Distributed Embedded Systems Final Report

Taspon Gonggiatgul, Apurva Patel, Jiaqi Song, Simon Spivey

Table of Contents

1.) Introduction

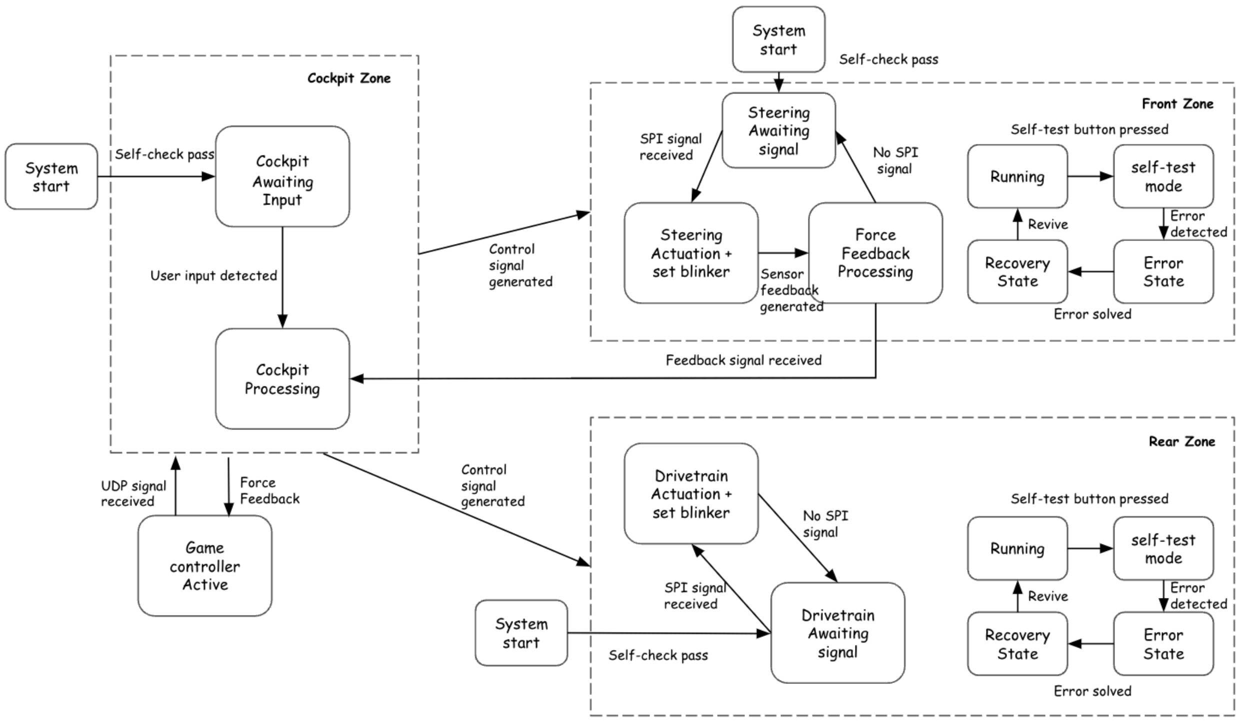
The advent of Drive-by-Wire technology marks a revolutionary shift in automotive design, pushing change in mechanical control systems to embrace the precision and adaptability of digital interconnectivity. In this project, we endeavor to engineer a Drive-by-Wire system which is both safety-critical and resource-constrained.

The drive-by-wire system outlined here is divided into three interconnected regions: the Cockpit Zone, the Front (Steering) Zone, and the Rear (Drivetrain) Zone. The system springs into action upon initialization with the Cockpit Zone poised to receive a UDP signal triggered by user input via a game controller interface. Upon receipt of the UDP signal, the Raspberry Pi (RPi) in the Cockpit Zone processes this signal and dispatches an SPI message to the Microcontroller Units (MCUs) located in both the Front and Rear Zones.

In the Rear Zone, the dedicated MCU processes the incoming SPI message to modulate the Pulse Width Modulation (PWM) of the motor and to control the turn signal blinkers. Concurrently, in the Front Zone, the MCU receives and processes the SPI signal to adjust the steering actuator's position. Additionally, it captures force feedback from the wheel’s interaction with the road and relays this sensory feedback back through the Cockpit Zone, and then back to the game controller.

Both the Front and Rear Zones are equipped with a test button that allows entry into a self-test mode—a critical feature for verifying the integrity of each zone independently. This test mode is designed to simulate failures and confirm the system's ability to transition to and recover from error states, ensuring robustness and safety in the drive-by-wire system's operation.

