Distributed Embedded Systems: RC Car Final Report

Lexi Batrachenko (abatrach), Shenai Chan (swchan),
Devan Grover (dsgover), Sanjana Shriram (sshriram),

May 2, 2024

1 Introduction

The goal of this project is to design the electronics and embedded software for a drive-by-wire system that is able to take inputs from an automotive cockpit and transmit them to table-top steering and drivetrain assembly. The linkage from the cockpit to the vehicle body needs to be bi-directional such that: (1) when the steering wheel moves, the wheels on the vehicle correspondingly move, and (2) if force is detected on the vehicle wheels then that force is applied as haptic feedback on the steering wheel. Throttle and brake are unidirectional in that the pedals simply control the drive motors. The throttle should set a velocity proportional to pedal position that is maintained using encoder feedback.

2 Requirement Design Document

We outline the requirements of the system, specifically looking at 5 key Use Case scenarios that will ensure core functionality. These are as follows:

1. **S1: Throttle Control**
   - Motor Velocity is set proportional to the Throttle Pedal angle when brake not pressed.
2. **S2: Brake Control**
   - Motor PWM halted and H-Bridge shorted when Brake depressed.
   - Brake takes priority over Throttle under all circumstances.
3. **S3: Steering Wheel**
   - The steering actuator needs to reflect the angle of the cockpit steering wheel input.
   - The force on the steering actuator should impart a proportional force to the steering wheel for user feedback.
4. **S4: Turn Signal and Hazard Blinkers**
   - When the right blinker button is pressed, the front and rear right blinkers should blink and the left blinker should stop.
   - When the left blinker button is pressed, the front and rear left blinkers should blink and the right blinker should stop.
   - When the right blinkers are blinking and the steering wheel passes a right turn threshold and then the wheel returns past the right turn threshold, the blinkers stop.
   - When the left blinkers are blinking and the steering wheel passes a left turn threshold and then the wheel returns past the left turn threshold, the blinkers stop.
   - In case of an error, all blinkers should rapidly flash indicating a hazard.
5. **S5: Self Test**
   - Each zone will have a self-test button that disables that zone. A single press causes the zone to fail, while a double press restarts the zone.
   - The fail safe state of the system will be to disable the Motor PWM output, engage the H-Bridge brake, and triggers a system-wide error that enables the hazard signal.

2.1 State Charts and Timing Constraints

State charts, also known as state diagrams or state machines, are graphical representations used in software engineering to model the behavior of complex systems and provide a visual of the states that an object can be in and the transitions between the states in response to events. These are some of the key elements of state charts:

1. States: Represent the different conditions or modes that an object or system can be in at any given time. States are typically depicted as rounded rectangles with descriptive labels.

2. Transitions: Arrows connecting states, indicating the possible transitions between them in response to events or conditions. Transitions are triggered by events and may be accompanied by actions or guards specifying conditions for transition execution.

3. Events: External stimuli or occurrences that trigger state transitions. Events can be user inputs, system signals, or changes in external conditions.

4. Actions: Tasks or operations that are performed when a state transition occurs. Actions can include computations, updates to internal variables, or interactions with other parts of the system.

Refer to Figure 1 for the state charts.

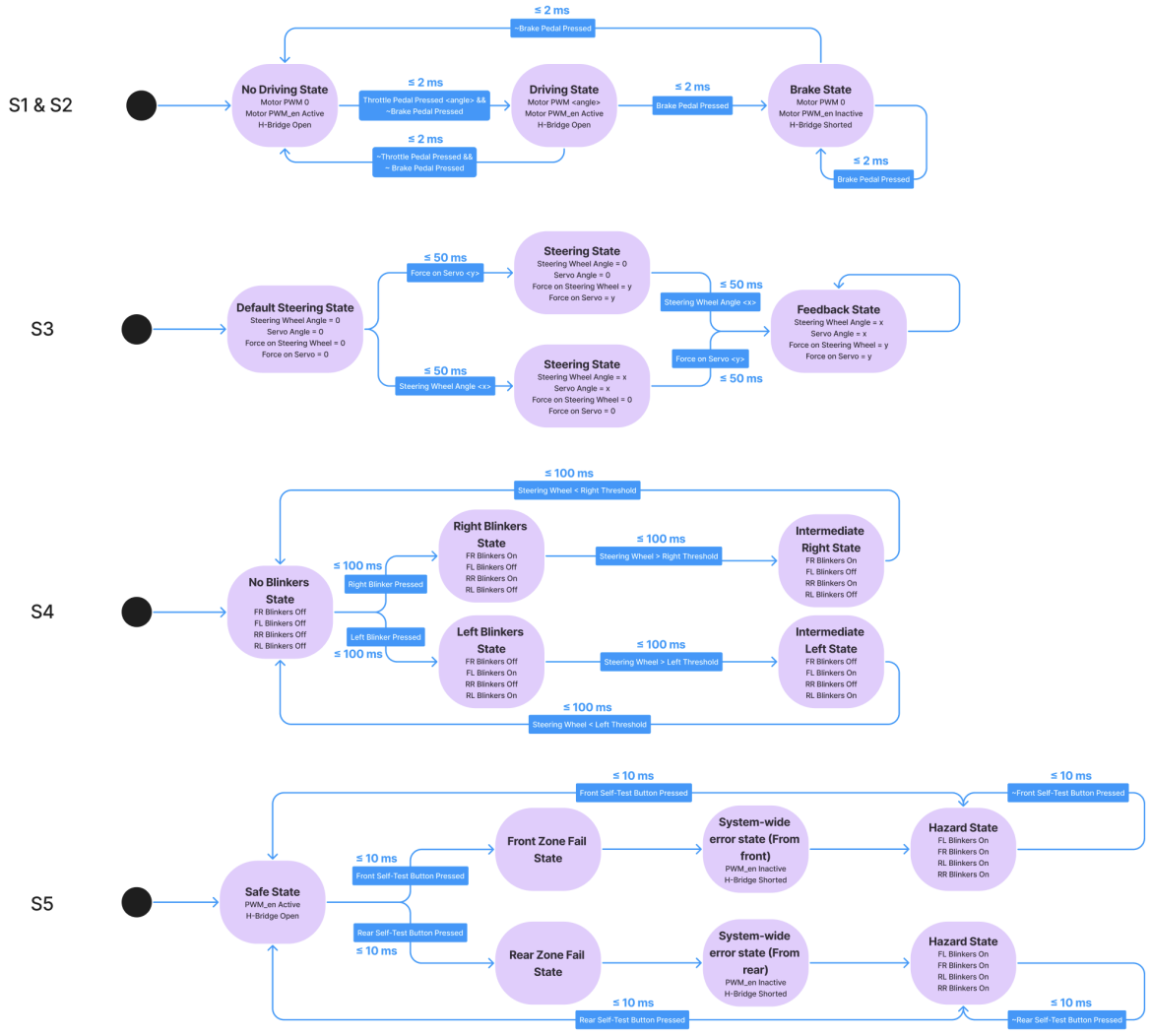

Figure 1: State charts

2.2 Sequence Diagrams

Sequence diagrams are tools in software engineering that visually represent the interactions between various components or objects within a system. They provide a clear and concise overview of how different parts of a system communicate with each other over time, depicting the flow of messages and the sequence of events. Here are a few key elements of Sequence Diagrams:

1. Objects: Represent the entities or components involved in the interaction. Each object is depicted as a box with its name on the top.

