Abstract

This document will explain how to build the ADMIT14 car, starting with firmware of microcontroller, followed by mechanical design. Afterwards, I will talk about future improvements and developments for the system.

This is still a work-in-progress, many additions are to come. This document and other informations will be available on Github, in a dedicated repository. Link to repository: https://github.com/TuanMinh2803/HS_ESS_ADMIT14

Contributions to the repo is always welcomed, I hope to see the platform to be developed and achieved a more matured, refined design.

Contents

1 Working with STM32

This document will help with developing the control system for ADMIT14. The system will be based on a STM32F303K8 Nucleo board. With modifications, the system can be designed with any STM32 MCU, using the same methods in this document.

1.1 STM32 and STM32 Cube IDE

1.1.1 STM32 Microcontroller

The STM32 microcontroller family from STMicroelectronics is a series of 32-bit microcontrollers based on ARM-Cortex-M cores. As there are multiple microcontrollers in this series, choosing a suitable chip is important. The naming convention of the series is as follows STM32XYZZ, with X indicating the type, Y indicating the core of MCU, ZZ is the line of MCU.

The type includes:

- F: Foundation
- G: Mainstream
- L: Low power
- U: Ultra low power
- H: High performance

Letter Y in naming convention represents the core, Y can be from 0 to 7, higher number equals core with higher performance. F3 MCU or above is recommended for control system, as they provide necessary peripherals. ZZ is the line of MCU. MCU with line Z1 should be avoided, since they do not have CAN protocol peripheral, and an external CAN controller is required to connect the system to CAN Bus.

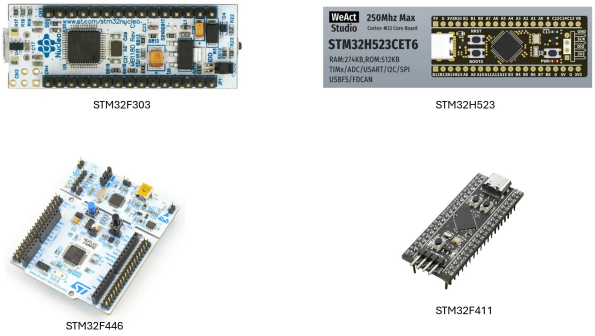

Figure 1.1: STM32 Boards

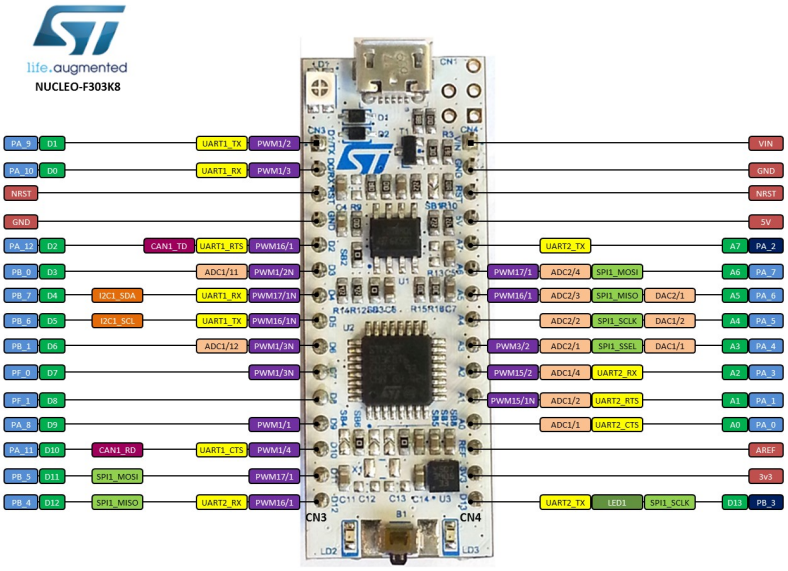

Figure 1.2: STM32F303 Pinout

Recommended MCU: STM32F446RE, STM32H523CET6,STM32F303K8. Avoid: STM32F01,F411.

For the Thesis, the F303K8 is used. In the future, a better board can be used such as the H523 board from Weact. Another option is designing a custom PCB that can use any MCU chip.

Figure **??** shows the pinout for F303 used in the Thesis.

1.1.2 STM32 Cube IDE

STM32CubeIDE is an integrated development environment (IDE) from STMicroelectronics, for programming and debugging STM32 MCU. Main functions of IDE includes:

- Setup peripherals, pinouts, clock of MCU
- Write code, firmware

- Debugging

Alternatives to developing firmware for STM32 includes: Simulink with STM32 Hardware support package, Arduino IDE.

