical contacts. The main controller enclosure should be in close proximity to the vehicle's inverters and grounded low voltage (GLV) system. The auxiliary controller enclosure should be in close proximity to the pedal box and steering rack/steering sensor.

In order to connect the main controller enclosure to the auxiliary controller enclosure, the custom wiring harness must be assembled. Measure to the appropriate length three (3) strands of 18 AWG stranded wire and four (4) strands of 18-22 AWG wire such that the wires can reach each enclosure without strain or risk of snagging. Crimp the appropriately sized MX150 blade receptacle onto both ends of the strand, and install into the provided 12-contact Molex MX150 weather-sealed female plug per the pinout described in Appendix D.2.

In order to connect the main controller to the vehicle, run the appropriate wires from the inverters and GLV system, crimp into the appropriately sized MX150 blade receptacle, and install into provided 12-contact Molex MX150 weather sealed female plug per the pinout described in Appendix D.1. There is no additional startup sequence. Ensuring the switch on the main controller enclosure is turned to "Active", the system will automatically activate when power is supplied via the 12V input to the main controller enclosure.

2.3.2 Algorithmic setup

A key component of this project is the relative ease of adapting it for any vehicle with independent rear drives. To do so, the user must modify the Matlab Simulink model with their particular parameters. This model can be exported and merged with the codebase for the main controller. Finally, the new firmware needs to be flashed to the main controller. The main controller, which is the top of the two microcontrollers housed in the main controller enclosure can be connected to a computer with the Espressif IoT Development Framework (ESP-IDF) installed via the free micro-USB port on the board. In a future iteration of this manual, we will go into much greater detail regarding how to edit the Simulink model and integrate it into the main controller codebase and how to program the microcontroller.

2.3.3 Software setup

In order to compile and run the code associated with our ESP32-S3 microcontrollers and other components, installation of the Espressif Standard Toolchain setup is required, which also includes the ESP-IDF library repository. Espressif provides users with a detailed step-by-step guide on how to complete these installation and configuration steps. Our team highly recommends users follow the link in the footnotes below to complete the setup process since various operating systems may require a different set of steps for configuration, so the website has the most detailed explanations and options.[1] Here is a general installation summary:

1. At the bottom of the first page linked below, a choice between using the integrated development environment (IDE) versus manual installation is presented. Due to IDE dependency and configuration issues, our team proceeded with the manual installation option; select the manual procedure that best aligns with the user's operating system.
2. Install the latest version of certain software packages based on the user's operating system, which includes Python 3 (mostly for Mac/Linux users).
3. For Mac/Linux users, clone the ESP-IDF Github repository (https://github.com/espressif/esp-idf.git) or install the ESP-IDF tool installer (Windows).
4. Set up the tools and environmental variables used by ESP-IDF on Mac/Linux, which include debuggers, compilers, and supporting packages. Windows users can simply follow the setup wizard instructions for ESP-IDF.
5. Start a project by navigating to the "hello_world" code example under the directory (./examples/get-started/hello_world). Connect the ESP32-S3 device to the computer and find the serial port name/number.
6. Configure, build, and flash the example code onto the ESP32-S3 device with these three commands in the following order:
    a. idf.py set-target esp32s3

[1] Link to ESP-IDF detailed installation:
https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/get-started/index.html

b. idf.py build
c. idf.py -p [*PORT*] flash monitor (replace *PORT* with user's serial port number connected to their ESP32-S3 chip)

Once those previous installation steps are successfully completed and verified with the example code, users can then navigate into the following three directories within our team's Github:

- SrDes-main_controller
- SrDes-motor_controller
- SrDes-watchdog

Within each of these directories, follow step 6 from above to upload the software onto their corresponding ESP32-S3 microcontrollers (3 in total). If no compilation errors are detected, then the programs have been successfully loaded onto the microcontrollers.

Users will also be required to have the latest Arduino IDE software and relevant libraries installed in order for the auxiliary controller to read data in from the accelerometer, as well as the steering and throttle inputs. This process is generalized as follows:

1. Navigate to the Arduino website and download the latest IDE version, which is also linked below in the footnotes.[2]
2. Once setup is completed, go to the library manager under the "Sketch" tab, and then install the "Adafruit LSM6DS" library by searching for it. If not included in that library bundle, please also install the "Adafruit BusIO" and "Adafruit Unified Sensor" libraries.
3. Example code is provided based on the sensor type. For our accelerometer, the Adafruit ISM330DHCX, navigate into the *File → Examples → Adafruit LSM6DS → adafruit_ism330dhcx_test* code file and run it to verify functionality with the proper Arduino controller type (our team used an Arduino Nano).
4. If step 3 is successful, then replace the test code with our team's auxiliary controller code under the "SrDes-aux_controller" directory and upload it to the Arduino.

After both the Arduino and ESP-IDF components are successfully installed and tested with our controller codebases, then the software and hardware is ready for user input testing.

2.3.4 Linear model integration setup

After the C code is generated from the Matlab Simulink, the model needs to be integrated with the main controller code. To do this, users first transfer all the files that were generated by Matlab and add them to the main folder of *SrDes-main_controller*, replacing the previous versions. Then copy and paste all the lines in the *Linear_Model.c* file into *main_controller.c*,

[2] Link to Arduino IDE installation: https://www.arduino.cc/en/software

replacing any old linear model code if necessary. Next, the input variables from the linear model will need to be renamed in accordance with our data nomenclature. All instances of *Linear_Model_U.throttle_Input* in the file must be replaced with *throttle_scaled*, all instances of *Linear_Model_U.steeringAngleEncoder* must be replaced with *steering_scaled*, and all instances of *Linear_Model_U.V_CG* must be replaced with *speed_scaled*. To avoid compilation errors, users must delete or comment out all the functions that relate to Matlab data logging. These same lines need to be commented out in *Linear_Model.c* and *Linear_Model.h* as well. After making these changes in the code, users can build the updated linear model with the rest of the controller code and upload it to the ESP32-S3 with the most up-to-date version of the linear model.

