UNIVERSITÀ DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

FINAL DISSERTATION

Embedded Systems in a Racing EV Prototype: Firmware and Architecture of its DAS and ECU

Supervisors

Paolo Bosetti, Luigi Palopoli

Student

Alessandro Sartori

Academic Year 2021/22

Acknowledgments

As this is not only a purely academical achievement, but also a personal celebration of the accomplishment of my efforts, I wish to use this small space to embed in this work the names of all those people who, during my life, have had the most incisive influences and led me to who I am today.

Obligatory beginning is family: being grateful towards an unconditional type of love is often hard, but if there's one person whose love I know I will never meet the limit to, he is my father. At any time I felt alone, or misplaced on this Earth, I could only carry on when recalling how he looks at me: sometimes, when looking a parent into their eyes, you can really tell how to them you are their entire world, and this is simply the most powerful and instinctive push to keep trying I have ever experienced, because somehow, there is a worth you will never ever loose. The same holds for my mother and most relatives, with some special mentions due: my grandfather Leone and my uncle Omar have perhaps been my closest and most valued examples in life, always proving how an open and sweet heart can be a surprising point of strength, by all means. Aunts Lucia and Ileana also deserve a spot in this paragraph, as two of the women I most respect for their life styles and personalities. Last but not least on the family paragraph is uncle Esteban, possibly the very person who made me fall in love with Computer Science: as a kid I died of curiosity towards any technical field, from mechanics to electronics to robotics. Each one seemed an infinite world to explore, until the day of my Confirmation when Esteban gifted me a book about HTML 4 and one about Java. Being still at elementary school, I could only comprehend a small part of those magic books, but something really sparked in me and I realized that this particular infinite world to explore was somehow different from all the others, as if it was even more infinite and full of secrets to unveil. To this day, I'm still convinced that that is the day in which my path of Computer Scientist had begun.

On the friendships side the list would grow enormous, so I'll try to slim it down and only mention those who really got into my heart over the years. In approximate chronological order, I'll start with Arianna, one of those acquaintances that are woven so deep into your life that you even forget how you met them in the first place. Despite our dominant characters often clashing into arguments, Arianna holds a special place in my memories for how much she's been close to me when I needed a friendly presence the most. She helped me get through my unspeakably worst years and, safe to say, I might very well be still alive thanks to her. Many other people nonetheless have been important cornerstones for understanding how to build my life during the same years: most notably, Luca with his unmatchable spirit and always wide-open arms, and Niccolò with his admirable rigor. After them, a whole lot of new acquaintances appeared in my life: I want to especially remember Claudia, a person so interesting to me that somehow turned my life around for the best without even doing anything.

Fast-forward to today, and the friendships I'm most grateful for are two: Emma, whose personality always manages to make my day, and all of my Formula-SAE buddies with which I've spent some of the best afternoons of my life, in a place where one can really feel like being at his very right place.

Contents

Abstract

Formula SAE is one of the most challenging, fun, and ambitious projects that a student can take part in. It's a worldwide championship between college Universities where each team has the objective of designing, building, and racing a formula-style vehicle.

In this work, after an obligatory introduction on the topic, the team, and the vehicle being designed (Chapters 1 and 2), the software aspects of the prototyping process will be examined thoroughly, with particular focus on microcontrollers and embedded systems. Successively, a chapter will be dedicated to the **TECS** unit, the primary subject of this thesis. The **Traction and Electronic Control System** is an on-board device composed of a Data Acquisition System (DAS) and an Electronic Control Unit (ECU). The former is a Micro Controller Unit (MCU) responsible for reading many sensors' data, operating the inverters and the motors, and keeping the vehicle in a coherent and safe state at all times, while the latter is a Single Board Computer (SBC) acting on the driver's input to elaborate traction control strategies that the DAS will apply.

On one hand, a remarkable dedication has been put in the development process of hardware and firmware for said devices, but a perhaps even greater effort has been dedicated to always aim at producing solid code with efficient troubleshooting and maintenance schemes. For this reason, Chapter 4 contains an overview of all practices that have been adopted to reach this goal with more proficiency, while Chapter 5 reports a document that has been produced to aggregate all solutions and relative explanations to several problems that have been encountered during design or development.

1 Introduction

1.1 What is Formula SAE

In the motorsport world, formula-style championships range from widely known open-wheeled single-seater races (such as Formula One or Formula E) to less acclaimed events, as Formula SAE often is. Said competition requires students to gather in teams and design, develop, and test a prototype to be evaluated by judges under many aspects, during static and dynamic events. In static events the teams pitch their ideas and solutions and receive points according to how effective and innovative these prove to be, while in dynamic events the vehicles are put on track and driven (again by students) across different challenges to test their performance. Additionally, extra points may be awarded for superior design accomplishments, regarding for instance innovative uses of electronics or noteworthy analytical approaches.

Being part of an F-SAE team is an incredible opportunity in a student's life, as it can give a unique real-world approach on the engineering world and strengthen their project-management and problem-solving skills. Furthermore, industries show as well a notable interest in this matter, since it provides them with already well-formed and very experienced candidates.

1.1.1 Structure of a Competition

As premised, F-SAE competitions are composed of several types of challenges, beginning with a thorough inspection to ensure that all vehicles comply with safety regulations and successively proceeding with the static and dynamic tests.

Technical Inspection

In order to take part in the dynamic on-track events, a car must be approved in all the following stages:

- **Pre-Inspection**: all equipment is checked for compliance with the SAE[1] and FIA[2] rules.
- **Accumulator Inspection**: the high-voltage battery pack is again checked for compliance with the rules (specifically its insulation and data acquisition methods) and permanently sealed by the judges for the rest of the competition to avoid later tampering.
- **Electrical Inspection**: every electrical component's datasheet is verified to meet the safety requirements, plus grounding effectiveness is ensured across all exposed conductive surfaces.
- **Mechanical Inspection**: specifications of the chassis materials are shown to the judges and the vehicle dimensions are checked to be compatible with the rulebook.
- **Tilt Test**: a special platform inclines the car with the driver seated (at an angle of 60 degrees) and no slip or fluid leakage must be observed.
- **Rain Test**: proper insulation of every electrical component is ensured by sprinkling the powered-on vehicle with water from every direction for two minutes. Humidity resilience is confirmed if power is kept on for an additional 120 seconds.
- **Brake Test**: the driver accelerates the car and performs an emergency brake test. All four wheels must lock and no deviation from a straight path must be observed.

Static Events

Static events aim at evaluating the efficiency of the team in theoretical aspects such as organization, business management or design processes. Point are awarded for the following challenges:

- **Engineering Design**: judges assess each work-group's design processes and choices, which have to be explained and possibly justified with data and simulations. The evaluation spans from the early stages of design up to the final realization and testing methods.
- **Cost and Manufacturing**: this presentation is focused on the discussion of a Bill Of Materials (BOM) in which the cost of every component or manufacturing process has to be reported and justified with respect to possible equivalent alternatives.
- **Business Plan Presentation**: officials impersonate potential investors to whom the team has to pitch their business model with which they plan to mass-produce the prototype. Costs and profitability are key points, whereas original ideas and innovation are appreciated and often rewarded.

Dynamic Events

In dynamic events the vehicle is put on track and four different performance aspects are put to test:

- **Skidpad**: the car must complete and 8-shaped path twice, occasionally on damp tarmac in the shortest time possible.
- **Acceleration**: timing is measured from a standing start to the completion of a 75 meters long straight line.
- **Autocross**: a track shorter than 1.5 kilometers must be completed in the least time possible, while facing slaloms, straights, and sharp turns.
- **Endurance**: a track laid out similarly to the Autocross setting has to be run for 22 kilometers and the efficiency in relation to the timing of the vehicle is awarded.

[1]Society of Automotive Engineers
[2]Fédération Internationale de l'Automobile

1.2 UniTN's *E-Agle Trento Racing Team*

Since 2016, the University of Trento has been participating to this world-known competition with a racing team named **E-Agle Trento Racing Team**, where the "E-" prefix is meant to hint at the category it belongs, the one of **Electric Vehicles**.

1.2.1 Organization

Being composed of more than 70 students from 5 different departments, coordination of work and people is key. A directive board acts as **Core Team** and is composed by the following parties:

- The **Faculty Advisor** (prof. Paolo Bosetti)
- One **Team Leader** (TL)
- Five **Chief Technicians** (CTs) for the respective technical areas
- One **Chief Technician** for the business-marketing area
- One **Treasurer**

Each of the five CTs oversees their specific technical area:

- **DMT**: Dynamics and Modeling Team
- **MT**: Mechanics Team
- **EET-HW**: Electric & Electronics Team - Hardware
- **EET-SW**: Electric & Electronics Team - Software
- **DV**: Driverless Team
- **EMT**: Economics and Marketing Team

Each of these teams is then subdivided in work-groups which at times intersect (e.g. for SW and HW aspects) and are coordinated by a dedicated **Project Manager** (PM):

- Accumulator - High Voltage
- Accumulator - Low Voltage
- Aerodynamics
- Chassis
- ECU
- Powertrain
- Steering
- Steering-Wheel
- Wheels
- Suspensions
- Telemetry
- Handcart
- Wiring
- Sponsorships
- Marketing
- Business Plan

Additionally, whenever the vehicle is expected to participate to an event, a parallel **Racing Team** (RT) is assembled comprising a number of drivers, at least one representative per area (DMT, EMT, MT, HW, SW, DV), and in general anyone essential to an efficient operation of the car and analysis of its data. In addition to technical duties related to the prototype, the RT head needs to ensure proper preparation of the rest of team members for the FSAE event, including a thorough understanding of the rulebook, of any safety procedure, and how to generally behave and operate at each type of event.

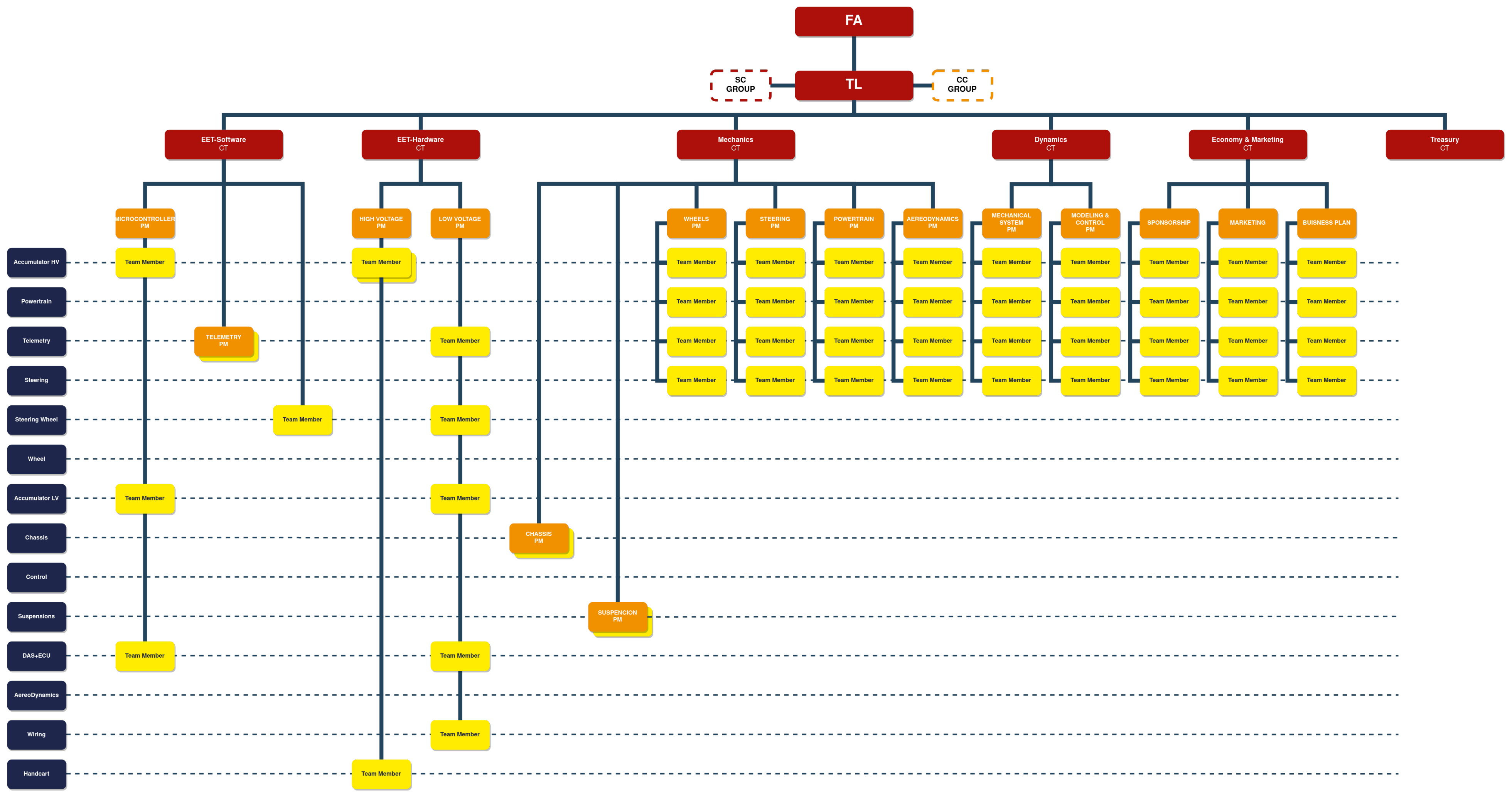


Figure 1.1: Full organogram of the team

1.2.2 History

The history of the team begins in summer 2016 when it was founded, on initiative of professor Paolo Bosetti, now Faculty Advisor.

By 2017 the first prototype, **Chimera**, was ready to compete in its first event: Formula SAE Italy in Varano De' Melegari. Although it failed the technical inspection and could not race, it nonetheless earned the team a trophy for best telemetry system and showed off its innovative tubular chassis built with 3D-printed steel joints (Figure 1.2).

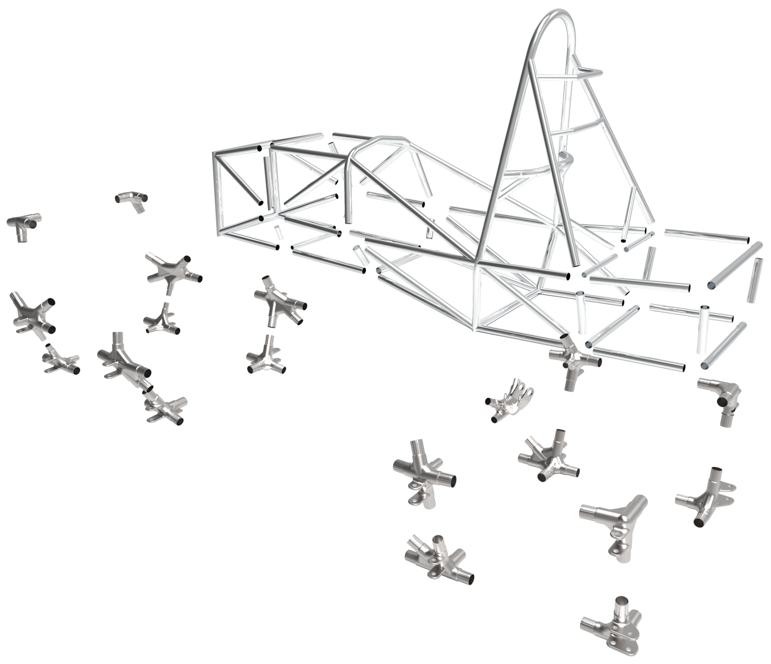


Figure 1.2: Chimera's chassis with 3D-printed steel joints

For the 2018 season Chimera underwent many notable design improvements, ranging from powertrain and battery pack performance tweaks to the inclusion of aerodynamic elements in carbon fiber composite materials, reaching a mass of 243 kg. Renamed **Chimera Evoluzione**, at Varano the vehicle didn't qualify for the the technical inspection once again, but caught the attention of the jury and won the prize for best Human-Machine-Interaction solution, awarded by *Lamborghini Automobili*. In Barcelona instead, it ranked 4th place at the Business Plan event.

Figure 1.3: Chimera Evoluzione

In 2019 the team began the design of yet another prototype, named Fenice to underline its birth from the metaphorical ashes of Chimera Evoluzione (even though it later found new life in the Driverless Cup). At present, due to the many consequences of the SARS-CoV-2 pandemic, Fenice is still under heavy development, and therefore, the subject of this thesis' work, thoroughly described in the chapters that follow.

The primary goal for this vehicle was to keep its mass below 200 kg with the use of a hybrid chassis, composed of a primary tubular structure with 3D printed steel joints plus a rear carbon fiber monocoque to support the posterior axle. The adaptor gear setup has been redesigned as well to reduce its volume and mass, opting for epicycloidal gears. Wheel rims are now laminated in carbon fiber as well, and brake calipers have been designed with generative techniques and 3D printed directly in metal. Moreover, the driver seat is now incorporated to the firewall, and motors have been upgraded to the Emrax 188 model.

