Newbie VCU Documentation

1 Introduction

This document provides an overview of the Red Bird Racing (RBR) EVRT Vehicle Control Unit (VCU) firmware made for learning purposes. The VCU firmware is designed to manage pedal inputs, CAN communication, and vehicle state transitions used to control the Formula Student electric race car under the RBR team.

This document contains references for setup, debugging, as well as system design for

2 Setup and Tuning

1. Adjust pedal ratios in pedal.h according to the voltages ranges of the APPS. Tolerance threshold should be adjusted accordingly by calculation using either the Desmos linked, or equivalent calculations.
   Ideally, an equivalent set of calculations in a python script should be used, as the Desmos linked is not well optimised.
   The partitions used for the validation settings should be chosen such that the number of elements in each partition does not exceed 10000 (limitation of Desmos).
2. Flash firmware, ensure the wheels are not in contact with any surface and the car is powered off during the process.
3. Clear the area around the car, especially the rear.
4. Test the minimum and maximum pedal input voltages and adjust the table accordingly. The minimum and maximum of the table should not be equal to the minimum and maximum pedal input voltages.

3 Debugging

3.1 Introduction

Debugging is done only VIA CAN, due to lack of serial implementations for debugging. Enable specific debug messages by setting flags in debug.hpp. Note that enabling debugging will introduce significant delay as most actions in the code will send some form of debug message if enabled.

3.2 Debug options

Debugging can be turned on or off by section by setting the appropriate values to true or false at the top of debug.hpp. It is recommended to leave DEBUG_MOTOR_STOP and DEBUG_BMS_AWAIT false when not specifically debugging those parts, as they can flood the CAN bus with unneeded frames.

3.3 CAN reference

The following is a reference for CAN messages on different CAN IDs. CAN IDs that are used in the code but lacking a reference do not send any data. Note endianness is little-endian (0x069420 -> 20 then 94 then 06).

0	1	2	3	4	5
90	69	42			
torque cmd	torque_val				
CAN ID 0x201 – motor command					

0	1	2	3	4	5
69	42				
torque_val					
CAN ID 0x290 – torque value					

Note the provided .dbc file will translate pedal_1 and pedal_2 to their minimum voltages, and pedal_1_scaled and pedal_2_scaled to minimum % values

0	1	2	3	4	5	6	7
02	42	01	83	4A	82	4B	96
pedal_1		pedal_2		pedal_1_scaled		pedal_2_scaled	
CAN ID 0xD01 – pedal_values							

0	1	2	3	4	5
03					
car_status					
CAN ID 0xD02 - car_status					

0	1	2	3	4	5
10	00	00	00	00	
fault_status	value				
CAN ID 0xBAD - fault					

3.3.1 Fault reference

enum	fault_status	Meaning of value
00	NONE	unused
10	DIFF_DAPPS	scaled_pedal_difference
19	DIFF_RISING	0
1A	DIFF_EXCEED_DURATION	0
1C	DIFF_FALLING	0

FF	STATUS_INVALID	0

3.3.2 car_status reference

enum	CarStatus	Meaning of value
-1	INVALID	An invalid status was reached
0	INIT	Car just started or restarted
1	STARTING	1st Transition state. Driver holds start button and fully presses brakes, lasts for at least STARTING_DELAY milliseconds. Will not transition to next state until BMS ready.
2	BUZZING	2nd transition state. Buzzer buzzing, driver can release start button and brakes
3	DRIVE	Ready to drive. Motor starts responding according to pedal inputs. Drive mode LED lights up.

4 System design

4.1 Data structures

Implementation includes 1 struct and 1 class. The struct is a helper object only used once.

4.1.1 Motor class

Constructor

Accepts pointer to the MCP2515 instance for the motor can controller. Initialises an instance of the Motor class that contains the MCP 2515 instance pointer.

Public

Contains functions to set torque, stop motor, and update motor.

Private

Contains the MCP2515 instance pointer for the CAN interface, the stored torque to be sent, the CAN frame to be sent, and a CAN frame specifically to stop the motor.

4.2 Logic flow

There are tasks preformed regardless of state. The motor is updated at the start of every cycle, followed by the status of the brake pedal, then the brake light.