3 Operation of the Project

3.1 Operating Mode 1: Normal Operation

The system can operate in one of two modes: "Active" and "Passthrough". Additionally, although the delivered product is configured for an accelerator sensor that takes 12V VCC (such as the Honeywell RTY###HVNAX family), the system can be reconfigured to accept a 5V power supply from the inverter to drive an accelerator sensor that takes 5V VCC (such as the Honeywell RTY###LVNAX family). System behavior between each configuration is functionally the same, and all of the following applies to both configurations unless explicitly noted otherwise.

In "Active" mode, the torque vectoring system is fully operational. On startup, the user will hear a series of five relay clicks. These represent the "watchdog", main controller, auxiliary controller, steering, and throttle safety systems coming online respectively. These five relays are closing the "Safety Shutdown Circuit" (SSC), a preexisting required system on the vehicle, which must be closed for the accumulator to be energized and power to be sent to the wheels. The torque vectoring system will respond to the inputs of the driver in accordance with the vehicle dynamics model.

In "Passthrough" mode, the torque vectoring system is deactivated. On startup, the user will hear only one click for the relay for the throttle pedal. The output of the throttle pedal is directly sent to each inverter; it is not modulated by a DAC or analog-to-digital converter (ADC) at any point.

If the user desires the aforementioned 5V configuration, they must make the alterations detailed in Appendix C.1, C.2. In this configuration, there will not be a relay associated with the throttle sensor, and therefore there will be one less "click" in each mode. Additionally, the user must add a wire to the Molex MX150 connector from the main controller enclosure to the vehicle for the 5V supply.

The system will automatically and safely boot up and shut down with the vehicle.

3.2 Operating Mode 2: Abnormal Operations

The torque vectoring system may open the SSC for a number of reasons, immediately shutting down the rest of the system and the vehicle. If any of the microcontrollers encounter a fault that disrupts the execution of their program, the relay associated with that microcontroller will open, and the SSC will open. If either the throttle or steering sensor experiences a short of VCC to Output or Ground, the SSC will open. If the "watchdog" observes abnormal voltages being supplied to the inverters, the characterization of which can be defined by the user, it will deliberately open the relay associated with it, and the SSC will open.

In any of these cases, the user should know that is intended behavior in the face of a potentially dangerous failure. Barring a hardware fault, the vehicle can be restarted with no further attention

to either of the controllers. If it was due to a hardware fault, the faulty component would need to be replaced, and the system should operate as designed afterward.

3.3 Safety Issues

The system itself presents no direct hazard to those operating or working on the vehicle. However, as the system is breaking the direct connection from the throttle pedal to the inverter when activated, it inherently increases the risk of abnormal vehicle behavior. These risks have been mitigated with several safety systems, but the user is ultimately responsible for their own safety.

4 Technical Background

Torque vectoring employs a computer system that controls the power of automobile differentials more effectively in order to improve grip on road surfaces and allow the vehicle to accelerate more quickly, especially on turns. So, this system will integrate electrical components, a control algorithm, and mechanical implementation.

The main foundation of the hardware/software integration lies within our choice of microcontrollers: the ESP32-S3 developed by Espressif. We chose these parts to drive the main, motor, and watchdog controllers of our system due to the wealth of communication features they provide for us to integrate our components, including two discrete 10-channel ADCs, integrated UART/Serial, I2C, and Serial Peripheral Interface (SPI) controllers, and two logical cores. Other features that were of vital importance included 512kB SRAM, ideal for lookup tables, and its dual-core design allowing multi-task programs. By using the ESP32-S3, our primary software functions were automatically encompassed by ESP-IDF, Espressif's official IoT Development Framework. ESP-IDF contains libraries and configurations within their SDKs to compile and run programs on the ESP32 chips. ESP-IDF is completely free and open source on Github, making it easily accessible to use example ESP-IDF code and modify it to our project deliverables.

Initially, the auxiliary controller was also based on the ESP-IDF functionality as well. However, integrating the Adafruit ISM330DHCX accelerometers with the ESP32-S3 proved to be a difficult challenge, given the complexity of the data byte register mapping on the datasheet and the lack of Internet resources. However, this exact accelerometer had plentiful guidance for integration with an Arduino-based setup. Realizing that the auxiliary controller's main function was transferrable to the Arduino as well, our team decided to switch gears and use a spare Arduino Nano instead of an ESP32-S3 to read sensor data from the accelerometer, steering, and throttle inputs, then package that data to the main controller over UART.

Communication between multiple ESP32-S3s and the Arduino Nano is fundamental to our system design, and we decided to stick with the available UART protocols. UART has a simple data transmission wiring procedure (generally just two wires) that facilitates low hardware complexity and a one-to-one connection between two microcontrollers to send an active digital signal. UART protocol also features reliability for long-distance communication, which aligns with our system design separating the auxiliary and main controllers. Located on different sections of the car, the auxiliary controller can easily feed the input data through packets free of interference to the main controller.

In terms of sensors, our suite collection originally encompassed the following three inputs:

- Adafruit ISM330DHCX accelerometers

- Honeywell RTY360HVNAX rotary steering sensor
- Honeywell RTY050HVNAX throttle sensor

However, due to functional issues with the rotary steering sensor, we could not proceed with using our originally defined steering and throttle sensors. Instead, our team 3D-printed a steering wheel component and collected a formula car throttle pedal from our client, then wired those to the input channels of the auxiliary controller. Our accelerometers were chosen due to the compact packaging and high precision performance, especially with I2C communication to the controllers. Like UART, I2C communication involves a simple hardware setup of two wires: one for the data channel (SDA- serial data) and the other for clock signal (SCL- serial clock). It also supports multi-master and multi-slave buses for high-speed, device connections within a short distance from the ESP32-S3s and Arduino Nano, allowing multiple functionalities and devices to be supported simultaneously.