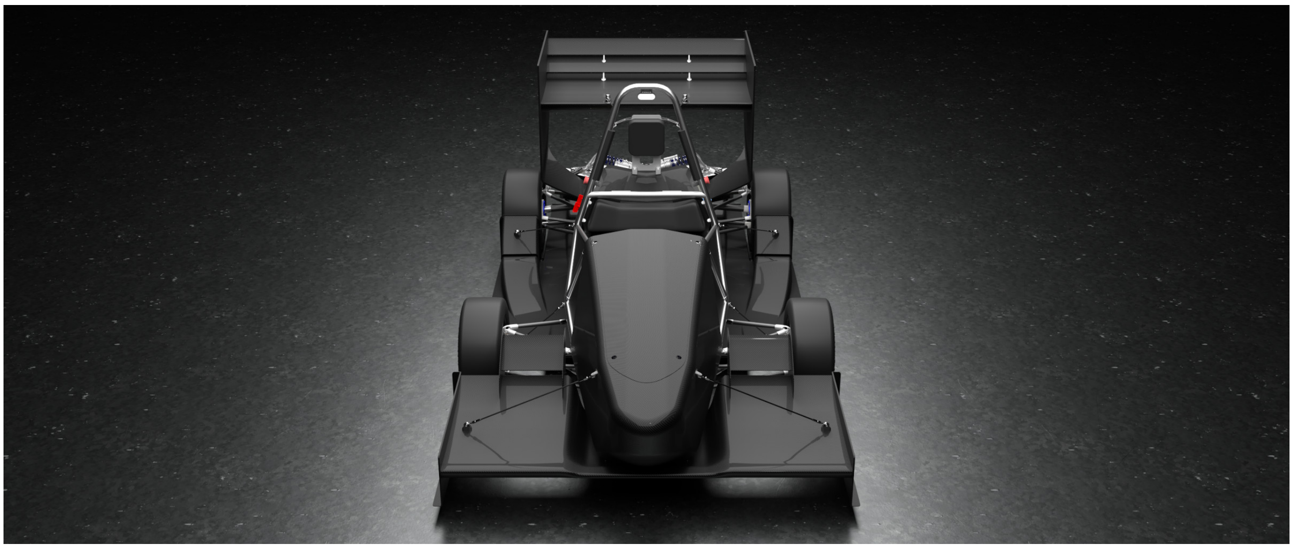


Figure 1.4: Render of Fenice - Overview

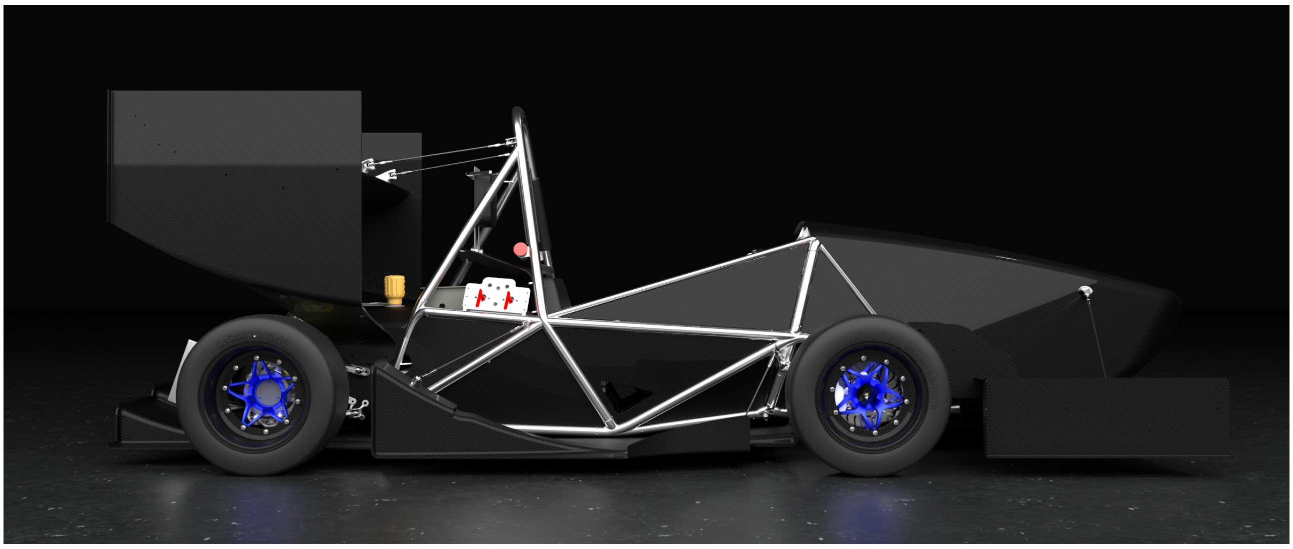


Figure 1.5: Render of Fenice - Side view

1.2.3 Team Values and Priorities

One of the most felt values of the team is perhaps the will to design and build by ourselves as many components as we can. As opposed to many others, rather that buying existing and complete modules from industries, we prefer spending several iterations on prototyping the right PCB or mechanical component, therefore having the added bonus of complete and extreme flexibility on their use, configuration, expansion and, most importantly, thorough knowledge of the underlying hardware or running software. Secondly, in terms of software, we are very passionate about supporting the philosophy of open source code and free licenses, .

1.3 My Journey In The Team

When in September 2020 I saw the advertising of the team for the recruitment of new members I immediately became interested, even though at first I didn't feel exactly up to the possible tasks. Nonetheless, I applied for an interview anyway and quickly found myself among very comforting people. I felt evaluated with a reassuring amount of detail and friendliness from well prepared colleagues that would soon let me know that my many and varied skills were interesting and very welcome in the team. By October I started (virtually) meeting my first team mates of the Software/Microcontrollers workgroup and getting to know the basics of the new vehicle.

During these first months, partly because I could not reach the lab due to Covid restrictions, my tasks were mainly focused on studying and refactoring existing C code so that I could have a good grasp of how embedded programming really looked like.

By roughly April I could finally reach the lab and meet my colleagues in person. I immediately felt at home, among people who were 100% like me on so many aspects. Touching and programming an actual microcontroller, at last, felt awesome and I immediately fell in love with my duties. My primary task was working on the Acquisition Control Unit (ACU), which is what eventually got renamed Data Acquisition System (DAS) and is, in fact, my own task to this day and the subject of this thesis.

On the last week of August, E-Agle took part in a competition held in Novi Marof, Croatia, for which I was chosen to be part of the Racing Team assembled for the occasion. Since the car was not yet complete we could only take part in static events, which was nonetheless a incredibly worthy experience. We could witness in first person how technical inspections are carried out, what it feels like to pitch a project to the judges and, of course, attend dynamic events at the track and enjoy other teams' astonishing performances. Last but not least, I think that during that single week so many bonds were born between us that will surely last for quite some time.

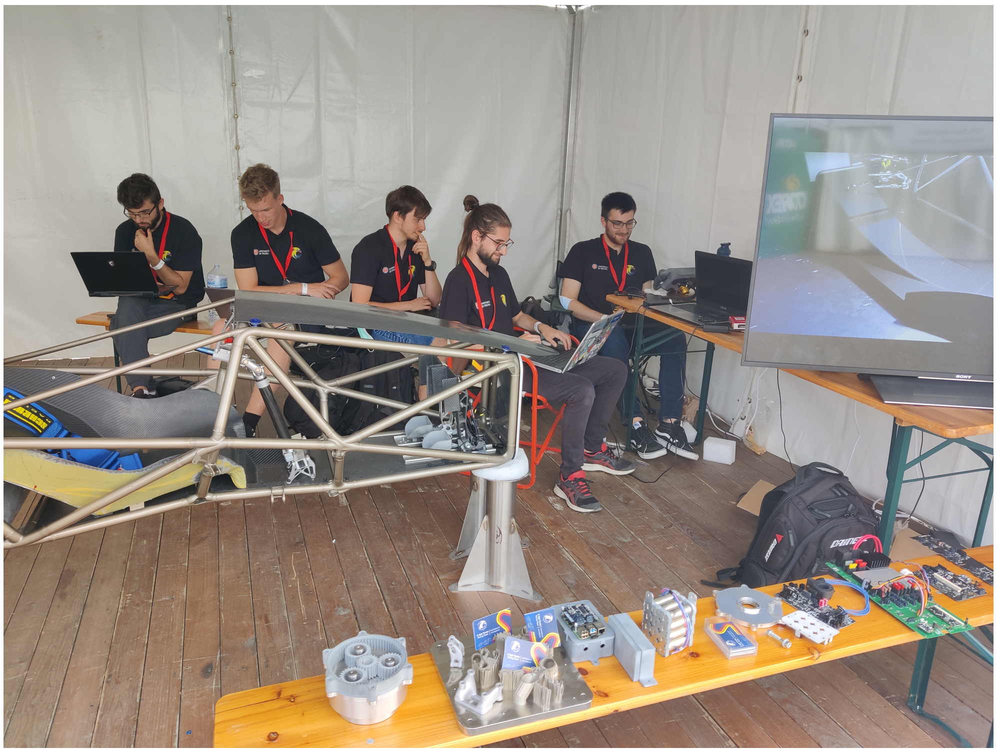


Figure 1.6: Team booth at Novi Marof, Croatia

At the end of the academic year, a plenary reunion was held to sum up the team's progress and discuss objectives for the new year. When reviewing new entries in the team and members that were finishing their careers, I was nominated to be the new Project Manager of the Microcontrollers workgroup. Finding myself in a position in which I could give back at least a bit of what I had received felt a privilege. At this time I'm in close contact both with my new workgroup colleagues and my old teammates, with whom we continuously exchange ideas and opinions.

2 Overview of the Vehicle

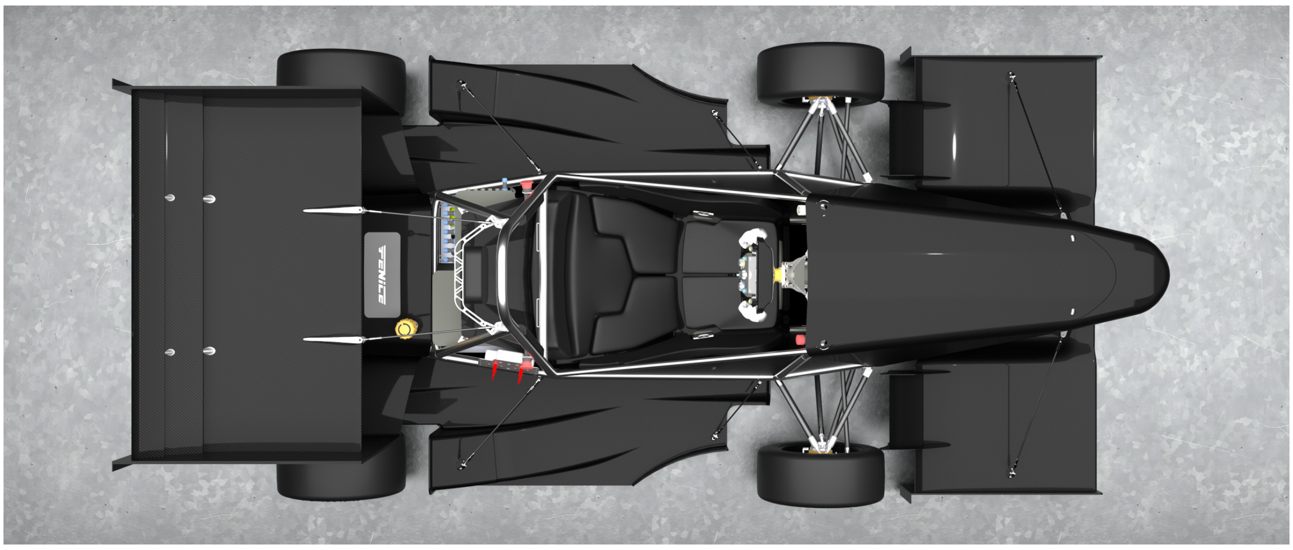


Figure 2.1: Render of Fenice - Top view

The dream of Fenice began in 2019, shortly before the Covid-19 pandemic would hit and put a stop to all in-presence activities. To this day, many industries and sponsors are still under-equipped, while the electronics market is also suffering a shortage, further obstructing the already difficult development.

The primary goals that drove the various technical decisions were to diminish the overall mass, lower the Center of Gravity (CoG), improve handling and ergonomics in general, and allow for easier packaging, maintenance, and troubleshooting. Broadly speaking, with respect to Chimera, these changes are mainly reflected by the upgrade to more efficient motors, carbon fiber rims, redesigned pedals and steering assemblies, and all-new electronics for the low-voltage and high-voltage systems, both on the hardware and on the software side. Lastly, Fenice marked the introduction of a full aerodynamic package, composed of front and rear wings, undertray, and diffuser.

2.1 Dynamics and Mechanics

2.1.1 Chassis

With the goal of reaching a total mass below 200 kilograms, Fenice is based on a hybrid chassis, composed of a main tubular steel structure (Figure 2.2) and a rear coque in carbon fiber to enclose the powertrain and to support the axles, brakes, and suspensions. A notable characteristic is, as with Chimera, the employment of 3D-printed steel joints that allow the integration of many mounting brackets (for floor panels, HV accumulator, bodywork, and other assemblies) and therefore less parts to weld and failure points. Additionally, since the coupling between the tubes is done by nesting, the structure can sustain itself and the welding jig requires less components, eventually leading to higher precision, resistance, and lower manufacturing effort and complexity.

This setup is estimated to yield a torsional rigidity of about $1800 Nm/°$ (an increase of 25% with respect to having no joints) and a total mass of 32 Kg (with respect to 46 Kg for Chimera). Lastly, the driver seat position has also been adjusted, further reclining it to lower the pilot's CoG.

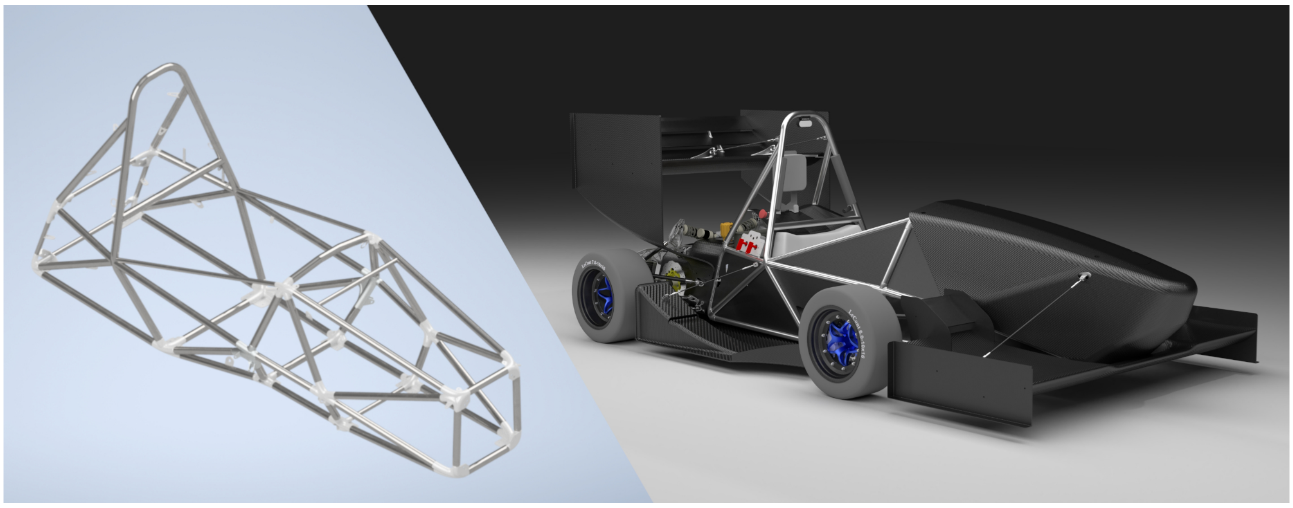


Figure 2.2: Render of Fenice and the tubular part of the chassis

2.1.2 Suspensions

Due to its stiffness and ease of design, a double A-arm (double wishbone) design was considered to be the most efficient choice, in particular for its simplicity in tuning geometrical parameters. Dampers have also faced several improvements in their models. Instead of assuming a simplified linear behavior, a more accurate case function was developed to represent their strong non-linearity according to six parameters: jounce at high and low speed, rebound at low and high speed, and transition velocities for jounce and rebound. One last notable added component is the Anti-Roll Bar (ARB). The initial specifications aimed at holding a roll angle smaller than 2° under 2g of lateral acceleration, but its design allows for simple tuning trading off mechanical resistance with desired stiffness.

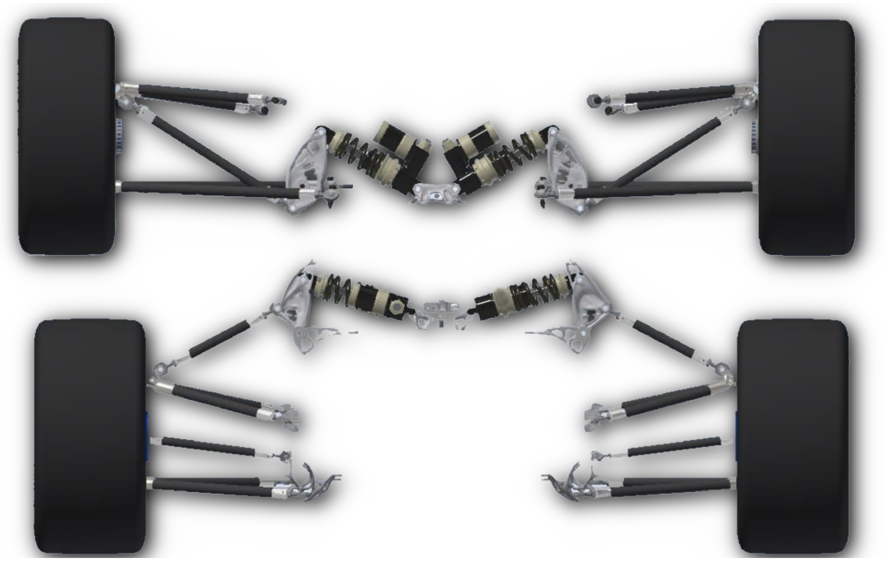


Figure 2.3: Suspension system of Fenice

2.1.3 Drivetrain

Fenice's drivetrain is based on two **Emrax 188** motors that, to obtain better torque characteristics for the acceleration event, need to go through a set of reduction gears. A pair of single-stage planetary gears provides the best compromise between weight and mechanical efficiency, yielding a reduction ratio of 3.7.

