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

Red Bird Racing EVRT

June 6, 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:

```
1  .
2  +-- include
3  |   +-- Debug.h
4  |   +-- pinMap.h
5  |   +-- README
6  +-- lib
7  |   +-- Pedal
8  |   |   +-- Pedal.cpp
9  |   |   +-- Pedal.h
10 |   |   +-- library.json
11 |   +-- Queue
12 |   |   +-- Queue.cpp
13 |   |   +-- Queue.h
14 |   +-- Signal_Processing
15 |   |   +-- Signal_Processing.cpp
16 |   |   +-- Signal_Processing.h
17 |   +-- README
18 +-- src
19 |   +-- main.cpp
20 +-- test
21 |   +-- README
22 +-- platformio.ini
23 +-- .vscode
24     +-- launch.json
25     +-- extensions.json
26     +-- 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:

-

-

-

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`.

```
1  #include <Arduino.h>
2  #include "pinMap.h"
3  #include "Pedal.h"
4  #include <mcp2515.h>
5  #include "Debug.h"
6
7  // === Pin setup ===
8  // Pin setup for pedal pins are done by the constructor of Pedal object
```

```cpp
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;
struct can_frame *tx_debug_msg = nullptr; // If DBC is not enabled, we don'
    t need to send debug messages

// 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
```

```cpp
52      pedal = Pedal(APPS_5V, APPS_3V3, REVERSE_BUTTON, LED1, millis());
53
54      // Init input pins
55      for (int i = 0; i < 4; i++)
56          pinMode(pin_in[i], INPUT);
57      // Init output pins
58      for (int i = 0; i < 4; i++)
59          pinMode(pin_out[i], OUTPUT);
60
61      // Init mcp2515
62      mcp2515.reset();
63      mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ); // 8MHZ for testing on uno
64      mcp2515.setNormalMode();
65
66      // Init serial for testing if DEBUG flag is set to true
67      if (DEBUG == true)
68      {
69          while (!Serial)
70              ;
71          Serial.begin(9600);
72      }
73
74      DBGLN_STATUS("Entered State 0 (Idle)");
75      if (DBC)
76      {
77          struct can_frame local_debug_msg; // Only created if needed
78          tx_debug_msg = &local_debug_msg;  // Point to the local variable
79      }
80  }
81
82  void loop()
83  {
84      // Update pedal value
85      pedal.pedal_update(millis());
86
87      /*
88      For the time being:
89      BTN1 = "Start" button
90      BTN2 = Brake pedal
91      LED1 = Buzzer output
92      LED2 = "Drive" mode indicator
93      */
94      DBG_PEDAL("Pedal Value: ");
95      DBGLN_PEDAL(pedal.final_pedal_value);
96
97      if (car_status == INIT)
98      {
99          // car_status = 3; // For testing drive mode
100
101          pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
102          mcp2515.sendMessage(&tx_throttle_msg);
103          DBGLN_CAN("Holding 0 torque during state 0");
104
105          if (digitalRead(BTN1) == HIGH && digitalRead(BTN2) == HIGH) //
106              Check if "Start" button and brake is fully pressed
107          {
108              car_status = IN_STARTING_SEQUENCE;
                 car_status_millis_counter = millis();
```

6