2. Messages: Arrows indicating the flow of communication between objects.

In this project, we use sequence diagrams to illustrate the interactions between different subsystems, components, and external entities. By creating sequence diagrams for key functionalities such as throttle, braking, steering, turn signals, and error buttons, we gain insight into how these components work together to achieve desired system behavior.

Our Sequence Diagrams are the following figures: 2, 4, and 3.

Figure 2: Sequence Diagram for Throttle Control (left) and Brake Control (right)

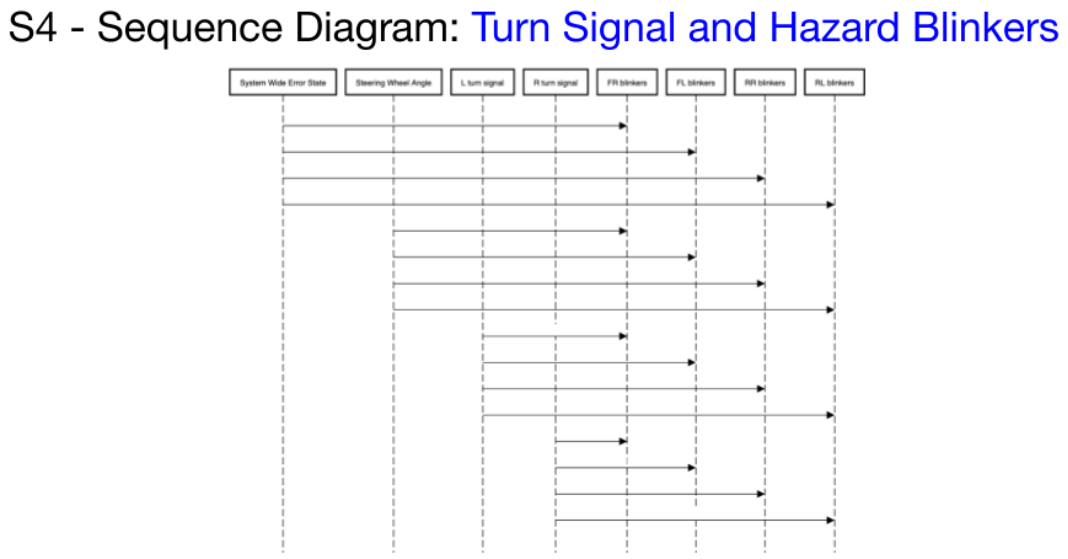

Figure 3: Sequence Diagram for Turn Signal and Hazard Blinkers

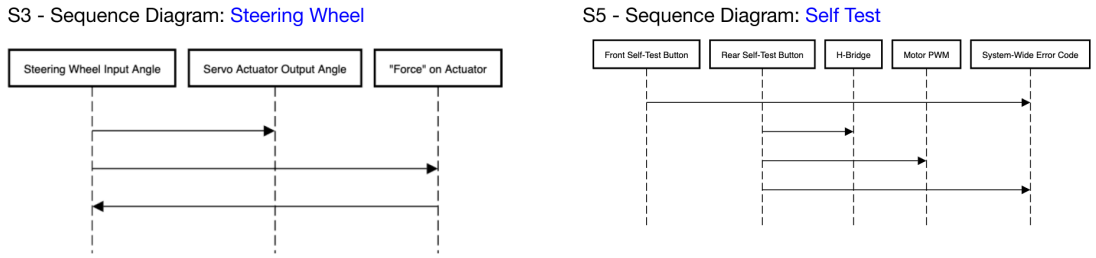

Figure 4: Sequence Diagram for Steering Wheel (left) and Self Test (right)

2.3 Traceability

Traceability tables are documentation artifacts that establish links between various system requirements, design elements, and implementation details throughout the development lifecycle. By maintaining traceability tables, we can track the evolution of system components, ensure compliance with regulatory standards, and enhance overall system reliability and maintainability. Our traceability tables are included in the report in Figures 1, 2, 3, and 4.

Use Cases	Test Requirements			
	R1. PWM_en enabled	R2. H Bridge shorts	R3. Motor velocity corresponds to accel pedal angle	R4. Motor PWM 0
U1. Presses on accelerator without brake	X		X	
U2. Presses on brake without accelerator		X		X
U3. Presses on neither	X		X	X
U4. Presses on brake and accelerator		X		X

Table 1: S1 and S2 Test Requirements

Use Cases	Test Requirements	
	R1. Servo rotates	R2. Force on cockpit steering wheel
U1. Turns steering wheel out of default position	X	
U2. Steering wheel set in default position		
U3. Force on steering actuator (current)		X
U4. No force on steering actuator		

Table 2: S3 Test Requirements

Use Cases	Test Requirements			
	R1. FR blinker blinks	R2. FL blinker blinks	R3. RR blinker blinks	R4. RL blinker blinks
U1. Right blinker button pressed	X		X	
U2. Left blinker button pressed		X		X
U3. When the right blinkers are blinking and the steering wheel passes a right turn threshold and then the wheel returns past the right turn threshold				
U4. When the left blinkers are blinking and the steering wheel passes a left turn threshold and then the wheel returns past the left turn threshold				

Table 3: S4 Test Requirements

Use Cases	Test Requirements			
	R1. System Wide error status	R2. Return to normal operation	R3. Blinking hazard lights	R4. PWM_en enabled, H Bridge open
U1. Front self-test button pressed once	X		X	
U2. Rear self-test button pressed once	X		X	
U3. Front self-test button pressed twice		X		X
U4. Rear self-test button pressed twice		X		X

Table 4: S5 Test Requirements

2.4 Proposed Timing Analysis

In order to meet strict timing constraints, we added test points to our custom PCB. We are able to set GPIO pins high and low on both the Raspberry Pi and our PCB. These will be sufficient along with an oscilliscope and modified code to test timing.

We foresee some potential bottlenecks in our system. Some of the most concerning ones are processing delays in the cockpit proxy that handles UDP messages. Network latency and congestion can impact response times. Execution delays in the motor controllers for adjusting throttle and brakes and those associated modes. There may also be communication delays between the cockpit proxy, front zone, and rear zone - as we will have different MCUs responsible for processing incoming commands and generating outputs, any processing delays can affect the system's ability to meet timing requirements. We could be limited by hardware: The motor, servo, and encoder have inherent response times and operational limitations, so those could become bottlenecks. Intermediate or error states may introduce additional overhead. And finally, integration is probably the biggest challenge and may prove to be a bottleneck due to a variety of issues that could arise.