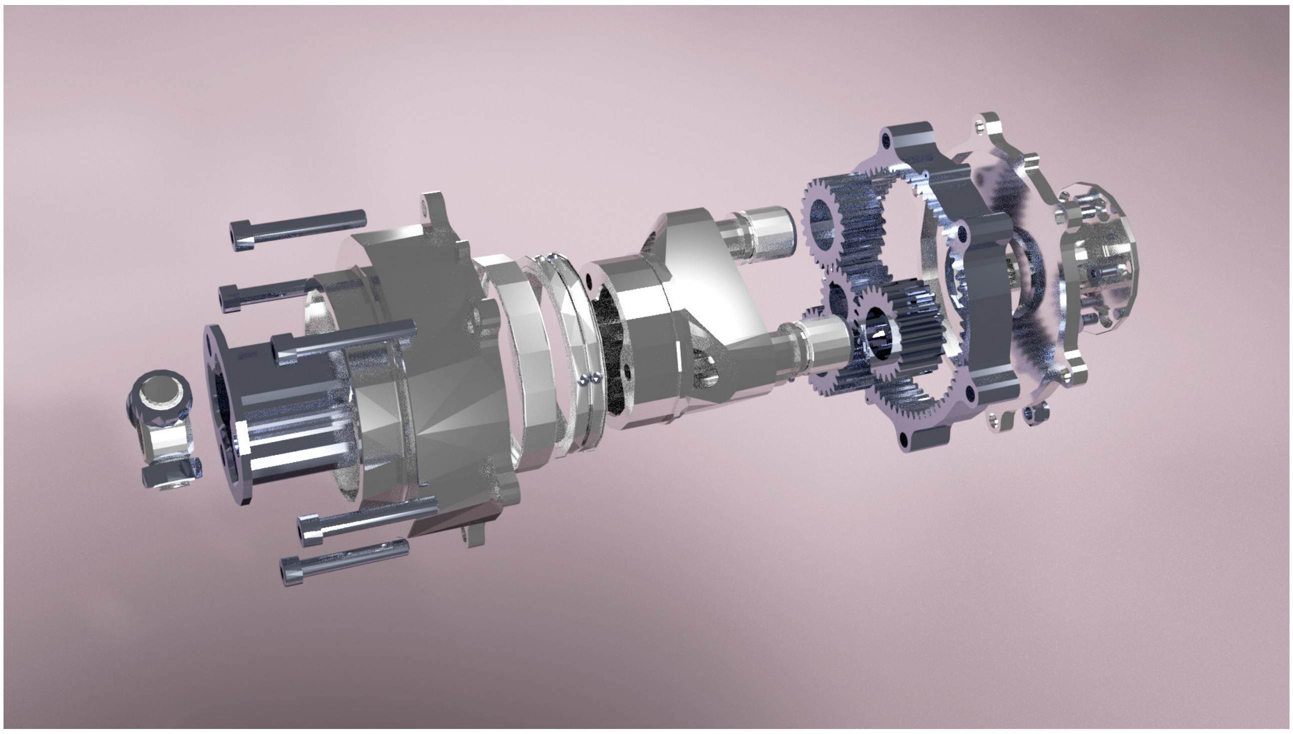


Figure 2.4: Planetary reduction gears

Another component of the drivetrain that has been improved in order to reduce mass is the half-shaft. Half-shafts are now made in carbon fiber, specifically by two different coaxial tubes terminated by 3D-printed titanium fittings and tripods to efficiently transfer torque.

Wheel assemblies have as well been redesigned from scratch to minimize unsprung masses (almost 40% less) and rotational inertia. Rims have been upgraded to carbon fiber as for many other components, and are bolted to bell-shaped optimized hubs made in Ergal.

The cooling system is composed by two symmetrical hydraulic circuits that run through two custom radiators to control the temperature of the motors and the inverters. The HV accumulator is instead cooled by a forced air induction that causes a convective exchange between the flow and the surfaces of the cells. Overall, the increase in efficiency of this system with respect to Chimera's led to a weight drop of 20%.

2.1.4 Brake System and Pedals Assembly

One of the most incisive changes to the rear axle is perhaps the relocation of the brake calipers and rotors from the classic in-wheel position to an on-board setting, reducing both unsprung masses and mechanical stress on the uprights. Brake disks also present new ventilation holes and waved outer profiles that ensure better heat dissipation and self-cleaning characteristics during use, on top of higher energy efficiency. Furthermore, brake lines have been equipped with a proportioning valve to control the balancing bar of the master cylinder, that can be actuated directly from the cockpit.

The pedals assembly has been redesigned to allow quick adjusting through a quick-release mechanism.

2.1.5 Traction Control

The control unit of Fenice will be equipped with two traction control algorithms that will be able to also operate simultaneously: Slip-Control and Torque-Vectoring. Both of them work on continuous inputs from the Data Acquisition System (DAS) and apply sensor values to virtual models of the vehicle to obtain an output response with which the DAS will command the inverters. Slip-Control mode compares the speeds of the two rear wheels with an estimation of the vehicle's velocity and, if the longitudinal slip exceeds a reference value, the engine torque is limited proportionally to maintain maximum traction. In Torque-Vectoring mode instead, motor torque is used to control the steering behavior. A desired yaw rate is compared with the measured one and is input into a model that outputs a yaw moment to compensate for the difference between the two. Successively, this yaw moment is split between the two rear wheels to guarantee differential torque.

2.2 Low Voltage Systems

While Chimera was based on a centralized power rail that delivered 5 volts to all peripherals, Fenice is based on a 12V rail that each device individually steps down to 5V, drastically reducing voltage drops across the network. Additionally, high power components such as fans and pumps can be directly attached to the LV bus.

2.2.1 HV Insulation and Shutdown Circuit

To achieve maximum safety, the high-voltage battery pack has its two poles normally disconnected from the external connector pins by means of two Accumulator Insulation Relays (AIRs), one for each pole (Figure 2.5).

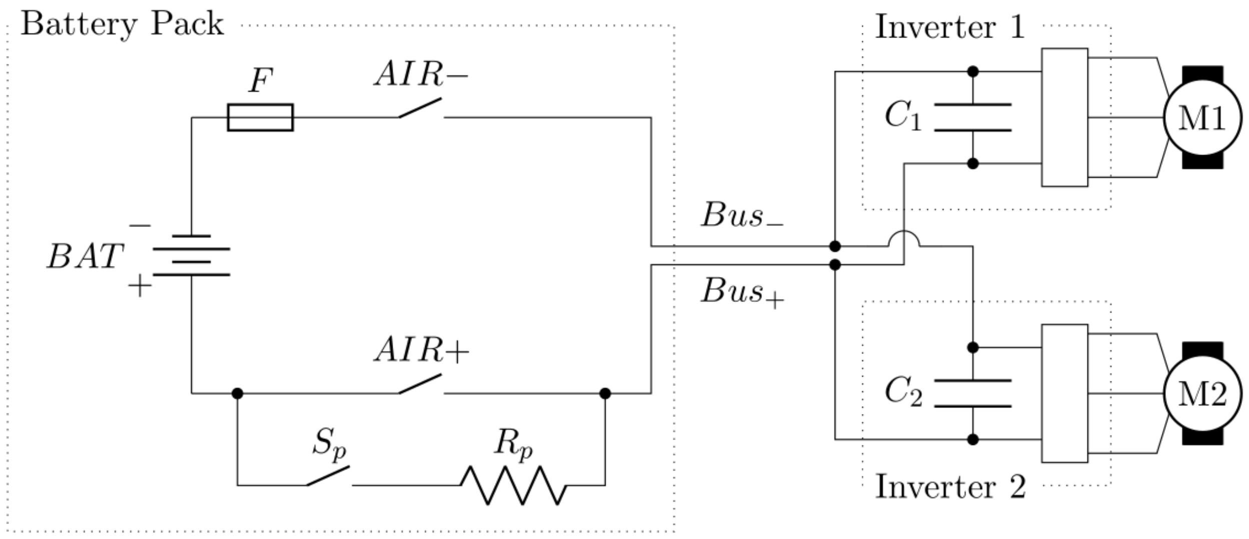


Figure 2.5: Logical diagram of the connection of the AIRs

These relays are activated only if a series of devices are in the correct state simultaneously. Formally, all these conditions create the so-called Shutdown Circuit, represented in a block diagram in Figure 2.6.

Namely, the devices that compose such circuit are the following:

- **Low-Voltage Master Switch (LVMS)**: master switch operated by the user to turn on and off the low voltage system.
- **Insulation Monitoring Device (IMD)**: a device that ensures insulation between the battery poles and the low voltage system by measuring the resistance between the two and interrupting the circuit if this value falls below $500\Omega/V_{max}$, where V_{max} is the maximum pack voltage.
- **Brake Switch Plausibility Device (BSPD)**: non-programmable circuit that triggers when the accelerator and brake pedals are pressed simultaneously.
- **Accumulator Management System (AMS)**: switch opened by the BMS-HV in case of internal errors.

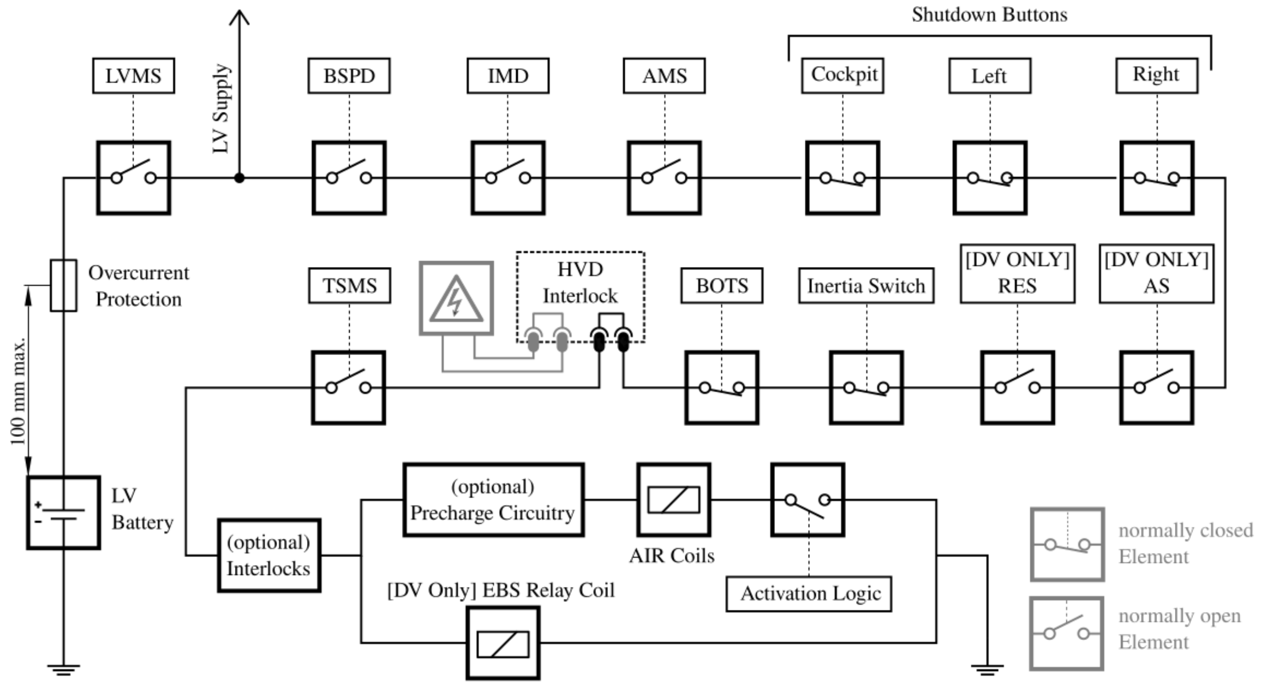


Figure 2.6: Shutdown circuitry

- **Shutdown Buttons**: mushroom safety buttons located in the cockpit and on both sides of the vehicle.
- **Inertial Switch**: pops open upon excessive acceleration or a violent shock (for example, the vehicle crashes into something).
- **Brake Over-Travel Switch (BOTS)**: a contact at the end of the brake pedal assembly that triggers (and therefore powers off the Tractive System) if excessive force is applied.
- **Tractive System Master Switch (TSMS)**: similar to the LVMS, a master switch operated by the user but to arm the Tractive System.
- **Interlocks**: auxiliary low-voltage pins located inside HV connectors that make contact to each other only when plugged.

2.2.2 Traction and Electronic Control System (TECS)

The heart of the vehicle controls reside in the TECS unit, which aggregates the functionalities of the Electronic Control Unit (ECU) and the Data Acquisition System (DAS). The former is composed by a Raspberry Pi 4 running a Matlab model of the vehicle, while the latter is a microcontroller that reads many sensors scattered through the car, interacts with the ECU to control the inverters and the motors, and during state transitions (e.g., from IDLE to RUN) coordinates all devices such that said transitions happen in a safe manner and leave all components in a coherent state. Examples of peripherals that the DAS needs to interact with, in addition to all devices on the CAN bus to monitor error states and warnings, are the two wheel encoders, the steering wheel encoder, the inverters, the PCU, the brake-light PWM signals, and more.

2.2.3 Steering Wheel / Human-Machine Interface (HMI)

Given the extremely promising feedback received from judges and sponsors on the previous model, the team decided to continue pursuing the effort for a completely custom telemetry and steering wheel systems, taking the occasion to redesign the latter to refine all its electronics and improve the driver's ergonomy.

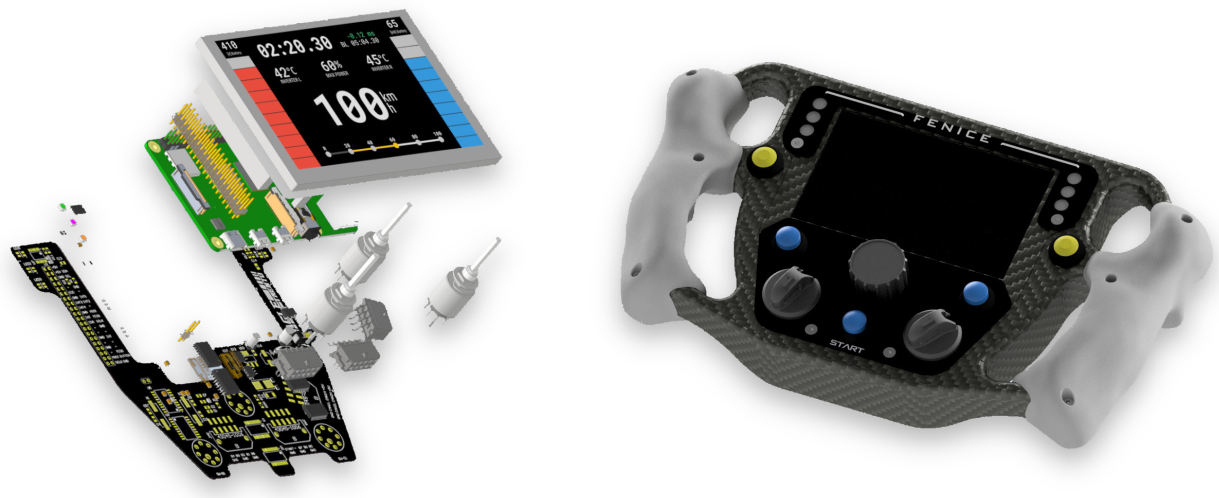


Figure 2.7: Render of the steering wheel

The interface is composed of five buttons, four rear paddles, three rotary switches, and a 5” LCD screen that allow the user to fully control all the car functions from within the cockpit. The software has been perfected as well, in particular to provide the driver with more verbose information which is now available thanks to an improved CAN-bus infrastructure. One important addition to the displayed statistics, such as speed, battery load, systems temperatures, etc., is the integration of a Lap Counter. Essentially, the driver sets a starting location by pressing a button when positioned at the start line, and at each lap the system computes an accurate time delta thanks to differential GPS technology.

2.2.4 Telemetry

The telemetry system captures all CAN frames that are transmitted in the network and logs them into a local database. At the same time, these data are also transmitted to a base station via an LTE link for real-time analysis on an exposed web interface. Moreover, after an event, race engineers can export the logs to Matlab using a self-developed software to perform fast and advanced elaboration and visualizations.

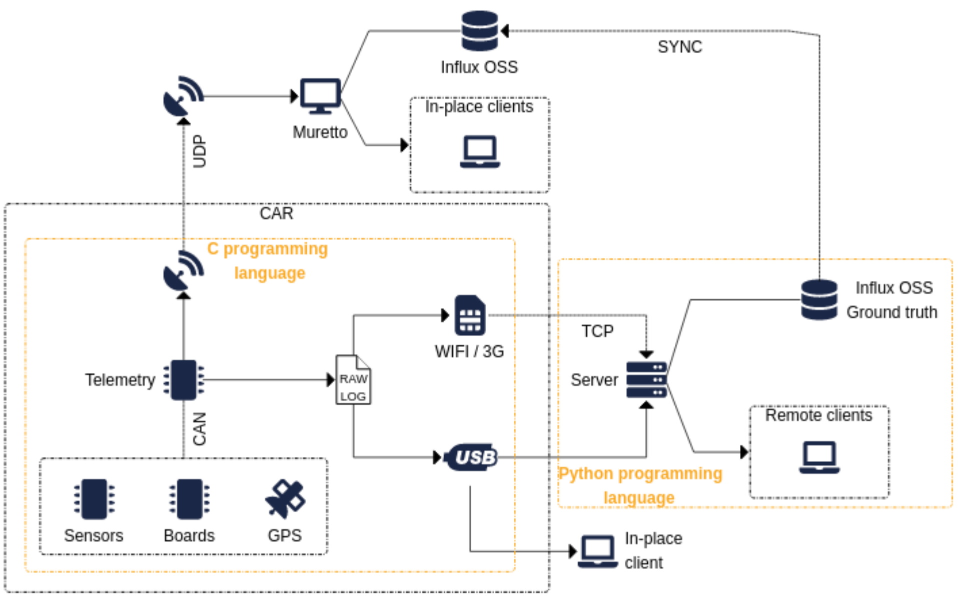


Figure 2.8: Telemetry architecture

2.2.5 Battery Management System (BMS-LV)

The low voltage power supply consists of a Lithium-Polymer battery which combines a relatively low weight with a high energy density and provides a capacity of 326 Wh. The BMS board hosts two DC-DC converters, one which feeds a 12V line at up to 400W of power, and another one that delivers 24V at 150W. Secondly, the BMS-LV has the important duty of constantly checking several temperatures (e.g., DC-DC converters, LiPo battery, etc.), voltages, and drawn currents in order to be able to promptly disconnect the master relay and power off the vehicle in case of issues.

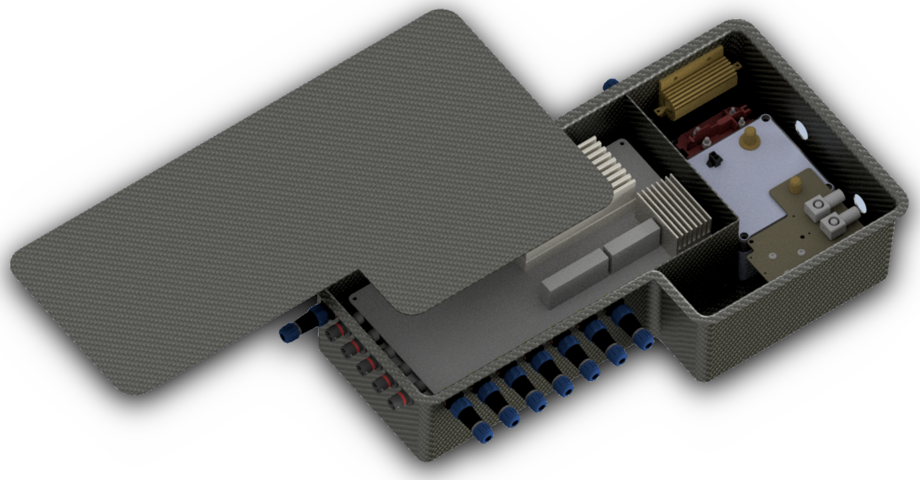


Figure 2.9: Render of the low voltage supply system

2.2.6 Pedals Control Unit (PCU)

The Pedals Control Unit is soon to be integrated into the Data Acquisition System (TECS/DAS), but as of now it consists of a dedicated microcontroller that reads the Accelerator Pedal Position Sensor's (APPS) potentiometers and forwards their values into the CAN bus. Secondly, the PCU employs two additional ADCs to read the brake pedal's pressure values (front and rear pressures). While in Chimera the APPS' potentiometers were rotative, therefore restricting their used range to only a section and not taking full advantage the ADCs' precision either, in Fenice the potentiometers are of linear type which, despite having a slightly less linear response, nonetheless allow for a better signal to be acquired and later filtered by software.