To design a control algorithm for the torque vectoring system, a variation of PID, a PI controller, was determined to be the easiest to implement, as the team had previous experience with this and there are ample resources online pertaining to PIDs. A PI controller is a PID controller without the derivative term, as it can introduce instability in the controller, especially since our system will have a user input (steering encoder), which can be inherently noisy due to vibrations and the user themselves being unstable[3]. Once the PI controller was determined to be the control algorithm, developing a method of tuning the controller is necessary. To do so, a linear model will be developed to use a root locus method of tuning the controller to streamline the tuning. Due to the nature of the linear model, the PI controller will need to be further linearized at different longitudinal velocities.

Before the model is discussed, a list of variables is necessary:

$$C_{y,f},\ Cornering\ Stiffness\ of\ the\ front\ axle$$
$$C_{y,r},\ Cornering\ Stiffness\ of\ the\ rear\ axle$$
$$m,\ mass\ of\ the\ vehicle$$
$$V_{x0},\ Initial\ Longitudinal\ Velocity$$
$$l_f,\ Distance\ between\ the\ center\ of\ gravity\ and\ the\ front\ axle$$
$$l_r,\ Distance\ between\ the\ center\ of\ gravity\ and\ the\ rear\ axle$$
$$I_{zz},\ Moment\ of\ inertia\ around\ the\ vertical\ vehicle\ axis$$
$$k_u,\ under-steer\ gradient$$

[3] https://www.controleng.com/articles/understanding-derivative-in-pid-control/

The linear model of the car will be a bike model that condenses the front two wheels and the back two wheels into a single wheel, respectively. This effectively reduces the model to exclude all lifting, rolling, and pitching motion, the aligning torque from the side slip angle, and the wheel-load distribution that would be present in the nonlinear model. From the textbook "Vehicle Dynamics: Theory and Application" by R.N. Jazar, the equations work out to:

$$\dot{v_y} = -\frac{C_{y,f} + C_{y,r}}{mv_{x0}} v_y + \left(\frac{-l_f C_{y,f} + l_r C_{y,r}}{mv_{x0}} - v_{x0}\right)\dot{\psi} + \frac{C_{y,f}}{mv_{x0}}\delta$$

$$\ddot{\psi} = \left(\frac{-l_f C_{y,f} + l_r C_{y,r}}{I_{zz} v_{x0}}\right) v_y - \left(\frac{l_f^2 C_{y,f} + l_r^2 C_{y,r}}{I_{zz} v_{x0}}\right)\dot{\psi} + \frac{l_f C_{y,f}}{I_{zz}}\delta$$

And can be written in their state-space equations:

$$\dot{x} = Ax + Bu_1 + Eu_2$$

$$\dot{x} = \begin{bmatrix} \dot{v_y} \\ \ddot{\psi} \end{bmatrix}; \qquad u_1 = M_z; \qquad u_2 = \delta$$

$$\begin{bmatrix} \dot{v_y} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} -\frac{C_{y,f} + C_{y,r}}{mv_{x0}} & \frac{-l_f C_{y,f} + l_r C_{y,r}}{mv_{x0}} - v_{x0} \\ \frac{-l_f C_{y,f} + l_r C_{y,r}}{I_{zz} v_{x0}} & -\frac{l_f^2 C_{y,f} + l_r^2 C_{y,r}}{I_{zz} v_{x0}} \end{bmatrix} \begin{bmatrix} v_y \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{I_{zz}} \end{bmatrix} M_z + \begin{bmatrix} \frac{C_{y,f}}{mv_{x0}} \\ \frac{l_f C_{y,f}}{I_{zz}} \end{bmatrix} \delta$$

With this model in mind, we can start building a Simulink model. However, we still need to calculate the desired yaw rate. This can be calculated from equations found in "Fundamentals of Vehicle Dynamics" by T.D. Gillespie. The equation works out to be:

$$\dot{\psi}_{desired} = \frac{v_{CG}}{(l_r + l_f) + K_u v_{GC}^2}\delta$$

The variables used to determine the motor power output was:

$MotorOutput_{right} = input\ voltage\ given\ to\ motor\ inverter\ for\ the\ right\ motor$

$MotorOutput_{left} = input\ voltage\ given\ to\ motor\ inverter\ for\ the\ left\ motor$

$ThrottleInput = throttle\ signal\ between\ -0.5\ to\ 0.5$

$VoltageMax = max\ voltage\ given\ to\ motor\ inverter\ (5\ V)$

$\Delta T = torque\ difference$

$$maxT = max\ torque\ from\ motors$$

To calculate the power distributed to each motor these equations are used:

When steering right:

$$MotorOutput_{right} = (1 - ThrottleInput)VoltageMax$$
$$MotorOutput_{left} = MotorOutput_{right} - (\Delta T/maxT)VoltageMax$$

When steering left:

$$MotorOutput_{left} = (1 - ThrottleInput)VoltageMax$$
$$MotorOutput_{right} = MotorOutput_{left} - (\Delta T/maxT)VoltageMax$$

