Red Bird Racing EVRT Vehicle Control Unit (VCU) (2025)
Project Documentation

Red Bird Racing EVRT

May 3, 2025

Contents

1 Introduction

This document provides an overview of the Red Bird Racing EVRT Vehicle Control Unit (VCU) (2025). The VCU firmware is designed to manage pedal input, CAN communication, and vehicle state transitions for our Formula Student electric race car.

1.1 Project Structure

The project is organized as follows:

```
.
+-- include
|   +-- Debug.h
|   +-- pinMap.h
|   +-- README
+-- lib
|   +-- Pedal
|   |   +-- Pedal.cpp
|   |   +-- Pedal.h
|   |   +-- library.json
|   +-- Queue
|   |   +-- Queue.cpp
|   |   +-- Queue.h
|   +-- Signal_Processing
|   |   +-- Signal_Processing.cpp
|   |   +-- Signal_Processing.h
|   +-- README
+-- src
|   +-- main.cpp
+-- test
|   +-- README
+-- platformio.ini
+-- .vscode
    +-- launch.json
    +-- extensions.json
    +-- c_cpp_properties.json
```

2 Setup and Tuning

1. Adjust pedal input constants in `Pedal.h`.
2. Flash the VCU firmware. Ensure the car is jacked up and powered off during this process.
3. Clear the area around the car, especially the rear.
4. Test the minimum and maximum pedal input voltages and adjust the constants accordingly.

3 Debugging

Debugging is performed using the serial monitor. Enable specific debug messages by setting flags in `Debug.h`. Note that enabling debugging may introduce delays due to the slow serial communication.

4 Reverse Mode

Reverse mode is implemented for testing purposes only and is prohibited in competition. The driver must hold the reverse button to engage reverse mode. Releasing the button places the car in neutral. The car would re-enter reverse mode if criteria are met; else forward mode is engaged if its criteria are met.

Important Notes:

- **Do NOT use in actual competition!**
- **Rules 5.2.2.3: 禁止通过驱动装置反转车轮。**
- Rough translation: It is prohibited to use the motor to turn the wheels backwards.

4.1 Reverse Mode Logic

The reverse mode logic in `Pedal.cpp` allows the driver to toggle between reverse and forward modes using a single button. Below is the updated workflow and key components:

4.1.1 Key Functions

- `void pedal_can_frame_update(can_frame *tx_throttle_msg, unsigned long millis)`: Updates the CAN frame with the current throttle value and handles reverse mode logic. The reverse button toggles between reverse and forward modes:
  - If the reverse button is pressed, the mode toggles between reverse and forward.
  - If `reverseMode` is `true`, the buzzer is activated, and reverse torque is calculated.
- `int calculateReverseTorque(float throttleVolt, float vehicleSpeed, int torqueRequested)`: Calculates the torque value for reverse mode with the following constraints:
  - The throttle voltage must be less than one-third of the maximum throttle voltage (`MAX_THROTTLE_IN_VOLT / 3`).
  - The vehicle speed must not exceed the reverse speed limit (`REVERSE_SPEED_MAX`).
  - The torque is scaled down to 30% of the requested torque to ensure reverse mode is slow and controllable.

  If any of the constraints are violated, the torque is set to zero.

4.1.2 Reverse Mode Workflow

1. The reverse button state is read using `digitalRead(reverse_pin)`.
2. If the reverse button is pressed, and conditions are met:
   - If the vehicle is in forward mode (`reverseMode = false`), it switches to reverse mode (`reverseMode = true`).
   - If the vehicle is in reverse mode (`reverseMode = true`), it switches to forward mode (`reverseMode = false`).
3. In reverse mode:
   - A buzzer is activated with a periodic beep to alert nearby individuals. The cycle time is BUZZER_CYCLE_TIME milliseconds.
   - The reverse torque is calculated using `calculateReverseTorque`.
4. The throttle torque value is updated and sent via CAN messages.
5. If the motor direction needs to be flipped (e.g., for forward mode), the torque value is negated.

4.1.3 Safety Notes

- Reverse mode is implemented for testing purposes only and should not be used in competition.
- **Rules 5.2.2.3**
- Rough translation: It is prohibited to use the motor to turn the wheels backwards.
- The reverse mode logic ensures that the vehicle operates safely by limiting throttle and speed in reverse mode.
- However, care should be taken any time the vehicle is maneuvering, or if the buzzer is heard.