4.2.1 INIT 0

The firmware starts at this state when turned on. Also acts as a reset state.

Transitions into STARTING when both start button and brake pedal have been pressed.

4.2.2 STARTING (1)

First buffer state between INIT and DRIVE. Ensures BMS is ready.

Transitions into BUZZING when both start button and brake pedal have been held for at least 2 seconds (configurable). Will not transition until BMS sends a ready message via CAN.

Transitions into INIT if either start button or brake pedal is released during the delay. This includes after the 2 seconds while awaiting BMS sending the ready message.

4.2.3 BUZZING (2)

Second buffer state between INIT and DRIVE. Makes the buzzer turn on. Let's driver be aware that the car will transition into DRIVE state.

Transitions into DRIVE after 2 seconds (configurable) and turns off buzzer.

4.2.4 DRIVE (3)

Main state of the car. Turns on Drive mode LED and reads pedals. Detects for faults in the pedal inputs. Maps the pedal inputs to torque value by linear interpolation. Updates torque value to be sent to the motor.

Transitions into INIT if pedal fault has been detected continuously for over 100ms (configurable).

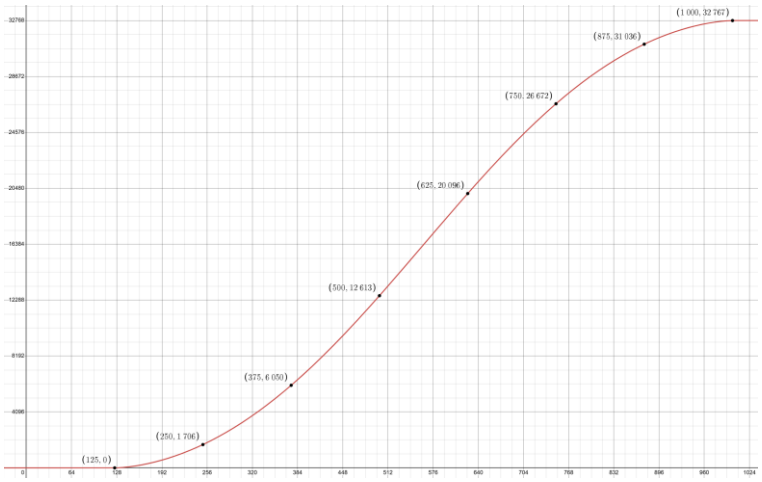
4.2.5 INVALID (-1)

Unreachable state. All other invalid states get set to –1. Stops motor and sets state to INVALID.

Transitions into INVALID at the end of every cycle.

4.3 Current pedal mapping

Small difference in the torque output near the lower and upper bounds of the mapping. Lower dead zone is larger so the driver resting their foot on the pedal does not trigger the motor.



5 Folder/File descriptions

5.1 src

Contains main firmware for the VCU

5.1.1 main.cpp

The main file initialises the pedals, CAN communications, indicator devices, and state machine for the car. Handles transitions between states when starting. Checking for pedal faults is done here.

5.2 include

Contains all common header files for the project. Defines pin mappings and enums.

5.2.1 pinMap.h

Centralised pin assignments for use across the project. No explicit pin assignments are used in the rest of the codebase for ease of maintainence and modification.

5.3 lib

Contains all libraries except headers (see include).

5.3.1 Debug

Defines macros for managing all debug messages across the codebase. Contains all debug message implementation.

5.3.2 Motor

Implements logic for constructing and sending CAN frames for motor control. Currently only does torque.

5.3.3 Pedal

Implements logics for pedal input, fault detection, reverse mode and mapping for throttle values.

5.4 platformio.ini

PlatformIO environment configuration file for the project. Specifies target board, framework, build flags and library dependencies required to build and upload the firmware.

6 Future development

Listed in order of priority:

Implement serial debug messages.

Convert pedal implementations into a class.

Streamline the passing of torque value from the pedal class to motor class.

Add built in testing scripts.

7 References

Tolerances for ADC https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

Pedal mapping graph https://www.desmos.com/calculator/wtzefqttlb