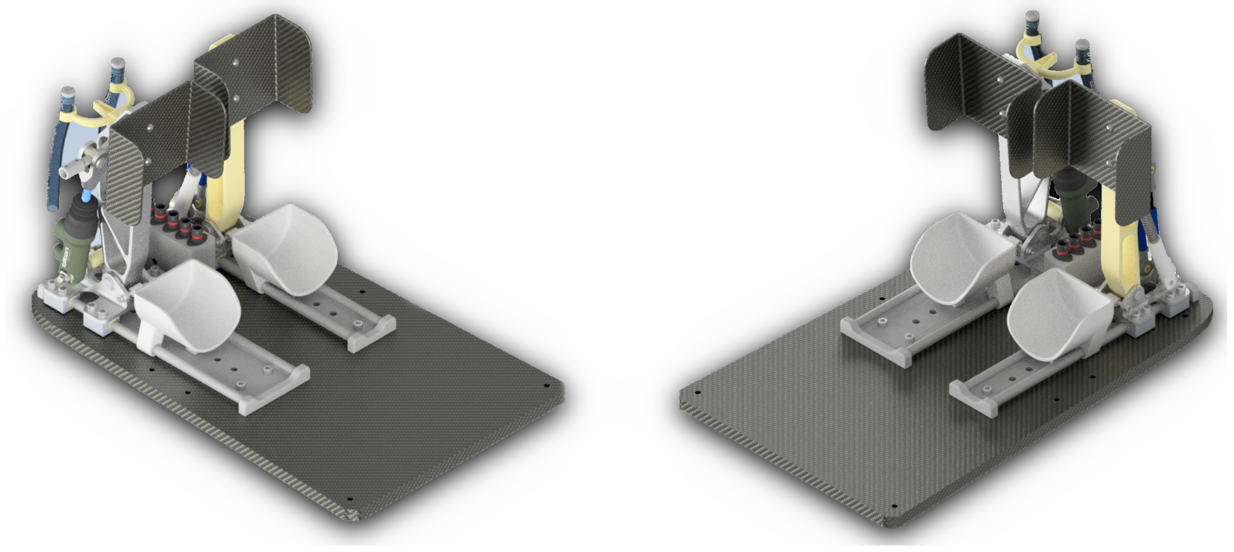


Figure 2.10: 3D model of the pedals assembly

2.3 High Voltage System and Powertrain

Fenice's high voltage system is composed of three main subsystems: the HV battery pack (and its BMS), the Powertrain (composed of motors and inverters), plus the charging handcart. The first two components together are also referred to as Tractive System.

2.3.1 Battery Pack and Battery Management System (BMS-HV)

The design of the high voltage battery pack was subject to several design constraints and trade-offs. Specifically, on one hand, the acceleration event requires to output as much power as allowed and the endurance event needs 22 kilometers of autonomy, but on the other hand size and weight are to be kept at minimum to achieve better efficiency and handling. Given that the rulebook imposes a maximum voltage limit of 600V and a maximum power limit of 80kW [1], and that to limit weight and volume as much as possible cells need to be arranged in the most efficient way, the pack adopts a **108s4p** configuration of high discharge rate cells, i.e. 108 cells in series and 4 in parallel for a total of 432 cells. This yields a nominal voltage of 388.8V (rewiring the pack in a 144s3p configuration would exceed the 600V limit) and a continuous discharge current up to 180A, approaching 70kW of power output and 6.2kWh of energy capacity. As a note, each one of the 432 cells was hand picked, measured, input into a balancing script[1], and manually sorted into the proper module to achieve maximum energy balance. A much thorough description and analysis of the BMS-HV can be found in [3].

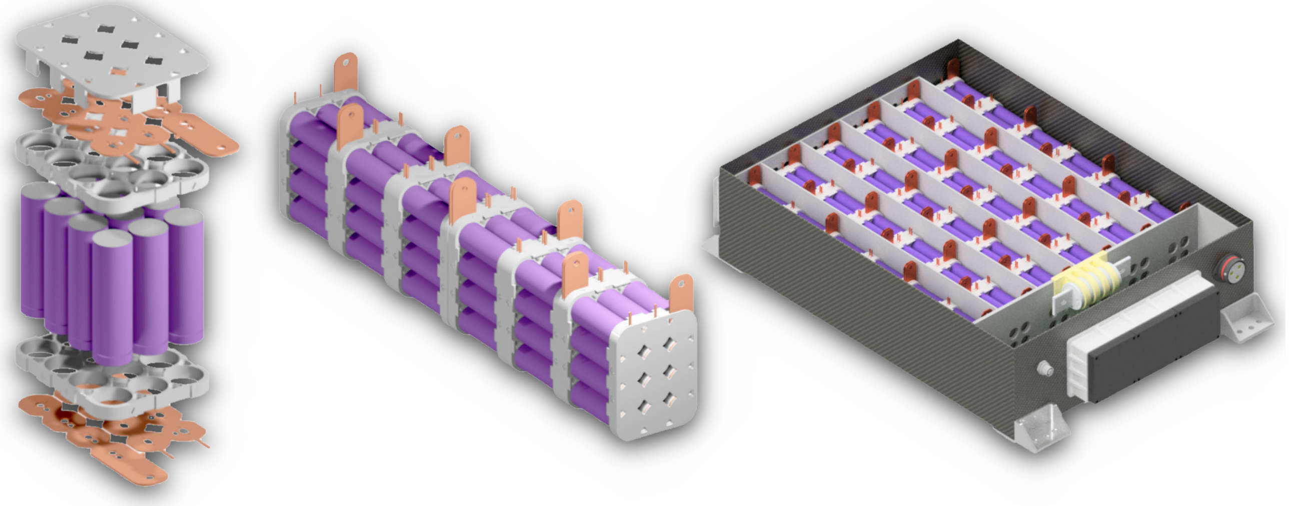


Figure 2.11: 3D models of one 3s4p module, one segment, and the complete battery pack

In terms of microcontrollers, the BMS-HV relies on two main components: a main-board and a set of cell-boards. The former is effectively the central brain of the BMS, managing all its peripherals such as CAN bus, insulated ADCs, EEPROMs, serial ports, SD card slot, and more. The latter is instead a set of low-power microcontrollers (one for each of the 6 battery modules) continuously reading voltages and temperatures of the cells via dedicated chips. These data are transferred from the chip to the cellboard via SPI and then forwarded to the mainboard via a dedicated internal CAN bus.

One other innovation of this BMS with respect to Chimera's is the ability to perform cell balancing: essentially, because of their intrinsic uniqueness, with time each cell slowly deviates from the others in terms of nominal voltage and capacity, which both limits the charging limit of the modules, but also leads to the anticipation of the cut-off threshold during discharge, causing the pack to shut off even if many modules still contain enough energy. The balancing hardware and logic is located on the cell-boards and is of the passive type (excess energy is dissipated instead of transferred to other

[1]https://github.com/AlexSartori/Battery-Pack-Partitioner, forked and improved from @BuckarewBanzai

cells) since the advantages of active-balancing approaches would not apply in our use case but only raise the overall complexity and create delicate failure points. Many details on the ad-hoc balancing algorithm are presented in [3].

One last important feature of the BMS is its solid error checking and reporting procedures, since any undetected hardware failure has the potential to be, quite literally, catastrophic. As a matter of fact, the rulebook imposes many strict regulations for Tractive System components, as they are the most safety-critical section of the vehicle. An efficient, time-sensitive, and easy to troubleshoot system to check and handle warnings and errors allows for unproblematic adhesion to the safety specifications.

2.3.2 Handcart

The handcart is the device that, by compliance with the rulebook, has to be built in order to transport and charge the HV accumulator. It consists of a four-wheeled aluminum structure geared with appropriate hardware to safely control the charging rate and the status of the cells. It is composed of three main components: the actual charging hardware (a BRUSA NLG513), a Raspberry Pi 4 to run the control software, and a CAN bus that connects the above two devices with the BMS-HV to receive information on its internal state and plan a safe and efficient charging process. Figure 2.12 shows a render of the handcart plus the web interface that it exposes through a Python Flask server.

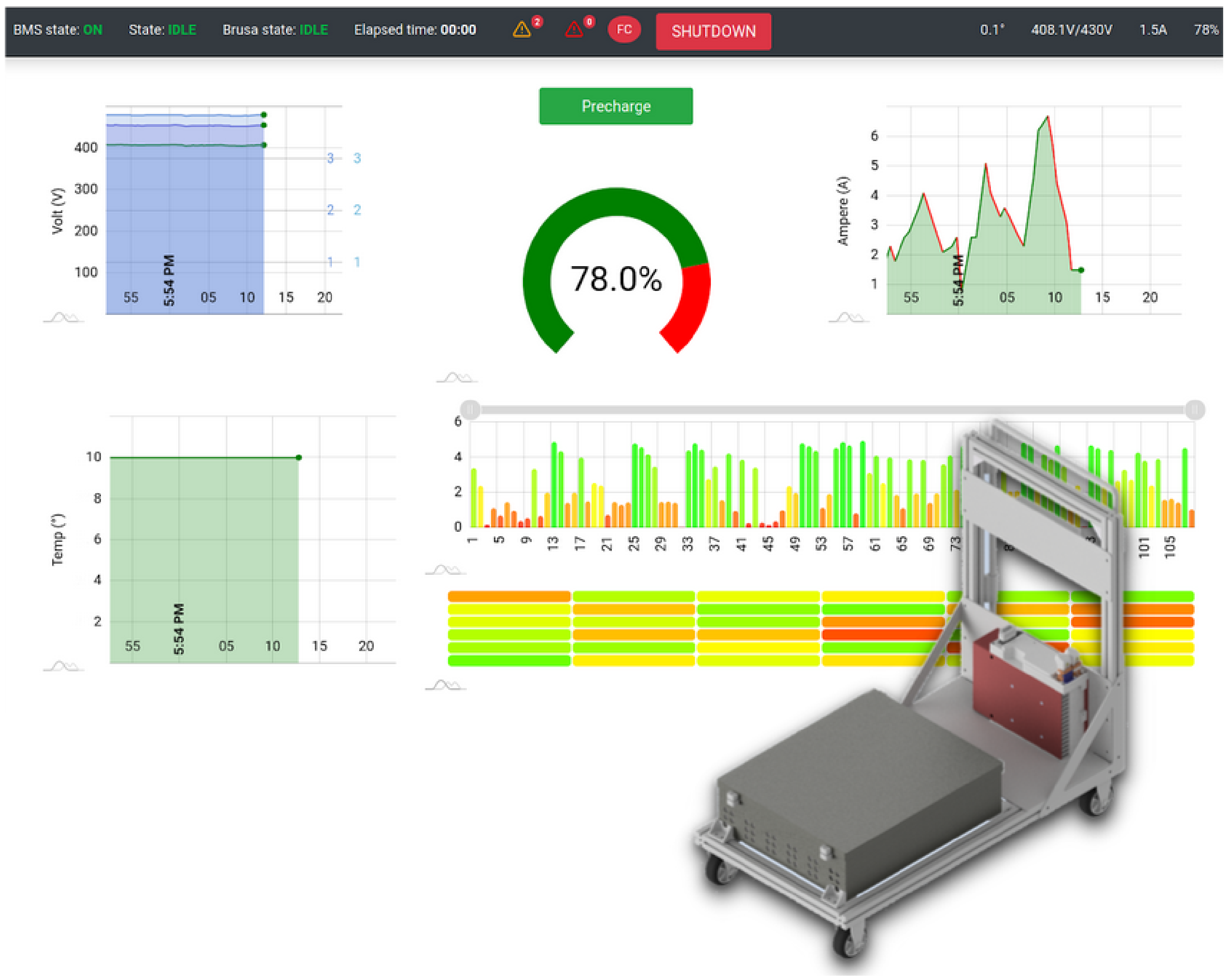


Figure 2.12: Render of the charging handcart overlayed to the web interface

3 TECS in Depth

The Traction and Electronic Control System (TECS) unit is the conjugation of two crucial devices: the Electronic Control Unit (ECU), responsible for running a virtual model of the vehicle for traction control purposes, and the Data Acquisition System (DAS), a microcontroller that ensures coherent operation of the car and safe state transitions at all times.

While the ECU logic has been studied and developed at high level by the **Dynamics and Modeling Team**, the Microcontrollers workgroup has been in charge of planning its low-level details such as hardware platform and communication protocol with the DAS, with the main goals of having both real-time responses, and a 100% reliable behavior in case the model crashes, becomes unresponsive, or starts behaving incoherently leading to otherwise unsafe control signals. On the other hand, the DAS has been developed entirely by my workgroup, with the obvious exception of the circuit board which has nonetheless required strict collaboration with the Electronics division to correctly plan the needed hardware features that had to be implemented.

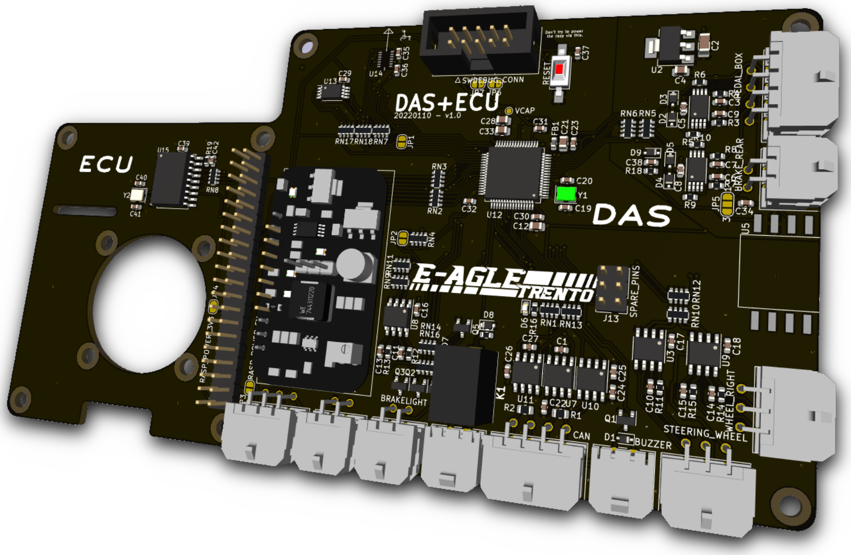


Figure 3.1: TECS PCB: on the left side, the 40-pin header to host the Raspberry Pi

3.1 ECU Hardware

The ECU runs on a Raspberry Pi 4 SBC, which provides the following features:

- **CPU**: Quad-Core Cortex on ARMv8 architecture (64-bit, 1.5 GHz)
- **RAM**: 4 Gb LPDDR4
- **Networking**: Gigabit Ethernet, 2.4 GHz and 5.0 GHz 802.11 b/g/n/ac wireless, and Bluetooth 5.0/BLE connectivities
- **Graphics support**: double micro-HDMI 4kp60 ports
- **Extra connectivity**: backwards-compatible 40-pin GPIO header, USB 2.0, USB 3.0, Micro-SD slot

3.2 ECU Software

The ECU software is a set of object-files (`*.o`) automatically generated by Matlab, included and used by a custom developed wrapper that handles the exchange of parameters over the SPI or UART interfaces with the DAS.

3.3 DAS Hardware

The DAS microcontroller is an STM32F446RETx chip soldered on a custom made PCB by the Electronics team. A render of the board project is visible in Figure 3.1.

3.3.1 External Crystal

STMs are built with an integrated oscillator that is factory calibrated to a 1% precision. However, being located inside the silicon chip, it is very susceptible to temperature variations of the MCU that can alter its oscillating frequency. For most applications this skew is of no concern, but for high speed peripherals such as the CAN controller, even the slightest variation can severely impact the timing correctness of a transmitted frame, especially if other devices on the bus are configured with small Synchronization Jump Width values and cannot correctly interpret the signal.

For this reason, STMs provide the possibility to attach an external clock source to the chip, which in our case proved to be a strict necessity to obtain correct behavior.

3.3.2 Timers

STM Timers - Theory and Specifications

Timers essentially consist of a 16-bit auto-reload counter register driven by a programmable prescaler that can be employed in a variety of purposes, such as measuring the pulse length of input signals (input capture) or generating output waveforms (output compare, PWM, complementary PWM with dead-time insertion). Their most relevant characteristics are:

- 16-bit up, down, up/down auto-reload counter
- 16-bit programmable prescaler to divide the counter clock frequency
- Synchronization circuit to control the timer with external signals or chain them together
- Repetition counter to update timer registers only after a given number of counter cycles
- Interrupt or DMA request generation upon:
    - Counter initialization
    - Counter overflow or underflow
    - Counter start, stop, or initialization by external trigger
    - Input capture or output compare
- Support for incremental (quadrature) encoder and Hall-sensor circuitry
- Trigger input for external clock source

Figure 3.2 from [7] displays the whole architecture that ST timers implement. Of particular interest for our use case are the PWM mode to generate a buzzer sound and the quadrature encoder mode to read the two wheel encoders.

Pulse Width Modulation mode allows to generate a square wave with a frequency determined by the value of the `TIMx_ARR` register and a duty cycle determined by the value of the `TIMx_CCRx` register.