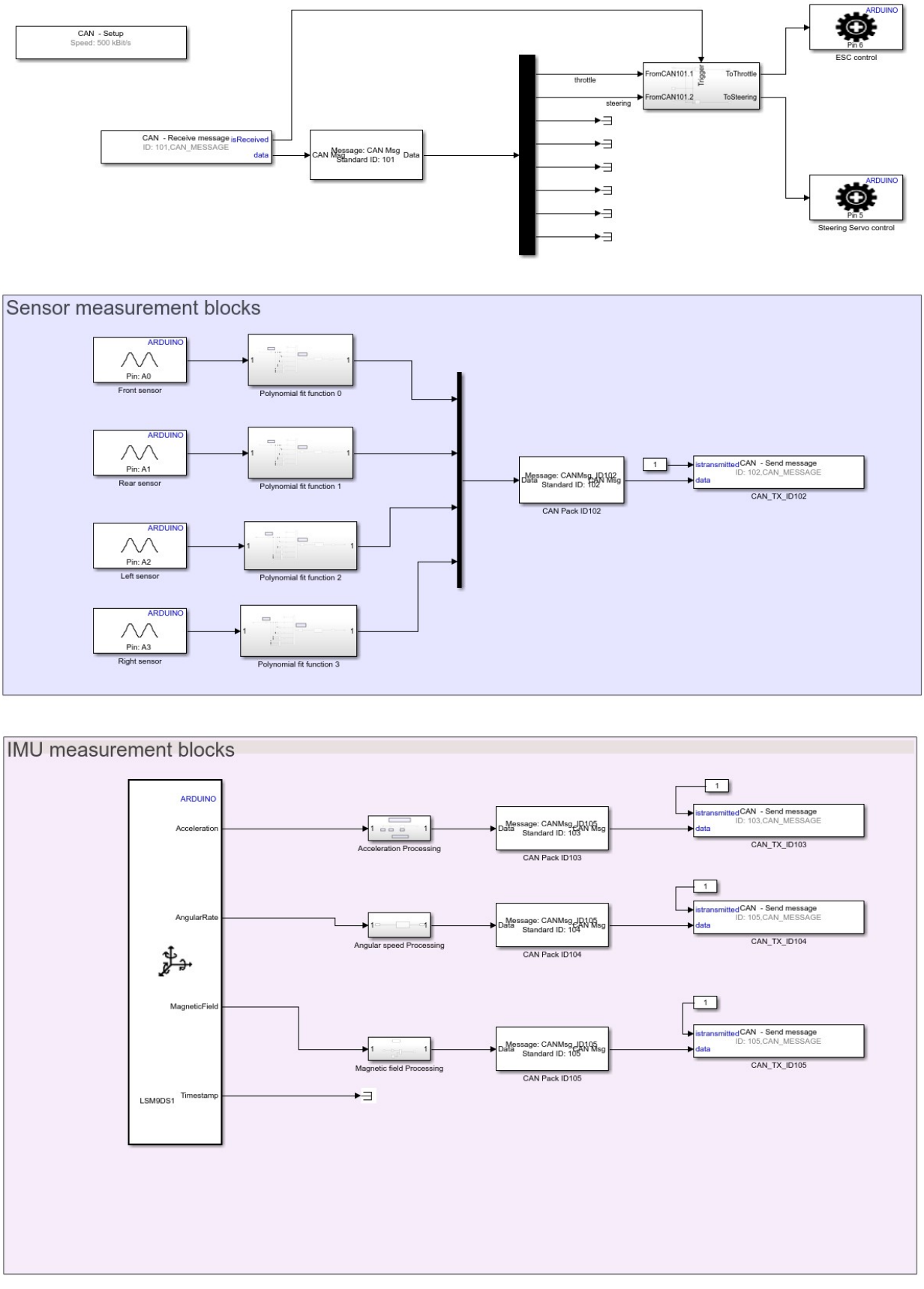

Figure 1.3: Model of Embedded Software for Microcontroller

1.2 Getting started with Cube IDE

This section will talk about developing the firmware for STM32F303 for the control system. Figure 1.3 shows the firmware using Model-based design method, made in Matlab Simulink. Originally the firmware on Simulink is used for Arduino UNO, but it can be replicated on STM32. The firmware can be divided into 3 parts:

- Receive message from CAN Bus and control motors
- Sample info from distance sensors and send to CAN Bus
- Sample from IMU and send to CAN Bus

To perform these functions, the MCU will need these peripherals: Analog-to-Digital Converter, or ADC; I2C for communication with IMU; CAN communication; PWM generation for controlling motors.

1.2.1 Peripheral initialization with CubeMX

After starting Cube IDE, the interface is shown in fig- 1.4. Click right mouse in the section under “Project explorer” to create new projects.

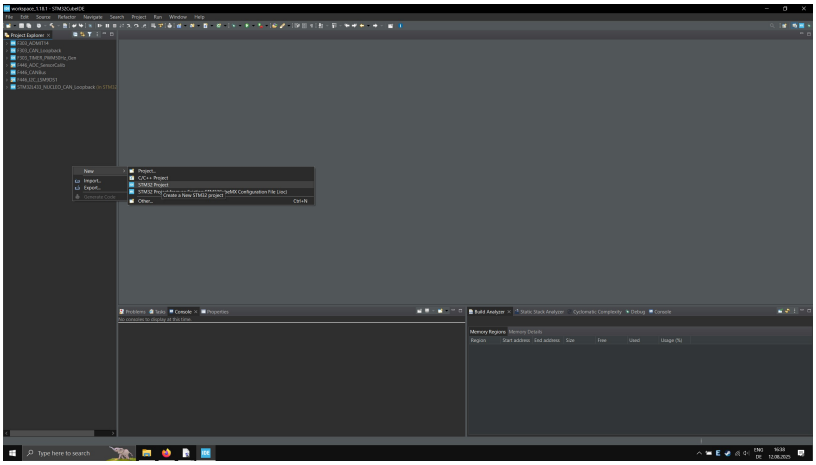

Figure 1.4: Interface of STM32 Cube IDE

Figure 1.5 shows the project initialization screen. Use the tab "MCU selector" or "Board selector" to select suitable MCU. "Board selector" is recommended, as using this option initialize necessary pins that connect to components already found on the board such as oscillator, LED, etc.

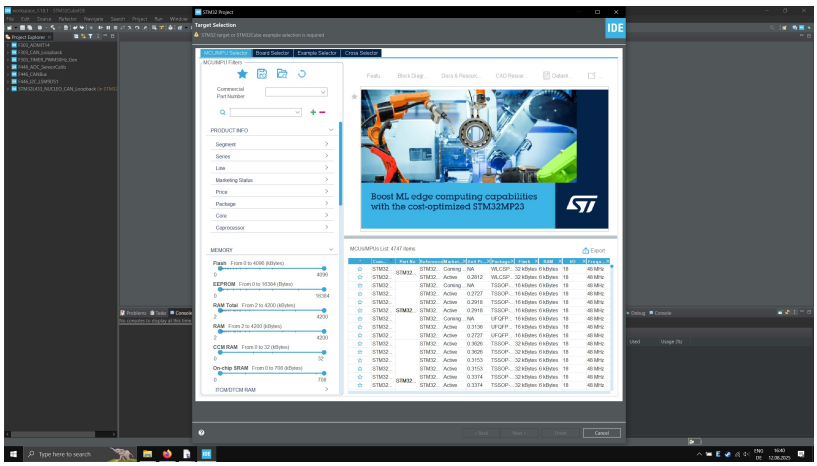

Figure 1.5: Project Initialization

After selecting the correct MCU, the interface will display the Pinouts in the IOC file, shown in Fig 1.6.