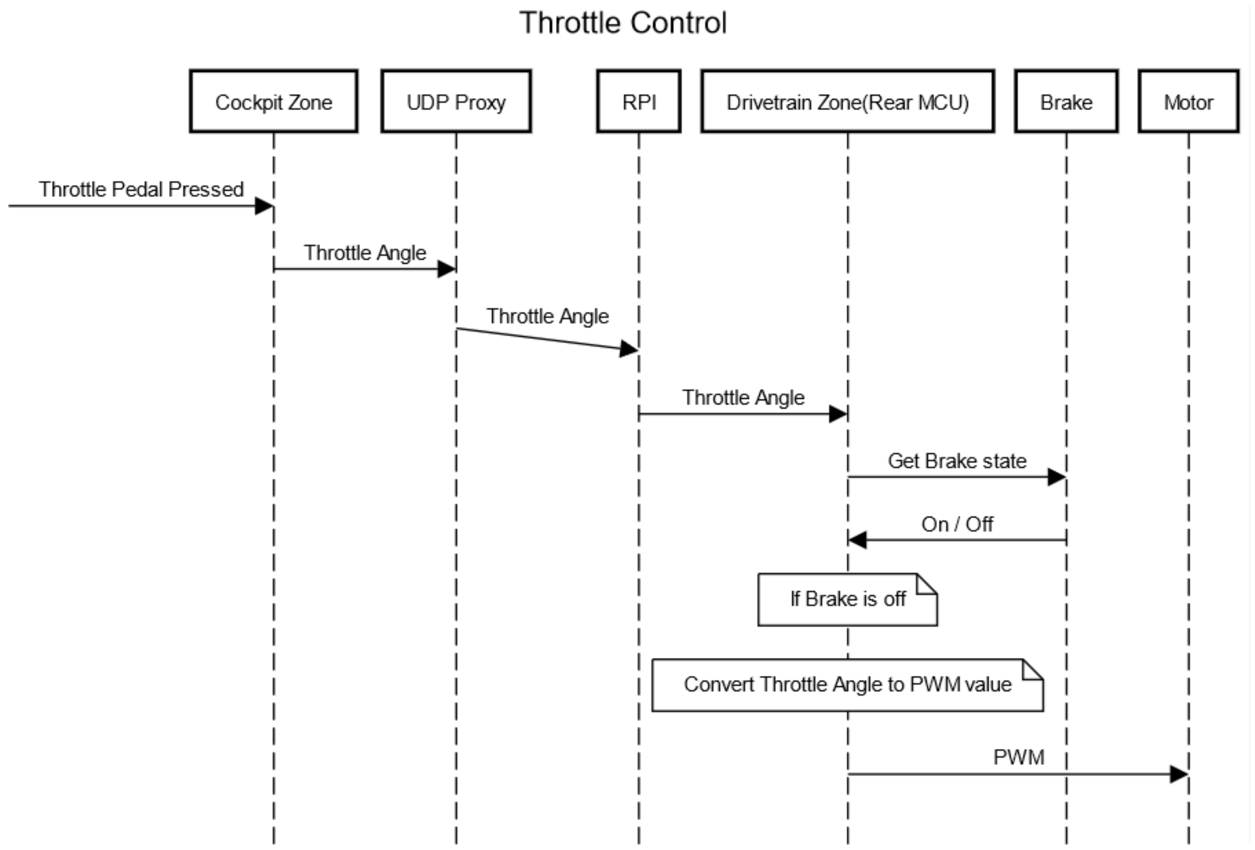
2.) System state chart



3.) Sequence diagrams

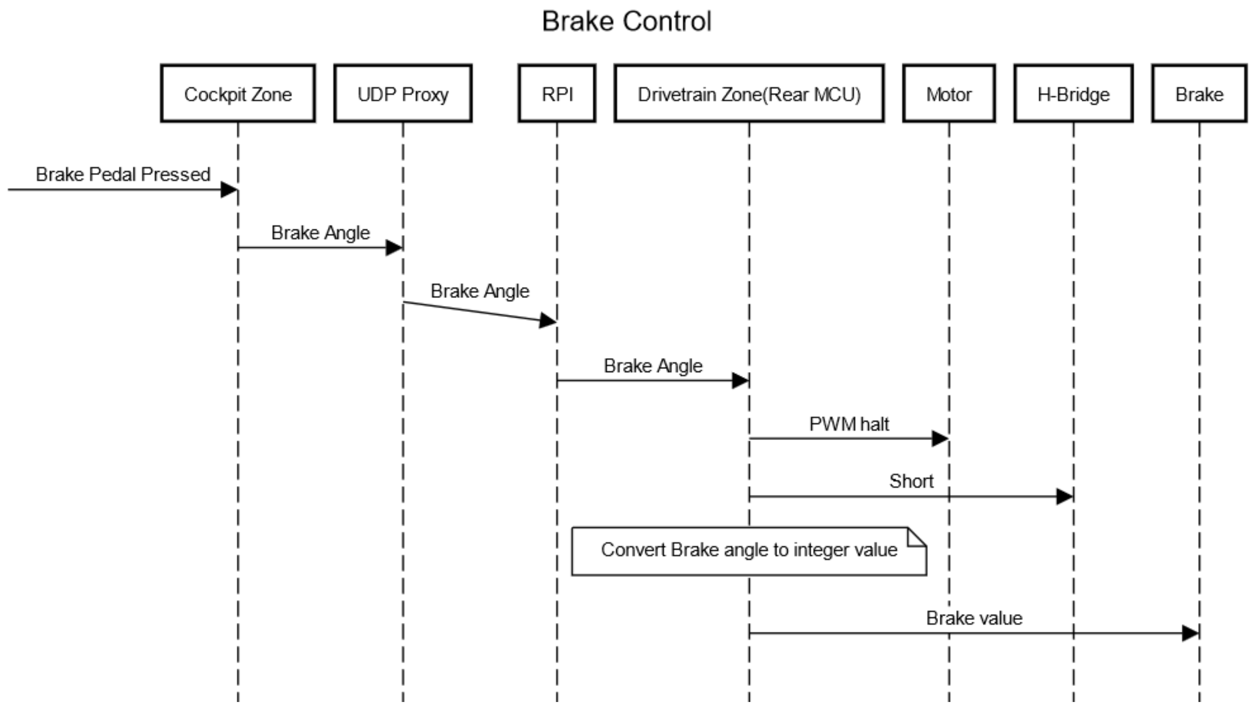
1. Throttle Control

```
Unset
title Throttle Control

[->Cockpit Zone:Throttle Pedal Pressed
Cockpit Zone->UDP Proxy: Throttle Angle
UDP Proxy->(1)RPI: Throttle Angle
RPI->Drivetrain Zone(Rear MCU): Throttle Angle
Drivetrain Zone(Rear MCU)->Brake: Get Brake state
Brake->Drivetrain Zone(Rear MCU):On / Off
note over Drivetrain Zone(Rear MCU): If Brake is off
note over Drivetrain Zone(Rear MCU):Convert Throttle Angle to PWM value
Drivetrain Zone(Rear MCU)->Motor:PWM
```