In **Encoder mode**, the timer is capable of decoding quadrature-encoded outputs by using two channels connected respectively to the A and B lines of an incremental encoder. These lines output two square waves with a 90 degrees phase difference which, at any time, will be positive or negative depending on the direction of rotation. Additionally, the frequency of pulses on A and B is directly proportional to the encoder's speed, with high frequencies indicating rapid movement and lower, or

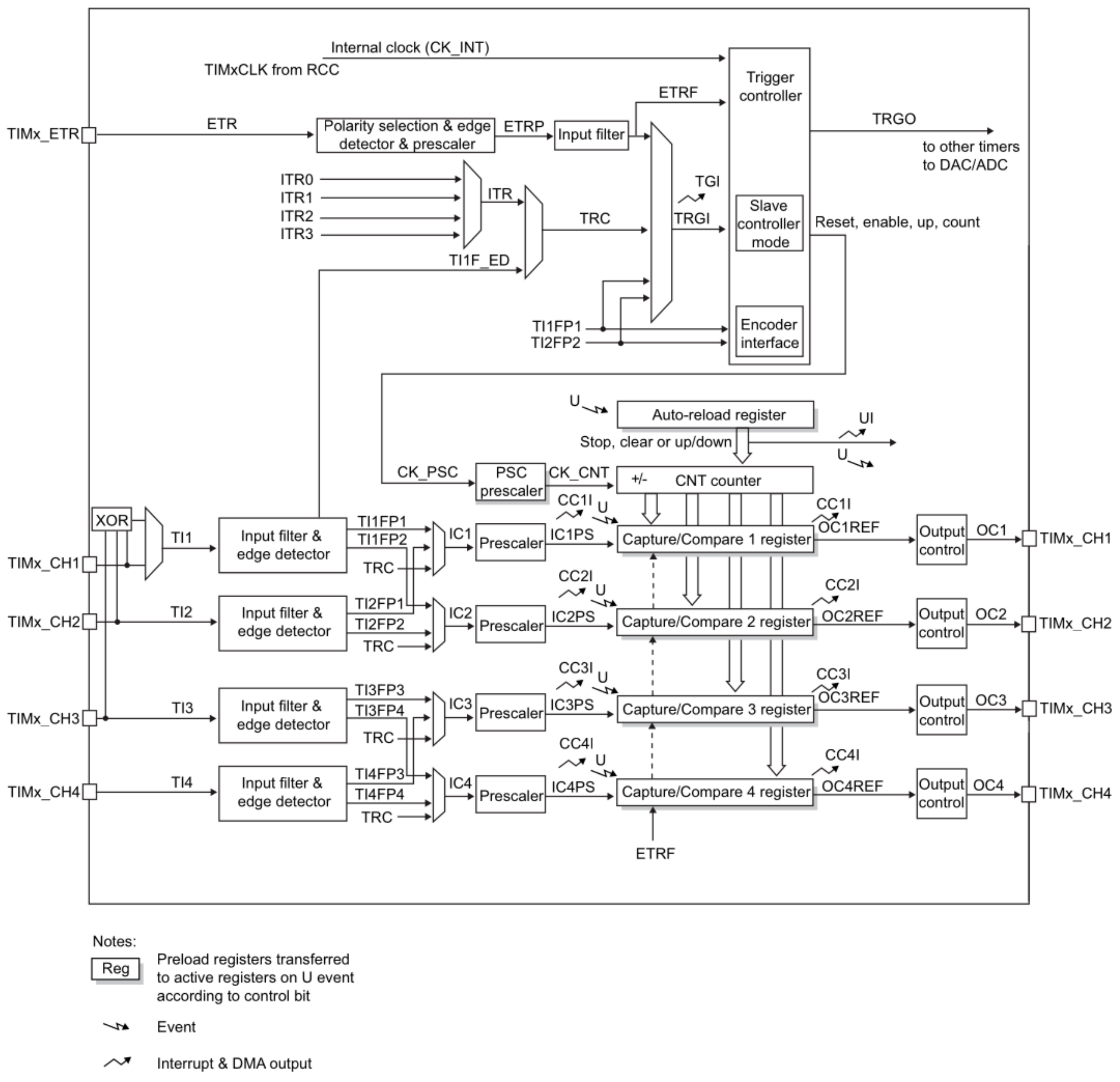


Figure 3.2: Timers block diagram

static and unchanging signals, indicating slow progression or absence of motion. Moreover, to reject common-mode noise, these signal lines are usually propagated on RS-422 differential pairs with, at the end of the path, an appropriate IC that converts them back to the single-ended logical signals required by digital logic (e.g., the DAS timer inputs).

Essentially, with the A and B lines connected to channel 1 and channel 2 of a timer, the counter continuously counts between 0 and the auto-reload value (0 to ARR or ARR down to 0 depending on the direction): in this way, the counter is modified automatically following the speed and the direction of the incremental encoder. Its content, therefore, always represents the encoder's position (since the count direction correspond to the rotation direction of the connected sensor). Possibly, the third encoder output which indicates the mechanical zero position, might be connected to an external interrupt input and trigger a counter reset. Figure 3.3 from [7] shows an illustrative example of this mode.

Dynamic information on the encoder speed, acceleration, or deceleration, can be obtained by measuring the time period between two events using a second timer configured in capture mode or, alternatively, the output of the encoder which indicates the mechanical zero (Z) might be used to trigger an interrupt.

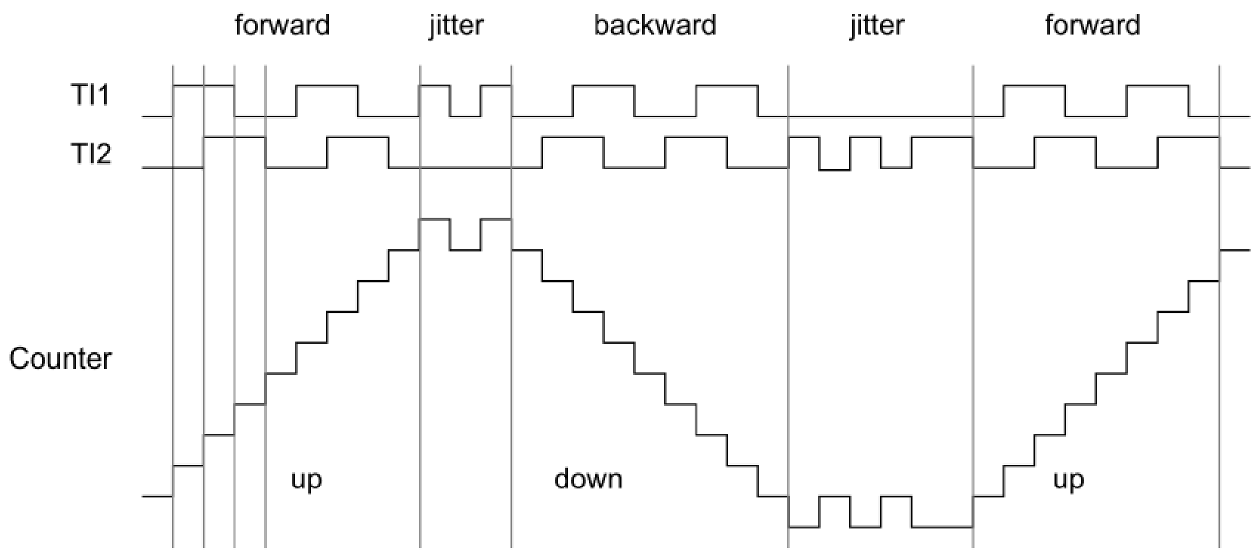


Figure 3.3: Example of a timer counter behavior in quadrature encoder mode.

Used Timers

Table 3.1 lists the used timers in the DAS.

Timer	Mode	Purpose
TIM1	PWM Generation	Buzzer frequency generation
TIM2	Quadrature-Encoder	Left wheel magnetic encoder
TIM5	Quadrature-Encoder	Right wheel magnetic encoder

Table 3.1: Timers employed by the DAS

3.3.3 Controller Area Network (CAN)

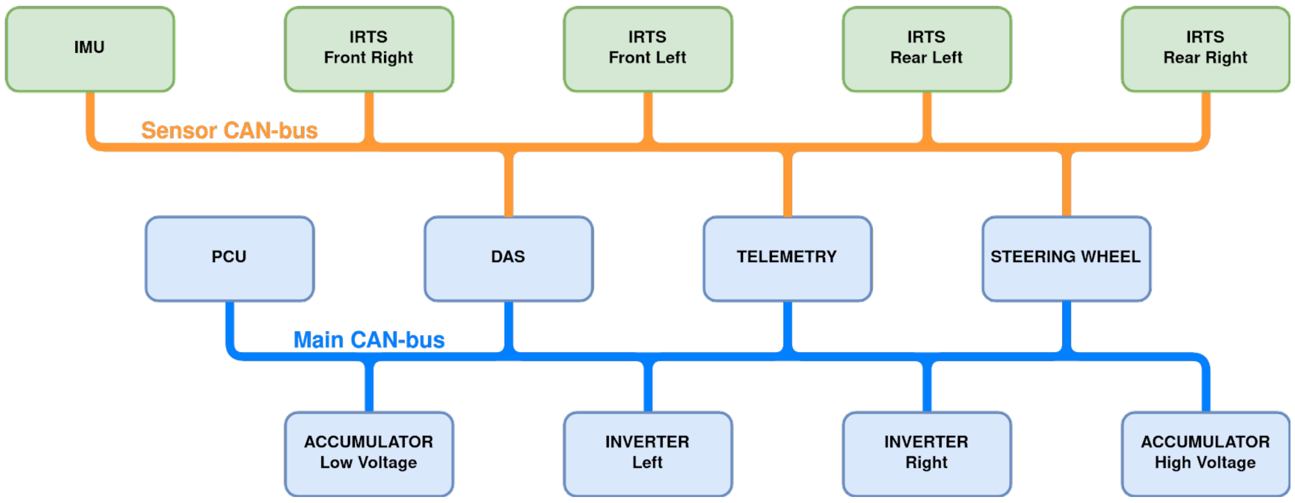


Figure 3.4: CAN topology in Fenice

Introduction to CAN Buses

A CAN bus is a robust vehicle standard designed to allow low-level devices to communicate with each other without the need for a host computer. It is a message-based protocol, designed originally for multiplex electrical wiring within vehicles to save on copper, but it is often used in many other contexts. The data of a frame is transmitted sequentially, but in such a way that if more than one device transmits at the same time, the highest priority one can continue while the others back off. Frames are broadcast to all devices, including to the transmitting device itself. The latest CAN specification is 2.0, which has two parts; part A is for the standard format with an 11-bit identifier (CAN 2.0A devices), and part B is for the extended format with a 29-bit identifier (CAN 2.0B devices). Modern cars can have as many as 70 control units for various subsystems, from the (usually) most powerful processor for the engine control unit to others dedicated to transmission, airbags, ABS, cruise control, electric power steering, audio systems, power windows, doors, mirror adjustment, battery and recharging systems for hybrid/electric cars, and many more. While some of these form independent subsystems, communication among others is essential, for instance for a subsystem that might need to control actuators or receive feedback from sensors. One key advantage is that interconnection between different systems can allow a wide range of safety, economy and convenience features to be implemented using software alone, a functionality which would add cost and complexity if it were "hard wired" using traditional automotive electrics.

On the hardware side, CAN is a multi-master bus that requires at least two nodes in order to communicate. All nodes are connected to each other through a two wire bus, usually a twisted pair with a 120Ω (nominal) characteristic impedance. Two signals, CAN high (CAN_H) and CAN low (CAN_L) are either driven to a "dominant" state with CAN_H > CAN_L, or not driven and pulled by passive resistors to a "recessive" state with CAN_H ≤ CAN_L. A 0 data bit encodes a dominant state, while a 1 data bit encodes a recessive state, supporting a wired-AND convention, which gives nodes with lower ID numbers priority on the bus.

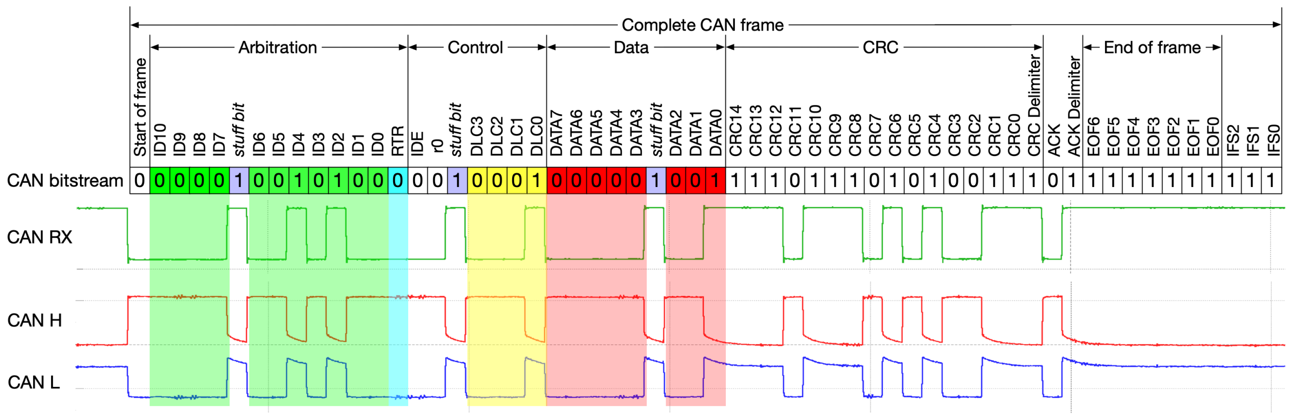


Figure 3.5: Example of the transmission of a CAN frame

The DAS needs to access two separate CAN networks. A primary one where critical messages flow, and a secondary one where lower-priority sensors can operate at high speed without the risk of congesting the line. STM32F4s implement the **Basic Extended CAN** peripheral, named bxCAN, that supports the CAN protocols version 2.0A and 2.0B. It has been designed to manage a high number of incoming messages efficiently with a minimum CPU load by also meeting the priority requirements for transmit messages. Its main characteristics that make it a good fit for our use-case are:

- Support for CAN 2.0A and 2.0B
- Bit rates up to 1 Mbit/s
- Three transmit mailboxes
- Configurable transmit priority
- Two receive FIFOs with three stages
- 28 scalable filter banks shared between CAN1 and CAN2 for dual CAN
- Software-efficient mailbox mapping at a unique address space

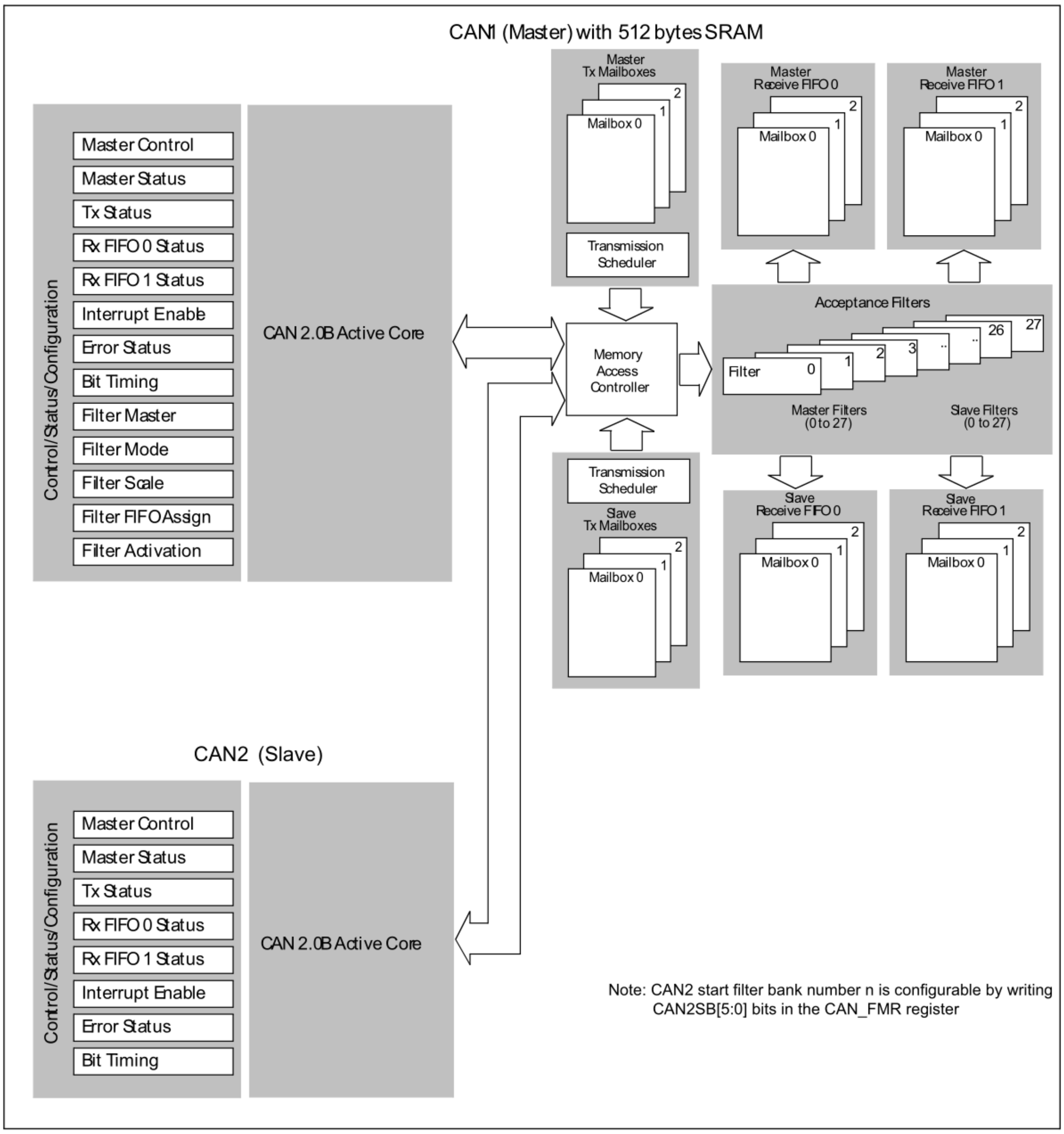


Figure 3.6: Block diagram of the bxCAN setup

Transmission is handled by three mailboxes filled by software that are automatically scheduled by the controller based on their contained message priorities. Received frames are instead stored by hardware into two FIFOs of three slots each, of course after checking that the configured filters actually accept the message. These filters are, again, completely managed by hardware and do not require any CPU cycle to be dedicated; up to 28 banks can be configured in **ID Mask Mode** or **in ID List Mode**: in the former, one register acts as the mask and the second as the ID, while in the latter mode, both registers represent IDs of the list. Filter banks can also be configured to operate in 16-bit mode, therefore doubling their capacity (four 16-bit registers instead of two 32-bit registers, which are split in two). In ID Mask mode this yields two pairs of Mask+ID, while in ID List mode, of course, it yields 4 ID registers. While list mode is the most straightforward, as the filter is simply a list of acceptable IDs, mask mode is the most powerful and effective. In this mode the filtering is based on a mask register which tells which bits of the ID register need to be compared with the incoming message ID. For instance, with a mask of `11110000000` and an ID of `01110000000`, only messages whose ID begins with `0111` (`0x3`) will be accepted. Figure 3.7 displays with better clarity how filter mode and scale influence their interpretation by the controller.