```cpp
109              DBGLN_STATUS("Entered State 1");
110          }
111      }
112      else if (car_status == IN_STARTING_SEQUENCE)
113      {
114          pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
115          mcp2515.sendMessage(&tx_throttle_msg);
116          DBGLN_CAN("Holding 0 torque during state 1");
117
118          if (digitalRead(BTN1) == LOW || digitalRead(BTN2) == LOW) // Check
                 if "Start" button or brake is not fully pressed
119          {
120              car_status = INIT;
121              car_status_millis_counter = millis();
122              DBGLN_STATUS("Entered State 0 (Idle)");
123          }
124          else if (millis() - car_status_millis_counter >=
                 STATUS_1_TIME_MILLIS) // Check if button held long enough
125          {
126              car_status = BUZZING;
127              digitalWrite(LED1, HIGH); // Turn on buzzer
128              car_status_millis_counter = millis();
129              DBGLN_STATUS("Transition to State 2: Buzzer ON");
130          }
131      }
132      else if (car_status == BUZZING)
133      {
134          pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
135          mcp2515.sendMessage(&tx_throttle_msg);
136          DBGLN_CAN("Holding 0 torque during state 2");
137
138          if (millis() - car_status_millis_counter >= BUSSIN_TIME_MILLIS)
139          {
140              digitalWrite(LED2, HIGH); // Turn on "Drive" mode indicator
141              digitalWrite(LED1, LOW);  // Turn off buzzer
142              car_status = DRIVE_MODE;
143              DBGLN_STATUS("Transition to State 3: Drive mode");
144          }
145      }
146      else if (car_status == DRIVE_MODE)
147      {
148          // In "Drive mode", car_status won't change, the drvier either
                 continue to drive, or shut off the car
149          DBGLN_STATUS("In Drive Mode");
150      }
151      else
152      {
153          // Error, idk wtf to do here
154          DBGLN_STATUS("ERROR: Invalid car_status encountered!");
155      }
156
157      // Pedal update
158      if (car_status == DRIVE_MODE)
159      {
160          // Send pedal value through canbus
161          pedal.pedal_can_frame_update(&tx_throttle_msg, millis(),
                 tx_debug_msg);
162          // The following if block is needed only if we limit the lower
```

```
                    bound for canbus cycle period
163         // if (millis() - final_throttle_time_millis >=
                THROTTLE_UPDATE_PERIOD_MILLIS)
164         // {
165         //     mcp2515.sendMessage(&tx_throttle_msg);
166         //     final_throttle_time_millis = millis();
167         // }
168         mcp2515.sendMessage(&tx_throttle_msg);
169
170 #if DBC // If DBC is enabled, send debug message; else don't compile this
        part
171         tx_debug_msg->data[0] = static_cast<uint8_t>(car_status);
172         tx_debug_msg->data[5] = BRAKE_5V;
173         mcp2515.sendMessage(tx_debug_msg); // Send debug message to CAN bus
174 #endif
175
176         DBGLN_CAN("Throttle CAN frame sent");
177     }
178     else
179     {
180         if (pedal.final_pedal_value > MIN_THROTTLE_OUT_VAL)
181         {
182             car_status = INIT;
183             car_status_millis_counter = millis(); // Set to current time,
                    in case any counter relies on this
184             pedal.pedal_can_frame_stop_motor(&tx_throttle_msg);
185             mcp2515.sendMessage(&tx_throttle_msg);
186             DBGLN_STATUS("Throttle pressed too early - Resetting to State 0
                ");
187         }
188     }
189
190     // mcp2515.sendMessage(&tx_throttle_msg);
191     // uint32_t lastLEDtick = 0;
192     // Optional RX handling (disabled for now)
193     // if (mcp2515.readMessage(&rx_msg) == MCP2515::ERROR_OK)
194     // {
195     //     // Commented out as currenlty no need to include receive
        functionality
196     //     // if (rx_msg.can_id == 0x522)
197     //     //     for (int i = 0; i < 8; i++)
198     //     //         digitalWrite(pin_out[i], (rx_msg.data[0] >> i) & 0x01
        );
199     // }
200 }
```

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.

```
1 #include "Pedal.h"
2 #include "Arduino.h"
3 #include "Signal_Processing.cpp"
4 #include "Debug.h"
```