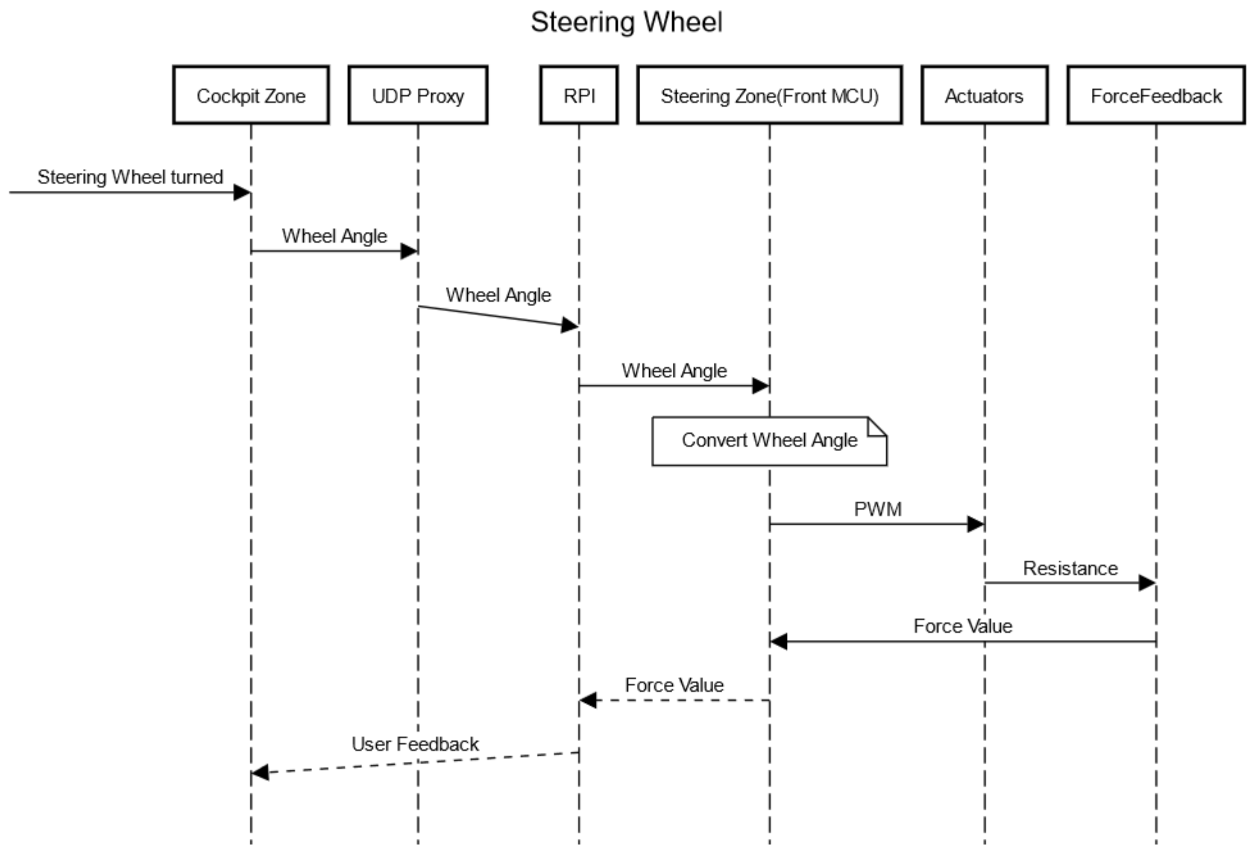
2. Brake Control

Unset
```
title Brake Control

[->Cockpit Zone:Brake Pedal Pressed
Cockpit Zone->UDP Proxy: Brake Angle
UDP Proxy->(1)RPI: Brake Angle
RPI->Drivetrain Zone(Rear MCU): Brake Angle
Drivetrain Zone(Rear MCU)->Motor: PWM halt
Drivetrain Zone(Rear MCU)->H-Bridge: Short
note over Drivetrain Zone(Rear MCU):Convert Brake angle to integer value
Drivetrain Zone(Rear MCU)->Brake: Brake value
```



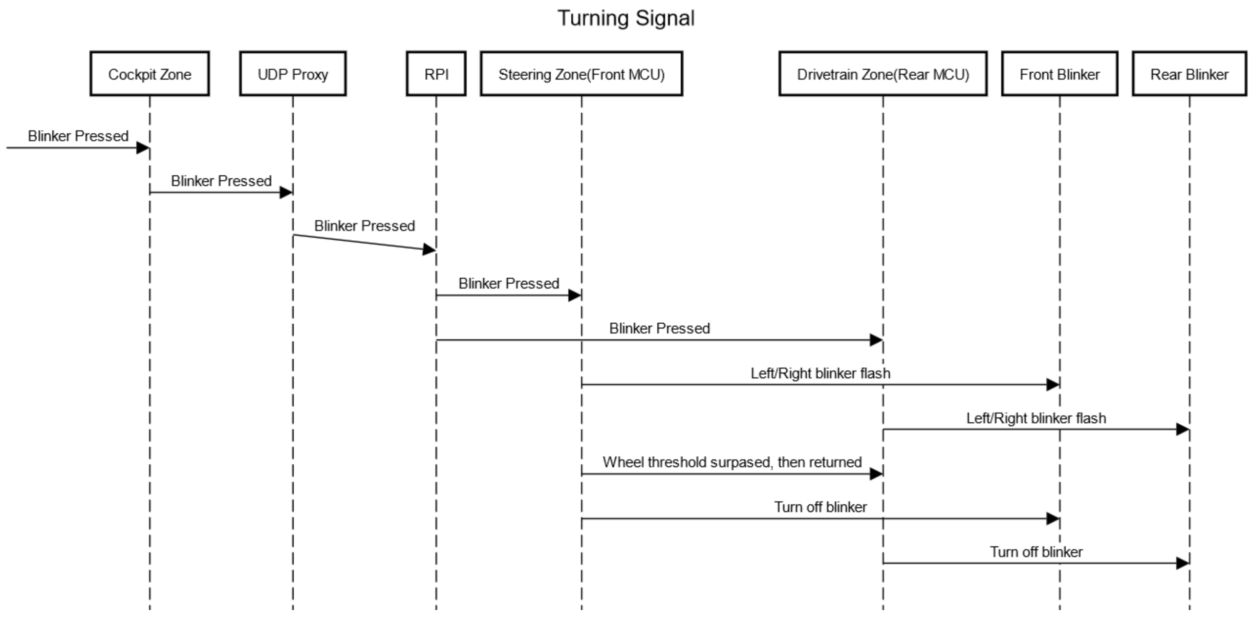
3. Steering Wheel

```
Unset
title Steering Wheel

[->Cockpit Zone:Steering Wheel turned
Cockpit Zone->UDP Proxy: Wheel Angle
UDP Proxy->(1)RPI: Wheel Angle
RPI->Steering Zone(Front MCU): Wheel Angle
note over Steering Zone(Front MCU):Convert Wheel Angle
Steering Zone(Front MCU)->Actuators: PWM
Actuators->ForceFeedback: Resistance
ForceFeedback->Steering Zone(Front MCU): Force Value
Steering Zone(Front MCU)-->RPI: Force Value
RPI-->(1)Cockpit Zone: User Feedback
```



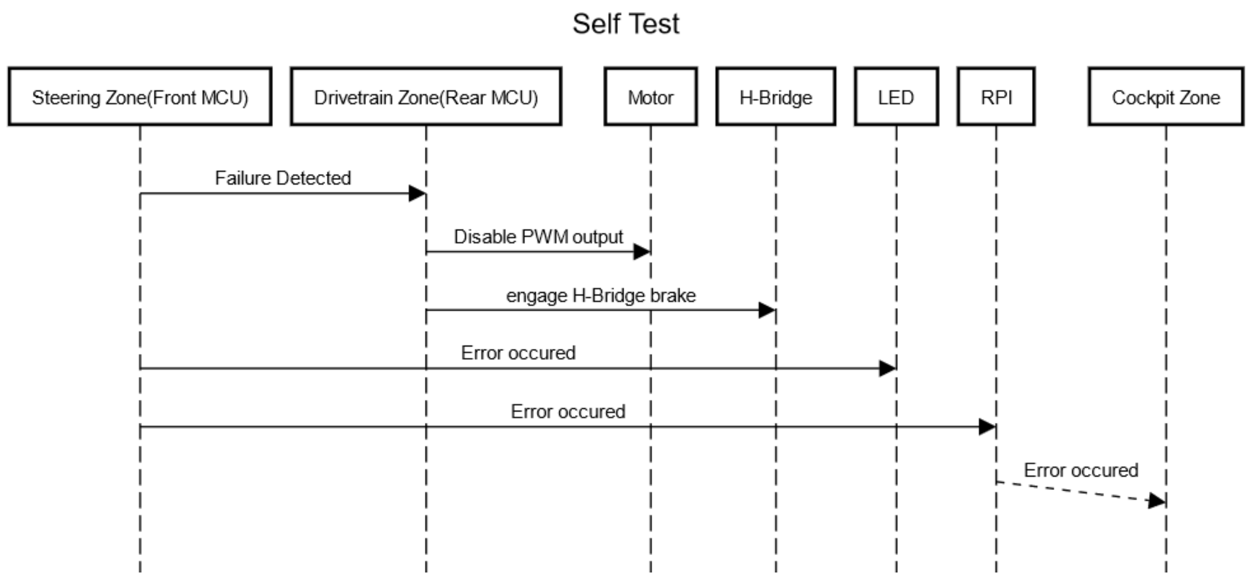
4. Turning Signal

```
Unset
title Turning Signal

[->Cockpit Zone:Blinker Pressed
Cockpit Zone->UDP Proxy:Blinker Pressed
UDP Proxy->(1)RPI:Blinker Pressed
RPI->Steering Zone(Front MCU):Blinker Pressed
RPI->Drivetrain Zone(Rear MCU):Blinker Pressed
Steering Zone(Front MCU)->Front Blinker:Left/Right blinker flash
Drivetrain Zone(Rear MCU)->Rear Blinker:Left/Right blinker flash
Steering Zone(Front MCU)->Drivetrain Zone(Rear MCU):Wheel threshold surpased,
then returned
Steering Zone(Front MCU)->Front Blinker:Turn off blinker
Drivetrain Zone(Rear MCU)->Rear Blinker:Turn off blinker
```