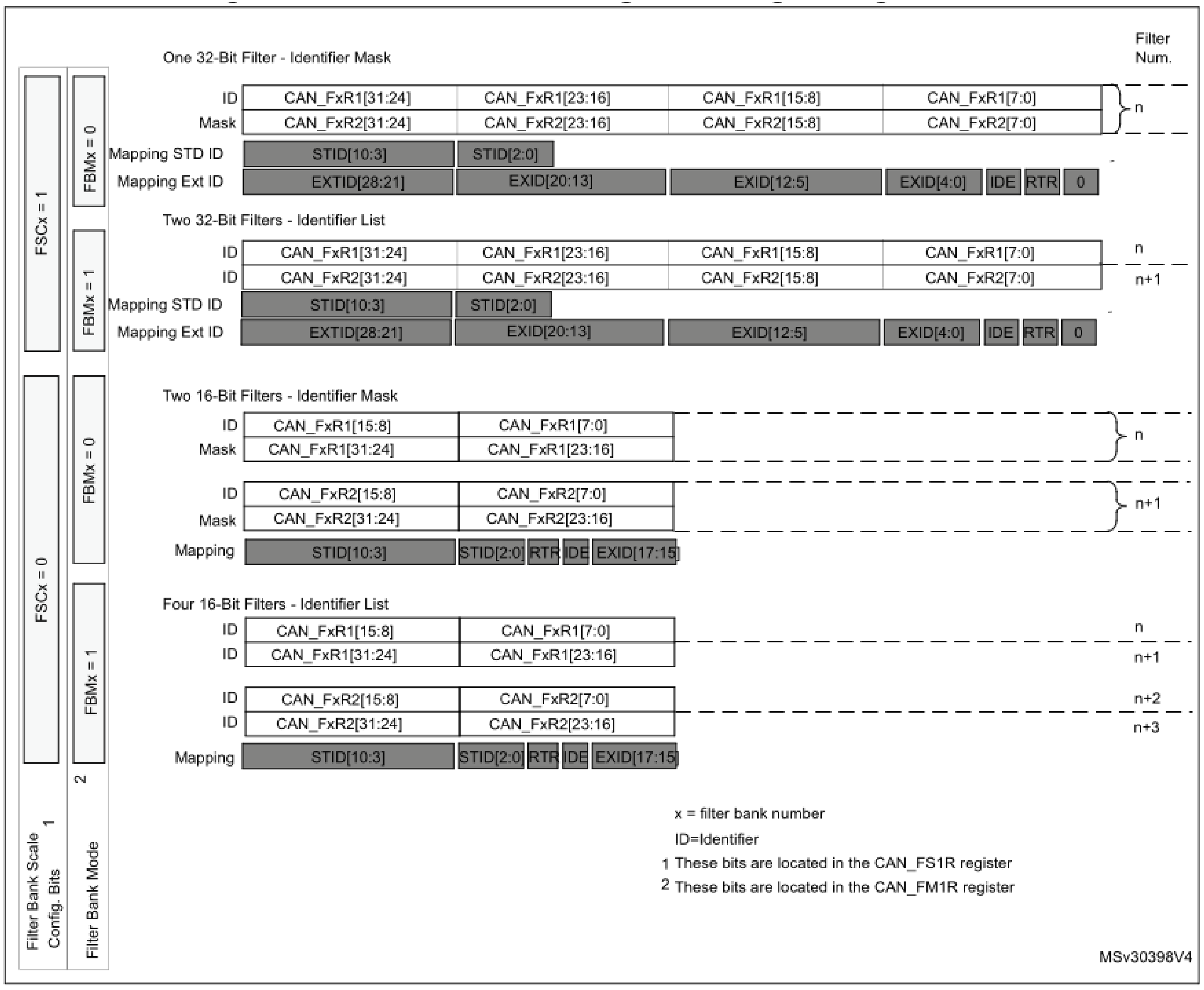


Figure 3.7: Layout and mapping of the bxCAN registers responsible for message filtering

3.3.4 Encoders

Encoders make use of two timers in **Encoder Mode** for the wheels, and one SPI interface for the steering wheel.

Wheels Encoders

Wheel encoders are composed of a magnetic ring and a read-head (model `LM13ICD40AB10F00`, Figure 3.8) [10].

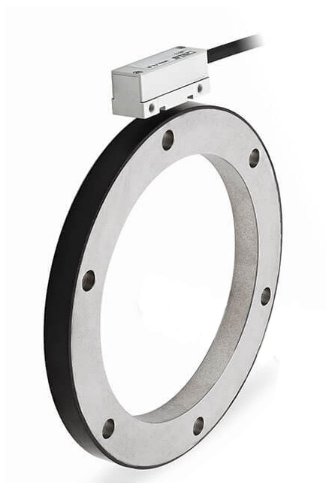


Figure 3.8: Wheel encoder composed of a magnetic ring and a read head

Figure 3.9 instead reports the main characteristics of the digital read-head hovering the magnetic ring, while its part number is decoded in Figure 3.10 in order to understand how to decode its output data: essentially, each signal transition corresponds to a progression of 0.000005 meters on the magnetic ring circumference.

Steering Wheel Encoder

Decoding the steering encoder part number (`RM44SC0012B10F2F10`) and referencing the various tables yields the key points detailed in Figure 3.12 [11].

Essentially, as shown in Figure 3.13, the encoder accepts a clock signal on one line and outputs a binary value on another. Figure 3.12 instead specifies its precision, that is, 360 degrees equal 4096 increments.

3.3.5 Inertial Measurement Unit (IMU)

The chosen IMU by Izze-Racing measures acceleration and angular rate for all three orthogonal axes and outputs data at 200Hz via CAN. Table 3.2 lists its electric and physic specifications [9].

Acceleration measurement range	±2 / ±4 / ±8 (default) / ±16 g
Angular measurement range	±245 (default) / ±500 / ±2000 dps
Acceleration accuracy	< 1% FS
Angular rate accuracy	< 1.5% FS
Temperature resolution	1.0 °C
Sampling frequency	10 / 50 / 120 / 240 (default) / 480 Hz
Supply voltage	5 to 8 V
Supply current	25 mA
CAN bitrate	1 Mbit/s / 500 / 250 / 100 Kbit/s
CAN byte order	Big-Endian
CAN ID (acceleration)	`0x4EC`
CAN ID (angular rate)	`0x4ED`

Table 3.2: IMU sensor specification

LM13IC – Digital output signals, RS422

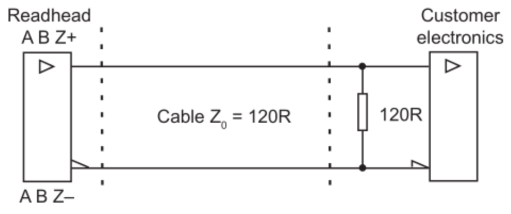
Square wave differential line driver to RS422

Power supply *	4.7 V to 7 V – voltage on readhead Reverse polarity protection
Power consumption	< 35 mA
Voltage drop over cable	~ 13 mV/m – without load ~ 54 mV/m – with 120 Ω load
Power supply rise time	< 1 ms (for PRG option only)
Response time **	< 100 ms < 10 µs (special option 02)
Output signals	3 square-wave signals A, B, Z and their inverted signals A–, B–, Z–
Reference signal	1 or more square-wave pulse Z and its inverted pulse Z–
Signal level	Differential line driver to EIA standard RS422: $U_H \geq 2.5$ V at $-I_H = 20$ mA $U_L \leq 0.5$ V at $I_L = 20$ mA

Permissible load	$Z_0 \geq 100\ \Omega$ between associated outputs $I_L \leq 20$ mA max. load per output Capacitive load ≤ 1000 pF Outputs are protected against short circuit to 0 V and to +5 V Only one output shorted at a time
Alarm	High impedance on output lines A, B, A–, B– Special option 02: Alarm is not signalled by high impedance state ** Special option 07: Alarm signal is output parallel as line driver signal
Switching time (10 to 90 %)	t+, t– < 30 ns (with 1 m cable and recommended input circuit)
Cable length *	Max. 100 m

* Please consider voltage drop over cable.
** See description on page 8.

Recommended signal termination



Timing diagram
Complementary signals not shown

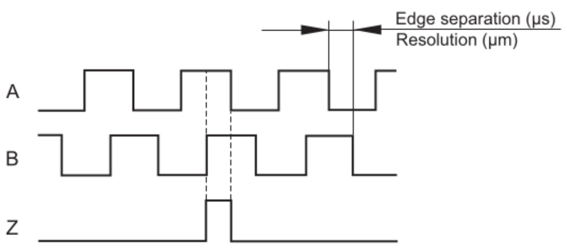
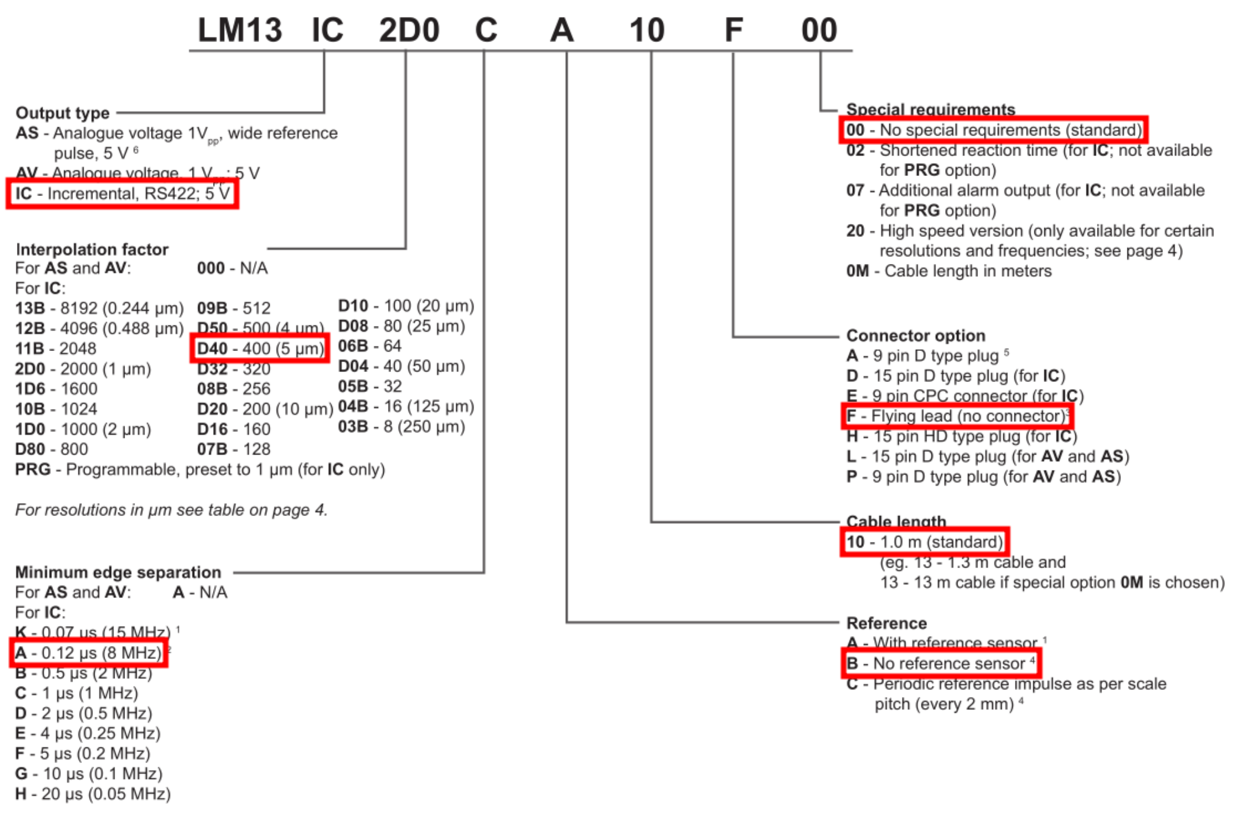


Figure 3.9: Main characteristics of the wheel encoder's read-heads



Output type
AS - Analogue voltage 1V$_{pp}$, wide reference pulse, 5 V [6]
AV - Analogue voltage, 1 V$_{pp}$; 5 V
[✓ **IC** - Incremental, RS422; 5 V]

Interpolation factor
For **AS** and **AV**: **000** - N/A
For **IC**:

13B - 8192 (0.244 µm)	**09B** - 512	**D10** - 100 (20 µm)
12B - 4096 (0.488 µm)	**D50** - 500 (4 µm)	**D08** - 80 (25 µm)
11B - 2048	[✓ **D40** - 400 (5 µm)]	**06B** - 64
2D0 - 2000 (1 µm)	**D32** - 320	**D04** - 40 (50 µm)
1D6 - 1600	**08B** - 256	**05B** - 32
10B - 1024	**D20** - 200 (10 µm)	**04B** - 16 (125 µm)
1D0 - 1000 (2 µm)	**D16** - 160	**03B** - 8 (250 µm)
D80 - 800	**07B** - 128	

PRG - Programmable, preset to 1 µm (for **IC** only)

For resolutions in µm see table on page 4.

Minimum edge separation
For **AS** and **AV**: **A** - N/A
For **IC**:
K - 0.07 µs (15 MHz) [1]
[✓ **A** - 0.12 µs (8 MHz) [2]]
B - 0.5 µs (2 MHz)
C - 1 µs (1 MHz)
D - 2 µs (0.5 MHz)
E - 4 µs (0.25 MHz)
F - 5 µs (0.2 MHz)
G - 10 µs (0.1 MHz)
H - 20 µs (0.05 MHz)

Special requirements
[✓ **00** - No special requirements (standard)]
02 - Shortened reaction time (for **IC**; not available for **PRG** option)
07 - Additional alarm output (for **IC**; not available for **PRG** option)
20 - High speed version (only available for certain resolutions and frequencies; see page 4)
0M - Cable length in meters

Connector option
A - 9 pin D type plug [5]
D - 15 pin D type plug (for **IC**)
E - 9 pin CPC connector (for **IC**)
[✓ **F** - Flying lead (no connector) [3]]
H - 15 pin HD type plug (for **IC**)
L - 15 pin D type plug (for **AV** and **AS**)
P - 9 pin D type plug (for **AV** and **AS**)

Cable length
[✓ **10** - 1.0 m (standard)]
(eg. 13 - 1.3 m cable and 13 - 13 m cable if special option **0M** is chosen)

Reference
A - With reference sensor [1]
[✓ **B** - No reference sensor [4]]
C - Periodic reference impulse as per scale pitch (every 2 mm) [4]

Figure 3.10: Decoding of the read-head part number to understand how to read its output signal

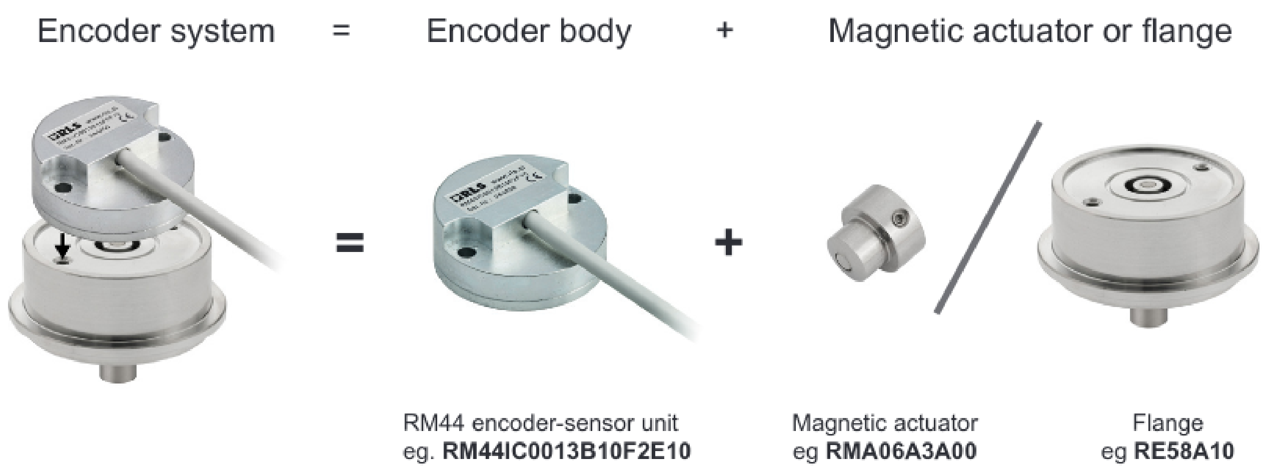
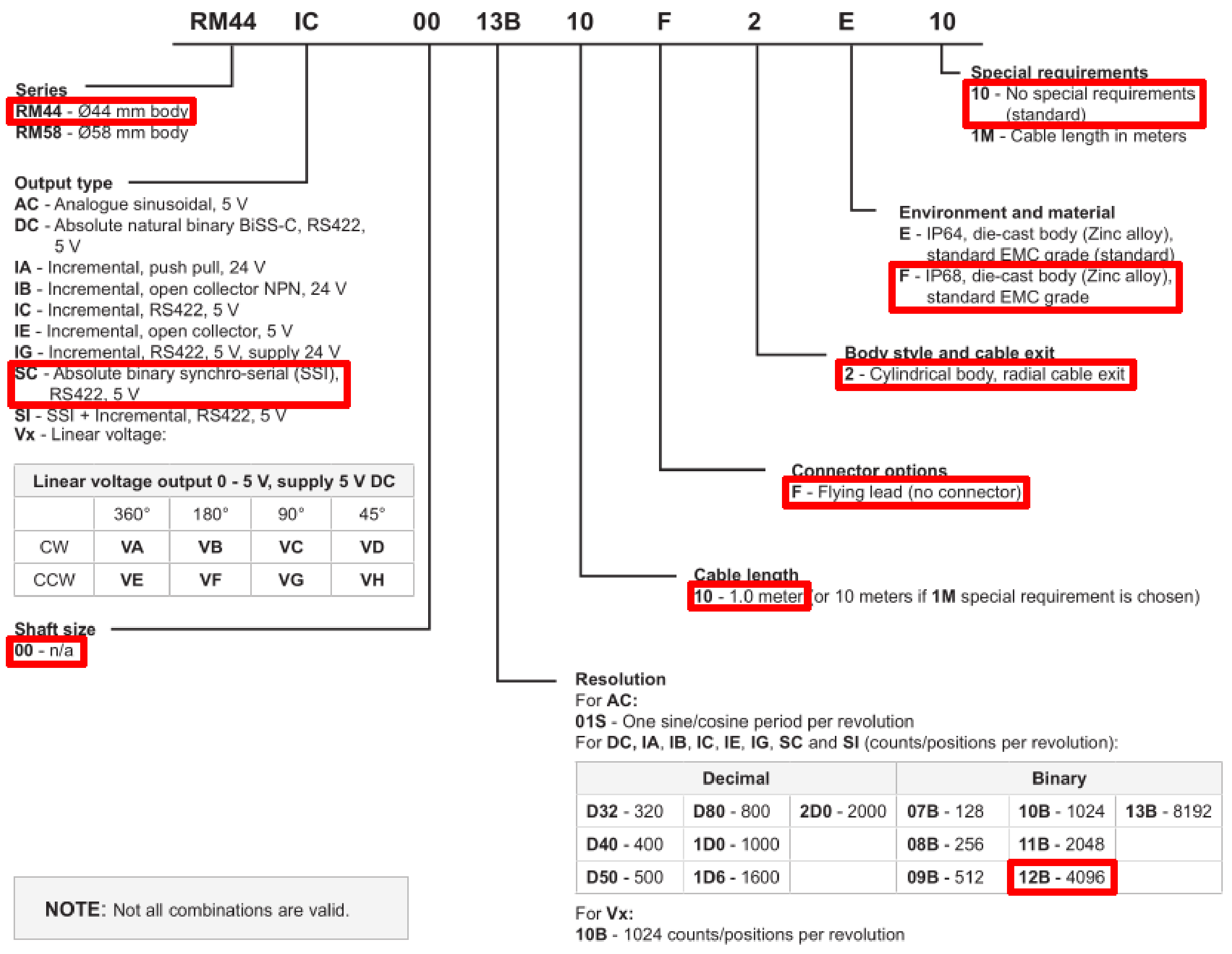