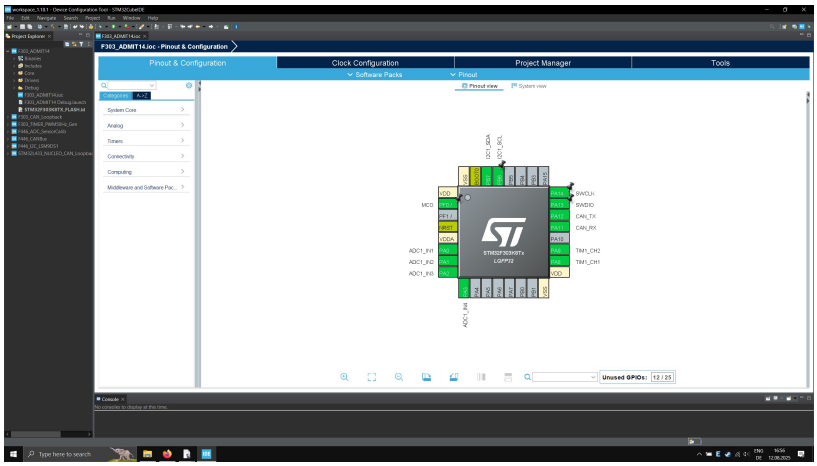

Figure 1.6: Editing IOC file

Before starting with initialization, in the IOC file screen, go to Project Manager ⇒Code Generator and enable "Generate peripheral initialization as a pair of .c/.h files per peripheral". This helps with cleaning main.c file and separating the peripheral codes from main code.

The first thing to consider is the MCU clock speed. The clock speed can be configured in the "Clock Configuration" tab. Typically there are 2 sources

for clock speed: internal and external. Internal source is found within the MCU itself, while external is usually an oscillator soldered on board. For the F303, the maximum clock allowed is 72MHz with external oscillator found on PCB board. This however require soldering to connect solder bridges on the board. The internal clock source has a frequency of 8MHz, and with PLL the maximum clock achievable with internal source is 64MHz. PLL is Phase-locked loop, used for multiplication of frequency.

Selecting suitable clock speed is important, as it decides the power consumption of MCU, as well as effectiveness of peripherals, as the peripherals' speeds are based on this clock speed. It is recommended to select the highest achievable clock speed for the MCU, then slowly decrease to a clock speed that is easily divisable for peripherals. Clock speed for peripherals will be discussed more in their dedicated sections.

1.3 Clock Setup

To setup clock speed, in "Pinout& Configuration" tab, select System Core→RCC. Depending on MCU model, there will be high-speed, low-speed external sources (HSE, LSE respectively) or both. For F303, there is only HSE. Select BYPASS Clock Source to use internal clock source.

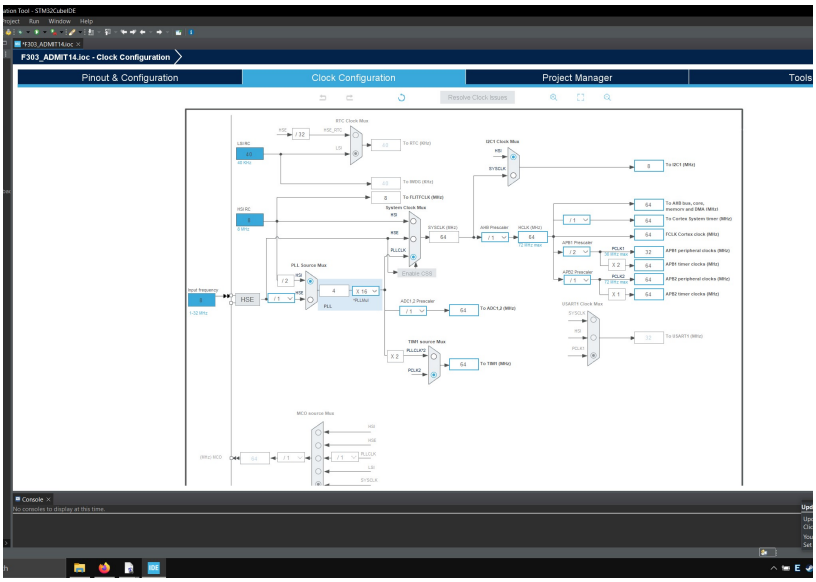

Figure 1.7: Clock configuration of F303

Figure 1.7 shows the configuration for the clock speed of F303. High speed internal source (HSI) has frequency of 8MHz, multiplied with PLL by 16 times, which results in a clock speed of 64MHz.

1.4 ADC setup and sensor interaction

The control system uses 4 distance sensors from Sharp to measure distance and detect objects surrounding the vehicle. For this we need to enable 4 pins of the MCU to connect to the output pin of the sensor. The sensor is Sharp GP2Y0A41SK0F, and its information is found in its datasheet[1].

1: Datasheet for Sharp sensor: https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a41sk_e.pdf

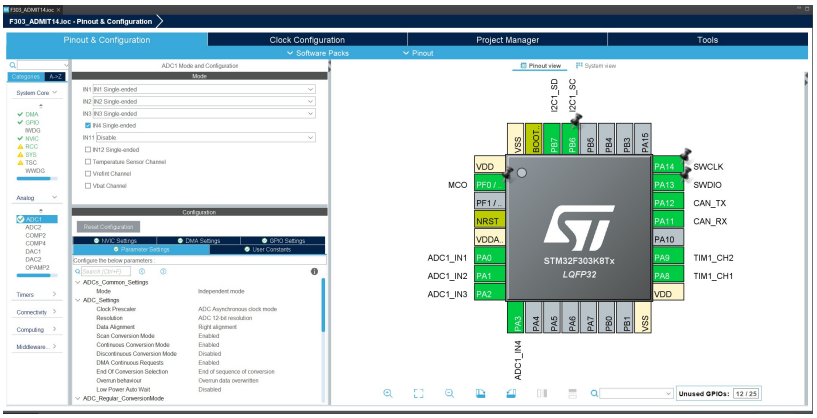

Figure 1.8: Setup of ADC,1

In total, 4 input pins are required to sample 4 sensors. They are IN1 to IN4, correspond to PA0, PA1, PA2, PA3 pins of MCU. There are 2 ways for conversions: Continuous and Discontinuous. Discontinuous will sample 4 pins once and stop, to continue sampling, extra line of code is included in main.c file to re-initiate the conversion process. On the other hand, in continuous mode, the conversion happens continuously and the conversion only needs to be initiated once. To setup, in the Configuration screen (Lower half , left side of fig. 1.8), choose "Enabled" for Continuous Conversion Mode, "Disabled" for Discontinuous Conversion Mode. An interrupt will be raised at the end of conversion to inform CPU about completion. This can be at the end of each conversion or at the end of sequence of conversion. Choose "end of sequence" for End of Conversion selection.