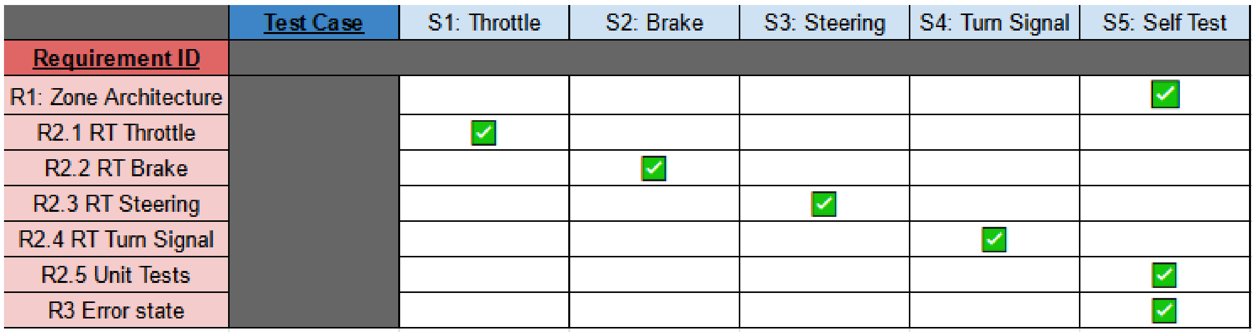


5. Self Test

```
Unset
title Self Test
Steering Zone(Front MCU)->Drivetrain Zone(Rear MCU):Failure Detected

Drivetrain Zone(Rear MCU)->Motor: Disable PWM output
Drivetrain Zone(Rear MCU)->H-Bridge:engage H-Bridge brake
Steering Zone(Front MCU)->LED:Error occured
Steering Zone(Front MCU)->RPI:Error occured
RPI-->(1)Cockpit Zone:Error occured
```



4.) Traceability per use-case



	Test Case	S1: Throttle	S2: Brake	S3: Steering	S4: Turn Signal	S5: Self Test
Requirement ID						
R1: Zone Architecture						✓
R2.1 RT Throttle		✓				
R2.2 RT Brake			✓			
R2.3 RT Steering				✓		
R2.4 RT Turn Signal					✓	
R2.5 Unit Tests						✓
R3 Error state						✓

5.) Test cases

Criteria	**Action**	**Expected Result**	**Pass / Fail**
Left blinker is on. Wheel is NOT past left threshold	Wheel turns left past threshold	Left blinker is still on.	✅
Left blinker is on. Wheel is past left threshold	Wheel returns from threshold.	Left blinker turns off.	✅
Right blinker is on. Wheel is in NOT past right threshold	Wheel turns right past threshold	Right blinker is still on.	✅
Right blinker is on. Wheel is past right threshold	Wheel returns from threshold.	Right blinker turns off.	✅
Left blinker is off	Press left turn button	Left blinker is on.	✅
Right blinker is off	Press right turn button	Right blinker is on.	✅
Car must be able to turn left	Turn the steering wheel left	Car wheels turn left	✅
Car must be able to turn right	Turn the steering wheel right	Car wheels turn right	✅

Wheel must have feedback force	-	User can feel resistance force feedback from the car wheel	✅
The system must be able to recover from an error state	Press the two reset buttons at the same time	Whole system resets	✅
The car moves forward when the throttle is pressed	Press throttle while brake is not pressed	Motor speed proportional to the throttle angle	✅
The car must come to a stop when the speed is non-zero and neither brake nor throttle are pressed	Both the brake and throttle are not pressed	Motor speed decay in a constant speed	✅
The brake must bring the car quickly to a stop	The brake is pressed	Motor speed converges to zero quickly	✅
The self-test button must bring the entire car into an error state	Press the self-test button on front zone	The whole system will enter an error state and enable the hazard signal	✅
The self-test button must bring the entire car into an error state	Press the self-test button on rear zone	The whole system will enter an error state and enable the hazard signal	✅
The throttle must be able to propel the car forward	Press the forward mode button	Pressing the throttle will move the car forward	✅
The throttle must be able to propel the car backwards	Press the backward mode button	Pressing the throttle will move the car backward	✅

6.) Proposed timing analysis

Proposed timing analysis test for R2.1 (Throttle) and R2.2 (Brake) and R2.4 (Turn signal)
Once the RPi receives a UDP message, a GPIO pin on the RPi is toggled. When the front/rear MCU receives the SPI command from the RPI, it sets the servo/motor/LED outputs and then toggles a GPIO pin on its board. The GPIO of RPi and STM32 toggle at two different time intervals. These signals will be read from two different sample channels of an oscilloscope, and the time difference between these two toggles in the sample channel should be measured to be less than the response time limit for that scenario (2ms for R2.1 and R2.2, 100ms for R2.4). In R2.4, the duty cycle of the front/rear blinker can be measured also using an oscilloscope and they should be synchronized within 1ms.