An anti-windup protocol is also implemented to limit the integrator gain if the torque difference was greater than the max torque output from the motors:

$$If\ \Delta T > maxT:$$
$$I_{gain} = 0$$

5 Relevant Engineering Standards

Formula Hybrid Requirements

EV3.5.4	All acceleration control signals (between the accelerator pedal and the motor controller) must have error checking. For analog acceleration control signals, this error checking must detect open circuit, short to ground and short to sensor power For digital acceleration control signals, this error checking must detect a loss of communication.
EV3.5.5	TS circuitry, even at low voltage or current levels, is not allowed in the cockpit. All control signals that are referenced to Tractive System (TS) and not GLV, such as non-isolated motor controller inputs, must be galvanically isolated and referenced to GLV ground.
EV3.5.6	The accelerator signal limit shutoff may be tested during electrical tech inspection by replicating any of the fault conditions listed in EV3.5.4
EV7.1	Shutdown Circuit The shutdown circuit is the primary safety system within a Formula Hybrid + Electric vehicle. It consists of a current loop that holds the Accumulator Isolation Relays (AIRs) closed. If the flow of current through this loop is interrupted, the AIRs will open, disconnecting the vehicle’s high-voltage systems from the source of that voltage within the accumulator container.
EV7.1.1	The shutdown circuit must consist of at least: (a) Grounded Low Voltage Master Switch (GLVMS) See: EV7.3 (b) Tractive System Master Switch (TSMS) See: EV7.4 (c) Two side-mounted shutdown buttons (BRBs) See: EV7.5 (d) Cockpit-mounted shutdown button. See: EV7.6 (e) Brake over-travel switch. See: T7.3 (f) A normally open (N.O.) relay controlled by the insulation monitoring device (IMD). See: EV7.9 and Figure 38. (g) A normally open (N.O.) relay controlled by the accumulator management system (AMS). See: EV2.11 and Figure 38. (h) All required interlocks. (i) Both IMD and AMS must use mechanical latching means as described in Appendix G. Latch reset must be by manual push button. Logic-controlled reset is

	not allowed. (j) If CAN communication is used for safety functions, a relay controlled by an independent CAN watchdog device that opens if relevant CAN messages are not received. This relay is not required to be latching.
EV7.1.2	Any failure causing the GLV system to shut down must immediately deactivate the tractive system as well.
EV7.1.3	The safety shutdown loop must be implemented as a series connection (logical AND) of all devices included in EV7.1.1. Digital logic or microcontrollers may not be used for this function. Normally open auxiliary relays may be used to power high-current devices such as fans, pumps etc. The AIRs must be powered directly by the current flowing through the loop.
EV7.1.4	All components in the shutdown circuit must be rated for the maximum continuous current in the circuit (i.e. AIR and relay current).
EV7.7.2	After enabling the shutdown circuit, at least one action, such as pressing a “start” button must be performed by the driver before the vehicle is “ready to drive”. i.e. it will respond to any accelerator input.

The engineering resiliency of the electronics is most strongly rooted in the waterproof enclosures that were purchased to hold the electronics. The enclosure is a Polycase WC-24F rated to National Electrical Manufacturers Association (NEMA) 4X, which means it is rated to be impervious to foreign objects, waterproof, and dust resistant. While our drilling into the enclosure reduces the usefulness of the rating in terms of guaranteed performance, our choices of smart machining practices and waterproof cable connections should mean that the case is still as strong and water-resistant as needed.

A basic standard of the project is that it’s built to withstand g-forces of 2 or less, and more generally able to withstand the normal drops, shakes, vibrations, and other perturbations inherent in being mounted on a racing vehicle. The way that this was accomplished was by designing custom additive manufactured mounts for each piece of hardware. These mounts are glued to the floor of the enclosure and have the electronics screwed into them. Along with all the wiring connecting the electronics, the fittings, and the electronics themselves, we have designed the interior of the case to protect its contents from most blows that it can expect.

6 Cost Breakdown

Listed below is the cost breakdown for our torque vectoring system and vehicle prototype. We have included a few quantities of extra components to account for any potential malfunctions.

Project Costs for Production of Beta Version (Next Unit after Prototype)				
Item	Quantity	Description	Unit Cost	Extended Cost
1	4	Adafruit ESP32-S3 DEVKITC Microcontroller	$15	$60
2	2	Adafruit ISM330DHCX Accelerometer	$20	$40
3	1	Grayhill Rotary Switch	$50	$50
4	2	Adafruit MCP4725 12-Bit DAC with I2C Interface	$5	$10
5	1 (pack of 8)	AZKO 5v Relay Board Relay Module 1 Channel Opto-Isolated High or Low Level Trigger	$12	$12
6	1 (pack of 5)	Solderable Breadboards GK1007	$12	$12
7	1 (pack of 2)	Waveshare SN65HVD230 CAN BUS Control Board	$18	$18
8	2 (pack of 2)	DC-DC Buck Converter Module 12V to 5V Micro USB Power Adapter	$12	$24
9	2	Polycase WC-24F Outdoor Enclosure with Clear Cover (including pack of 100 screws)	$32	$64
10	3	Molex MX150 12 pin male panel mount	$5	$15
11	3	Molex MX150 12 pin female inline plug	$5	$15
12	36	Molex MX150 blade	$0.10	$3.60
13	36	Molex MX150 socket	$0.10	$3.60
14	6	TE Connectivity AMP Connector Blade	$0.15	$.90
15	6	TE Connectivity AMP Connector Socket	$0.40	$2.40
16	2	TE Connectivity AMP Superseal 1.5 3-pin	$1.50	$3
17	42	TE Connectivity Single Line Seal	$0.03	$1.26
18	1	Hilitchi Professional Pin Crimping Tool	$17	$17

19	1	TUOFENG 22 AWG Wire Solid Core Hookup Jumper Wires (6 colors)	$15	$15
20	2	McMaster Cast Acrylic Sheet	$16	$32
			Beta Version-Total Cost	$398.76

All of our components listed were purchased within our allocated budget from the ECE department (up to $1000). Except for a handful of spare jumper wires in the Terrier Motorsport inventory, all remaining hardware components had to be purchased from external vendors including DigiKey Electronic, Mouser Electronics, Polycase, McMaster-Carr, and Amazon. All software requirements (such as the Espressif ESP-IDF development framework and the Arduino IDE) should be open source and free of charge for the user, assuming a computer device is available to run the programs and frameworks as mentioned in the earlier sections.