After the voltage is sampled by ADC, the accquired digital value will be saved to the memory of MCU. Typically there are 3 ways to do so: Polling, interrupt, DMA (Direct memory access)[2].

2: More information about these methods at https://deepbluembedded.com/stm32-adc-read-example-dma-interrupt-polling/

- In polling method, after each conversion, the value is manually assigned to a variable declared in the code.
- In interrupt method, after conversion, an interrupt is raised, triggering ISR (interrupt service routine) handler to handle this interrupt. The code in this handler is similar to the code in polling method, manually assigning value from ADC to declared variables.
- IN DMA method, the converted digital value can be directly assigned to variables in memory, no manual code needed, the CPU is not directly involved.

In the Thesis, DMA method is used, as the code is simple compared to other 2 methods. However DMA has limited resource, and the data

throughput is finite. For this reason, caution is needed when using DMA for other peripherals.

Following this, DMA configuration is needed [3]. Afterwards, the ranks for conversion need to be determined. The rank will decide the order of converted values in the array of variables. Rank 1 to 4 is Channel 3-1-4-2, corresponding to Front-Rear-Left-Right sensor. This order of conversion is influenced by the location of the pins and sensors. In each rank, we can set sampling time individually. If this is set too low, conversion will be finished in a short time, which will make interrupt call more frequent, affecting other functions.

3: DMA Setup followed in this video: https://www.youtube.com/watch?v=Re60MW2xNmM

After the voltage signals from sensors are sampled and saved in memory of MCU, they need to be converted to measured distance. To do this we need to use polynomial with sampled digital values as variables. The process to determine coefficients for the polynomial is sensor calibration [4]. The calibration process is discussed in the Thesis. To summarise the process: Record measured voltage and corresponding distance, use Curve fitter toolbox in MATLAB to generate coefficients, use coefficients to make function in main.c for calculation.

4: Calibration process: https://www.youtube.com/watch?v=Vtj0w2V1bRo

In main.c, the process for sensor measurement is as follows:

Declare 2 arrays: ADC_input[] and Dist[] →ADC initialization with HAL_ADC_Start_DMA() func, sampled value is assigned directly to ADC_input array →Each element of ADC_input is converted to distance with DistCalc func and assigned to Dist[]. CalcDist func returns:

- calculated distance-35 to fit into CAN message (limit 8 bit)
- 1 if distance is more than 290mm
- 0 if distance is less than 40mm

The Dist[] array contains the measured distances from 4 sensors. This array will be used to be packed into a CAN message and sent to CAN Bus.

1.5 I2C setup and IMU interaction

The Inertial Measurement Unit (IMU) used for the control system is LSM9DS1. This IMU can communicate using either SPI or I2C. I2C is used for the system, as it requires fewer connections (2 pins compared to up to 4 for SPI). The driver for this IMU is found in https://github.com/yazar/LSM9DS1-STM32-Library.

To enable I2C connection, in IOC edit screen, go to Connectivity →I2C1 and enable I2C. I2C runs in standard mode and timing configuration is kept as default.

IMU is initiated with 4 functions: begin(), setMagScale(), setAccelScale(), setGyroScale(). Scales are mentioned in datasheet. A higher scale means wider measurement range, with worse resolution.

After initialization, the IMU works normally. To access measured data, use readX functions. The function contain codes that send read request to IMU so the IMU will send out data from its registers. Up to 9 variables are needed for measurement: ax, ay, az for acceleration; mx, my, mz for magnetometer, gx, gy, gz for gyroscope.

Information from IMU is divided into 3 parts: acceleration, angular velocity (gyroscope), orientation in space (magnetometer). Each part is packed into 1 CAN message with unique ID.

1.6 PWM generation

The control system uses PWM signal to control external hardwares, such as servo motor and ESC.

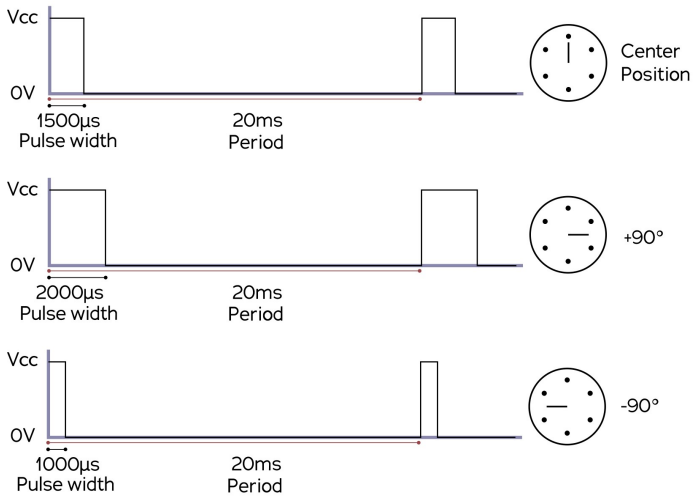

Figure 1.9: Servo Signal

Figure 1.9 shows the control signal for a servo. Signal has frequency of 50Hz or 20ms period, duty cycle varies from 10 to 20%, corresponds to full left and full right for the servo. The ESC is used to control propulsion motor and it also uses PWM as input signal with the same parameters. As such, the ESC can be treated as a servo, full throttle reverse and full throttle forward can be 10 to 20%, respectively. The Mamba ESC used in the Thesis will need calibration before operation. Calibration process is documented in its instruction. Currently, connection between Arduino-calibrated and STM32-calibrated ESC is not possible, and each MCU need a separate calibrated ESC. There are 2 calibrated ESC in locker, each for 1 MCU.