To ensure that the system meets the performance requirements, we plan to conduct thorough testing to identify any potential failures to meet the requirements we have set out to meet. We also intend to implement fail-safe mechanisms to handle situations where requirements cannot be met to prevent hazards or unintended behavior. We are monitoring system behavior over time, using good style practices and version control.

3 Implementation Details

3.1 Hardware Implementation

3.1.1 PCB Design

Our two-layer PCB, TeSTla, was designed using Altium Designer. We organized the overall schematic into four sections: Interfaces, Front Zone, Rear Zone, and Power Step Down.

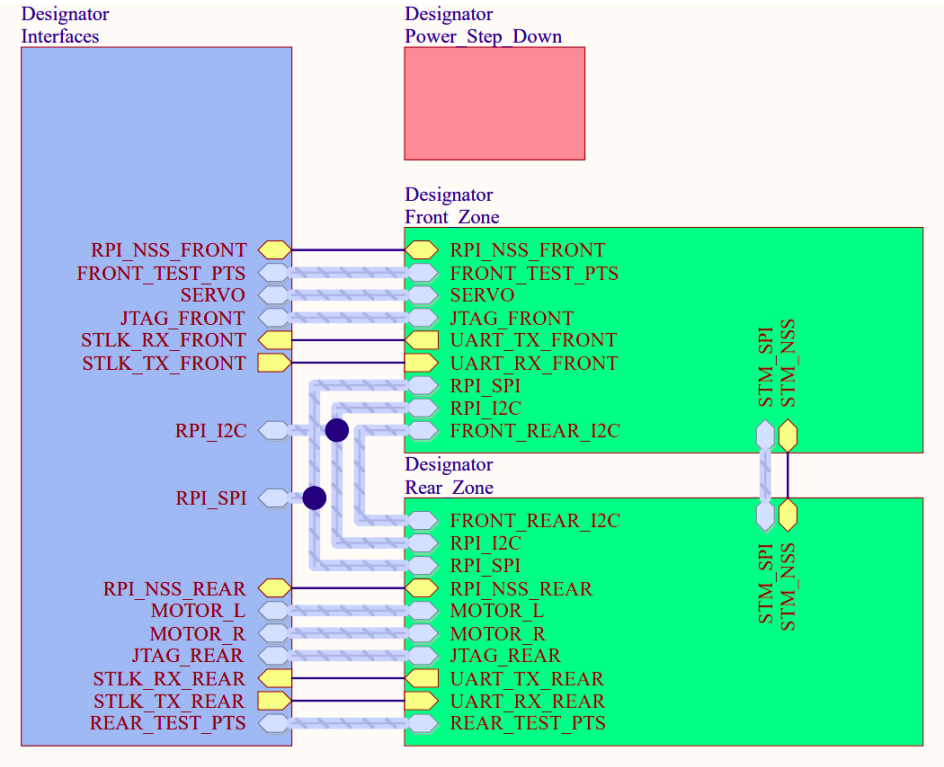

Figure 5: PCB Schematic Overview

Interfaces

This section includes connectors and pins that we require to interface our PCB with external devices.

- To provide power to our PCB, we used a 2-position pin header to take in 12 Volts. The power was provided by a battery pack that connected to these two pins.
- To program our STMs, we had a JTAG interface for each STM - this used a 6-position pin header to connect to the ST-Link.
- For communication, we had multiple headers for various protocols we were planning on possibly using. Our PCB is able to interface with external UART, I2C, and SPI interfaces. We set our PCB up to be used with multiple communcation protocols as a contingency plan, but we ended up only using the UART interfaces for communication.

- For the actuators on our car we used different headers. For the servo, we used a 3-position pin header to provide voltage, ground, and a PWM output. For the motors, we used two 6-position right angle JST headers for motor control and the encoder output.
- In order to perform timing analysis and testing, we also connected two test points to GPIO pins on each MCU on our PCB. These test points are set high on certain events based on the message we are attempting to test.

This supported our original vision of whole-system I2C communication, while still accounting for potential communication design pivots.

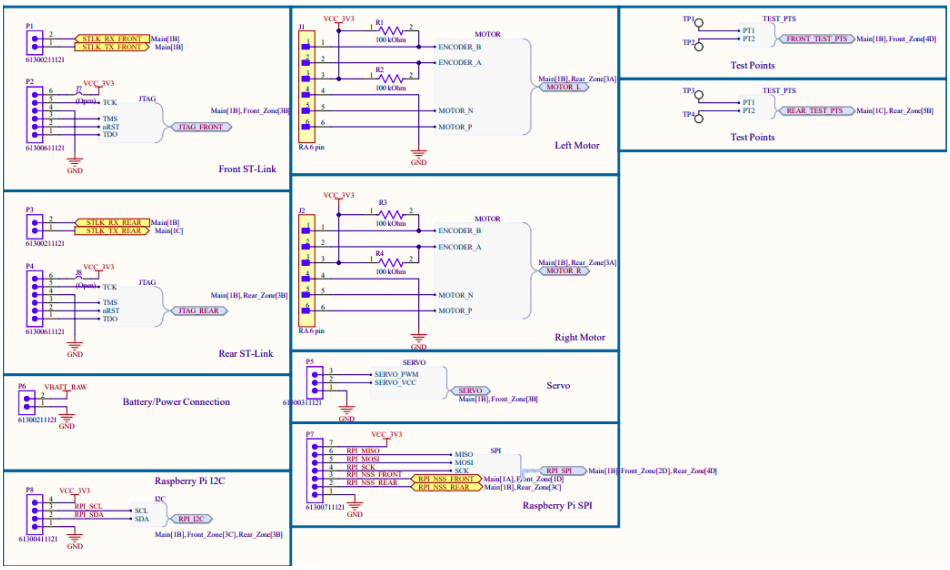

Figure 6: Interfaces Schematic

Power Step Down

Our system uses a 12 Volt battery pack with an additional 5V USB output. Our PCB takes in the 12V directly, whereas the 5V USB output powers the Raspberry Pi mounted on our PCB. We chose 12 Volts as our input voltage based on the voltage rating of the motors. Each voltage level on our PCB has an LED indicating that that power bus is alive. Furthermore, we added jumpers between our voltage levels to isolate each level and allow for iterative physical testing when bringing the board up.

- The 12 Volt input is primarily used to power the motors on our car.
- We used a buck converter to step the 12 Volt input down to 7.4 Volts to power the servo. Our servo takes in a voltage range of 6-7.4 Volts, which is why we decided to have a 7.4 Volt voltage level on our PCB. A buck converter was used for this step down rather than an LDO in order to account for the high current that the servo can draw under load (3.6A).
- Our 12 Volt input is also stepped down to 5 volts using a low dropout regulator. The 5V powers the H-Bridge motor driver and the Hall-Effect sensor.
- Lastly, our 5 Volt voltage level is stepped down to 3.3 Volts using another LDO. The 3.3V powers our STM microcontrollers and is also used as a reference voltage for pull-up resistors on our test buttons, I2C lines, and encoders.