```cpp
// 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;

```

```cpp
33      // Init ADC buffers
34      for (int i = 0; i < ADC_BUFFER_SIZE; ++i)
35      {
36          pedalValue_1.buffer[i] = 0;
37          pedalValue_2.buffer[i] = 0;
38      }
39  }
40
41  void Pedal::pedal_update(unsigned long millis)
42  {
43      // If is time to update
44      if (millis - previous_millis > conversion_period)
45      {
46          // Updating the previous millis
47          previous_millis = millis;
48          // Record readings in buffer
49          pedalValue_1.push(analogRead(input_pin_1));
50          pedalValue_2.push(analogRead(input_pin_2));
51
52          // By default range of pedal 1 is APPS_PEDAL_1_RANGE, pedal 2 is
                  APPS_PEDAL_2_RANGE;
53
54          // this is current taking the direct array the circular queue
                  writes into. Bad idea to do anything other than a simple average
55          // if not using a linear filter, pass the pedalValue_1.
                  getLinearBuffer() to the filter function to ensure the ordering
                  is correct.
56          // can also consider injecting the filter into the queue if need
57          // depends on the hardware filter, reduce software filtering as
                  much as possible
58          int pedal_filtered_1 = round(AVG_filter<float>(pedalValue_1.buffer,
                  ADC_BUFFER_SIZE));
59          int pedal_filtered_2 = round(AVG_filter<float>(pedalValue_2.buffer,
                  ADC_BUFFER_SIZE));
60
61          // int pedal_filtered_1 = round(FIR_filter<float>(pedalValue_1.
                  buffer, SINC_128, ADC_BUFFER_SIZE, 6.176445));
62          // int pedal_filtered_2 = round(FIR_filter<float>(pedalValue_2.
                  buffer, SINC_128, ADC_BUFFER_SIZE, 6.176445));
63          final_pedal_value = pedal_filtered_1; // Only take in pedal 1 value
64
65          DBG_PEDAL("Pedal 1: ");
66          DBG_PEDAL(pedal_filtered_1);
67          DBG_PEDAL(" | Pedal 2: ");
68          DBG_PEDAL(pedal_filtered_2);
69          DBG_PEDAL(" | Final: ");
70          DBGLN_PEDAL(final_pedal_value);
71
72          if (check_pedal_fault(pedal_filtered_1, pedal_filtered_2))
73          {
74              if (fault)
75              { // Previous scan is already faulty
76                  if (millis - fault_start_millis > 100)
77                  { // Faulty for more than 100 ms
78                      // TODO: Add code for alerting the faulty pedal, and
                              whatever else mandated in rules Ch.2 Section 12.8,
                              12.9
79
```

```cpp
80                        // Turning off the motor is achieved using another
                          //     digital pin, not via canbus, but will still send 0
                          //     torque can signals
81                        fault_force_stop = true;
82
83                        DBGLN_PEDAL("FAULT: Pedal mismatch persisted > 100ms!")
                          ;
84
85                        return;
86                    }
87                }
88                else
89                {
90                    fault_start_millis = millis;
91                    DBGLN_PEDAL("FAULT: Pedal mismatch started");
92                }
93
94                fault = true;
95                return;
96            }
97        }
98 }
99
100 void Pedal::pedal_can_frame_stop_motor(can_frame *tx_throttle_msg)
101 {
102     tx_throttle_msg->can_id = 0x201;
103     tx_throttle_msg->can_dlc = 3;
104     tx_throttle_msg->data[0] = 0x90; // 0x90 for torque, 0x31 for speed
105     tx_throttle_msg->data[1] = 0;
106     tx_throttle_msg->data[2] = 0;
107
108     DBGLN_PEDAL("CAN STOP");
109 }
110
111 void Pedal::pedal_can_frame_update(can_frame *tx_throttle_msg, unsigned
        long millis, can_frame *tx_debug_msg)
112 {
113     if (fault_force_stop)
114     {
115         pedal_can_frame_stop_motor(tx_throttle_msg);
116         return;
117     }
118     float throttle_volt = (float)final_pedal_value * APPS_PEDAL_1_RANGE /
            1024; // Converts most update pedal value to a float between 0V and
            5V
119
120     int16_t throttle_torque_val = 0;
121     /*
122     Between 0V and THROTTLE_LOWER_DEADZONE_MAX_IN_VOLT: Error for open
            circuit
123     Between THROTTLE_LOWER_DEADZONE_MAX_IN_VOLT and MIN_THROTTLE_IN_VOLT:
            0% Torque
124     Between MIN_THROTTLE_IN_VOLT and MAX_THROTTLE_IN_VOLT: Linear
            relationship
125     Between MAX_THROTTLE_IN_VOLT and THORTTLE_UPPER_DEADZONE_MIN_IN_VOLT:
            100% Torque
126     Between THORTTLE_UPPER_DEADZONE_MIN_IN_VOLT and 5V: Error for short
            circuit
```