5 Source Code Overview

5.1 `main.cpp`

The main file initializes the pedal, CAN communication, and state machine for the car. It handles transitions between states such as `INIT`, `IN_STARTING_SEQUENCE`, `BUZZING`, and `DRIVE_MODE`.

```
#include <Arduino.h>
#include "pinMap.h"
#include "Pedal.h"
#include <mcp2515.h>
#include "Debug.h"

// === Pin setup ===
// Pin setup for pedal pins are done by the constructor of Pedal object
```

```
uint8_t pin_out[4] = {LED1, LED2, LED3, BRAKE_OUT};
uint8_t pin_in[4] = {BTN1, BTN2, BTN3, BTN4};

// === CAN + Pedal ===
MCP2515 mcp2515(CS_CAN);
Pedal pedal;

struct can_frame tx_throttle_msg;
struct can_frame rx_msg;

// For limiting the throttle update cycle
// const int THROTTLE_UPDATE_PERIOD_MILLIS = 50; // Period of sending canbus signal
// unsigned long final_throttle_time_millis = 0;  // The last time sent a canbus message

/* === Car Status State Machine ===
Meaning of different car statuses
INIT (0):  Just started the car
IN_STARTING_SEQUENCE (1):  1st Transition state -- Driver holds the "Start" button and is on full brakes, lasts for STATUS_1_TIME_MILLIS milliseconds
BUZZING (2):  2nd Transition state -- Buzzer bussin, driver can release "Start" button and brakes
DRIVE_MODE (3):  Ready to drive -- Motor starts responding according to the driver pedal input. "Drive mode" LED lights up, indicating driver can press the throttle

Separately, the following will be done outside the status checking part:
1.  Before the "Drive mode" LED lights up, if the throttle pedal is pressed (Throttle input is not euqal to 0), the car_status will return to 0
2.  Before the "Drive mode" LED lights up, the canbus will keep sending "0 torque" messages to the motor

Also, during status 0, 1, and 2, the VCU will keep sending "0 torque" messages to the motor via CAN
*/
enum CarStatus
{
    INIT = 0,
    IN_STARTING_SEQUENCE = 1,
    BUZZING = 2,
    DRIVE_MODE = 3
};
CarStatus car_status = INIT;
unsigned long car_status_millis_counter = 0; // Millis counter for 1st and 2nd transitionin states
const int STATUS_1_TIME_MILLIS = 2000;       // The amount of time that the driver needs to hold the "Start" button and full brakes in order to activate driving mode
const int BUSSIN_TIME_MILLIS = 2000;         // The amount of time that the buzzer will buzz for

void setup()
{
    // Init pedals
    pedal = Pedal(APPS_5V, APPS_3V3, REVERSE_BUTTON, LED1, millis());

```

```
    // Init input pins
    for (int i = 0; i < 4; i++)
        pinMode(pin_in[i], INPUT);
    // Init output pins
    for (int i = 0; i < 4; i++)
        pinMode(pin_out[i], OUTPUT);

    // Init mcp2515
    mcp2515.reset();
    mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ); // 8MHZ for testing on uno
    mcp2515.setNormalMode();

    // Init serial for testing if DEBUG flag is set to true
    if (DEBUG == true)
    {
        while (!Serial)
            ;
        Serial.begin(9600);
    }

    DBGLN_STATUS("Entered State 0 (Idle)");
}

void loop()
{
    // Update pedal value
    pedal.pedal_update(millis());

    /*
    For the time being:
    BTN1 = "Start" button
    BTN2 = Brake pedal
    LED1 = Buzzer output
    LED2 = "Drive" mode indicator
    */
    DBG_PEDAL("Pedal Value: ");
    DBGLN_PEDAL(pedal.final_pedal_value);

    if (car_status == INIT)
    {
        // car_status = 3; // For testing drive mode

        pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
        mcp2515.sendMessage(&tx_throttle_msg);
        DBGLN_CAN("Holding 0 torque during state 0");

        if (digitalRead(BTN1) == HIGH && digitalRead(BTN2) == HIGH) // Check if "Start" button and brake is fully pressed
        {
            car_status = IN_STARTING_SEQUENCE;
            car_status_millis_counter = millis();
            DBGLN_STATUS("Entered State 1");
        }
    }
    else if (car_status == IN_STARTING_SEQUENCE)
    {
        pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
        mcp2515.sendMessage(&tx_throttle_msg);
```

```
        DBGLN_CAN("Holding 0 torque during state 1");

        if (digitalRead(BTN1) == LOW || digitalRead(BTN2) == LOW) // Check if "Start" button or brake is not fully pressed
        {
            car_status = INIT;
            car_status_millis_counter = millis();
            DBGLN_STATUS("Entered State 0 (Idle)");
        }
        else if (millis() - car_status_millis_counter >= STATUS_1_TIME_MILLIS) // Check if button held long enough
        {
            car_status = BUZZING;
            digitalWrite(LED1, HIGH); // Turn on buzzer
            car_status_millis_counter = millis();
            DBGLN_STATUS("Transition to State 2: Buzzer ON");
        }
    }
    else if (car_status == BUZZING)
    {
        pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
        mcp2515.sendMessage(&tx_throttle_msg);
        DBGLN_CAN("Holding 0 torque during state 2");

        if (millis() - car_status_millis_counter >= BUSSIN_TIME_MILLIS)
        {
            digitalWrite(LED2, HIGH); // Turn on "Drive" mode indicator
            digitalWrite(LED1, LOW);  // Turn off buzzer
            car_status = DRIVE_MODE;
            DBGLN_STATUS("Transition to State 3: Drive mode");
        }
    }
    else if (car_status == DRIVE_MODE)
    {
        // In "Drive mode", car_status won't change, the drvier either continue to drive, or shut off the car
        DBGLN_STATUS("In Drive Mode");
    }
    else
    {
        // Error, idk wtf to do here
        DBGLN_STATUS("ERROR: Invalid car_status encountered!");
    }

    // Pedal update
    if (car_status == DRIVE_MODE)
    {
        // Send pedal value through canbus
        pedal.pedal_can_frame_update(&tx_throttle_msg, millis());
        // The following if block is needed only if we limit the lower bound for canbus cycle period
        // if (millis() - final_throttle_time_millis >= THROTTLE_UPDATE_PERIOD_MILLIS)
        // {
        //     mcp2515.sendMessage(&tx_throttle_msg);
        //     final_throttle_time_millis = millis();
        // }
        mcp2515.sendMessage(&tx_throttle_msg);
```

```
        DBGLN_CAN("Throttle CAN frame sent");
    }
    else
    {
        if (pedal.final_pedal_value > MIN_THROTTLE_OUT_VAL)
        {
            car_status = INIT;
            car_status_millis_counter = millis(); // Set to current time, in case any counter relies on this
            pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
            mcp2515.sendMessage(&tx_throttle_msg);
            DBGLN_STATUS("Throttle pressed too early - Resetting to State 0");
        }
    }

    // mcp2515.sendMessage(&tx_throttle_msg);
    // uint32_t lastLEDtick = 0;
    // Optional RX handling (disabled for now)
    // if (mcp2515.readMessage(&rx_msg) == MCP2515::ERROR_OK)
    // {
    //     // Commented out as currenlty no need to include receive functionality
    //     // if (rx_msg.can_id == 0x522)
    //     //     for (int i = 0; i < 8; i++)
    //     //         digitalWrite(pin_out[i], (rx_msg.data[0] >> i) & 0x01);
    // }
}
```