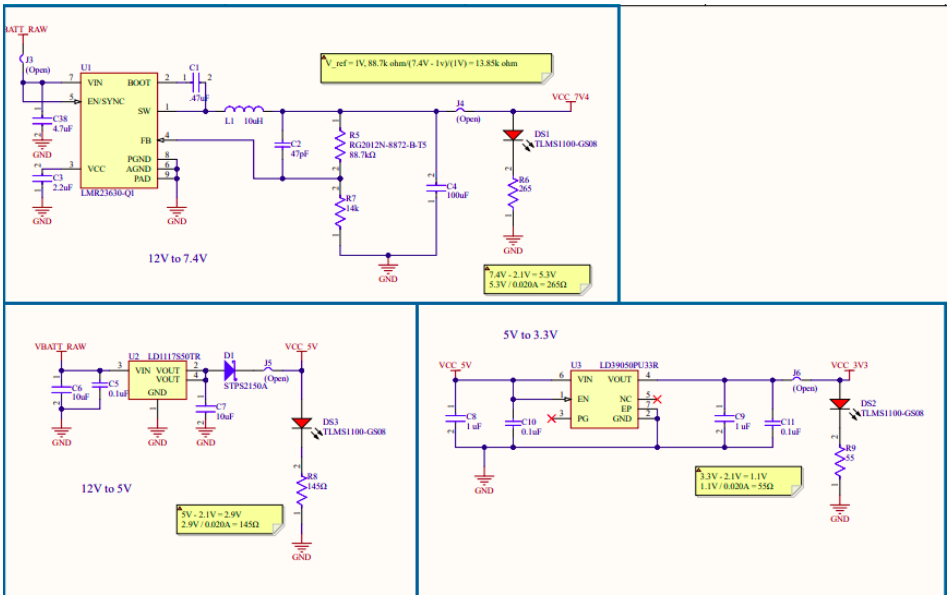

Figure 7: Power Schematic

Front Zone

This zone is governed by the front STM processor. Notably, the front microcontroller connects to the servo, the hall effect sensor, the front test button, the front blinker LEDs, two test points for timing analysis, and relevant communication protocol connector pins. A status LED was also included for debugging purposes.

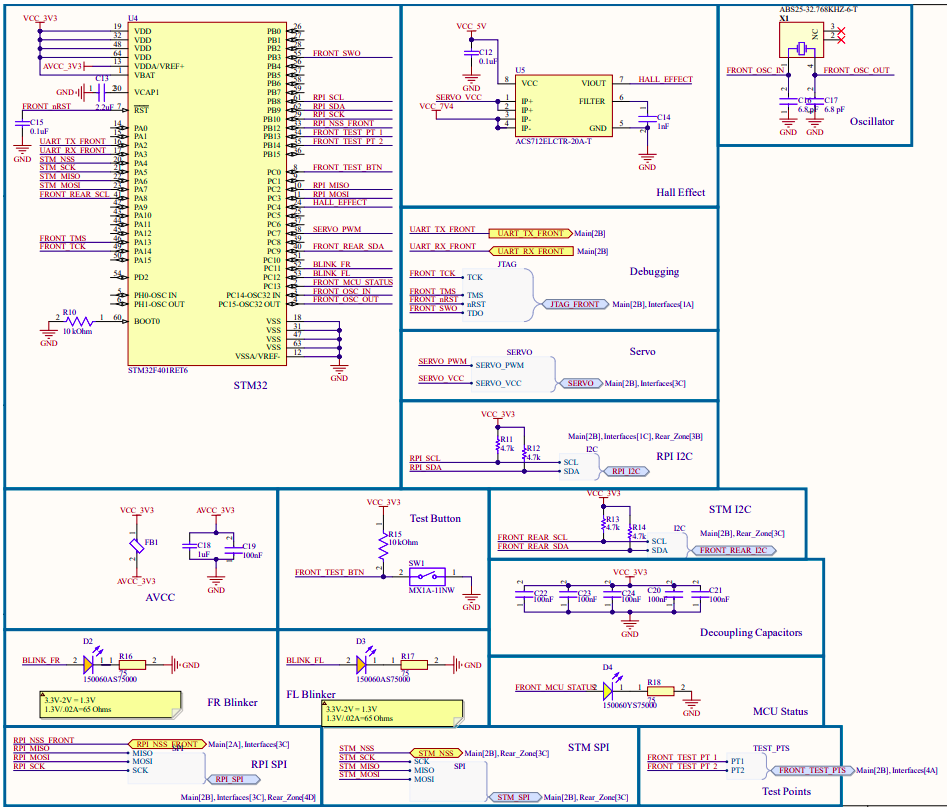

Figure 8: Front STM Schematic

Rear Zone

This zone is governed by the rear STM processor. The rear microcontroller is connected to the motor and motor driver, the motor encoder, the rear test button, the rear blinker LEDs, two test points for timing analysis, and relevant communication protocol connector pins. A status LED was included for debugging purposes.

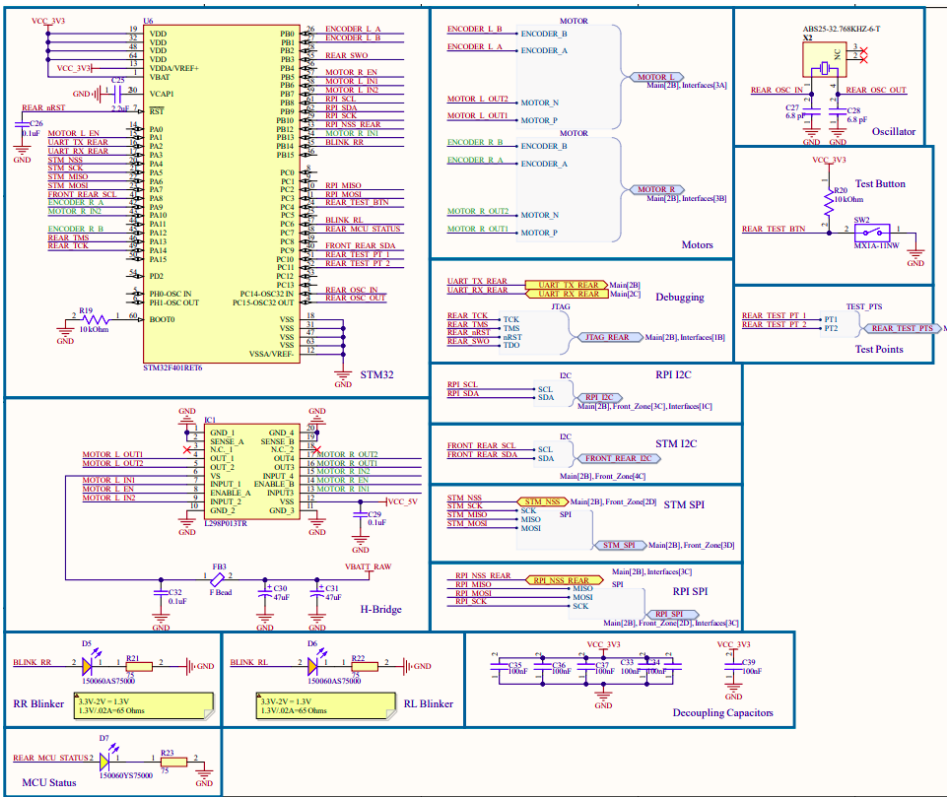

Figure 9: Rear STM Schematic

Our 2 layer PCB contains two polygon pours - the top layer contains a 3.3V pour and the bottom layer contains a ground pour. In order to mount the Raspberry Pi to our PCB, we added M2.5 mounting holes to the middle.