```
127        */
128        if (throttle_volt < THROTTLE_LOWER_DEADZONE_MIN_IN_VOLT)
129        {
130            DBG_PEDAL("Throttle voltage too low");
131            DBGLN_PEDAL(throttle_volt);
132            throttle_torque_val = 0;
133        }
134        else if (throttle_volt < MIN_THROTTLE_IN_VOLT)
135        {
136            throttle_torque_val = MIN_THROTTLE_OUT_VAL;
137        }
138        else if (throttle_volt < MAX_THROTTLE_IN_VOLT)
139        {
140            // Scale up the value for canbus
141            throttle_torque_val = (throttle_volt - MIN_THROTTLE_IN_VOLT) *
142                MAX_THROTTLE_OUT_VAL / (MAX_THROTTLE_IN_VOLT -
143                MIN_THROTTLE_IN_VOLT);
142        }
143        else if (throttle_volt < THROTTLE_UPPER_DEADZONE_MAX_IN_VOLT)
144        {
145            throttle_torque_val = MAX_THROTTLE_OUT_VAL;
146        }
147        else
148        {
149            DBG_PEDAL("Throttle voltage too high");
150            DBGLN_PEDAL(throttle_volt);
151            // For safety, this should not be set to other values
152            throttle_torque_val = 0;
153        }
154
155        //
156        // Reverse mode logic
157        // Do NOT use in actual competition! Read Documentation
158        //
159
160        reverseButtonPressed = digitalRead(reverse_pin);
161        // temp override for testing
162        float brakePercentage = 0.0;
163        float vehicleSpeed = 0.0;
164
165        // check enter reverse mode
166        if (reverseButtonPressed)
167        {
168            if (!reverseMode)
169            {
170                reverseMode = check_enter_reverse_mode(brakePercentage,
171                    throttle_volt, vehicleSpeed);
172            }
173            else
174            {
175                reverseMode = check_enter_forward_mode(brakePercentage,
176                    throttle_volt, vehicleSpeed);
177            }
178        }
179        if (reverseMode)
180        {
181            // Reverse mode
182            // buzzer
```

```cpp
181        if (millis % (2 * REVERSE_BEEP_CYCLE_TIME) <
               REVERSE_BEEP_CYCLE_TIME)
182        {
183            digitalWrite(buzzer_pin, HIGH);
184        }
185        else
186        {
187            digitalWrite(buzzer_pin, LOW);
188        }
189        // Reverse mode torque calculation
190        throttle_torque_val = calculateReverseTorque(throttle_volt,
               vehicleSpeed, throttle_torque_val);
191    }
192
193
194    // motor reverse is car forward
195    if (Flip_Motor_Dir)
196    {
197        throttle_torque_val = -throttle_torque_val;
198    }
199
200    DBG_PEDAL("CAN UPDATE: Throttle = ");
201    DBGLN_PEDAL(throttle_torque_val);
202
203    tx_throttle_msg->can_id = 0x201;
204    tx_throttle_msg->can_dlc = 3;
205    tx_throttle_msg->data[0] = 0x90; // 0x90 for torque, 0x31 for speed
206    tx_throttle_msg->data[1] = throttle_torque_val & 0xFF;
207    tx_throttle_msg->data[2] = (throttle_torque_val >> 8) & 0xFF;
208
209 #if DBC
210    // CAN DBC debug
211    tx_debug_msg->can_id = 0x102; // ID for debug message
212    tx_debug_msg->can_dlc = 6;    // Length of the message
213    // created in main tx_debug_msg->data[0] = static_cast<uint8_t>(
           car_status); // State of the car
214    tx_debug_msg->data[1] = input_pin_1; // Pedal 1 voltage
215    tx_debug_msg->data[2] = input_pin_2; // Pedal 2 voltage
216    tx_debug_msg->data[3] = throttle_torque_val & 0xFF;
217    tx_debug_msg->data[4] = (throttle_torque_val >> 8) & 0xFF; // Torque
           value
218    // VCU currently not handling brake voltage
219    // tx_debug_msg->data[5] = input_pin_3;                        //
           Brake voltage
220 #endif
221 }
222
223 bool Pedal::check_pedal_fault(int pedal_1, int pedal_2)
224 {
225    float pedal_1_percentage = (float)pedal_1 / 1024;
226    float pedal_2_percentage = (float)pedal_2 * (APPS_PEDAL_1_RANGE /
           APPS_PEDAL_2_RANGE) / 1024;
227
228    float pedal_percentage_diff = abs(pedal_1_percentage -
           pedal_2_percentage);
229    // Currently the only indication for faulty pedal is just 2 pedal
           values are more than 10% different
230
```