Listing 1: main.cpp

5.2 Pedal.cpp and Pedal.h

These files define the `Pedal` class, which encapsulates functionality for reading pedal input, filtering signals, and constructing CAN frames.

```
#include "Pedal.h"
#include "Arduino.h"
#include "Signal_Processing.cpp"
#include "Debug.h"

// Sinc function of size 128
float SINC_128[128] = {0.017232, 0.002666, -0.013033, -0.026004, -0.032934, -0.031899, -0.022884, -0.007851, 0.009675, 0.025427,
                       0.035421, 0.036957, 0.029329, 0.014081, -0.005294, -0.024137, -0.037732, -0.042472, -0.036792, -0.021652,
                       -0.000402, 0.021937, 0.039841, 0.048626, 0.045647, 0.031053, 0.007888, -0.018512, -0.041722, -0.055750,
                       -0.056553, -0.043139, -0.017994, 0.013320, 0.043353, 0.064476, 0.070758, 0.059540, 0.032321, -0.005306,
                       -0.044714, -0.076126, -0.090908, -0.083781, -0.054402, -0.007911, 0.045791, 0.093940, 0.123670, 0.125067,
```

```
                            0.093855, 0.033095, -0.046569, -0.128280, -0.191785,
                                 -0.217229, -0.189201, -0.100224, 0.047040,
                                0.239389,
                            0.454649, 0.664997, 0.841471, 0.958851, 1, 0.958851,
                                 0.841471, 0.664997, 0.454649, 0.239389,
                                0.047040,
                            -0.100224, -0.189201, -0.217229, -0.191785,
                                -0.128280, -0.046569, 0.033095, 0.093855,
                                0.125067, 0.123670,
                            0.093940, 0.045791, -0.007911, -0.054402, -0.083781,
                                 -0.090908, -0.076126, -0.044714, -0.005306,
                                0.032321,
                            0.059540, 0.070758, 0.064476, 0.043353, 0.013320,
                                -0.017994, -0.043139, -0.056553, -0.055750,
                                -0.041722,
                            -0.018512, 0.007888, 0.031053, 0.045647, 0.048626,
                                0.039841, 0.021937, -0.000402, -0.021652,
                                -0.036792,
                            -0.042472, -0.037732, -0.024137, -0.005294,
                                0.014081, 0.029329, 0.036957, 0.035421, 0.025427,
                                 0.009675,
                            -0.007851, -0.022884, -0.031899, -0.032934,
                                -0.026004, -0.013033};

Pedal::Pedal()
    : input_pin_1(-1), input_pin_2(-1), reverse_pin(-1), buzzer_pin(-1),
        previous_millis(0), conversion_rate(0), fault(true),
        fault_force_stop(false) {}

Pedal::Pedal(int input_pin_1, int input_pin_2, int reverse_pin, int
    buzzer_pin, unsigned long millis, unsigned short conversion_rate)
    : input_pin_1(input_pin_1), input_pin_2(input_pin_2), reverse_pin(
        reverse_pin), buzzer_pin(buzzer_pin), previous_millis(millis),
        conversion_rate(conversion_rate), fault(false), fault_force_stop(
        false)
{
    // Init pins
    pinMode(input_pin_1, INPUT);
    pinMode(input_pin_2, INPUT);
    pinMode(buzzer_pin, OUTPUT);
    conversion_period = 1000 / conversion_rate;

    // Init ADC buffers
    for (int i = 0; i < ADC_BUFFER_SIZE; ++i)
    {
        pedalValue_1.buffer[i] = 0;
        pedalValue_2.buffer[i] = 0;
    }
}

void Pedal::pedal_update(unsigned long millis)
{
    // If is time to update
    if (millis - previous_millis > conversion_period)
    {
        // Updating the previous millis
        previous_millis = millis;
        // Record readings in buffer
```

```
        pedalValue_1.push(analogRead(input_pin_1));
        pedalValue_2.push(analogRead(input_pin_2));

        // By default range of pedal 1 is APPS_PEDAL_1_RANGE, pedal 2 is APPS_PEDAL_2_RANGE;

        // this is current taking the direct array the circular queue writes into. Bad idea to do anything other than a simple average
        // if not using a linear filter, pass the pedalValue_1.getLinearBuffer() to the filter function to ensure the ordering is correct.
        // can also consider injecting the filter into the queue if need
        // depends on the hardware filter, reduce software filtering as much as possible
        int pedal_filtered_1 = round(AVG_filter<float>(pedalValue_1.buffer, ADC_BUFFER_SIZE));
        int pedal_filtered_2 = round(AVG_filter<float>(pedalValue_2.buffer, ADC_BUFFER_SIZE));

        // int pedal_filtered_1 = round(FIR_filter<float>(pedalValue_1.buffer, SINC_128, ADC_BUFFER_SIZE, 6.176445));
        // int pedal_filtered_2 = round(FIR_filter<float>(pedalValue_2.buffer, SINC_128, ADC_BUFFER_SIZE, 6.176445));
        final_pedal_value = pedal_filtered_1; // Only take in pedal 1 value

        DBG_PEDAL("Pedal 1: ");
        DBG_PEDAL(pedal_filtered_1);
        DBG_PEDAL(" | Pedal 2: ");
        DBG_PEDAL(pedal_filtered_2);
        DBG_PEDAL(" | Final: ");
        DBGLN_PEDAL(final_pedal_value);

        if (check_pedal_fault(pedal_filtered_1, pedal_filtered_2))
        {
            if (fault)
            { // Previous scan is already faulty
                if (millis - fault_start_millis > 100)
                { // Faulty for more than 100 ms
                    // TODO: Add code for alerting the faulty pedal, and whatever else mandated in rules Ch.2 Section 12.8, 12.9

                    // Turning off the motor is achieved using another digital pin, not via canbus, but will still send 0 torque can signals
                    fault_force_stop = true;

                    DBGLN_PEDAL("FAULT: Pedal mismatch persisted > 100ms!");

                    return;
                }
            }
            else
            {
                fault_start_millis = millis;
                DBGLN_PEDAL("FAULT: Pedal mismatch started");