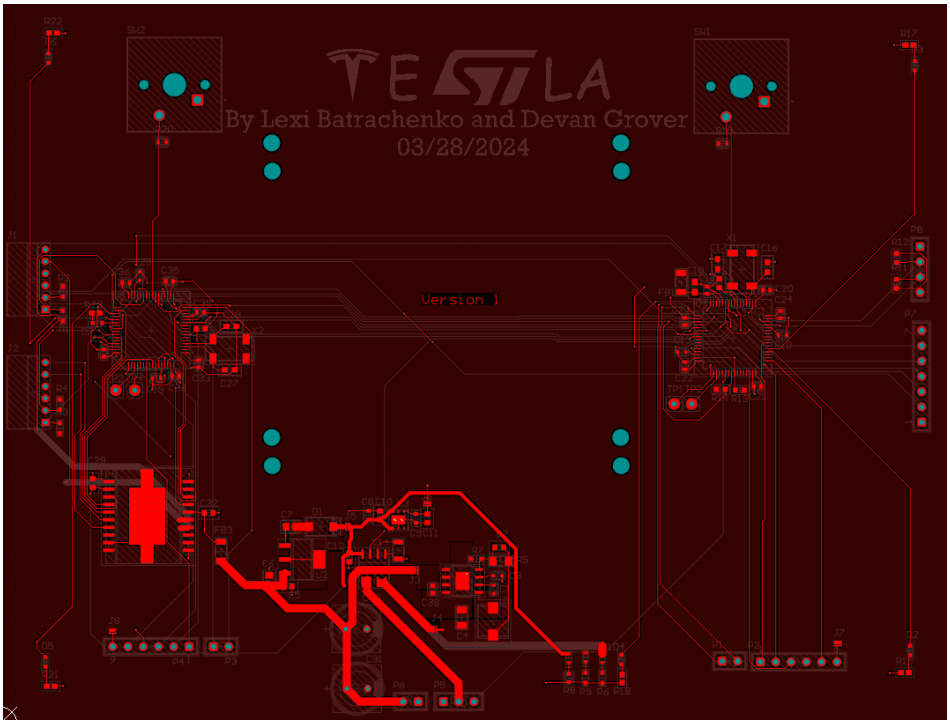

Figure 10: PCB 2D View

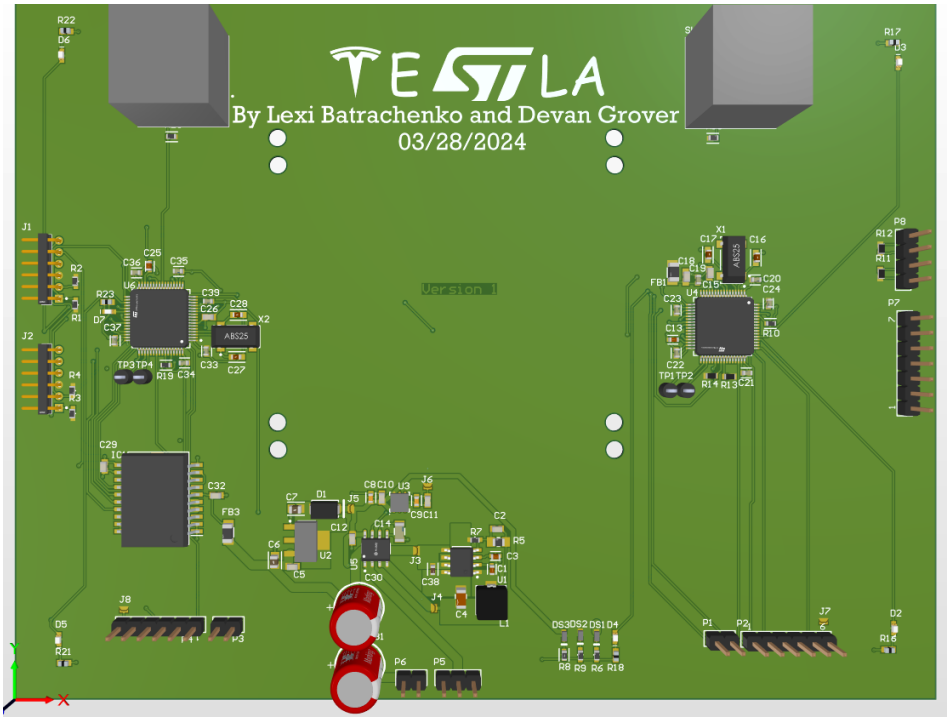

Figure 11: PCB 3D View

3.2 Software Implementation

3.2.1 Communication System Design

As mentioned in the PCB Design section, we originally intended to communicate from the RPI to the STMs via I2C. We chose I2C because we thought its affordances for bidirectional communication between a master (I2C) and multiple slaves (the STMs) would support our vision for the RPI sending the state of the cockpit, and the STMs responding with heartbeat messages (as well as the force on the servo from the front STM).

However, we encountered difficulties in executing this plan, most importantly the fact that we hadn't developed an interrupt-driven I2C driver for the STMs, much less the RPI, which would be necessary for receiving spontaneous messages from the RPI. We thus pivoted to two UART channels: one between each STM and the RPI.

We decided that the back and front STMs would not really need to communicate directly with each other. We debated sending heartbeat keep-alives between the STMs over I2C, but ultimately scrapped this idea, due to the aforementioned lack of an interrupt-based I2C driver. Instead, to synchronize performance across both processors, we sent status updates from the STMs, processed those within an FSM on the RPI, and then instigated cross-zone functionality with the RPI.

In conclusion, in addition to the UDP protocol driving the wireless communication between the Windows cockpit and the RPI, we are utilizing two UART channels: to communicate from the RPI to the front STM, and the RPI to the rear STM.

We also devised a bit-packing structure for all data sent as follows:

1. Data from the RPI to both STMs:
   We send 3 types of bytes. The first 2 bits are used to decode the remaining 6 bits of data. We refer to the 2 MSBs as the code bits.
   code: 00 = other 6 bits are for steering
   code: 01 = other 6 bits are for throttle
   code: 10 = other 6 bits are a status vector. From MSB to LSB, we have left blinker, right blinker, error/hazard state, brake, an unused bit, and reverse in the LSB. Except for error state, where a 1 signals a fail state and 0 signals normal operation, all of the bits are 1 when the corresponding logitech button is pressed, and 0 when it isn't.