```cpp
231      if (pedal_percentage_diff > 0.1)
232      {
233          DBGLN_PEDAL("WARNING: Pedal mismatch > 10%");
234          return true;
235      }
236      return false;
237 }
238
239 //// Reverse mode functions
240
241 bool Pedal::check_enter_reverse_mode(float brakePercentage, float
        throttlePercentage, float vehicleSpeed)
242 // Enable reverse mode.
243 //
244 // Do NOT use in actual competition!
245 // Read documentation
246 //
247 // returns reverseMode status
248 {
249      if (brakePercentage > REVERSE_ENTER_BRAKE_THRESHOLD &&
250          throttlePercentage < REVERSE_ENTER_THROTTLE_THRESHOLD &&
             vehicleSpeed < CAR_STATIONARY_SPEED_THRESHOLD)
250      {
251          DBGLN_PEDAL("Entering reverse mode!");
252          return true;
253      }
254      return false;
255 }
256
257 bool Pedal::check_enter_forward_mode(float brakePercentage, float
        throttlePercentage, float vehicleSpeed)
258 // will see what additional criteria can be added
259 // returns reverseMode status
260 {
261      if (brakePercentage > REVERSE_ENTER_BRAKE_THRESHOLD &&
             throttlePercentage < MIN_THROTTLE_IN_VOLT && vehicleSpeed <
             CAR_STATIONARY_SPEED_THRESHOLD)
262      {
263          DBGLN_PEDAL("Entering forward mode!");
264          return false;
265      }
266      return true;
267 }
268
269 int Pedal::calculateReverseTorque(float throttleVolt, float vehicleSpeed,
        int torqueRequested)
270 // Calculate the torque value for reverse mode
271 // require throttle to be less than 1/3
272 // limit speed to threshold
273 {
274      if (throttleVolt > MAX_THROTTLE_IN_VOLT / 3)
275          return 0;
276      if (vehicleSpeed > REVERSE_SPEED_MAX)
277          return 0;
278      DBG_PEDAL("Reverse mode: ");
279      return torqueRequested * 0.3; // make reverse slow and controllable
280      // consider that throttle must be less than 1/3
281      // then max torque is 1/10 of the normal torque
```