Note: All costs in the table above do not include shipping & handling fees or taxes. All unit and extended costs are subject to change, depending on current demand of the components. These estimated costs were updated as of the date of the final user's manual submission.

7 Appendices

7.1 Appendix A - Specifications

Requirement	Value, range, tolerance, units
Performance Sensitivity	±2g with 0.05g sensitivity
Steering Sensitivity	5° steering sensitivity
Throttle Input Latency	< 50 ms throttle input latency
Steering Input Latency	< 250 ms steering input latency
Design Specifications	Fail-safe and runs on 12V power source
Packaging	< 8in x 8in x 8in Weather sealed
Prototype Cost	Below $500

7.2 Appendix B – Team Information

William Krska (CE 2023)
Worked primarily on overall system architecture, wiring, and physical connections.
Incoming Ph.D. Student at Carnegie Mellon University in Fall 2023.

Jonathan Ye (ME 2023)
Worked primarily on creating the linear PI controller model on Simulink.
Incoming Medtech Engineering Development Program Engineer at Johnson & Johnson in Fall 2023.

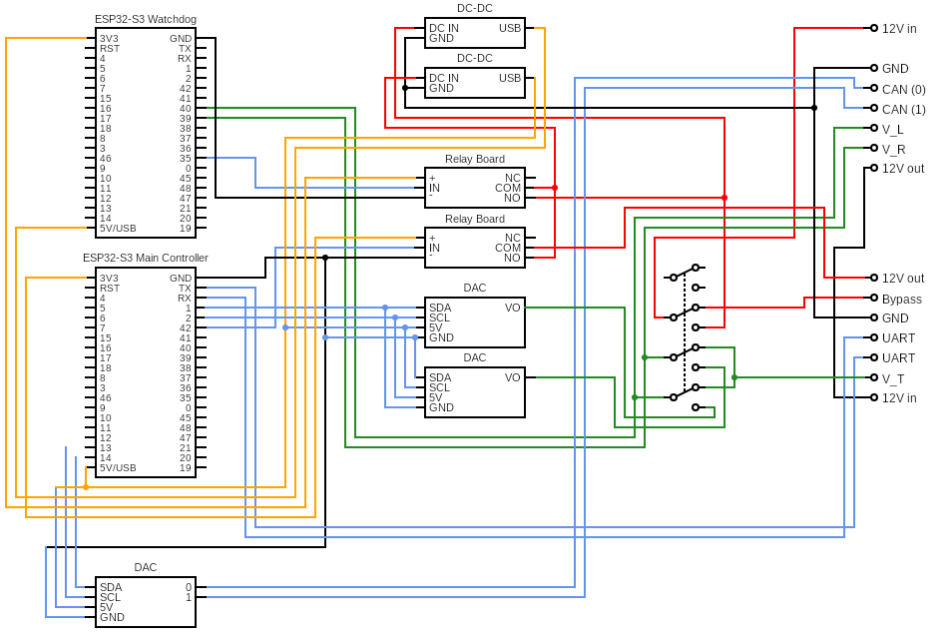
Alex Zhou (CE 2023)
Worked primarily on integrating communication protocols among the entire system.
Incoming Software Engineer at JPMorgan Chase in Fall 2023.

Giacomo Coraluppi (EE 2023)
Worked primarily on sensor/microcontroller interface and linear model integration.

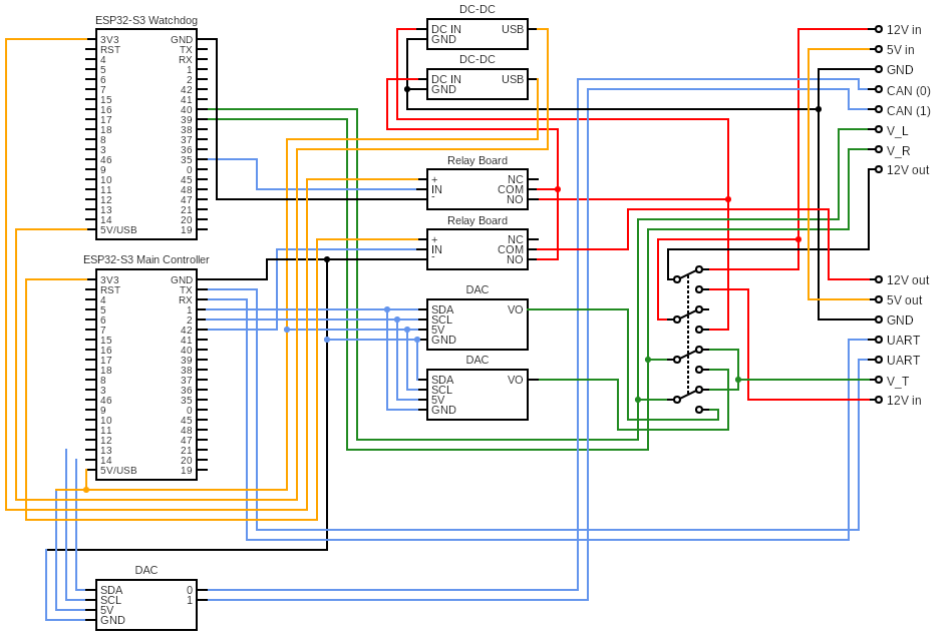
Nick Marchuk (ME 2023)
Worked primarily on the design of the electronic housing for the system.