            }
```

```
            fault = true;
            return;
        }
    }
}

void Pedal::pedal_can_frame_stop_motor(can_frame *tx_throttle_msg)
{
    tx_throttle_msg->can_id = 0x201;
    tx_throttle_msg->can_dlc = 3;
    tx_throttle_msg->data[0] = 0x90; // 0x90 for torque, 0x31 for speed
    tx_throttle_msg->data[1] = 0;
    tx_throttle_msg->data[2] = 0;

    DBGLN_PEDAL("CAN STOP");
}

void Pedal::pedal_can_frame_update(can_frame *tx_throttle_msg, unsigned
   long millis)
{
    if (fault_force_stop)
    {
        pedal_can_frame_stop_motor(tx_throttle_msg);
        return;
    }
    float throttle_volt = (float)final_pedal_value * APPS_PEDAL_1_RANGE /
       1024; // Converts most update pedal value to a float between 0V and
       5V

    int16_t throttle_torque_val = 0;
    /*
    Between 0V and THROTTLE_LOWER_DEADZONE_MAX_IN_VOLT: Error for open
       circuit
    Between THROTTLE_LOWER_DEADZONE_MAX_IN_VOLT and MIN_THROTTLE_IN_VOLT:
       0% Torque
    Between MIN_THROTTLE_IN_VOLT and MAX_THROTTLE_IN_VOLT: Linear
       relationship
    Between MAX_THROTTLE_IN_VOLT and THORTTLE_UPPER_DEADZONE_MIN_IN_VOLT:
       100% Torque
    Between THORTTLE_UPPER_DEADZONE_MIN_IN_VOLT and 5V: Error for short
       circuit
    */
    if (throttle_volt < THROTTLE_LOWER_DEADZONE_MIN_IN_VOLT)
    {
        DBG_PEDAL("Throttle voltage too low");
        DBGLN_PEDAL(throttle_volt);
        throttle_torque_val = 0;
    }
    else if (throttle_volt < MIN_THROTTLE_IN_VOLT)
    {
        throttle_torque_val = MIN_THROTTLE_OUT_VAL;
    }
    else if (throttle_volt < MAX_THROTTLE_IN_VOLT)
    {
        // Scale up the value for canbus
        throttle_torque_val = (throttle_volt - MIN_THROTTLE_IN_VOLT) *
           MAX_THROTTLE_OUT_VAL / (MAX_THROTTLE_IN_VOLT -
```

```
            MIN_THROTTLE_IN_VOLT);
    }
    else if (throttle_volt < THROTTLE_UPPER_DEADZONE_MAX_IN_VOLT)
    {
        throttle_torque_val = MAX_THROTTLE_OUT_VAL;
    }
    else
    {
        DBG_PEDAL("Throttle voltage too high");
        DBGLN_PEDAL(throttle_volt);
        // For safety, this should not be set to other values
        throttle_torque_val = 0;
    }

    //
    // Reverse mode logic
    // Do NOT use in actual competition! Read Documentation
    //

    reverseButtonPressed = digitalRead(reverse_pin);
    // temp override for testing
    float brakePercentage = 0.0;
    float vehicleSpeed = 0.0;

    // check enter reverse mode
    if (reverseButtonPressed)
    {
        if (!reverseMode)
        {
            reverseMode = check_enter_reverse_mode(brakePercentage,
                throttle_volt, vehicleSpeed);
        }
        else
        {
            reverseMode = check_enter_forward_mode(brakePercentage,
                throttle_volt, vehicleSpeed);
        }
    }
    if (reverseMode)
    {
        // Reverse mode
        // buzzer
        if (millis % (2 * REVERSE_BEEP_CYCLE_TIME) <
            REVERSE_BEEP_CYCLE_TIME)
        {
            digitalWrite(buzzer_pin, HIGH);
        }
        else
        {
            digitalWrite(buzzer_pin, LOW);
        }
        // Reverse mode torque calculation
        throttle_torque_val = calculateReverseTorque(throttle_volt,
            vehicleSpeed, throttle_torque_val);
    }

    DBG_PEDAL("CAN UPDATE: Throttle = ");
    DBGLN_PEDAL(throttle_torque_val);
```

```
    // motor reverse is car forward
    if (Flip_Motor_Dir)
    {
        throttle_torque_val = -throttle_torque_val;
    }

    tx_throttle_msg->can_id = 0x201;
    tx_throttle_msg->can_dlc = 3;
    tx_throttle_msg->data[0] = 0x90; // 0x90 for torque, 0x31 for speed
    tx_throttle_msg->data[1] = throttle_torque_val & 0xFF;
    tx_throttle_msg->data[2] = (throttle_torque_val >> 8) & 0xFF;
}

bool Pedal::check_pedal_fault(int pedal_1, int pedal_2)
{
    float pedal_1_percentage = (float)pedal_1 / 1024;
    float pedal_2_percentage = (float)pedal_2 * (APPS_PEDAL_1_RANGE / APPS_PEDAL_2_RANGE) / 1024;

    float pedal_percentage_diff = abs(pedal_1_percentage - pedal_2_percentage);
    // Currently the only indication for faulty pedal is just 2 pedal values are more than 10% different

    if (pedal_percentage_diff > 0.1)
    {
        DBGLN_PEDAL("WARNING: Pedal mismatch > 10%");
        return true;
    }
    return false;
}

bool Pedal::check_enter_reverse_mode(float brakePercentage, float throttlePercentage, float vehicleSpeed)
// Enable reverse mode.
//
// Do NOT use in actual competition!
// Read documentation
//
// returns reverseMode status
{
    if (brakePercentage > REVERSE_ENTER_BRAKE_THRESHOLD && throttlePercentage < REVERSE_ENTER_THROTTLE_THRESHOLD && vehicleSpeed < CAR_STATIONARY_SPEED_THRESHOLD)
    {
        DBGLN_PEDAL("Entering reverse mode!");
        return true;
    }
    return false;
}

bool Pedal::check_enter_forward_mode(float brakePercentage, float throttlePercentage, float vehicleSpeed)
// will see what additional criteria can be added
// returns reverseMode status
{
    if (brakePercentage > REVERSE_ENTER_BRAKE_THRESHOLD &&
```

```
        throttlePercentage < MIN_THROTTLE_IN_VOLT && vehicleSpeed <
        CAR_STATIONARY_SPEED_THRESHOLD)
    {
        DBGLN_PEDAL("Entering forward mode!");
        return false;
    }
    return true;
}

int Pedal::calculateReverseTorque(float throttleVolt, float vehicleSpeed,
    int torqueRequested)
// Calculate the torque value for reverse mode
// require throttle to be less than 1/3
// limit speed to threshold
{
    if (throttleVolt > MAX_THROTTLE_IN_VOLT / 3)
        return 0;
    if (vehicleSpeed > REVERSE_SPEED_MAX)
        return 0;
    DBG_PEDAL("Reverse mode: ");
    return torqueRequested * 0.3; // make reverse slow and controllable
    // consider that throttle must be less than 1/3
    // then max torque is 1/10 of the normal torque
}
```