   On the RPI side, we used a UART Termios library to handle sending and receiving from the STMs.

2. Data from Front STM to the RPI:
   We only sent one kind of byte from the STMs to the RPI: a status vector. Due to the nature of polling on the RPI side, we set the MSB of any valid data package sent from the STMs to "1". This way, ambient low voltage received from the STM by Termios would not register as valid UART packets.

   The second MSB on the front STM represented a front error state, with 1 signaling a front zone fail state, and 0 signaling normal operation. The third MSB encoded the feedback from the Hall Effect sensor, which measured current to determine whether or not we were receiving force on the steering servo. We thresholded this analog value, and sent a 1 when we were receiving significant force, and 0 when we were not. We unfortunately could not test this functionality as firewall issues prevented UDP packets from being sent back to the Windows cockpit, but this data was to be used to apply force feedback to the steering wheel. The remaining status vector bits were unused.

3. Data from Rear STM to the RPI:
   Similarly to the Front STM, we set 1 in the MSB when we were sending a valid data package, and a 1 in the second MSB when we were in a rear zone fail state.

3.2.2 Task Summary for Front STM

Our system currently operates on a cyclic-executive model, where we have one task that does each desired system function in succession. We originally had multiple tasks that performed various functions, with locks on the appropriate structs, but for the sake of simplicity, we created a task that uses the full time allotted by the scheduler.

This task performs the following functions:

1. Receives three bytes of data from RPI over UART
2. Sends one byte of data to RPI over UART which contains the status of the front STM (0 = no error, 1 = error) as well as the hall effect reading.
3. Activate the servo, mapping the 6-bit angle to an appropriate range to take the servo from -90 to 90 degrees.
4. Activate the blinkers in correspondence with the blinker state sent from the RPI.
5. Read from the hall effect sensor to see whether force is being applied on the servo.
6. Read from the fail state button to see whether the fail state is getting toggled.

3.2.3 Task Summary for Rear STM

The functionality performed by the rear STM is relatively similar to the front, in terms of the basic fetch-actuate cycle. The primary differences are that the rear controls the motors and braking.

The functionalities are as follows:

1. Receives three bytes of data from RPI over UART
2. Sends one byte of data to RPI over UART which contains the status of the front STM (0 = no error, 1 = error).
3. Activate the stepper motors, setting the PWM proportional to the throttle angle ONLY when the brake is not pressed.
4. Activate the blinkers in correspondence with the blinker state sent from the RPI.
5. Read from the fail state button to see whether the fail state is getting toggled.

3.2.4 Task Summary for Raspberry Pi

The Raspberry Pi sends data to both STMs with important system updates. It keeps track of several different pieces of incoming information from the STMs to make updates to the state that it then broadcasts to both STMs via UART.

Here is the information that the RPI receives from the Front and Rear STM and the cockpit and how that information is used within the RPI's code.

1. Front Status - from Front, used to set the hazard state which is then broadcast to the STMs
2. Rear Status - from Rear, used to set the hazard state which is then broadcast to the STMs
3. Hall Effect Status - from Front, theoretically used to put force back on the steering wheel, which we could not fully accomplish because of firewall issues.
4. Cockpit struct - from Cockpit, please refer to the following logitech wheel dev module for the breakdown of the struct. The only relevant pieces of information are the steering wheel angle, throttle angle, left/right paddle statuses, and B button status (for reverse).

Once it receives the appropriate data, most of the system state machine computation occurs in the RPI. Please refer to 1 for more details.

Then, once the RPI has performed appropriate state computation, it sends data to the STMs as described in 3.2.1.

4 Timing Analysis

We conducted a timing analysis for the following components with the following specifications.

1. **Throttle** Upon receiving a periodic UDP state update message from the cockpit proxy, the motor PWM must adjust to the throttle angle within 2ms.
2. **Brake** Upon receiving a periodic UDP state update message from the cockpit proxy, the motor must enter the brake mode of operation within 2ms.
3. **Steering** Upon receiving a periodic UDP state update message from the cockpit proxy, the steering output must update within 50ms.
4. **Blinker Synchronization** State changes of the front and rear blinkers must be synchronized to within 1ms.
5. **Turn Signal Duty Cycle** When active, the blinkers must blink at a 50 % duty-cycle at a rate between 0.9Hz and 1.1Hz.
6. **Turn Signal** Upon receiving a periodic UDP state update message from the cockpit proxy, the left (or right) blinker must start blinking within 100ms.

4.1 Methodology

We conducted all response time measurements externally using an oscilloscope. For subsystems that initiate on the Raspberry Pi (RPI), we implemented the following methodology: Upon reception of the UDP packet, we triggered a write operation to the GPIO pins of the RPI. Additionally, upon receiving the packet on the STM or actuating a command such as setting a motor or servo, we triggered a write operation to the Test Point GPIO. By simultaneously observing both GPIO signals on the oscilloscope, we were able to record the time difference between their respective high states.

In addition to timing subsystems end-to-end, we also provide a breakdown of the time the system spends UART polling. This was found by triggering a GPIO high right before the RPI sends data and right after it is received by the STM, then recording the difference.

We were unable to time smaller segments of the code present in the STM after UART polling - however we have accurate timing measurements by using the system time - UART polling time. Essentially, these two measurements are the same thing. To trigger a GPIO high signal in the STMs, we have to perform some computation first to determine if we are even in a case to trigger the test point GPIO.

The results of our extensive timing analysis along with the constraints placed upon the system and our results are presented in Table 6.

A more in depth analysis and breakdown of the two subsystems we focused on – brake and blinkers – is provided in Figure 12.

Subsystem (includes UART)	System Measurement	Constraint	Criteria Met
Throttle	315 µs	2 ms	✓
Brake	360 µs	2 ms	✓
Steering	210 µs	50 ms	✓
Blinker Synchronization	12.5 µs	1 ms	✓
Turn Signal Duty Cycle	1.006 Hz	0.9 Hz – 1.1 Hz	✓
Turn Signal	360 µs	100 ms	✓