7.3 Appendix C – Circuit Diagrams

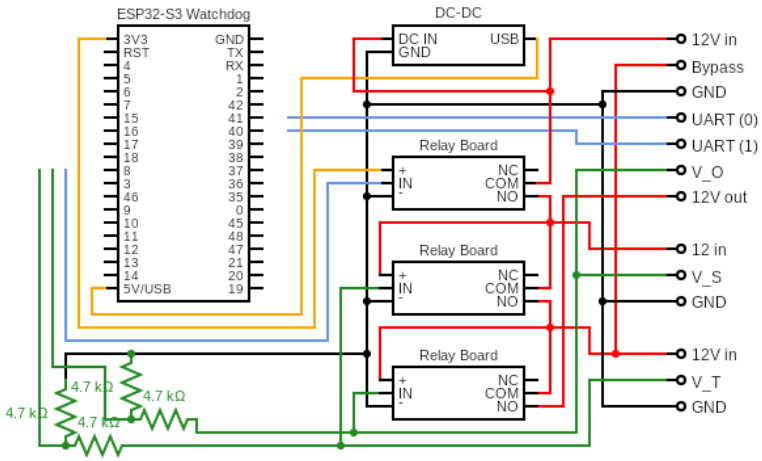
7.3.1 Main Controller, 12V throttle configuration



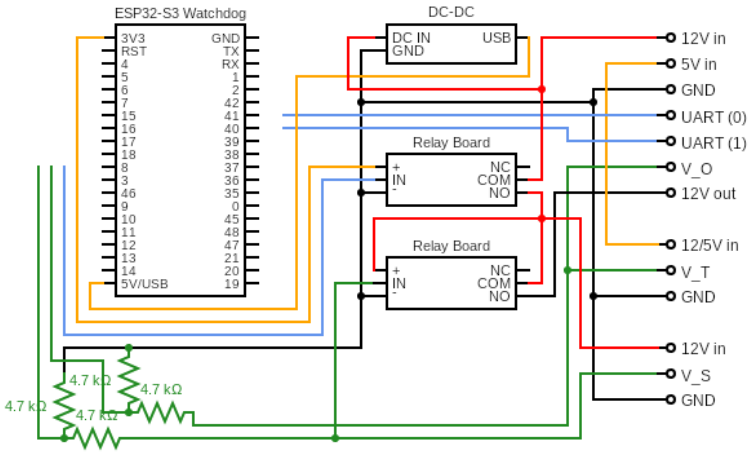
7.3.2 Main Controller, 5V throttle configuration



7.3.3 Auxiliary Controller, 12V Throttle Configuration



7.3.4 Auxiliary Controller, 5V Throttle Configuration



7.4 Appendix D - Molex Connector Pinout

7.4.1 Main Controller ⇔ Vehicle

6 12V supply	5 GND	4	3	2 CAN (Hi)	1 V_L
12 SSC out	11 5V supply	10	9	8 CAN (Lo)	7 V_R

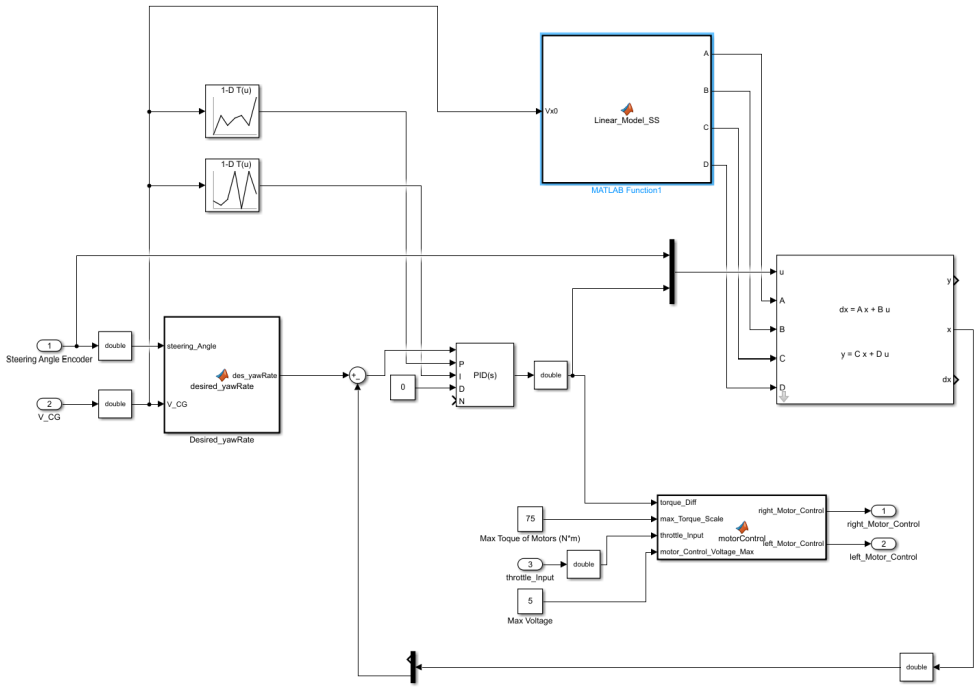
7.4.2 Main Controller ⇔ [Aux Controller] (same pinout both ends)

6 12V supply [SSC in]	5 GND	4	3	2 UART TX [RX]	1 V_T
12	11	10	9	8	7

SSC in [SSC out]	5V supply			UART RX [TX]	