Listing 2: Pedal.cpp

```
#ifndef PEDAL_H
#define PEDAL_H

#include "Queue.h"
#include "mcp2515.h"

// Constants
const float APPS_PEDAL_1_MIN_VOLTAGE = 0.0;
const float APPS_PEDAL_1_MAX_VOLTAGE = 5.0;
const float APPS_PEDAL_2_MIN_VOLTAGE = 0.0;
const float APPS_PEDAL_2_MAX_VOLTAGE = 3.3;

const float APPS_PEDAL_1_RANGE = APPS_PEDAL_1_MAX_VOLTAGE -
    APPS_PEDAL_1_MIN_VOLTAGE;
const float APPS_PEDAL_2_RANGE = APPS_PEDAL_2_MAX_VOLTAGE -
    APPS_PEDAL_2_MIN_VOLTAGE;

const float APPS_PEDAL_1_LOWER_DEADZONE_WIDTH = 0.0;
const float APPS_PEDAL_1_UPPER_DEADZONE_WIDTH = 0.4;
// const float APPS_PEDAL_2_LOWER_DEADZONE_WIDTH = 0.0;
// const float APPS_PEDAL_2_UPPER_DEADZONE_WIDTH = 0.0;

const float MIN_THROTTLE_IN_VOLT = APPS_PEDAL_1_MIN_VOLTAGE +
    APPS_PEDAL_1_LOWER_DEADZONE_WIDTH;
const float MAX_THROTTLE_IN_VOLT = APPS_PEDAL_1_MAX_VOLTAGE -
    APPS_PEDAL_1_UPPER_DEADZONE_WIDTH;
const float THROTTLE_LOWER_DEADZONE_MIN_IN_VOLT = APPS_PEDAL_1_MIN_VOLTAGE
    - APPS_PEDAL_1_LOWER_DEADZONE_WIDTH;
const float THROTTLE_UPPER_DEADZONE_MAX_IN_VOLT = APPS_PEDAL_1_MAX_VOLTAGE
    + APPS_PEDAL_1_UPPER_DEADZONE_WIDTH;