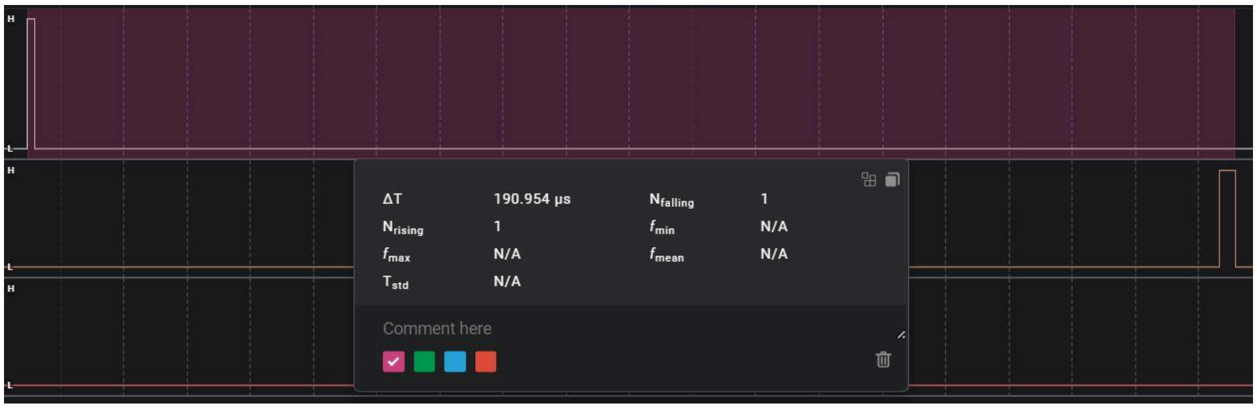
Proposed timing analysis test for R2.3 (Steering)
Upon reception of a periodic UDP message, a GPIO pin on the RPI is set to high. The RPI will send a SPI command to the front MCU to update the servo status, and the ADC will sample the torque applied to the servo. The front MCU will send the value back to the RPI via SPI. Upon sending the updated steering force value back to the steering wheel, the RPI will set the same GPIO pin to low. The time difference of the toggle which is measured using an oscilloscope should be less than 50ms.

Proposed timing analysis test for the unit test (Error)
Once the user presses the self-test button in the corresponding zone, a GPIO pin on that MCU is toggled and the MCU will send an error message to the RPI. Upon receiving the error message, the RPI will toggle its own GPIO pin. The time difference between these two GPIO toggles in the sample channel should be measured to be less than 10ms.

7.) Evaluation of Timing Analysis

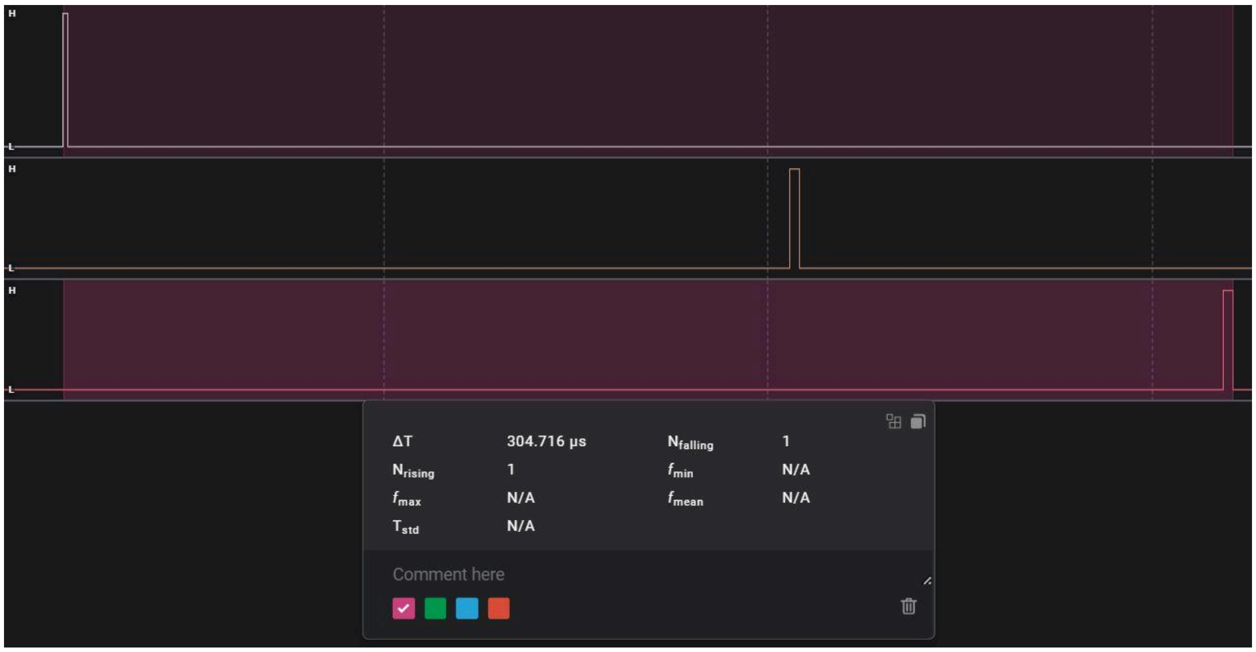
All timing measurements meet the requirements. Analysis was done using a logic analyzer. All tests except for the blinker sync test are done with respect to the RPi. We signal the start of the timing analysis by toggling a GPIO pin on the RPi high then low before the RPi transmits the SPI messages. We signal the end of the timing analysis by toggling a GPIO pin on the STM32 high then low after the STM32 completes the desired action. For the blinker sync test, the measurement starts before the front STM32 sets the blinker and ends after the rear STM32 sets the blinker.

Steering Timing:



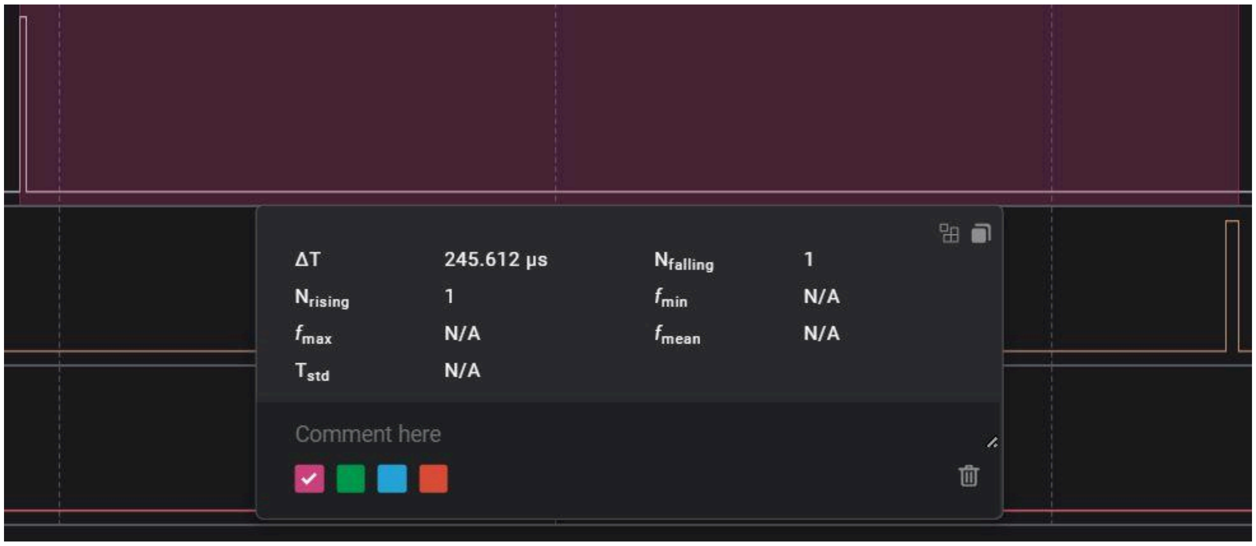
```
ΔT        190.954 µs    N_falling  1
N_rising  1             f_min      N/A
f_max     N/A           f_mean     N/A
T_std     N/A
Comment here
```