7.5 Appendix E - Simulink Model/ C code

7.5.1 Simulink Model



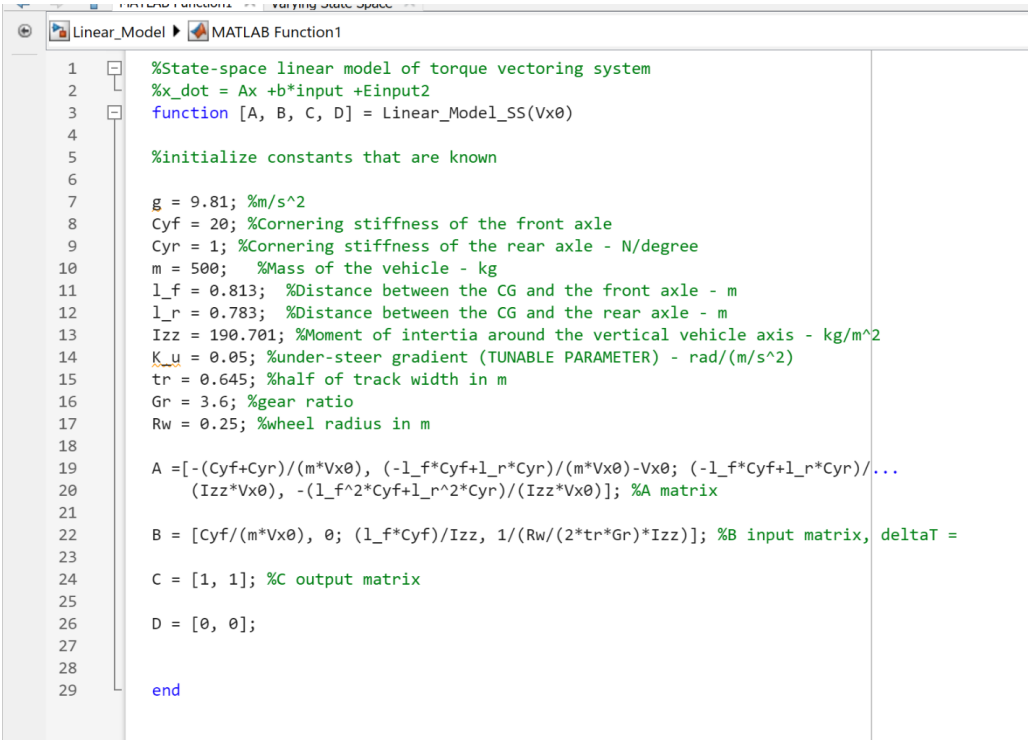
7.5.2 Desired Yaw Rate

Desired_yawRate | Varying State Space

Linear_Model ▸ Desired_yawRate

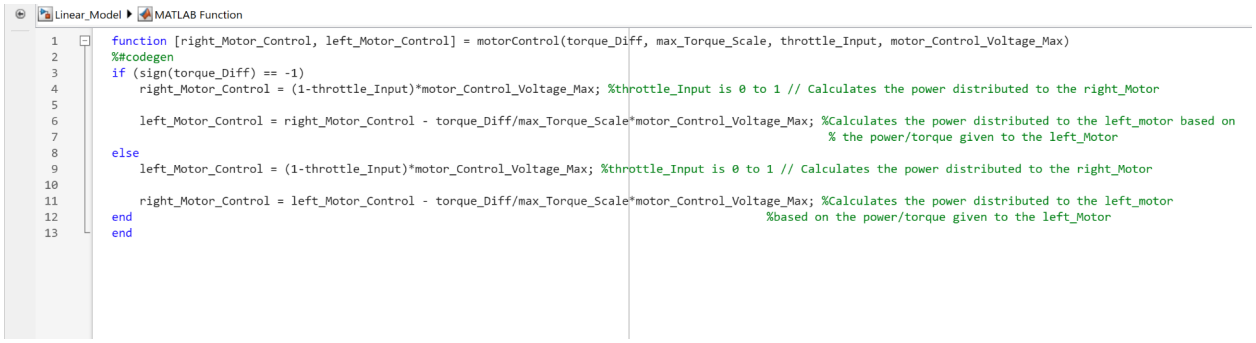
```
function des_yawRate = desired_yawRate(steering_Angle, V_CG)
%initialize constants that are known

g = 9.81; %m/s^2
Cyf = 20; %Cornering stiffness of the front axle
Cyr = 1; %Cornering stiffness of the rear axle - N/degree
m = 500;   %Mass of the vehicle - kg
l_f = 0.813;  %Distance between the CG and the front axle - m
l_r = 0.783;  %Distance between the CG and the rear axle - m
Izz = 190.701; %Moment of intertia around the vertical vehicle axis - kg/m^2
K_u = 0.05; %under-steer gradient (TUNABLE PARAMETER) - rad/(m/s^2)

    tire_road_coeff = 1; %dependant on road
    sig = 1; %tunable scalar

    yawRate_max = sig*tire_road_coeff*g/V_CG;
    yawRate = V_CG/((l_r+l_f)+K_u+V_CG^2) * steering_Angle;

if (yawRate <= yawRate_max)
    des_yawRate = yawRate;
else
    des_yawRate = yawRate_max;
end
```