```

```
const int MAX_THROTTLE_OUT_VAL = 32430; // Maximum torque value is 32760
    for mcp2515
// current set to a slightly lower value to not use current control
// see E,EnS group discussion, 20250425HKT020800 discussion
const int MIN_THROTTLE_OUT_VAL = 300; // Minium torque value tested is 300
    (TBC)

// To go forward, this should be true; false sets the motor to go in
    reverse
bool Flip_Motor_Dir = true; // Flips the direction of motor output
// set to true for gen 3

// Reverse mode "stationary" speed threshold
const float CAR_STATIONARY_SPEED_THRESHOLD = 0.2;
// Reverse mode entering brake threshold
const float REVERSE_ENTER_BRAKE_THRESHOLD = 0.5;
// Reverse mode entering throttle threshold
const float REVERSE_ENTER_THROTTLE_THRESHOLD = 0.1;
// Reverse mode maximum speed
const float REVERSE_SPEED_MAX = 0.2;
// Reverse mode buzzer cycle time
const unsigned short REVERSE_BEEP_CYCLE_TIME = 400; // in ms

#define ADC_BUFFER_SIZE 16

// Class for generic pedal object
// For Gen 5 car, only throttle pedal is wired through the VCU, so we use
    Pedal class for Throttle pedal only.
class Pedal
{
public:
    // Two input pins for reading both pedal potentiometer
    // Conversion rate in Hz
    Pedal(int input_pin_1, int input_pin_2, int reverse_pin, int buzzer_pin
        , unsigned long millis, unsigned short conversion_rate = 1000);

    // Defualt constructor, expected another constructor should be called
        before start using
    Pedal();

    // Update function. To be called on every loop and pass the current
        time in millis
    void pedal_update(unsigned long millis);

    // Updates the can_frame with the most update pedal value. To be called
         on every loop and pass the can_frame by reference.
    void pedal_can_frame_update(can_frame *tx_throttle_msg, unsigned long
        millis);

    // Updates the can_frame to send a "0 Torque" value through canbus.
    void pedal_can_frame_stop_motor(can_frame *tx_throttle_msg);

    // Pedal value after filtering and processing
    // Under normal circumstance, should store a value between 0 and 1023
        inclusive (translates to 0v - 5v)
    int final_pedal_value;

private:
```

```
    int input_pin_1, input_pin_2, reverse_pin, buzzer_pin;

    // Will rollover every 49 days
    unsigned long previous_millis;

    unsigned short conversion_rate;

    // If the two potentiometer inputs are too different (> 10%), the
       inputs are faulty
    // Definition for faulty is under FSEC 2024 Chapter 2, section 12.8,
       12.9
    bool fault = false;
    unsigned long fault_start_millis;

    // Forced stop the car due too long fault sensors, restart car to reset
        this to false
    bool fault_force_stop = true;

    // Period in millisecond
    unsigned short conversion_period;

    // Returns true if pedal is faulty
    bool check_pedal_fault(int pedal_1, int pedal_2);

    RingBuffer<float, ADC_BUFFER_SIZE> pedalValue_1;
    RingBuffer<float, ADC_BUFFER_SIZE> pedalValue_2;

    // reverse mode
    //
    // Do NOT use in actual competition!
    // Read documentation
    //

    // calculate reverse torque value
    int calculateReverseTorque(float throttleVolt, float vehicleSpeed, int
       torqueRequested);

    // reverse button pin to bool
    bool reverseButtonPressed = false;

    // Reverse mode status
    bool reverseMode = false;

    // function check and set reverse, return reverse mode status
    bool check_enter_reverse_mode(float brakePercentage, float
       throttlePercentage, float vehicleSpeed);

    // function check and set forward, return reverse mode status
    bool check_enter_forward_mode(float brakePercentage, float
       throttlePercentage, float vehicleSpeed);
};

#endif // PEDAL_H
```