Figure 3.11: Steering encoder composed of a body and a magnetic actuator



RM44 IC 00 13B 10 F 2 E 10

Series
RM44 - Ø44 mm body
RM58 - Ø58 mm body

Output type
AC - Analogue sinusoidal, 5 V
DC - Absolute natural binary BiSS-C, RS422, 5 V
IA - Incremental, push pull, 24 V
IB - Incremental, open collector NPN, 24 V
IC - Incremental, RS422, 5 V
IE - Incremental, open collector, 5 V
IG - Incremental, RS422, 5 V, supply 24 V
SC - Absolute binary synchro-serial (SSI), RS422, 5 V
SI - SSI + Incremental, RS422, 5 V
Vx - Linear voltage:

Linear voltage output 0 - 5 V, supply 5 V DC				
	360°	180°	90°	45°
CW	**VA**	**VB**	**VC**	**VD**
CCW	**VE**	**VF**	**VG**	**VH**

Shaft size
00 - n/a

Resolution
For **AC:**
01S - One sine/cosine period per revolution
For **DC**, **IA**, **IB**, **IC**, **IE**, **IG**, **SC** and **SI** (counts/positions per revolution):

Decimal			**Binary**		
D32 - 320	**D80** - 800	**2D0** - 2000	**07B** - 128	**10B** - 1024	**13B** - 8192
D40 - 400	**1D0** - 1000		**08B** - 256	**11B** - 2048	
D50 - 500	**1D6** - 1600		**09B** - 512	**12B** - 4096	

For **Vx:**
10B - 1024 counts/positions per revolution

Cable length
10 - 1.0 meter or 10 meters if **1M** special requirement is chosen)

Connector options
F - Flying lead (no connector)

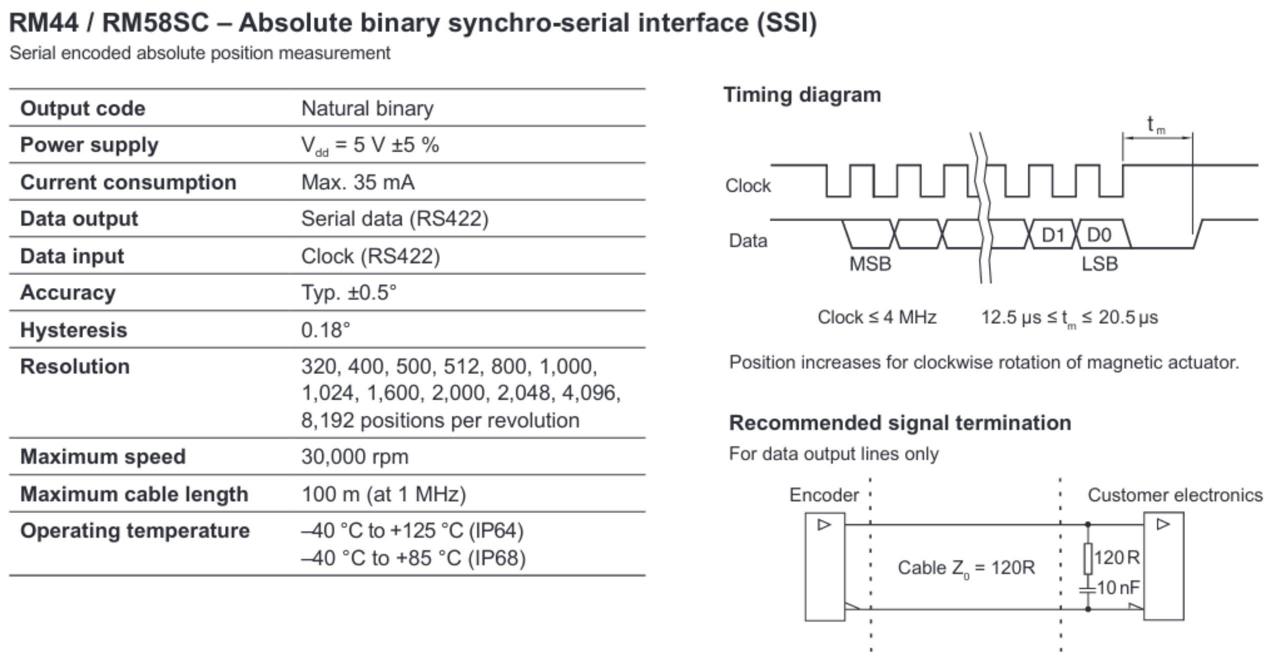
Body style and cable exit
2 - Cylindrical body, radial cable exit

Environment and material
E - IP64, die-cast body (Zinc alloy), standard EMC grade (standard)
F - IP68, die-cast body (Zinc alloy), standard EMC grade

Special requirements
10 - No special requirements (standard)
1M - Cable length in meters

NOTE: Not all combinations are valid.

Figure 3.12: Decoding of the center encoder to understand its output data



RM44 / RM58SC – Absolute binary synchro-serial interface (SSI)

Serial encoded absolute position measurement

Output code	Natural binary
Power supply	V_{dd} = 5 V ±5 %
Current consumption	Max. 35 mA
Data output	Serial data (RS422)
Data input	Clock (RS422)
Accuracy	Typ. ±0.5°
Hysteresis	0.18°
Resolution	320, 400, 500, 512, 800, 1,000, 1,024, 1,600, 2,000, 2,048, 4,096, 8,192 positions per revolution
Maximum speed	30,000 rpm
Maximum cable length	100 m (at 1 MHz)
Operating temperature	–40 °C to +125 °C (IP64) –40 °C to +85 °C (IP68)

Timing diagram
t_m
Clock
Data
D1
D0
MSB
LSB
Clock ≤ 4 MHz
12.5 μs ≤ t_m ≤ 20.5 μs
Position increases for clockwise rotation of magnetic actuator.
Recommended signal termination
For data output lines only
Encoder
Customer electronics
Cable Z_0 = 120R
120 R
10 nF

Figure 3.13: Characteristics of the steering encoder

3.3.6 Inverters

Fenice's motors are driven by two **BAMOCAR D3-700** inverters capable of handling up to 50 kW each (continuous), at 160 Amperes of peak current. Their key features are the following, from [5]:

- Power input range from 12 to 700V
- Liquid cooling
- Independent auxiliary voltage connection at 24 V
- Digital interfaces: RS232 and CAN BUS
- Digital inputs/outputs: programmable and optically decoupled
- Position, speed and torque control
- Feedback encoder systems: resolver, incremental encoder, SINCOS 1 Vss, rotor position
- Current limiting: static and dynamic
- Processor-independent hardware switch off in case of over voltage, under voltage, short circuits, circuits to earth, and over temperature at the amplifier or motor

Their connection to the DAS happens through the primary CAN bus; their transceivers are opto-coupled and do not terminate the line with resistors.

3.3.7 Infra-Red Temperature Sensors (IRTSs)

The selected Izze-Racing tire temperature sensors are specifically designed to measure the highly transient surface temperature of a tire with spatial fidelity, providing valuable information for chassis tuning, tire exploitation, compound selection, and driver development.

The sensor is capable of measuring temperature at 16, 8, or 4 laterally-spaced points, at a sampling frequency of up to 100Hz, object temperature between -20 to 300 °C and using CAN 2.0A protocol [8].

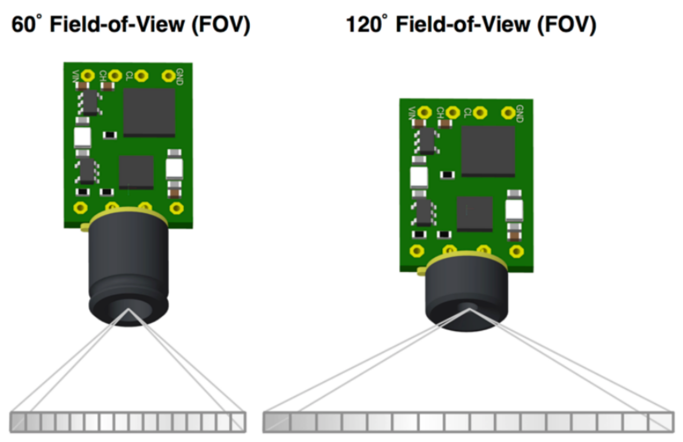


Figure 3.14: IRTS sensor

The predefined CAN IDs for the four sensors are:

- Left-Front: 0x4B0
- Right-Front: 0x4B4
- Left-Rear: 0x4B8
- Right-Rear: 0x4BC

Each sensor skips four units from the base ID because it sends four different types of messages:

- Base ID + 0: Channels 1, 2, 3, 4
- Base ID + 1: Channels 5, 6, 7, 8
- Base ID + 2: Channels 9, 10, 11, 12
- Base ID + 3: Channels 13, 14, 15, 16

Each message is composed by 8 bytes, encoding each channel's temperature as a 16-bit unsigned integer (MSB transmitted first). Data conversion happens by applying a -100°C offset and a precision of 0.1°C per bit.

3.3.8 Pedals Control Unit (PCU)

The Pedals Control Unit is an element that is currently being migrated from a dedicated board (on an STM32F3) to an extension of the DAS, which will read the potentiometers' values directly from 4 ADC channels. As of now though, the PCU reads the two Accelerator Pedal Position Sensors (APPSs) and the brake circuit pressures on a dedicated MCU and forwards them via CAN.

(a) APPS linear potentiometers

(b) Brake oil pressure transducers

Figure 3.15: PCU sensors

3.3.9 Tractive System

The Tractive System, in this context, is effectively the BMS-HV, composed of a Mainboard (on a STM32F4 MCU) and six cellboards (on STM32L4 MCUs) that control the state of the accumulator.

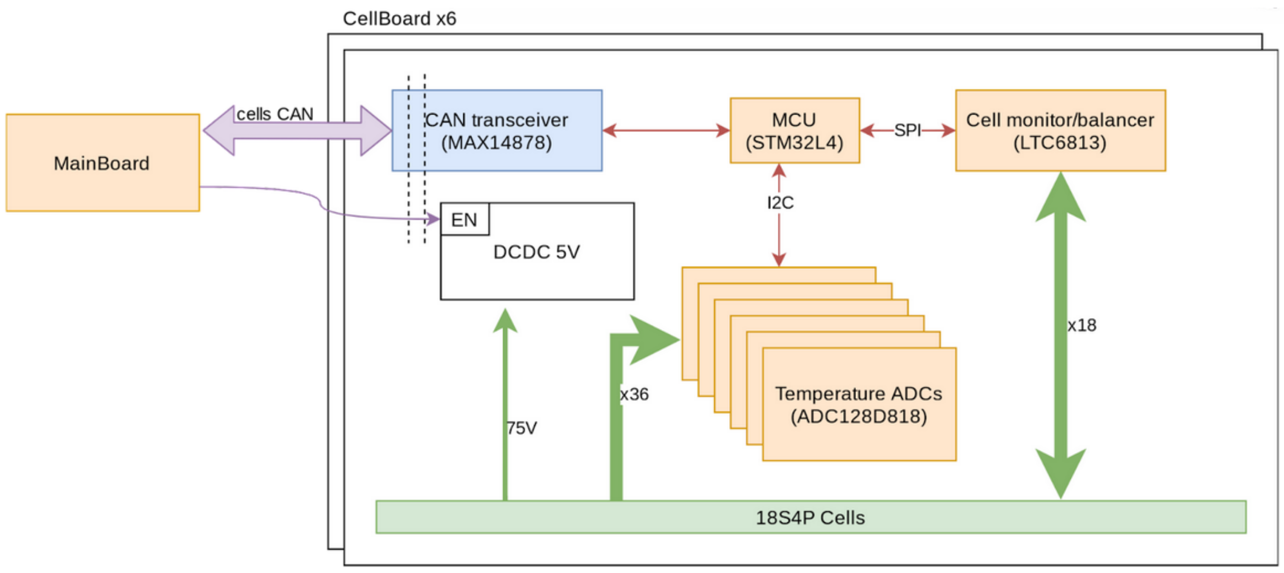

CellBoard x6
MainBoard
cells CAN
CAN transceiver (MAX14878)
MCU (STM32L4)
SPI
Cell monitor/balancer (LTC6813)
I2C
EN
DCDC 5V
Temperature ADCs (ADC128D818)
x18
x36
75V
18S4P Cells

Figure 3.16: Structure of the BMS-HV

3.4 DAS Firmware

In order to be efficiently debugged, tested, and possibly migrated to different architectures, the code has been developed in a highly modularized way. Ideally, each functionality is implemented in its own pair of `.h/.c` files, with extensive Doxygen documentation before each piece of code. The modules are listed below.

3.4.1 Buzzer

The buzzer.h module exposes a function called `BUZ_play_frequency_ms(uint16_t freq, uint16_t ms)` that starts a PWM signal on TIM1 that drives the buzzer at the given frequency for the specified time period. As per STM documentation, the signal frequency is given by the values of three registers (in relation to the internal clock): **ARR**, **CCR**, and **PSC**. ARR contains the value at which the timer will reset itself, CCR is essentially the duty cycle of the signal in relation to ARR, and PSC is a prescaler value applied to the incoming clock frequency. Their values are so calculated (where freq is the desired PWM frequency):

$$ARR = \frac{CLK_{APB} \cdot (PSC + 1)}{freq} - 1$$

$$CCR = \frac{1}{2} ARR \text{ (for 50\% duty cycle)}$$

The HAL is then invoked to start the timer, pause for `ms` milliseconds, and stop the timer.

```
/**
 * @brief     Play a frequency on the buzzer for a given number of milliseconds
 *
 * @param     freq The frequency in Hertz
 * @param     ms Duration of the sound in milliseconds
 */
```

```c
void BUZ_play_frequency_ms(uint16_t freq, uint16_t ms) {
    uint32_t APB_clk = 90000000;
    uint16_t psc = 1000;
    uint16_t arr, ccr;

    arr = APB_clk / freq / (psc + 1) - 1;
    ccr = arr / 2;

    BUZZER_TIM.Instance->PSC = psc;
    BUZZER_TIM.Instance->ARR = arr;
    BUZZER_TIM.Instance->CCR1 = ccr;

    if (HAL_TIM_PWM_Start(&BUZZER_TIM, TIM_CHANNEL_1) != HAL_OK)
        LOG_write(LOGLEVEL_ERR, "[BUZ] Error starting PWM timer");

    HAL_Delay(ms);

    if (HAL_TIM_PWM_Stop(&BUZZER_TIM, TIM_CHANNEL_1) != HAL_OK)
        LOG_write(LOGLEVEL_ERR, "[BUZ] Error stopping PWM timer");
}
```

Listing 3.1: Buzzer module code sample

3.4.2 CAN Networks

A first implementation made use of four queues (two for each network): a priority queue for messages to be transmitted and a FIFO queue for received messages. Since these queues were preallocated to accomodate 100 messages each to avoid runtime memory allocation issues, the amount of possibly wasted static memory was consistent. Additionally, enqueuing and dequeuing messages required several steps that would excessively slow down the associated Interrupt Service Routines, which is to be avoided. As one last inconvenience, this architecture would obstacolate an efficient check for message timeouts and correct transmitting frequency, given the computational load required to scan the entire queue each time and given that the serialization/deserialization process had to be scattered all over the various modules. The best solution found consists in preallocating only one struct for each message type (20 structs instead of 400 of similar size) and therefore keeping only one copy of each message (the latest arrived). This way, ISRs are only required to serialize/deserialize the message buffer (a `memcpy` of 8 bytes) to/from the appropriate struct and save there a timestamp. Consequently, timeouts are easily checked by reading one variable, and transmitting frequencies can be regulated in the same way at each call from the main loop.

```c
typedef struct {
    uint32_t timestamp;     /*< Timestamp of when the message was received/sent */
    bool is_new;            /*< If the message data has yet to be processed     */
} CANMSG_MetadataTypeDef;

typedef struct { CANMSG_MetadataTypeDef info; primary_TLM_STATUS data;          }
     CANMSG_TLMStatusTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_CAR_STATUS data;          }
     CANMSG_CarStatusTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_SET_CAR_STATUS data;      }
     CANMSG_SetCarStatusTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_HV_VOLTAGE data;          }
     CANMSG_HVVoltageTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_HV_CURRENT data;          }
     CANMSG_HVCurrentTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_HV_TEMP data;             }
     CANMSG_HVTemperatureTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_HV_ERRORS data;           }
     CANMSG_HVErrorsTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_TS_STATUS data;           }
     CANMSG_TSStatusTypeDef;
```

```
typedef struct { CANMSG_MetadataTypeDef info; primary_SET_TS_STATUS data;       }
     CANMSG_SetTSStatusTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_LV_CURRENT data;          }
     CANMSG_LVCurrentTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_LV_VOLTAGE data;          }
     CANMSG_LVVoltageTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_LV_TEMPERATURE data;      }
     CANMSG_LVTemperatureTypeDef;
typedef struct { CANMSG_MetadataTypeDef info; primary_COOLING_STATUS data;      }
     CANMSG_CoolingStatusTypeDef;

extern CANMSG_TLMStatusTypeDef          CANMSG_TLMStatus;
extern CANMSG_CarStatusTypeDef          CANMSG_CarStatus;
extern CANMSG_SetCarStatusTypeDef       CANMSG_SetCarStatus;
extern CANMSG_HVVoltageTypeDef          CANMSG_HVVoltage;
extern CANMSG_HVCurrentTypeDef          CANMSG_HVCurrent;
extern CANMSG_HVTemperatureTypeDef      CANMSG_HVTemperature;
extern CANMSG_HVErrorsTypeDef           CANMSG_HVErrors;
extern CANMSG_TSStatusTypeDef           CANMSG_TSStatus;
extern CANMSG_SetTSStatusTypeDef        CANMSG_SetTSStatus;
extern CANMSG_LVCurrentTypeDef          CANMSG_LVCurrent;
extern CANMSG_LVVoltageTypeDef          CANMSG_LVVoltage;
extern CANMSG_LVTemperatureTypeDef      CANMSG_LVTemperature;
extern CANMSG_CoolingStatusTypeDef      CANMSG_CoolingStatus;
```