Front Blinker Timing:



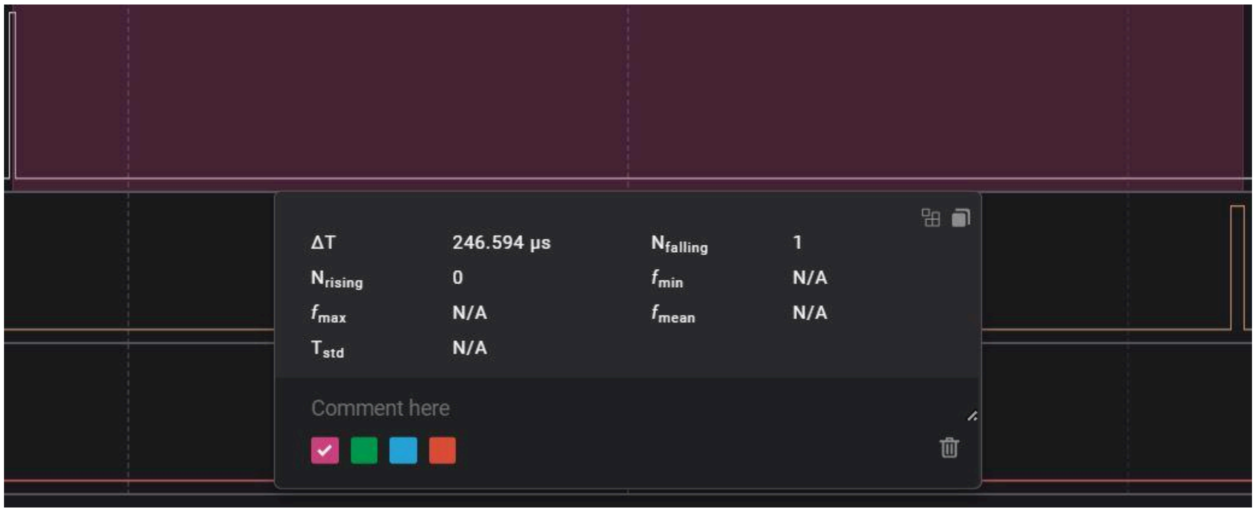
```
ΔT        304.716 µs    N_falling  1
N_rising  1             f_min      N/A
f_max     N/A           f_mean     N/A
T_std     N/A
Comment here
```

Rear Brake Timing:



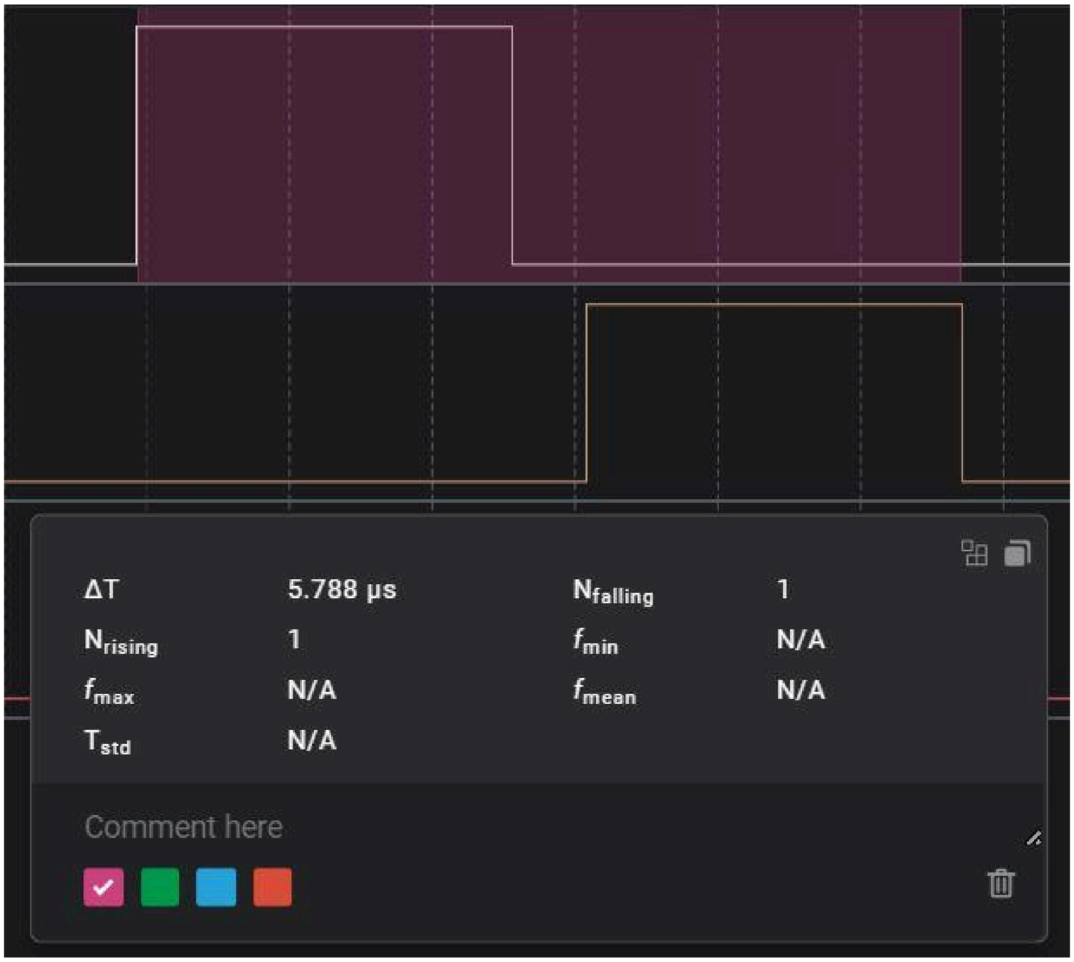
ΔT	245.612 µs	$N_{falling}$	1
N_{rising}	1	f_{min}	N/A
f_{max}	N/A	f_{mean}	N/A
T_{std}	N/A		

Rear Throttle Timing:



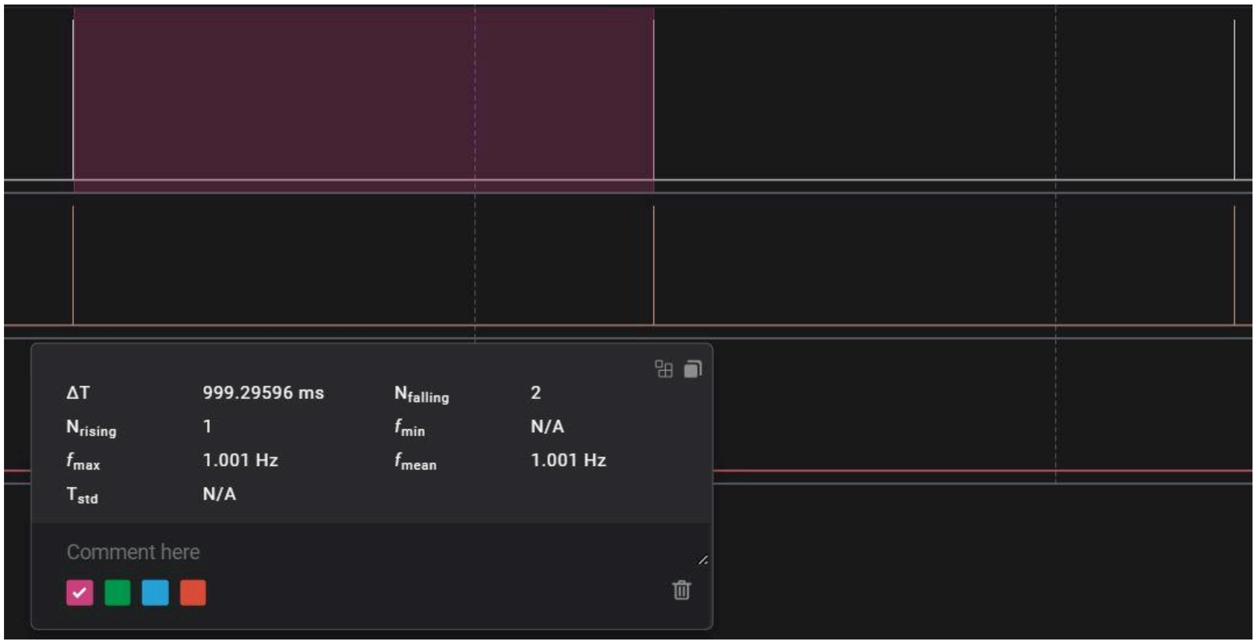
ΔT	246.594 µs	$N_{falling}$	1
N_{rising}	0	f_{min}	N/A
f_{max}	N/A	f_{mean}	N/A
T_{std}	N/A		

Blinker Sync Timing:



ΔT	5.788 µs	$N_{falling}$	1
N_{rising}	1	f_{min}	N/A
f_{max}	N/A	f_{mean}	N/A
T_{std}	N/A		

Blinker Period Timing:



ΔT	999.29596 ms	$N_{falling}$	2
N_{rising}	1	f_{min}	N/A
f_{max}	1.001 Hz	f_{mean}	1.001 Hz
T_{std}	N/A		

8.) Scheduling & Worst-case scenarios

Since our design is completely interrupt-driven, the worst case scenario is when all interrupts occur at the same time. However, since a MCU can only receive a message once every 50ms, the worst case scenario will have no effect on the schedulability of the system. As shown in the timing analysis, the maximum of both MCUs of the summation of worst-case computation time of tasks totals to under 1.5ms. This means that all interrupts will hit their deadlines, due to 1.5ms being less than the shortest deadline, 2ms. Since the period of a new set of interrupts is a minimum of 50ms, there is no feasible way that an interrupt will miss its deadline.

9.) Hardware Implementation

1. General Design

The car has a controller board that is used to control the front servo and the back motor. The controller board will interface with a RaspberryPi 4B that is used to wirelessly receive messages from the logitech steering wheel. In our design, the main PCB controller board can be separated into three different sections (steering, drive, and power management). The steering section of the PCB contains a STM32F410RB MCU which is responsible for steering of the front-wheels. Also, a current sense IC is used to sample the servo current to interpret the force that is being applied to the servo. In the drive section of the PCB, we have another STM32F410RB MCU that is used to interface with the H-bridge on the board to control the motor for the back-wheels. The H-bridge also contains encoder outputs that can be sampled to detect the current speed of the motor. Both sections of the board also contain different headers that we can use to establish SPI connections to the RPI, and SWD connectors can be used to flash the chip. Additionally, GPIO pinouts are also provided on the board to provide support for response time analysis by sampling the timing interval between GPIO toggles. The power management section of the connector contains a power jack for directly powering the board using the 12V power brick adapter or 9-12V battery pack. The board also has a separate USB-C port that can be connected to a 5V power bank to power the board. While the system can operate at 12V, we use a 15V boost converter module to boost the input to 15V. This provides consistent voltage and current to the motor for stable performance. The 15V output is directly fed to the H-bridge to power the motor. To power other parts of the board, including all of the onboard MCUs, a total of 4 linear power regulators are used. First, a 15V to 7V(1A) linear regulator is used to provide power to the servo. Next, two 15V to 5V(2A) linear regulators are used to supply enough current to power the RPi, current sense chip, H-bridge chip and provide backup power to encoders. Lastly, a 15V to 3.3V(2A) linear regulator is used as the main supply for the encoder and to power the two STM32 MCUs.

10.) Software Implementation

1. Front MCU

The design of the front MCU is entirely interrupt based. In the design, four hardware timers are used to control the blinker, heartbeat, and PWM for the servo. The communication protocol used to interface with the RPI is SPI, and SPI message handling is done via the SPI interrupt handler in an synchronous manner (RPI sends the message every 50 ms). A simple state machine is also implemented inside the message handler to ensure that the behavior of the blinker matches with the expected results from the test cases. In the message interrupt handler, the front MCU collects the SPI message that is sent from the RPI. The message can be casted to a struct (3 bytes) as shown below

```
C/C++
typedef struct {
      uint8_t turn_angle; //From 0 to 180 degree
      uint8_t left_blinekr_on;
      uint8_t right_blinker_on;
} message_t
```

Using the information given by the RPI, the handler then sets the current servo PWM by changing the compare and capture value register of the PWM timer, and the blinker states are also updated accordingly. After, the ADC current sense value of the servo is collected (ADC sampling time is around 3 cycles) and converted to a force feedback value (0 - 255) before it is sent back to the RPI. Since these are all very non-time consuming operations, the computation time of the message handler is estimated to be very short (less than 1ms).

As mentioned above, both the blinker and heartbeat module are also implemented using a timer interrupt. The frequency of the blinker module is set to 1Hz to ensure that the blinkers are toggled every second. However, in order to synchronize the front and the back blinkers, two GPIO pins are also toggled along with the blinker GPIO pins to send an external interrupt to the rear MCU. This design ensures that the synchronization timing requirement is met.

A heartbeat system is used to implement error detection between the front and rear MCU. This system utilizes two hardware timers with one that periodically toggles the GPIO pin for generating the heartbeat, and the other one is used as a countdown timer which should be cleared periodically by an external heartbeat interrupt coming from the rear MCU.