To initialize PWM generation, in IOC file edit screen, go to Timers →TIM1 →PWM Generation CH1 for Channel 1, PWM Generation CH2 for Channel 2. CH1 will be used for ESC control, while CH2 will be for servo control.

Calculation for PWM generation: PWM takes APB2 clock as clock source. The value of APB2 is displayed in right side of Clock screen in fig. 1.7.

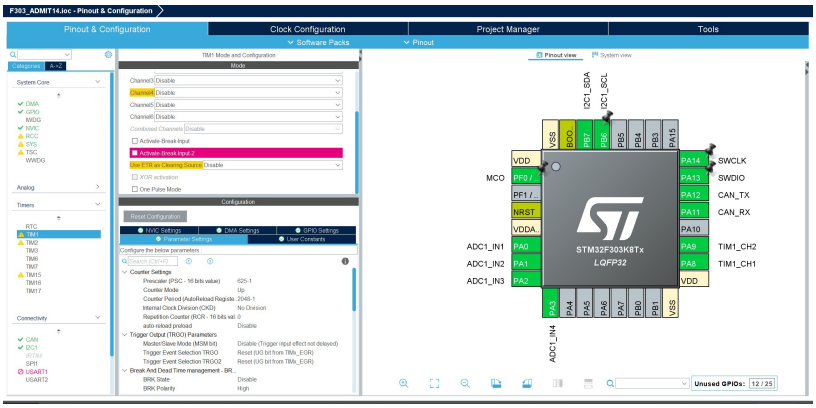

Figure 1.10: PWM signal configuration screen

Figure 1.10 shows the configuration screen for PWM. The parameters to adjust are: Prescaler or PSC, counter period or AutoReload Register or ARR, and Capture Compare Register or CCRx.

First, Timer peripheral clock is calculated by:

$$TIM\_CLOCK = \frac{APB2}{PSC+1}$$

PSC is the prescaler value, which can be set in IOC file edit. With APB2=64MHz, PSC set to 624, TIM_CLOCK becomes 102400Hz.

Next, the signal frequency is calculated by:

$$Freq = \frac{TIM\_CLOCK}{ARR+1}$$

To achieve 50Hz frequency, set ARR to 2047. ARR determine the resolution of duty cycle. Higher ARR value results in more step for controlling between 0% to 100%. After these 2 calculation, a PWM signal with frequency of 50Hz can be generated.

To determine duty cycle for PWM signal, we use registers CCRx, with x corresponds to the PWM channel; CCR1 controls PWM signal at channel 1, CCR2 for channel 2. Duty cycle is calculated by:

$$DutyCycle(\%) = \frac{CCRx}{ARR+1} * 100$$

to have a duty cycle from 5 to 10%, the CCRx values should be from 103 to 204 for throttle, and 103 to 223 for steering. Steering need higher value because swapping servo to right side changes geometry of steering system, and it requires more steering to one side.

The system will receive CAN message with ID 101 from the bus. This message contains information about desired throttle and steering values. We use these values to manipulate CCRx registers, controlling duty cycle. In main.c file, the ConvertToPWM_Signal will be used to convert the number in CAN message to appropriate CCRx value. TIM1→CCRx is used to assign value to these registers.

1.7 CAN protocol setup

Final part of working with firmware is setting up the CAN protocol. In Connectivity, choose CAN and then check box to Activate. In NVIC settings, enable CAN RX0 interrupt. This is for detecting incoming CAN messages.

In parameter settings, the parameters are Prescaler, Time Quanta Bit segment 1 and 2. The desired baud rate for CAN Bus is 500kb/s[5].

5: Calculation for baud rate http://www.bittiming.can-wiki.info/ and https://community.st.com/t5/stm32-mcus-products/can-bus-boudrate-calculate/td-p/382923?lightbox-message-images-382928=42250i4CECDEA5C8955E57, https://community.st.com/t5/stm32-mcus/can-bxcan-bit-time-configuration-on-stm32-mcus/ta-p/689864

First, calculate time quantum by:

$$t_q = \frac{Prescaler}{APB1\_Peripheral}$$

With APB1 at 32MHz, Prescaler 4, the time quantum is 125ns. Next we calculate Nominal bit time.

$$NominalBitTime = 1 * t_q + BS1 * t_q + BS2 * t_q$$

with BS1 and BS2 as coefficients for bit segment 1 and bit segment 2. BS1 should be 75 to 80% of BS1+BS2 to have suitable sampling point position. With BS1=11, BS2=4, the Nominal bit time will be 2000ns, or one bit is transmitted in 2000ns. This results in a baud rate of 500kb/s.

This concludes the setup process for CAN protocol.

In main.c file, there will be headers declared for CAN messages. Each header will contain standard ID and Data length DLC. Header is used during CAN message transmission, each header is attached to appropriate data to generate a CAN message. 4 headers for: Sensor message, IMU_Accel message, IMU_Gyro message, IMU_Magne message.

CAN Message is transmitted with function HAL_CAN_AddTxMessage().

To receive CAN message, we need to setup CAN Filter. This will be used to read and filter desired message with declared IDs. Filter is declared in form of struct with multiple informations. The most important information is the FilterID variables. Currently the system only needs to receive CAN message with ID 101 which contains desired control values for servo and propulsion motor. To do this set FilterIdHigh and FilterIdLow in canfilterconfig to 0x65, this is hexadecimal value for 101. FilterMaskId can be ignored, as the filter mode is set to id list and not id mask[6].

6: Further information about CAN filtering: https://schulz-m.github.io/2017/03/23/stm32-can-id-filter/

STM32 cannot directly connect to the CAN bus and a CAN Transceiver is required. Any 3.3V transceiver is sufficient, such as SN65HVD230D.

2 Electrical Design, Digital signal processing

This chapter talks about the Electrical design of the car.