Table 5: Summary of Timing Analysis

Subsystem	System Measurement
UART polling put byte (from RPI STM)	146 µs

Table 6: Timing Summary of UART portion

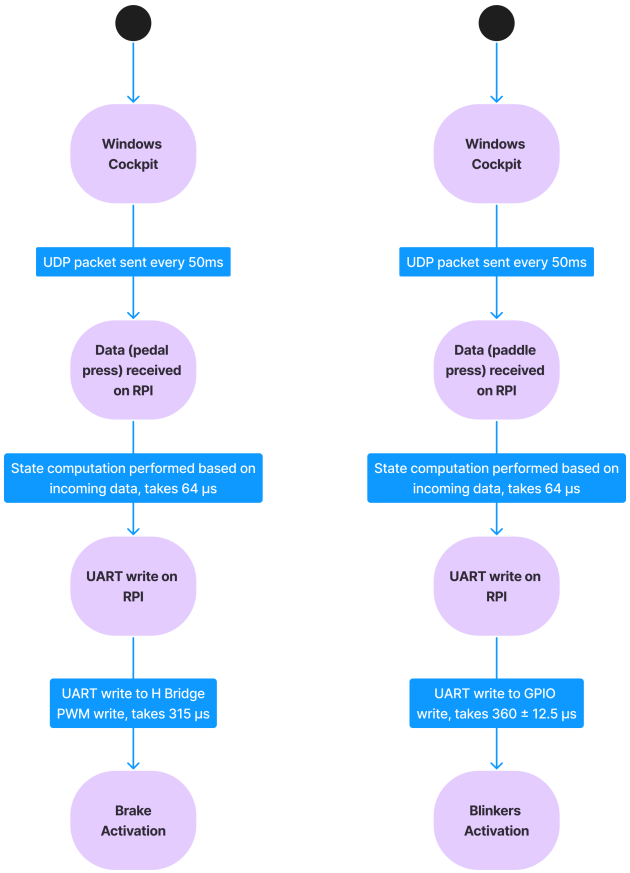

Figure 12: System Timing Breakdown: Brake System (left) and Blinker System (right)

There is a ±6.25 µs due to the recorded 12.5 µs difference in blinker activation between the front and rear STM.
This reasoning indicates that the computation time of the rear vs front STM differs by roughly 12.5 µs.

- Overall UART Polling Time: According to the right image of Figure 16, the best-case time spent for UART data sending from the RPI to STM is 146µs. This is the time spent waiting through one uart_polling_getbyte function (receiving a single byte). However, in Figures 13 and 14, the blinkers and throttle systems were measured with the worst-case UART condition: where it requires waiting through three uart polling getbyte functions on the STM before activating the necessary peripheral (blinker LEDs or motor PWM) - this accounts for the fact that the byte required to activate a peripheral on the STM may be the third out of the three total bytes that are received (consecutively) from the RPI. Intuitively, the best case in our STM code is when the necessary byte for activating a peripheral is received first out of the three.
- RPI Computation Time: According to Figure 16, CPU computation time in the RPI accounts for roughly 210µs - 146µs = 64µs of the time elapsed from the reception of the UDP packet to

activation of necessary peripherals by the STMs.
From the two points above, we can see that the time used to send and receive with UART is over twice the time used for CPU computation on the MCUs.

4.2 Worst Case Timing Discussion

Our software architecture makes it such that there are 3 UART bytes sent at once from the RPI for each UDP state update (the steering value, throttle value, and left—right—...reverse bitvector). The STMs on the other hand, read and process bytes one at a time using the codes (mentioned in section 3.2.1). Therefore, the worst case timing is when the third byte read by the STM is the one being processed. The worst case timing we were able to record for UART polling is 191 µs.

UART polling is the bottleneck of the system and consumes the most time out of all components in the system. Other paths of the system are not as perceptible to large variations in worst case timing.

4.3 Annotated Oscilloscope Measurements

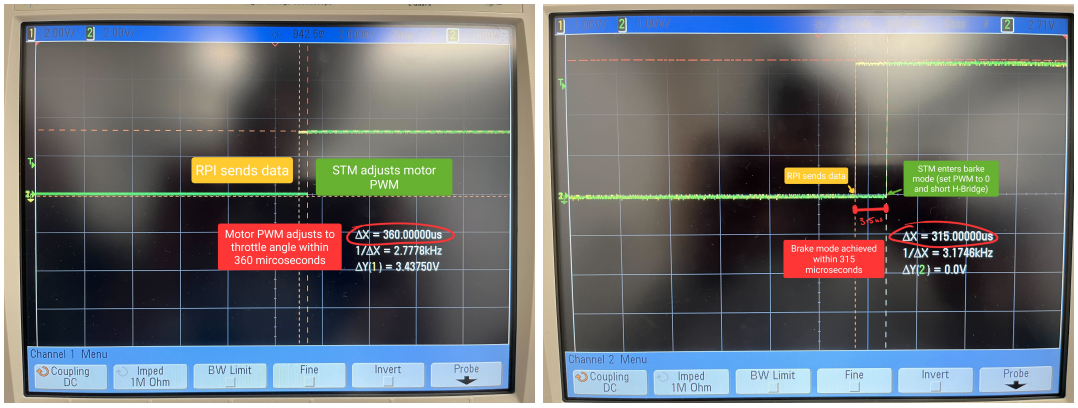

Figure 13: Measurement for Throttle (left) and Brake Activation (right)

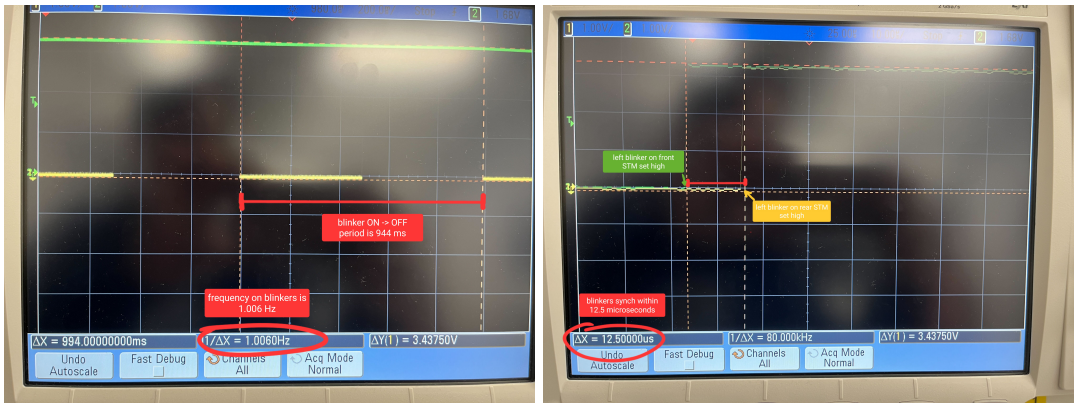

Figure 14: Measurement for Blinker Frequency (left) and Blinker Synchronization (right)