Listing 3: Pedal.h

5.3 Queue.cpp and Queue.h

These files implement a static FIFO queue and a ring buffer for managing pedal input data.

```cpp
#include "Queue.h"

template <typename T, int size>
Queue<T, size>::Queue() : queueFull(false), queueEmpty(true), queueCount(0) {}

template <typename T, int size>
void Queue<T, size>::push(T val)
{
    for (int i = size - 1; i > 0; i--)
    {
        buffer[i] = buffer[i - 1];
    }
    buffer[0] = val;

    if (!queueFull)
        ++queueCount;

    queueFull = (queueCount == size);
}

template <typename T, int size>
T Queue<T, size>::pop()
{
    if (queueCount == 0) // If the queue is empty and attempts to pop an object, the program will end
        this->exit(0);   // this->exit() somehow circumnavigates some errors

    --queueCount;
    queueEmpty = (queueCount == 0);

    return buffer[queueCount];
}

template <typename T, int size>
T Queue<T, size>::getHead()
{
    return buffer[queueCount - 1];
}

template <typename T, int size>
bool Queue<T, size>::isEmpty()
{
    return queueEmpty;
}

template <typename T, int size>
bool Queue<T, size>::isFull()
{
    return queueFull;
}
```

Listing 4: `Queue.cpp`

```
#ifndef QUEUE_H
#define QUEUE_H

// A simple FIFO object
// This object is completely static
template <typename T, int size>
class Queue
{
public:
    Queue();

    void push(T val);
    T pop();
    T getHead();

    bool isEmpty();
    bool isFull();

    T buffer[size];

private:
    bool queueFull, queueEmpty;
    int queueCount;
};

template <typename T, int size>
class RingBuffer
{
public:
    RingBuffer() : head(0), count(0) {}

    void push(T val)
    {
        buffer[head] = val;
        head = (head + 1) % size;
        if (count < size)
            ++count;
    }

    void getLinearBuffer(T *out)
    {
        for (int i = 0; i < count; ++i)
        {
            out[i] = buffer[(head + i) % size];
        }
    }

    T buffer[size];
    int head;
    int count;
};

#endif // QUEUE_H
```