2.1 Power supply

On the ADMIT14 car, there will be 2 batteries with about 12V voltage. One of them will be used to power the Jetson Orin Nano, while the other will be used to power the propulsion motor, servo and the control system. One buck-boost converter will be used to step down voltage from 12V to 5V. This will be used to power 4 sensors and the MCU. MCU can be powered using either 3.3V, 5V or Vin pins. In this case, the NRST pin needs to be pulled up to 3.3V. Simply short the NRST to 3.3V pin with a 10kΩresistor. Alternatively, we can cut a micro USB, exposing Ground and 5V wires and connect to buck converter as power supply. In this case, the pull up resistor for NRST pin is not needed. By powering the MCU with USB, the ST-Link onboard is also powered, this pulls the NRST pin to 3.3V automatically. When MCU is powered with 3.3V, 5V or Vin pin, ST-Link is not powered and NRST pin is pulled to low by default, forcing the MCU into a reset loop and need NRST pulled up to 3.3V to operate.

2.2 Connection of components

To help with connecting components to the control system, we should use a perf board and solder on female headers as connections for components.

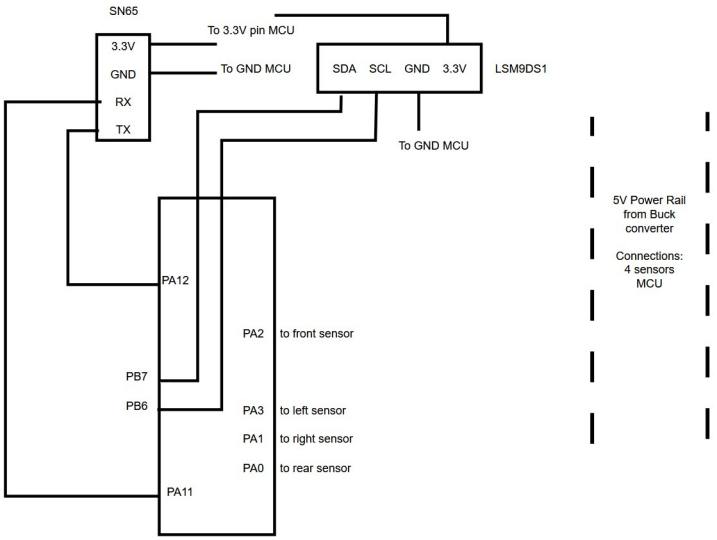

Figure 2.1: Connection of components

The connections for components are displayed in fig. 2.1. Pin names is also in the figure.

2.3 Filtering for ADC

I have tried applying an analog low pass filter, 100 Ω resistor and 100 μ F capacitor. Result and comparison is shown in fig. 2.2.

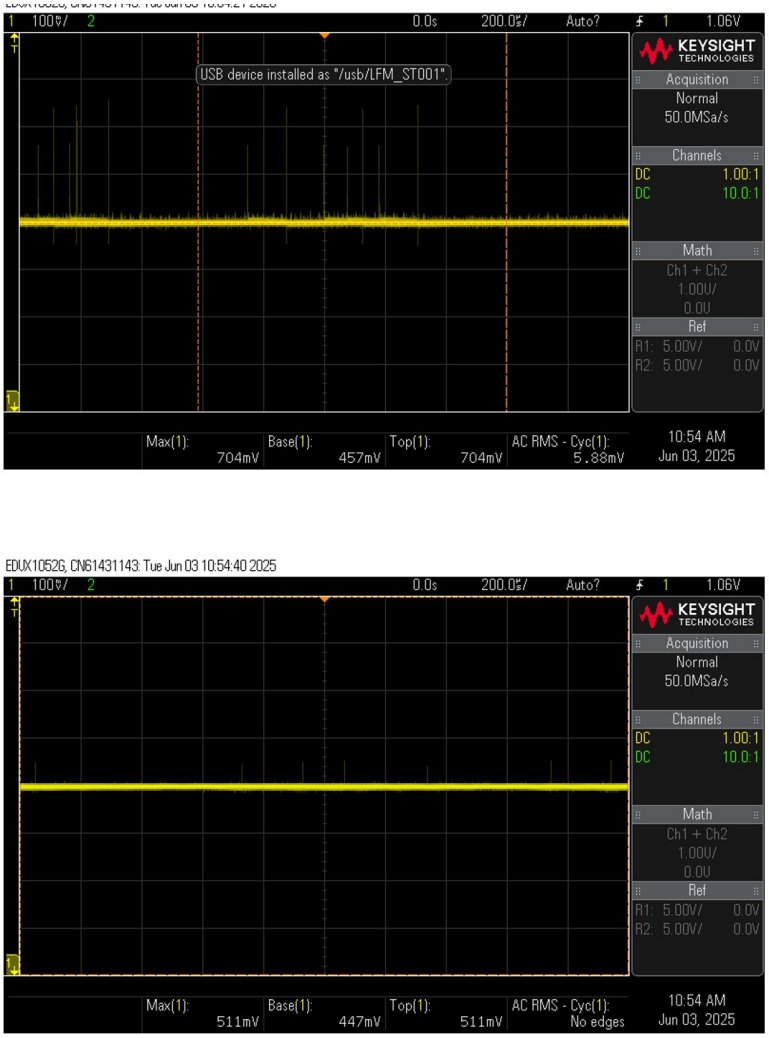

Figure 2.2: Sensor output, without (above) and with (below) low pass filter

However, I did not go too deep into this, and further experiment is needed. More suitable cut-off frequency is probably needed. Currently it is around 15Hz with above setup. In the datasheet of sensor, it states in the timing chart section that the sensor output around every 25ms, so around 40Hz?. Maybe try a LPF with a cut-off freq around this and see the response of output signal from sensor. Too low cut-off freq may make signal change become sluggish.

Additionally, digital filters can also be applied to further process signal. Notable example is median filter: https://github.com/mhtb32/EfficientMovingAverage.

Update 09.10.2025: Median filter is implemented for firmware.

Additions: Include header and source files named MedianFilter to the STM32 project. Changes to how the sensor output is sampled and calculated: Now DMA will operate in one-shot mode: At the beginning of DistSensTask, the DMA will be called to sample 4 channels. After finishing sampling and moving data to memory, a Call back is raised, signaling completion of DMA and DMA will stop. This Call back will signal MCU to calculate filtered ADC value. Look for HAL_ADC_ConvCpltCallback function at the end of freertos.c.

MedianFilter library: Contain definition for struct FilterTypeDef and 2 functions. The struct has 3 elements: Filter_Elements to store values for filtering, Sum of this array, WindowPointer to point to the current element.