Figure 15: Measurement for Blinker Activation

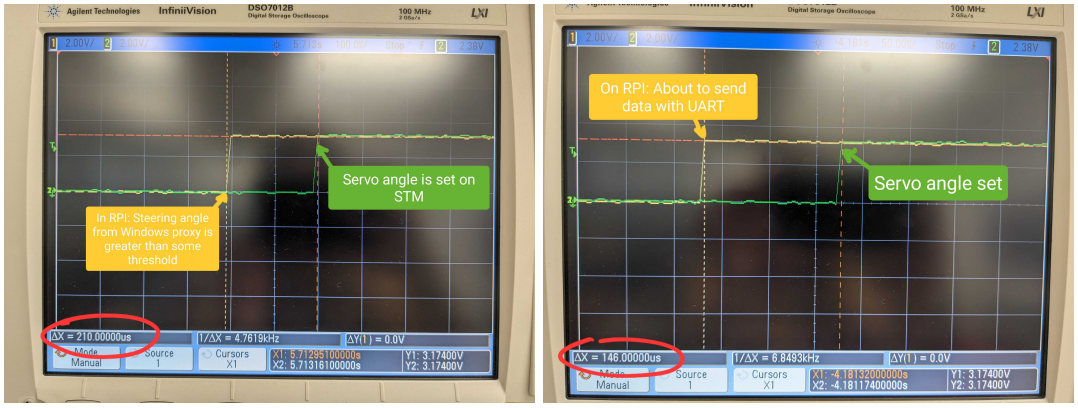

Figure 16: Time from Receiving Steering data on the RPI till Setting the Servo Angle on Front STM (left) and Time from Sending Data with UART on RPI till Setting the Servo Angle on Front STM (right)

4.4 Timing Challenges

Synchronizing the front and rear blinkers was our most significant timing challenge. As per the Lab 2 writeup, we connected the front left and right blinkers to the front STM and the rear left and right blinkers to the rear STM. We initially flashed each STM with a distinct, single-thread main file that determined the blinker state and triggered the LEDs to turn on or off in each loop iteration. The RPI also used two distinct UART TX lines/file descriptors to send bytes to each STM. This resulted in very noticeable asynchrony.

To solve this, we moved the blinker state and frequency logic to the RPI and only sent bits to indicate whether the front and rear STMs should write high or low to their left and right blinker LED pins. Less importantly, we implemented UART broadcast from the RPI to the STMs, tying the two STMs' RX lines to one RPI TX line (RPI writes to a single file descriptor).

We note that the criteria for Steering Feedback was not met. This is due to our use of CMU computers with a firewall that restricts messages from the RPI to the cockpit. We still wished for the steering to feel realistic, so we implemented force feedback on the steering wheel by directly sampling the wheel angle in proxy_gui.py. Since this feedback is implemented entirely on the Windows machine, there is no way to externally time this interaction.

5 Annotated Photographs of Car Hardware

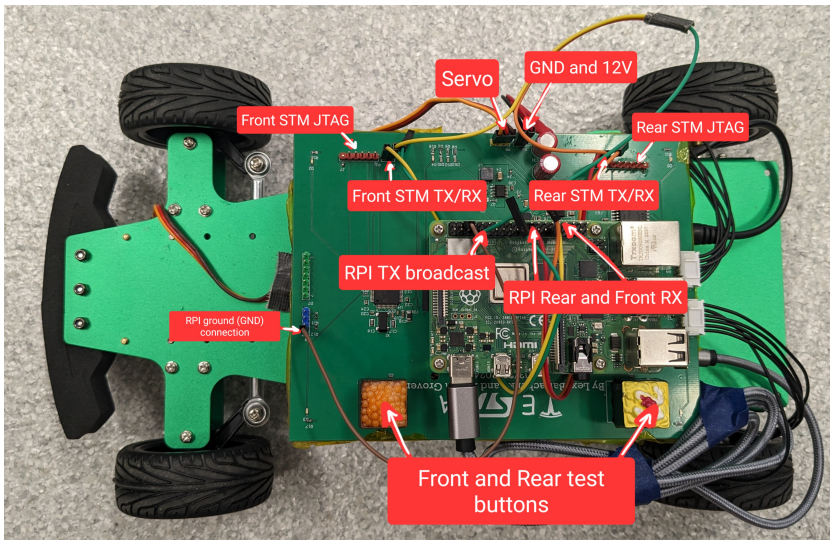

Figure 17: Car Top View

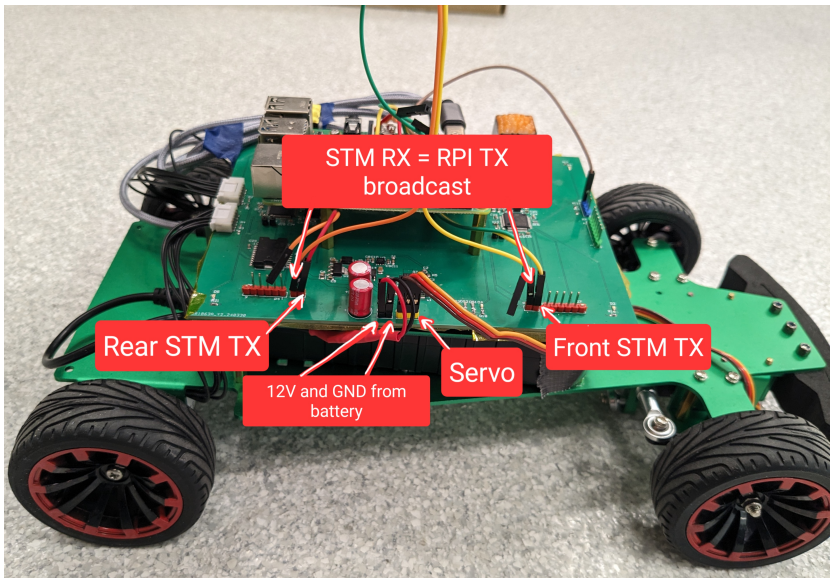

Figure 18: Car Side View

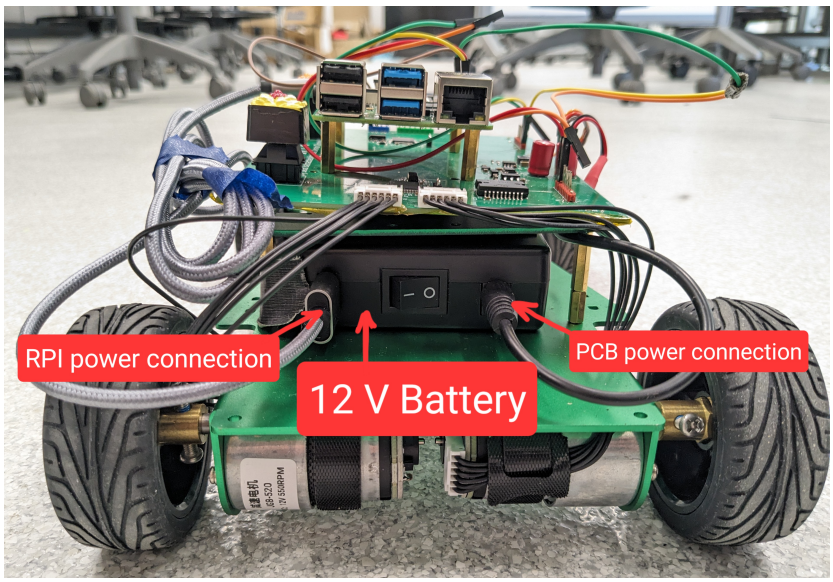

Figure 19: Car Back View