7.5.3 Linear State Space Model matrices



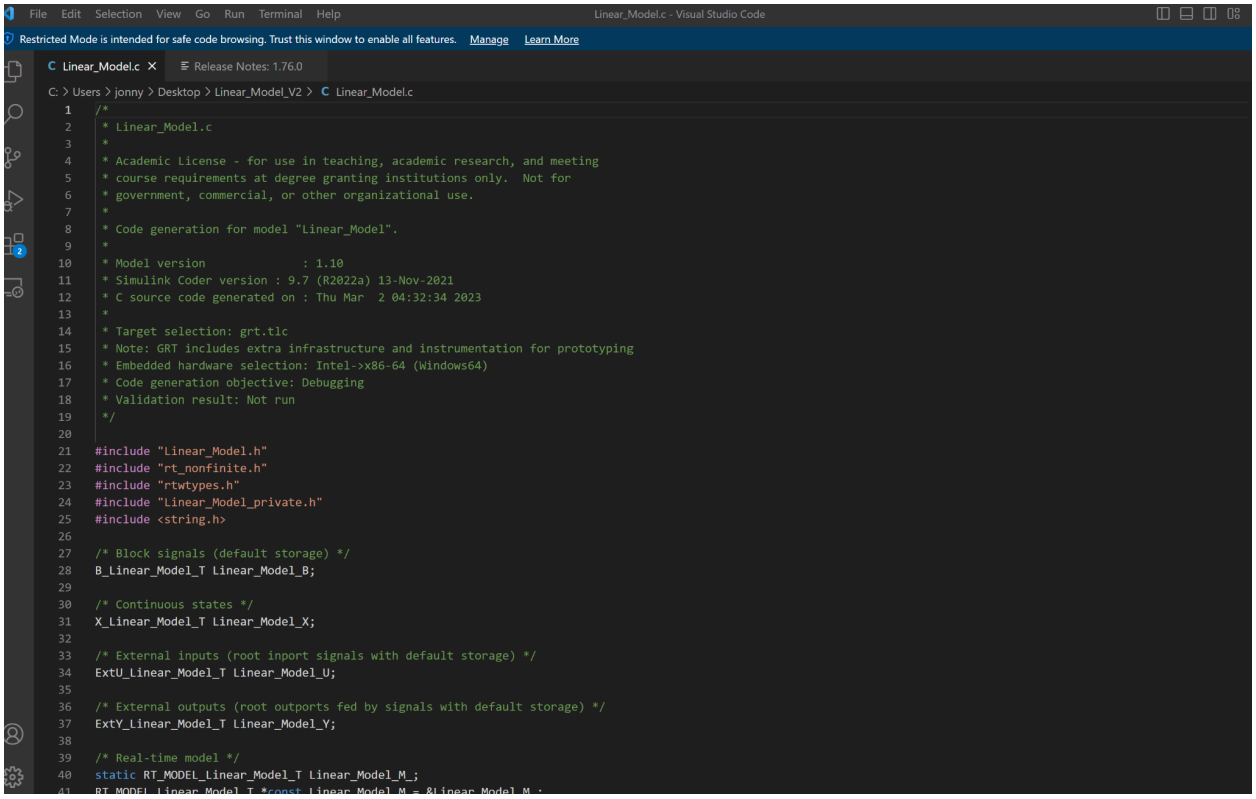

```
%State-space linear model of torque vectoring system
%x_dot = Ax +b*input +Einput2
function [A, B, C, D] = Linear_Model_SS(Vx0)

%initialize constants that are known

g = 9.81; %m/s^2
Cyf = 20; %Cornering stiffness of the front axle
Cyr = 1; %Cornering stiffness of the rear axle - N/degree
m = 500;   %Mass of the vehicle - kg
l_f = 0.813;  %Distance between the CG and the front axle - m
l_r = 0.783;  %Distance between the CG and the rear axle - m
Izz = 190.701; %Moment of intertia around the vertical vehicle axis - kg/m^2
K_u = 0.05; %under-steer gradient (TUNABLE PARAMETER) - rad/(m/s^2)
tr = 0.645; %half of track width in m
Gr = 3.6; %gear ratio
Rw = 0.25; %wheel radius in m

A =[-(Cyf+Cyr)/(m*Vx0), (-l_f*Cyf+l_r*Cyr)/(m*Vx0)-Vx0; (-l_f*Cyf+l_r*Cyr)/...
    (Izz*Vx0), -(l_f^2*Cyf+l_r^2*Cyr)/(Izz*Vx0)]; %A matrix

B = [Cyf/(m*Vx0), 0; (l_f*Cyf)/Izz, 1/(Rw/(2*tr*Gr)*Izz)]; %B input matrix, deltaT =

C = [1, 1]; %C output matrix

D = [0, 0];


end
```

7.5.4 Motor Control Function




```
function [right_Motor_Control, left_Motor_Control] = motorControl(torque_Diff, max_Torque_Scale, throttle_Input, motor_Control_Voltage_Max)
%#codegen
if (sign(torque_Diff) == -1)
    right_Motor_Control = (1-throttle_Input)*motor_Control_Voltage_Max; %throttle_Input is 0 to 1 // Calculates the power distributed to the right_Motor

    left_Motor_Control = right_Motor_Control - torque_Diff/max_Torque_Scale*motor_Control_Voltage_Max; %Calculates the power distributed to the left_motor based on
                                                                                                       % the power/torque given to the left_Motor
else
    left_Motor_Control = (1-throttle_Input)*motor_Control_Voltage_Max; %throttle_Input is 0 to 1 // Calculates the power distributed to the right_Motor

    right_Motor_Control = left_Motor_Control - torque_Diff/max_Torque_Scale*motor_Control_Voltage_Max; %Calculates the power distributed to the left_motor
end                                                                                                    %based on the power/torque given to the left_Motor
end
```

7.5.5 Generated C code


File Edit Selection View Go Run Terminal Help
Linear_Model.c - Visual Studio Code
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
Linear_Model.c
Release Notes: 1.76.0
C: > Users > jonny > Desktop > Linear_Model_V2 > Linear_Model.c
/*
* Linear_Model.c
*
* Academic License - for use in teaching, academic research, and meeting
* course requirements at degree granting institutions only. Not for
* government, commercial, or other organizational use.
*
* Code generation for model "Linear_Model".
*
* Model version : 1.10
* Simulink Coder version : 9.7 (R2022a) 13-Nov-2021
* C source code generated on : Thu Mar 2 04:32:34 2023
*
* Target selection: grt.tlc
* Note: GRT includes extra infrastructure and instrumentation for prototyping
* Embedded hardware selection: Intel->x86-64 (Windows64)
* Code generation objective: Debugging
* Validation result: Not run
*/
#include "Linear_Model.h"
#include "rt_nonfinite.h"
#include "rtwtypes.h"
#include "Linear_Model_private.h"
#include <string.h>
/* Block signals (default storage) */
B_Linear_Model_T Linear_Model_B;
/* Continuous states */
X_Linear_Model_T Linear_Model_X;
/* External inputs (root inport signals with default storage) */
ExtU_Linear_Model_T Linear_Model_U;
/* External outputs (root outports fed by signals with default storage) */
ExtY_Linear_Model_T Linear_Model_Y;
/* Real-time model */
static RT_MODEL_Linear_Model_T Linear_Model_M_;