2. Rear MCU

The design of the rear MCU is also entirely interrupt based. In the design, four hardware times are used to control the heartbeat, two PWM for the two motors. The communication protocol used to interface with the RPI is also SPI, and SPI message handling is done via the SPI interrupt handler in an synchronous manner (RPI sends the message every 50 ms). A simple state machine is also implemented inside the message handler to ensure that the behavior of the blinker matches with the expected results from the test cases. In the message interrupt handler, the front MCU collects the SPI message that is sent from the RPI. The message can be casted to a struct (4 bytes) as shown below

```
C/C++
typedef struct {
       uint8_t throttle; //From 0 to 100 degree
       uint8_t brake;  // From 0 to 100 degree
       uint8_t fmode; // Forward direction, 0 or 1
       uint8_t bmode; // Backword direction, 0 or 1
} message_t
```

Using the information given by the RPI, the handler then sets the current two motors PWM by changing the compare and capture value register of the two PWM timers. When the brake angle is more than 5, it has the higher priority which will let the speed go to zero; Othersize, the speed of the motor will be setted based on the angle of the throttle. When the angle of the brake and throttle are all zero, the speed will decay constantly. The speed control system utilizes a PID-like control function to adjust the speed of the two motors at the same time based on the speed output from the motor encoders. Moreover, the rear MCU can receive the mode change signal. When the forward mode is 1, it will change the current rotation direction of the motor to forward; When the backward mode is 1, it will change the current rotation direction of the motor to backward. The change of the direction of the motors is fulfilled by changing the current direction in the H-bridge.

The heartbeat module of the rear part is implemented using a timer interrupt which is the same as the front part. The blinkers of the rear part are toggled based on the external interrupt which comes from the front part. This design can ensure the synchronization of the blinkers at the front and rear parts.

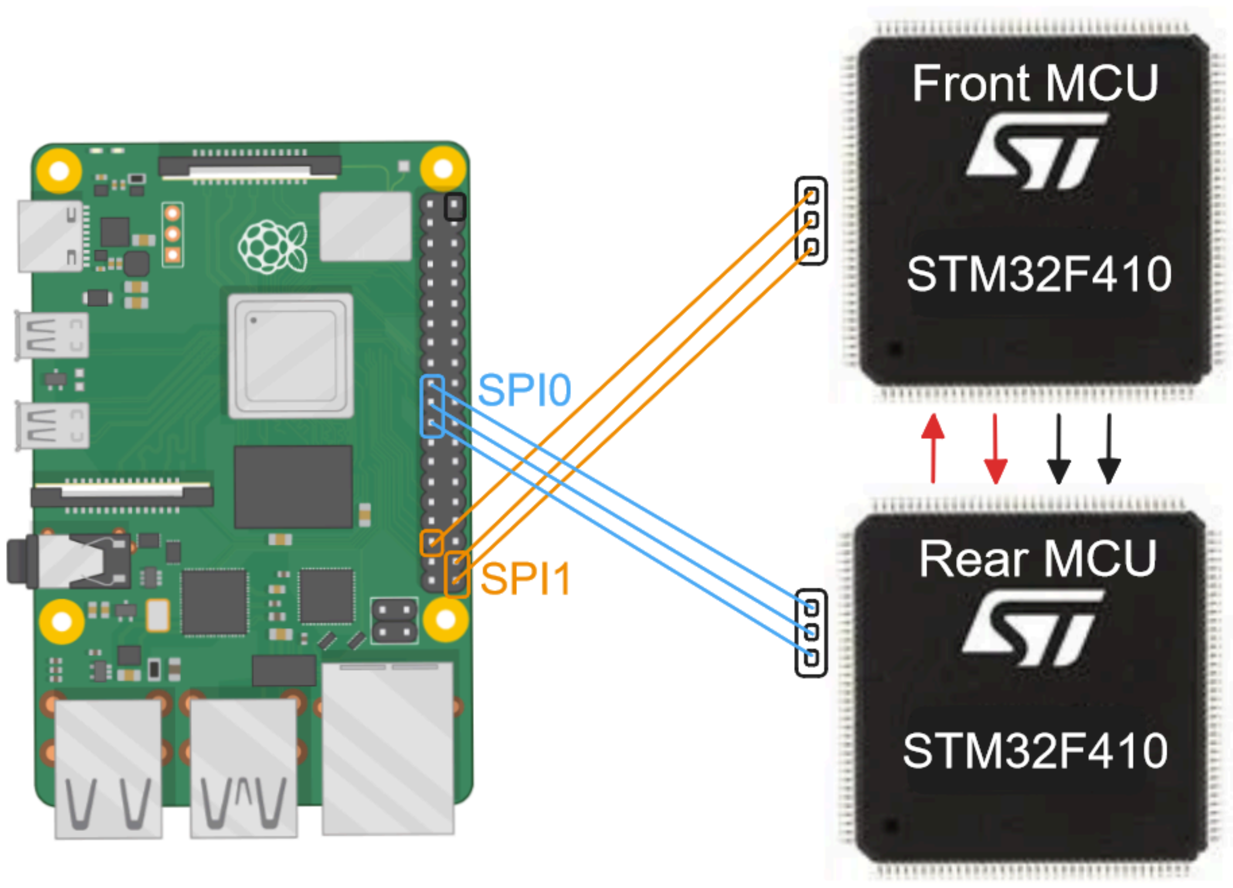
Also, a heartbeat system is used to implement error detection between the rear and front MCU. This system utilizes two hardware timers with one that periodically toggles the GPIO pin for generating the heartbeat, and the other one is used as a countdown timer which should be cleared periodically by an external heartbeat interrupt coming from the front MCU.

3. RPI

The implementation has been completely written in python, with the `spidev` package, which interfaces with SPI devices from user space via the SPI linux kernel driver. Per our original design, we planned to have the front MCU and RPI communicate via MISO and MOSI pins. We found due to the lack of proper support of the raspberry pi drivers, that true full-duplex is not supported for SPI. We are able to send data and receive data, but not simultaneously. To solve this problem, the RPI still communicates with the front MCU in full-duplex mode, but we are using the `xfer` function instead of assigning two threads to the `read` and `write` functions. This also means that the program running on the RPI is single-threaded. When a packet is received from the wheel, the RPI will break apart the packet and then forward the appropriate contents to each MCU. The RPI will write 4 bytes per 50ms to the rear MCU and write 3 bytes per 50ms to the front MCU. When the `xfer` function writes to the front MCU, the RPI expects a 1 byte return with the force feedback. The RPI will then forward this value back to the wheel.

11.) Car Overview

1. Network Architecture



For data transfer between RPi and the STM32s, we choose to use SPI as it allows bidirectional communication, maximizes transfer speed and gives a deterministic estimate of how long data transfer takes for timing analysis. Since the RPi supports multiple SPI buses, we decided to implement separate SPI connections to each STM32 so that we could have persistent connections as well as avoid the overhead of selecting which slave device to talk to using the chip select pin. We established several GPIO interrupts between the STM32s as seen in the diagram above, indicated by the red and black arrows. The rear STM32 uses one GPIO pin for sending heartbeat interrupts to the front STM32. The front STM32 also uses a GPIO pin for sending heartbeat interrupts to the rear STM32. We designed our software such that the front STM32 is responsible for synchronization of blinkers. The front STM32 uses 2 GPIO pins to send interrupts to the rear STM32 for synchronizing the left and right blinkers respectively.

2. Photos