```
282 }
```

Listing 2: `Pedal.cpp`

```cpp
1  #ifndef PEDAL_H
2  #define PEDAL_H
3
4  #include "Queue.h"
5  #include "mcp2515.h"
6
7  // Constants
8  const float APPS_PEDAL_1_MIN_VOLTAGE = 0.0;
9  const float APPS_PEDAL_1_MAX_VOLTAGE = 5.0;
10 const float APPS_PEDAL_2_MIN_VOLTAGE = 0.0;
11 const float APPS_PEDAL_2_MAX_VOLTAGE = 3.3;
12
13 const float APPS_PEDAL_1_RANGE = APPS_PEDAL_1_MAX_VOLTAGE -
       APPS_PEDAL_1_MIN_VOLTAGE;
14 const float APPS_PEDAL_2_RANGE = APPS_PEDAL_2_MAX_VOLTAGE -
       APPS_PEDAL_2_MIN_VOLTAGE;
15
16 const float APPS_PEDAL_1_LOWER_DEADZONE_WIDTH = 0.0;
17 const float APPS_PEDAL_1_UPPER_DEADZONE_WIDTH = 0.4;
18 // const float APPS_PEDAL_2_LOWER_DEADZONE_WIDTH = 0.0;
19 // const float APPS_PEDAL_2_UPPER_DEADZONE_WIDTH = 0.0;
20
21 const float MIN_THROTTLE_IN_VOLT = APPS_PEDAL_1_MIN_VOLTAGE +
       APPS_PEDAL_1_LOWER_DEADZONE_WIDTH;
22 const float MAX_THROTTLE_IN_VOLT = APPS_PEDAL_1_MAX_VOLTAGE -
       APPS_PEDAL_1_UPPER_DEADZONE_WIDTH;
23 const float THROTTLE_LOWER_DEADZONE_MIN_IN_VOLT = APPS_PEDAL_1_MIN_VOLTAGE
       - APPS_PEDAL_1_LOWER_DEADZONE_WIDTH;
24 const float THROTTLE_UPPER_DEADZONE_MAX_IN_VOLT = APPS_PEDAL_1_MAX_VOLTAGE
       + APPS_PEDAL_1_UPPER_DEADZONE_WIDTH;
25
26 const int MAX_THROTTLE_OUT_VAL = 32430; // Maximum torque value is 32760
       for mcp2515
27 // currently set to a slightly lower value to not use speed control (100%)
28 // see E,EnS group discussion, 20250425HKT020800 discussion
29 const int MIN_THROTTLE_OUT_VAL = 300; // Minium torque value tested is 300
       (TBC)
30
31 // To go forward, this should be true; false sets the motor to go in
       reverse
32 const bool Flip_Motor_Dir = true; // Flips the direction of motor output
33 // set to true for gen 3
34
35 // Reverse mode "stationary" speed threshold
36 const float CAR_STATIONARY_SPEED_THRESHOLD = 0.2;
37 // Reverse mode entering brake threshold
38 const float REVERSE_ENTER_BRAKE_THRESHOLD = 0.5;
39 // Reverse mode entering throttle threshold
40 const float REVERSE_ENTER_THROTTLE_THRESHOLD = 0.1;
41 // Reverse mode maximum speed
42 const float REVERSE_SPEED_MAX = 0.2;
43 // Reverse mode buzzer cycle time
44 const unsigned short REVERSE_BEEP_CYCLE_TIME = 400; // in ms
45
```

```cpp
#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, can_frame *tx_debug_msg);