2 Functions: Init and Compute. Init has input of pointer to a struct of type FilterTypeDef, used to initialize elements of struct, set array and sum to 0.

Compute calculate sum and fill the array each time it is called, cycle along the array. Compute has 2 inputs: raw_data and pointer to struct FilterTypeDef. Compute output filtered value, or Sum/WindowLength. WindowLength is defined in header, can be changed.

2.4 Processing IMU Output data

Processing data from IMU by the microcontroller should be considered, to reduce workload for central computer. Look for quarternion-based kalman filter.

3 Mechanical Design

This chapter talks about the mechanical design of the car.

With the replacement of the motor, the new motor will interfere with the servo. To fix this, we need to move the servo to the right side of the car.

Remove battery tray and drill holes on the car's plate on the right side to attach the servo. In total, 3 holes need to be drilled, old screw can be used to secure the servo to the chassis. The servo is connected to the steering system by a linkage and rocker arm. After changing the position of servo, we need a new, mirrored rocker arm, such as infig 3.1

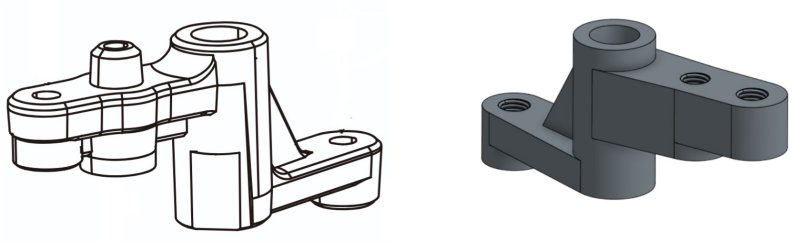

Figure 3.1: Original (left) and 3D-printed (right) rocker arms

CAD Model is available. All parameters are maintained relative to original rocker arm.

Next will be the plattorms for mounting hardwares. The car will consist of 2 platforms, attached to the chasis with two brackets. Both brackets have CAD models available.

Two brackets are needed. Each bracket will have two metal threaded rods with a diameter of 5mm. The rods can either be threaded directly into the bracket or we can use thread-insert [1] if the 3D print cannot print threads for the plastic parts.

1: Thread insert for 3D printed parts:https://formlabs.com/de/blog/schraubgewinde-fuer-3d-Druckteile/

The first platform is used to mount the batteries (bottom), Jetson and control system PCB (top). The items is displayed in fig.3.2.

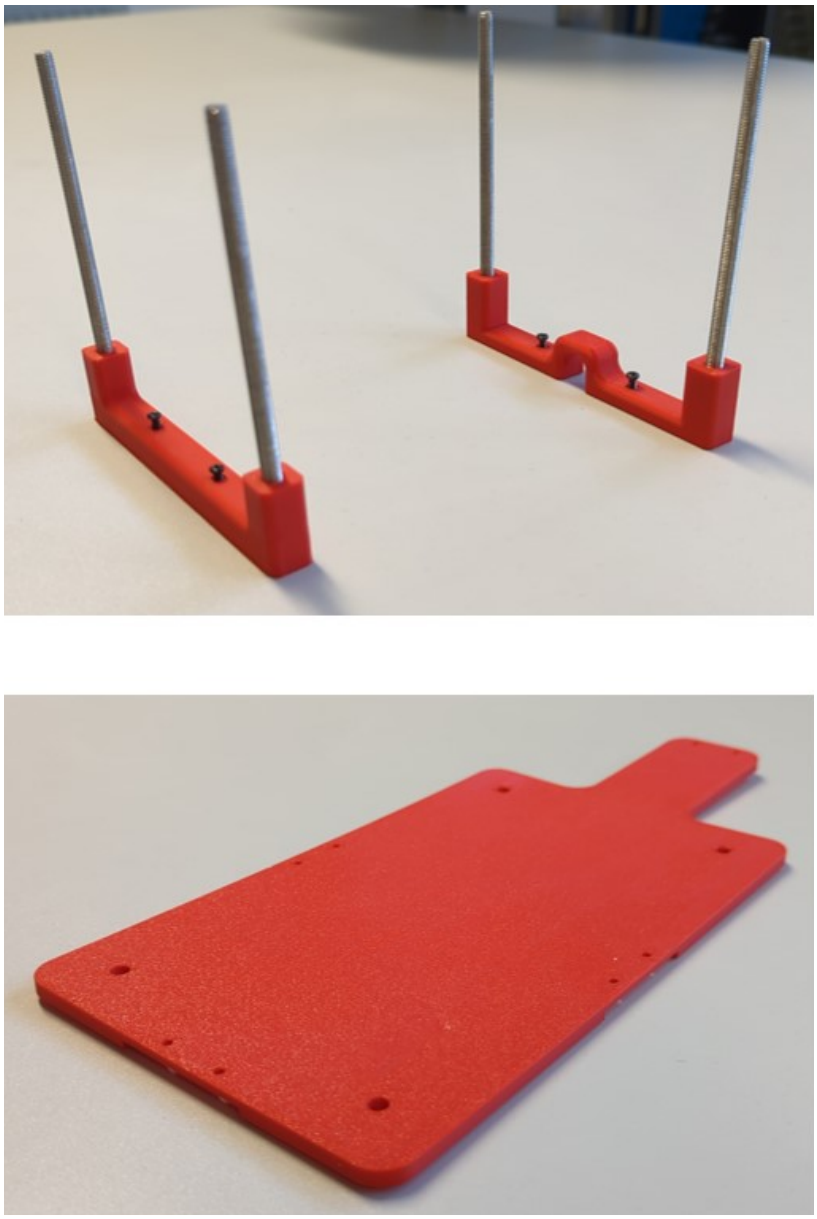

Figure 3.2: Brackets and platform

Features of 2 platforms:

- Both platforms have 4 holes to mount. Bolt pattern is a rectangle. Width of this rectangle is based on the distance between 2 metal rods of the bracket. Currently this is 100mm. Can be changed if there is problem with fitting parts on platform. Length is around $195 - 200mm$, based on the distance between two brackets.
- First platform has 4 attachment points, pattern $60 \times 60mm$ to attach control system PCB
- First platform has 4 slots to attach mounts for distance sensors.