Listing 5: `Queue.h`

5.4 Signal_Processing.cpp and Signal_Processing.h

These files provide simple DSP functions for filtering and processing pedal input signals.

```cpp
#include "Signal_Processing.h"

// Apply a FIR filter on the signal buffer
// The buffer size must be the same as the kernel
// Filtered output will be stored in the output_buf
template <typename T>
T FIR_filter(T *buffer, float *kernel, int buf_size, float kernel_sum)
{
    float sum = 0;

    for (int i = 0; i < buf_size; ++i)
    {
        sum += buffer[i] * kernel[i];
    }

    // Kernel sum is the sum of all values in the kernel. This normalize
       the output value
    return sum / kernel_sum;
}

template <typename T>
T average(T val1, T val2)
{
    return (val1 + val2) / 2;
}

template <typename T>
T AVG_filter(T *buffer, int buf_size)
{
    float sum = 0;

    for (int i = 0; i < buf_size; ++i)
        sum += buffer[i];
    return sum / (float)buf_size;
}
```

Listing 6: Signal_Processing.cpp

```cpp
// A library containing simple DSP functions, for ADC filtering, buffer
   comparisons and more
#ifndef SIGNAL_PROCESSING_H
#define SIGNAL_PROCESSING_H

template <typename T>
T FIR_filter(T *buffer, float *kernel, int buf_size, float kernel_sum);

template <typename T>
T average(T val1, T val2);

template <typename T>
T AVG_filter(T *buffer, int buf_size);

#endif
```

Listing 7: Signal_Processing.h

5.5 Debug.h

This file defines macros for enabling or disabling debug messages.

```
#ifndef DEBUG_H
#define DEBUG_H

// === Debug Flags ===

// ALWAYS LEAVE FALSE FOR GITHUB
#define DEBUG false // Oveall debug functionality

#define DEBUG_PEDAL true && DEBUG
#define DEBUG_SIGNAL_PROC false && DEBUG
#define DEBUG_GENERAL true && DEBUG
#define DEBUG_PEDAL true && DEBUG
#define DEBUG_CAN true && DEBUG
#define DEBUG_STATUS true && DEBUG

#if DEBUG_PEDAL
#define DBG_PEDAL(x) Serial.print(x)
#define DBGLN_PEDAL(x) Serial.println(x)
#else
#define DBG_PEDAL(x)
#define DBGLN_PEDAL(x)
#endif

#if DEBUG_SIGNAL_PROC
#define DBG_SIG(x) Serial.print(x)
#define DBGLN_SIG(x) Serial.println(x)
#else
#define DBG_SIG(x)
#define DBGLN_SIG(x)
#endif

#if DEBUG_GENERAL
#define DBG_GENERAL(x) Serial.print(x)
#define DBGLN_GENERAL(x) Serial.println(x)
#else
#define DBG_GENERAL(x)
#define DBGLN_GENERAL(x)
#endif

#if DEBUG_PEDAL
#define DBG_PEDAL(x) Serial.print(x)
#define DBGLN_PEDAL(x) Serial.println(x)
#else
#define DBG_PEDAL(x)
#define DBGLN_PEDAL(x)
#endif

#if DEBUG_CAN
#define DBG_CAN(x) Serial.print(x)
#define DBGLN_CAN(x) Serial.println(x)
#else
#define DBG_CAN(x)
#define DBGLN_CAN(x)
#endif

```

```
#if DEBUG_STATUS
#define DBG_STATUS(x) Serial.print(x)
#define DBGLN_STATUS(x) Serial.println(x)
#else
#define DBG_STATUS(x)
#define DBGLN_STATUS(x)
#endif

#endif // DEBUG_H
```

Listing 8: `Debug.h`

5.6 `pinMap.h`

This file maps the pins used in the project to meaningful names.

```
#ifndef PINMAP_H
#define PINMAP_H

#define BTN1 5
#define BTN2 6
#define BTN3 7
#define BTN4 8

// #define CS_CAN 14
#define CS_CAN 10 // For arduino testing

// #define APPS_5V 23
// #define APPS_3V3 24
// #define BRAKE_5V 25
// #define BRAKE_OUT 26
#define APPS_5V A0    // For arduino testing
#define APPS_3V3 A1   // For arduino testing
#define BRAKE_5V A2   // For arduino testing
#define BRAKE_OUT A3 // For arduino testing

#define REVERSE_BUTTON A4 // For arduino testing

#define LED1 2
#define LED2 3
#define LED3 4

#endif // PINMAP_H
```

Listing 9: `pinMap.h`

6 PlatformIO Configuration

The `platformio.ini` file configures the PlatformIO environment for the project. It specifies the board, framework, and library dependencies.

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
```

```
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:uno]
platform = atmelavr
board = uno
framework = arduino
lib_deps = autowp/autowp-mcp2515@^1.2.1
build_flags =
    -Wall
    -pedantic
    -Wextra
```

Listing 10: `platformio.ini`

7 Future Development

- Add more CAN channels for BMS, data logger, and other components.
- Improve the torque curve for better performance.
- Fully implement reverse mode.

8 References

- PlatformIO Documentation
- GCC Header File Documentation