    // 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);
```

```
94
95    RingBuffer<float, ADC_BUFFER_SIZE> pedalValue_1;
96    RingBuffer<float, ADC_BUFFER_SIZE> pedalValue_2;
97
98    // reverse mode
99    //
100   // Do NOT use in actual competition!
101   // Read documentation
102   //
103
104   // calculate reverse torque value
105   int calculateReverseTorque(float throttleVolt, float vehicleSpeed, int
          torqueRequested);
106
107   // reverse button pin to bool
108   bool reverseButtonPressed = false;
109
110   // Reverse mode status
111   bool reverseMode = false;
112
113   // function check and set reverse, return reverse mode status
114   bool check_enter_reverse_mode(float brakePercentage, float
          throttlePercentage, float vehicleSpeed);
115
116   // function check and set forward, return reverse mode status
117   bool check_enter_forward_mode(float brakePercentage, float
          throttlePercentage, float vehicleSpeed);
118 };
119
120 #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.

```
1  #include "Queue.h"
2
3  template <typename T, int size>
4  Queue<T, size>::Queue() : queueFull(false), queueEmpty(true), queueCount(0)
       {}
5
6  template <typename T, int size>
7  void Queue<T, size>::push(T val)
8  {
9      for (int i = size - 1; i > 0; i--)
10     {
11         buffer[i] = buffer[i - 1];
12     }
13     buffer[0] = val;
14
15     if (!queueFull)
16         ++queueCount;
17
18     queueFull = (queueCount == size);
```

```cpp
19 }
20
21 template <typename T, int size>
22 T Queue<T, size>::pop()
23 {
24     if (queueCount == 0) // If the queue is empty and attempts to pop an
             object, the program will end
25         this->exit(0);   // this->exit() somehow circumnavigates some
                 errors
26
27     --queueCount;
28     queueEmpty = (queueCount == 0);
29
30     return buffer[queueCount];
31 }
32
33 template <typename T, int size>
34 T Queue<T, size>::getHead()
35 {
36     return buffer[queueCount - 1];
37 }
38
39 template <typename T, int size>
40 bool Queue<T, size>::isEmpty()
41 {
42     return queueEmpty;
43 }
44
45 template <typename T, int size>
46 bool Queue<T, size>::isFull()
47 {
48     return queueFull;
49 }
```

Listing 4: `Queue.cpp`

```cpp
1 #ifndef QUEUE_H
2 #define QUEUE_H
3
4 // A simple FIFO object
5 // This object is completely static
6 template <typename T, int size>
7 class Queue
8 {
9 public:
10     Queue();
11
12     void push(T val);
13     T pop();
14     T getHead();
15
16     bool isEmpty();
17     bool isFull();
18
19     T buffer[size];
20
21 private:
22     bool queueFull, queueEmpty;
```

```
23    int queueCount;
24 };
25
26 template <typename T, int size>
27 class RingBuffer
28 {
29 public:
30     RingBuffer() : head(0), count(0) {}
31
32     void push(T val)
33     {
34         buffer[head] = val;
35         head = (head + 1) % size;
36         if (count < size)
37             ++count;
38     }
39
40     void getLinearBuffer(T *out)
41     {
42         for (int i = 0; i < count; ++i)
43         {
44             out[i] = buffer[(head + i) % size];
45         }
46     }
47
48     T buffer[size];
49     int head;
50     int count;
51 };
52
53 #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.

```
1 #include "Signal_Processing.h"
2
3 // Apply a FIR filter on the signal buffer
4 // The buffer size must be the same as the kernel
5 // Filtered output will be stored in the output_buf
6 template <typename T>
7 T FIR_filter(T *buffer, float *kernel, int buf_size, float kernel_sum)
8 {
9     float sum = 0;
10
11     for (int i = 0; i < buf_size; ++i)
12     {
13         sum += buffer[i] * kernel[i];
14     }
15
16     // Kernel sum is the sum of all values in the kernel. This normalize
           the output value
17     return sum / kernel_sum;
18 }
```

```
19
20  template <typename T>
21  T average(T val1, T val2)
22  {
23      return (val1 + val2) / 2;
24  }
25
26  template <typename T>
27  T AVG_filter(T *buffer, int buf_size)
28  {
29      float sum = 0;
30
31      for (int i = 0; i < buf_size; ++i)
32          sum += buffer[i];
33      return sum / (float)buf_size;
34  }
```

Listing 6: `Signal_Processing.cpp`

```
1   // A library containing simple DSP functions, for ADC filtering, buffer
        comparisons and more
2   #ifndef SIGNAL_PROCESSING_H
3   #define SIGNAL_PROCESSING_H
4
5   template <typename T>
6   T FIR_filter(T *buffer, float *kernel, int buf_size, float kernel_sum);
7
8   template <typename T>
9   T average(T val1, T val2);
10
11  template <typename T>
12  T AVG_filter(T *buffer, int buf_size);
13
14  #endif
```

Listing 7: `Signal_Processing.h`

5.5 `Debug.h`

This file defines macros for enabling or disabling debug messages.