- 6 additional holes, 3 each side to use velcro strips to secure 2 batteries to the platform

Second platform has not been designed. However there are a few notes:

- Used for mounting camera and LiDAR. The LiDAR should be mounted higher than the camera for it to function properly without being ostructed by the camera.
- To properly separate car functions from autonomous driving functions, I think the Jetson should be mounted to this second platform instead of first platform. Mounting points for Jetson should be considered, and the Jetson may be mounted up-side-down at the bottom of second platform.

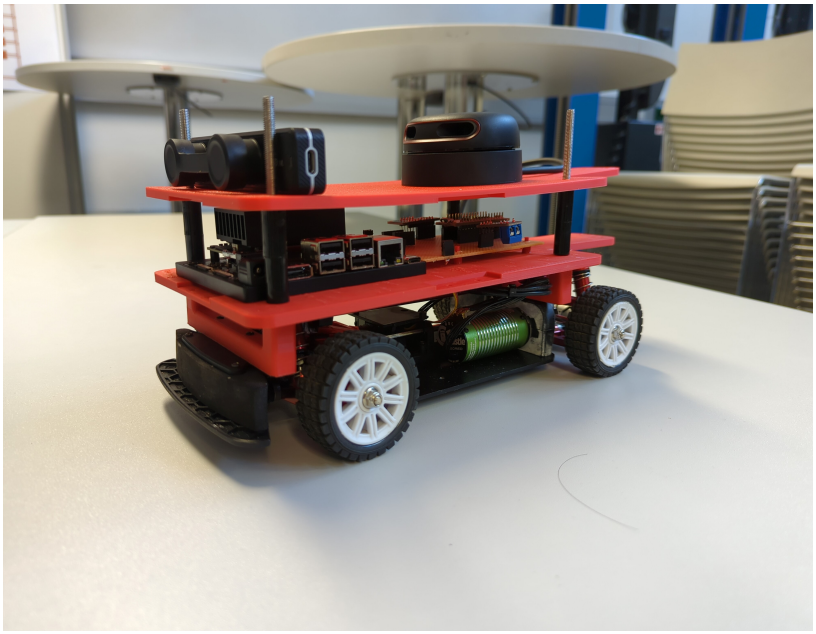

Figure 3.3: Complete car

The completed car should be like in fig.3.3

4 Future development

Future developments for the system:

- Improve electronic design
- Improve sensor suite
- Improve mechanical design

4.1 Electronic design

For signal integrity, it is better to design PCB for the control system. Currently, the components such as IMU is attached to perf board by rows of female headers. The pins of the IMU cannot mate completely with the headers and is not firmly attached to the car. This will affect the accuracy of this sensor.

With a custom PCB, it is possible to choose a different, more powerful MCU for the control system. The F303 offers enough performance for the time being, but if we want to improve sensor suite or implement filter algorithms, it may not be sufficient[1].

1: One good option is the STM32H523 black pill, cheap, high clock speed, FDCAN. https://docs.zephyrproject.org/latest/boards/weact/blackpill_h523ce/doc/index.html

One solution is to follow PCB design of this project: https://github.com/aktasenes/BLDC_driver. Solder SMD components on PCB and use headers to attach the MCU board.

We also need to consider new safety features for the control system. Currently the system has 1 safety feature: put throttle and steering value to neutral if connection to CAN bus is lost. The system checks if the throttle and steering value do not change after 500ms, then it will halt the car. I do not know how the self driving algorithm works and obviously this would not work if the algorithm of self driving car does not constantly send control signal. One way to fix this is that we can program the main computer such that it sends a dummy, low priority CAN message to check independent of other CAN messages.

Firmware wise, we can change and develop from the FreeRTOS firmware in my github. Both firmwares function the same but FreeRTOS version has better readability, and expanding firmware with FreeRTOS is easier to track. With larger, more complex firmware, a RTOS helps with determinism, tasks in the loop run on time.

4.2 Sensor suite

Currently the car has: 4 distance sensors and IMU. The motor from Castle Creations has 3 Hall sensors to detect motor position. However the output of these 3 sensors need to be amplified to be readable. I have yet found a solution to this. For detecting motor position, there are projects based on STM32, and we can use this to learn their algorithms: https://github.com/dimitris12kara/Hall_effect_sensor_STM32/tree/main.

I have implemented low pass filter for the distance sensor. The result was that it could reduce the noise for the sensor's output. However there is no more space on the perf board so solder the resistor and capacitor on. This can be fixed with a custom PCB, as the SMD components are smaller. Another solution is using Kalman filter or median filter. Kalman filter algorithm example: https://github.com/ibrahimcahit/STM32_MPU6050_KalmanFilter/blob/main/Core/Src/main.c. The same can be applied to the outputs of IMU, processing these sensor measurements will help reducing load on main computer.

For Kalman filter: change DMA continuous to discontinuous. Workflow: start sample →end sampling (interrupt raised) →KF →send results to CAN Bus →change to next task until start a new sample. Maybe change order, separate sampling and KF calculation into different tasks to save resource, avoid waiting time, optimize task flow? End sampling interrupt can be used for preemptive task scheduling?

New sensors to consider: battery temperature and SoC sensors[2]. Maybe make the control system stop the car if battery is low?

2: https://arxiv.org/pdf/2504.05936

4.3 Mechanical design

Design second platform to mount camera, LiDAR.

Current 1st platform has holes for the batteries at the wrong position. Currently the battery stays to close to front wheels. I have fixed this in the CAD model on Onshape, but have not printed out.

Steering system has a lot of free-play, even with metal parts. Currently no solution

Improve wiring for all components on the car. Maybe change servo motor for better response.

Suggestion for attaching jetson orin nano to platform: remove plastic frame, use screw hole to directly attach board to platform. However the atenna for wifi card is attached to the plastic frame. External attenna can be used, like in F1Tenth build[3].

Move to 1:10 scale?

3: F1Tenth build, attena part:https://f1tenth.readthedocs.io/en/stable/getting_started/build_car/autonomy_elements.html#preparing-the-wifi-antenna

References