```
1   #ifndef DEBUG_H
2   #define DEBUG_H
3
4   // === Debug Flags ===
5
6   // ALWAYS LEAVE FALSE FOR GITHUB
7   #define DEBUG false // Overall debug functionality
8
9   #define DBC true // Debug CAN bus functionality, independent of DEBUG
10
11  #define DEBUG_PEDAL true && DEBUG
12  #define DEBUG_SIGNAL_PROC false && DEBUG
13  #define DEBUG_GENERAL true && DEBUG
14  #define DEBUG_PEDAL true && DEBUG
15  #define DEBUG_CAN true && DEBUG
16  #define DEBUG_STATUS true && DEBUG
```

```
17
18  #if DEBUG_PEDAL
19  #define DBG_PEDAL(x) Serial.print(x)
20  #define DBGLN_PEDAL(x) Serial.println(x)
21  #else
22  #define DBG_PEDAL(x)
23  #define DBGLN_PEDAL(x)
24  #endif
25
26  #if DEBUG_SIGNAL_PROC
27  #define DBG_SIG(x) Serial.print(x)
28  #define DBGLN_SIG(x) Serial.println(x)
29  #else
30  #define DBG_SIG(x)
31  #define DBGLN_SIG(x)
32  #endif
33
34  #if DEBUG_GENERAL
35  #define DBG_GENERAL(x) Serial.print(x)
36  #define DBGLN_GENERAL(x) Serial.println(x)
37  #else
38  #define DBG_GENERAL(x)
39  #define DBGLN_GENERAL(x)
40  #endif
41
42  #if DEBUG_PEDAL
43  #define DBG_PEDAL(x) Serial.print(x)
44  #define DBGLN_PEDAL(x) Serial.println(x)
45  #else
46  #define DBG_PEDAL(x)
47  #define DBGLN_PEDAL(x)
48  #endif
49
50  #if DEBUG_CAN
51  #define DBG_CAN(x) Serial.print(x)
52  #define DBGLN_CAN(x) Serial.println(x)
53  #else
54  #define DBG_CAN(x)
55  #define DBGLN_CAN(x)
56  #endif
57
58  #if DEBUG_STATUS
59  #define DBG_STATUS(x) Serial.print(x)
60  #define DBGLN_STATUS(x) Serial.println(x)
61  #else
62  #define DBG_STATUS(x)
63  #define DBGLN_STATUS(x)
64  #endif
65
66  #endif // DEBUG_H
```

Listing 8: `Debug.h`

5.6 `pinMap.h`

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

```
1  #ifndef PINMAP_H
2  #define PINMAP_H
3
4  #define BTN1 5
5  #define BTN2 6
6  #define BTN3 7
7  #define BTN4 8
8
9  // #define CS_CAN 14
10 #define CS_CAN 10 // For arduino testing
11
12 // #define APPS_5V 23
13 // #define APPS_3V3 24
14 // #define BRAKE_5V 25
15 // #define BRAKE_OUT 26
16 #define APPS_5V A0   // For arduino testing
17 #define APPS_3V3 A1  // For arduino testing
18 #define BRAKE_5V A2   // For arduino testing
19 #define BRAKE_OUT A3 // For arduino testing
20
21 #define REVERSE_BUTTON A4 // For arduino testing
22
23 #define LED1 2
24 #define LED2 3
25 #define LED3 4
26
27 #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.

```
1  ; PlatformIO Project Configuration File
2  ;
3  ;   Build options: build flags, source filter
4  ;   Upload options: custom upload port, speed and extra flags
5  ;   Library options: dependencies, extra library storages
6  ;   Advanced options: extra scripting
7  ;
8  ; Please visit documentation for the other options and examples
9  ; https://docs.platformio.org/page/projectconf.html
10
11 [env:uno]
12 platform = atmelavr
13 board = uno
14 framework = arduino
15 lib_deps = autowp/autowp-mcp2515@^1.2.1
16 build_flags =
17     -Wall
18     -pedantic
19     -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