Listing 3.2: Subset of the CAN messages that are statically defined in the header file

```
CANMSG_TLMStatusTypeDef          CANMSG_TLMStatus          = { };
CANMSG_CarStatusTypeDef          CANMSG_CarStatus          = { };
CANMSG_SetCarStatusTypeDef       CANMSG_SetCarStatus       = { };
CANMSG_HVVoltageTypeDef          CANMSG_HVVoltage          = { };
CANMSG_HVCurrentTypeDef          CANMSG_HVCurrent          = { };
CANMSG_HVTemperatureTypeDef      CANMSG_HVTemperature      = { };
CANMSG_HVErrorsTypeDef           CANMSG_HVErrors           = { };
CANMSG_TSStatusTypeDef           CANMSG_TSStatus           = { };
CANMSG_SetTSStatusTypeDef        CANMSG_SetTSStatus        = { };
CANMSG_LVCurrentTypeDef          CANMSG_LVCurrent          = { };
CANMSG_LVVoltageTypeDef          CANMSG_LVVoltage          = { };
CANMSG_LVTemperatureTypeDef      CANMSG_LVTemperature      = { };
CANMSG_CoolingStatusTypeDef      CANMSG_CoolingStatus      = { };

void CANMSG_process_RX(CAN_MessageTypeDef msg) {
    CANMSG_MetadataTypeDef *msg_to_update = _CANMSG_deserialize_msg_by_id(msg);
    msg_to_update->timestamp = HAL_GetTick();
    msg_to_update->is_new = true;
}
```

Listing 3.3: Instantiation of the message structs and implementation of the updating of CAN packet data upon reception

3.4.3 Encoders

The encoder module exposes three functions. Two read the ground speeds measured from the velocity of the front-right wheel and the front-left wheel encoders, while the third reads the absolute angle of the steering wheel.

```
/**
 * @file      encoders.h
 * @author    Alex Sartori [alex.sartori1997@gmail.com]
 * @date      2021-10-11
 *
 * @brief     Handle reading of Fenice's wheel and steering encoders
 *
 */

#ifndef ENCODERS_H
#define ENCODERS_H

#include "stdint.h"
#include "stdbool.h"


#define ENC_TIM_L htim2 /*< Left encoder timer */
#define ENC_TIM_R htim5 /*< Right encoder tiemr */


/**
 * @brief     Compute the speed of the front-left wheel in meters/second
 * @return    double The speed in m/s
 */
double ENC_L_get_speed_ms();

/**
 * @brief     Compute the speed of the front-right wheel in meters/second
 * @return    double The speed in m/s
 */
double ENC_R_get_speed_ms();


/**
 * @brief     Get the steering encoder's absolute position in degrees
 * @return    double
 */
double ENC_C_get_angle_deg();

#endif
```

Listing 3.4: API of the encoders module

Wheel Encoders

Wheel encoders operate two STM timers in encoder-mode, sending two quadrature-encoded signals each (named A and B), with a phase difference of 90 degrees depending on the turning direction. When configured in this mode, timers receive lines A and B on CH1 and CH2, and increment or decrement their value according to A and B's phase difference. The encoder's magnetic heads hover a magnetic ring with 120 magnetic poles which are internally interpolated to give a resolution of 5 micrometers; therefore, each timer increment corresponds to a 0.000005 meters progress on the magnetic ring circumference.

At maximum measurable speed however, the signal reaches a frequency of 8 MHz, which would overflow a 16-bit timer register in a matter of few milliseconds. Timers with 32-bit counter registers are therefore strictly required, so that they can receive up to 4'294'967'296 pulses before overflowing (instead of 65536 for 16 bits). Consequently, while 16 bit timers would need to be read and reset at most every 16 milliseconds, TIM2 and TIM5 allow for correct counting for over 17 minutes. Nonetheless, for efficient decoding and to avoid periodic overflows, timer counters should be reset to 0 after every read, which as said, is now free to happen at lower frequencies than every 16 milliseconds, if needed.

The code logic is then relatively straightforward: each time a function `ENC_[L|R]_get_speed_ms` is called, the difference between the current and the previous timer value is computed (resulting in the total number of increments happened since the last measurement), and is multiplied by the resolution of the magnetic ring to obtain the total ring length covered. At this point the length value (in meters) is divided by the elapsed time since the last read (in seconds) to obtain the velocity (measured at the ring circumference). Lastly, to convert this value to the ground speed of the wheels, it is multiplied by the ratio between the circumference of the encoder and that of the wheels.

```c
/**
 * @brief     Calculate the ground speed in meters/second
 *
 * @param     tim_counter Timer ticks since the last read
 * @param     ellapsed_ms Milliseconds elapsed since the last read
 * @return    double Ground speed in meters/second
 */
double inline _ENC_calculate_wheel_speed(uint32_t tim_counter, uint32_t ellapsed_ms
    ) {
    return tim_counter * _ENC_encoder_resolution_m * _ENC_speed_multiplier / ((
    double)ellapsed_ms / 1000);
}

/**
 * @brief     Calulate the difference between the current value of the timer
    counter
 *            and its value from the last read
 * @param     htim The timer from which to read
 * @return    uint32_t Timer ticks since the last read
 */
uint32_t _ENC_get_tim_cnt(TIM_HandleTypeDef *htim) {
    uint32_t *last_cnt = (htim == &ENC_TIM_L) ? &_ENC_L_last_cnt : &_ENC_R_last_cnt
    ;
    uint32_t cnt = __HAL_TIM_GET_COUNTER(htim);
    uint32_t diff = min(
        *last_cnt - cnt,
        UINT32_MAX - (*last_cnt - cnt)
    );
    *last_cnt = cnt;
    return diff;
}

/**
 * @brief     Calculate the ground speed from the left wheel encoder
 * @return    double
 */
double ENC_L_get_speed_ms() {
    uint32_t ellapsed_ms = HAL_GetTick() - _ENC_L_last_ticks;
    uint32_t tim_counter = _ENC_get_tim_cnt(&ENC_TIM_L);
    double speed = _ENC_calculate_wheel_speed(tim_counter, ellapsed_ms);
    _ENC_L_last_ticks = HAL_GetTick();
    return speed;
}

/**
 * @brief     Calculate the ground speed from the right wheel encoder
 * @return    double
 */
double ENC_R_get_speed_ms() {
    uint32_t ellapsed_ms = HAL_GetTick() - _ENC_R_last_ticks;
    uint32_t tim_counter = _ENC_get_tim_cnt(&ENC_TIM_R);
    double speed = _ENC_calculate_wheel_speed(tim_counter, ellapsed_ms);
    _ENC_R_last_ticks = HAL_GetTick();
    return speed;
}
```

Listing 3.5: Actual logic that converts timer values to ground speed

More formally:

$$V_{m/s} = \frac{\text{Increments} \cdot \text{Resolution}_m}{\text{Time}_m} \cdot \text{SpeedMultiplier}$$

$$\text{SpeedMultiplier} = \frac{2\pi \cdot \text{WheelRadius}}{2\pi \cdot \text{EncoderRadius}} = \frac{\text{WheelDiameter}}{\text{WheelDiameter}}$$

Steering Encoder

The steering encoder (also referred to as Center/Central Encoder) communicates with the RS422 serial protocol over a clock and a data line.

The first and current implementation synchronously drives the clock signal using a GPIO pin and reads the returned data bit on another GPIO line, progressively shifting it into a 16 bit variable. After this, the value is divided by the max value of the 16 bits (which would represent the full angle) variable and multiplied by 360 degrees and cast as a double. A future, more efficient, implementation will make use of an additional SPI interface that will be able to read the two data bytes asynchronously on the MISO line.

```
/**
 * @brief     Read the absolute steering wheel angle
 * @return    double
 */
double ENC_C_get_angle_deg() {
    uint16_t data_buffer = 0U;
    GPIO_PinState curr_bit;

    for (int bit_i = 0; bit_i < 16; bit_i++) {
        HAL_GPIO_WritePin(CENTER_ENCODER_CLK_GPIO_Port, CENTER_ENCODER_CLK_Pin,
    GPIO_PIN_RESET);
        curr_bit = HAL_GPIO_ReadPin(CENTER_ENCODER_DATA_GPIO_Port,
    CENTER_ENCODER_DATA_Pin);
        data_buffer |= curr_bit;
        data_buffer <<= 1;
        HAL_GPIO_WritePin(CENTER_ENCODER_CLK_GPIO_Port, CENTER_ENCODER_CLK_Pin,
    GPIO_PIN_SET);
    }

    return (double)data_buffer/UINT16_MAX*360;
}
```

Listing 3.6: Code logic that drives the clock signal of the steering encoder and reads ts response bit by bit

3.4.4 Finite State Machine

Among all the sensing modules, the DAS has one specific primary task: holding and controlling the global state of the car: that is, ensure that when in **idle** or **run** mode, the vehicle behaves accordingly and performs all operations coherently.

Software-wise, the FSM logic is contained in the `fsm.h/.c` module, which is mostly generated by a specific library discussed in the next chapter, since its main purpose is to create FSMs as solid as possible by starting from a graph definition in a `.dot` formatted file. At present, the FSM design is still not final, as many tests have still to be carried out, but nonetheless will be here described for completeness' sake.

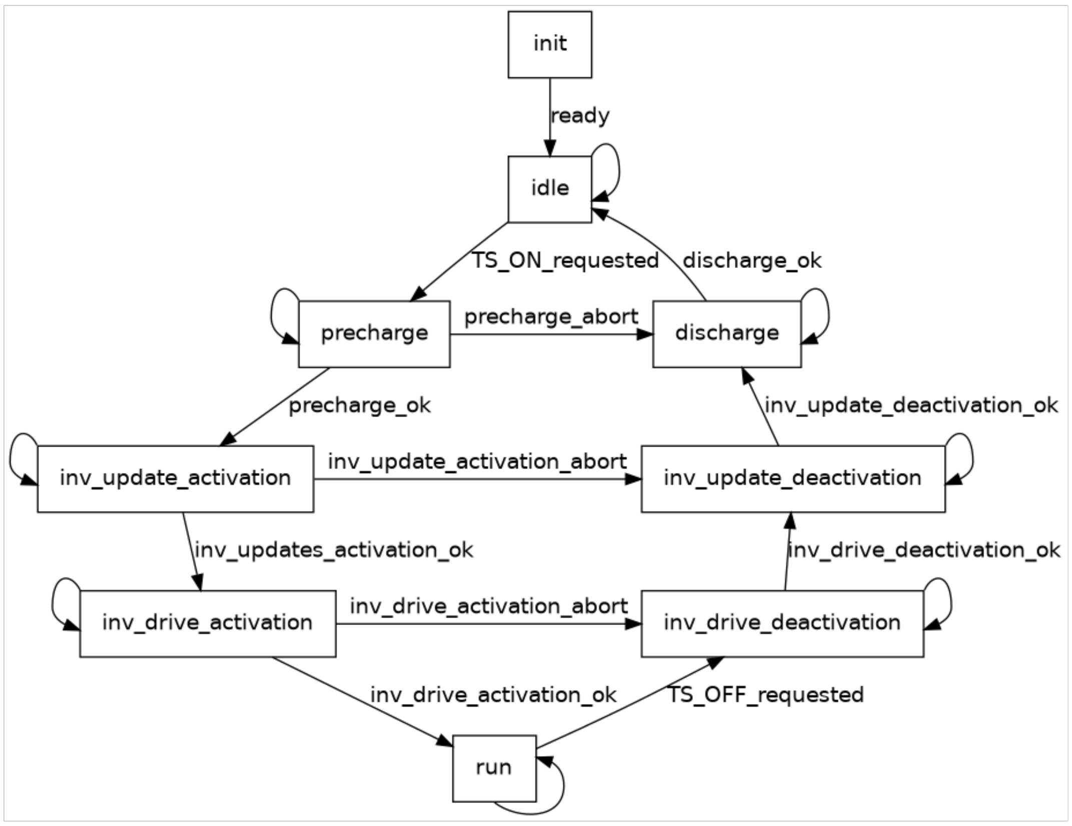


Figure 3.17: Flowchart of Fenice's FSM

Figure 3.17 displays the currently implemented states and transitions, described below:

Init The microcontroller is initializing its state and peripherals.

Idle The DAS is ready to operate and receive inputs. Upon request from the steering wheel to prepare the vehicle to run, enter the **precharge** state.

Precharge Send the BMS-HV a `TS_ON` message and wait for its precharge to complete; then, enter the **inv_updates_activation** state.

Inv. Updates Activation Ask the inverters to start reporting the contents of their status registers every 100ms. When the request is fulfilled, enter the **inv_drive_activation** state.

Inv. Drive Activation Enable the inverters (Drive Mode) and upon confirmation enter the **run** state.

Run Read pedal values from the PCU (or the ADCs directly, in the near future) and forward them to the ECU and the inverters. Upon request from the steering wheel to power down the drivetrain, enter the **inv_drive_deactivation** state.

Inv. Drive Deactivation Disable the inverters and upon confirmation enter the **inv_updates_deactivation** state.

Inv. Updates Deactivation Stop the inverters status reports and enter the **discharge** state.

Discharge Send the BMS-HV a **TS_OFF** request and, upon fulfillment, enter the **idle** state.

3.4.5 Inertial Measurement Unit (IMU)

The IMU sends its data over CAN in the formats detailed in Table 3.4 and 3.3.

Angular Rate, X-axis		Angular Rate, Y-axis		Angular Rate, Z-axis		(unused)	
Byte 0 (MSB)	Byte 1	Byte 2 (MSB)	Byte 3	Byte 4 (MSB)	Byte 5	Byte 6 (MSB)	Byte 7

Table 3.3: Format of an IMU CAN message reporting the angular rate

Acceleration, X-axis		Acceleration, Y-axis		Acceleration, Z-axis		Temperature	
Byte 0 (MSB)	Byte 1	Byte 2 (MSB)	Byte 3	Byte 4 (MSB)	Byte 5	Byte 6 (MSB)	Byte 7

Table 3.4: Format of an IMU CAN message reporting the acceleration rate and the temperature

3.4.6 Inverters

First of all, the inverters need to be configured to operate at the desired baud rate on the CAN bus by writing the appropriate register with the NDrive software. The admissible values are reported in Table 3.5 [4].

Transmission rate (kBaud)	**Register value** (hex)	**Maximum line length** (m)
1000	`0x4002`	20
500	`0x4025`	70
625	`0x4014`	70
250	`0x405c`	100
100	`0x4425`	500

Table 3.5: Configuration values for the inverters' CAN bus

Additionally, the user has to configure the transmitting and receiving addresses by modifying the registers described in Table 3.6.

Address	**Register**	**Default value**	**Range**
Receiving	RPD01	`0x201`	`0x201` to `0x27F`
Transmitting	TPD01	`0x181`	`0x181` to `0x1FF`

Table 3.6: Configuration values for the inverters' RX and TX CAN addresses

According to documentation, CAN frames are encoded as Little-Endian (Bit7 to 0 / Bit15 to 8 / Bit23 to 16 / Bit31 to 24). Messages are expected in a 3 or 5 byte data packets when received and 4 or 6 when sent. It is also possible to use messages of up to 8 bytes, which however, would still be evaluated as 5 bytes. The identifier is the standard 11 bit wide, plus the RTR function bit (Remote Transmission Request, which is however ignored) and the DLC information (Data Length Code). The byte 0 of the data field is for the REGID index (parameter number), the second to the fifth bytes (byte 1 to byte 4) instead contain data for the indicated REGID.

Example: to send a torque command of 50% (dec. 16380, hex. 0x3FFC), the CAN frame would look like so:

- Header
  - **ID**: **0x201** (default)

  - **RTR**: **0** (ignored)
  - **DLC**: **3** (REGID + 2 data bytes)

- Body
  - **Byte 0**: ID of the *TORQUE-CMD* register, **0x90**
  - **Byte 1**: bits 7 to 0, **0xFC**
  - **Byte 2**: bits 15 to 8, **0x3F**
  - **Byte 3**: –
  - **Byte 4**: –

The inverters firmware module exposes the following API:

```c
#ifndef INVERTERS_H
#define INVERTERS_H

#include "can.h"


typedef struct {
    uint32_t val;
    uint16_t Vout;
} INV_StatusTypeDef;


void INV_power_on();
void INV_power_off();

void INV_enable_status_updates();
void INV_disable_status_updates();
void INV_enable_vout_updates();
void INV_disable_vout_updates();

bool INV_are_updates_enabled();
bool INV_R_is_enabled();
bool INV_L_is_enabled();

INV_StatusTypeDef INV_R_read_status();
INV_StatusTypeDef INV_L_read_status();

void INV_send_torque(float);

void INV_process_CAN_msg(CAN_MessageTypeDef*);

#endif
```

Listing 3.7: API of the inverters module

Essentially, this module serves two purposes: reading detailed status updates and sending commands to operate the motors. The status struct holds the voltage being output and a bitmap of the inverter status as periodically reported after the `INV_enable_status_updates()` function enabled the function. More specifically, updates are enabled and disabled with a *transmission request command* (ID `0x3D`) which accepts as parameters the REGID to read and transmit back, and a timing value to define the repetition interval of the request (with **0x00** meaning to reply only once, and **0xFF** to stop).

Example The message that enables the updates on the status map register is composed as follows:

- **ID**: `0x201` (default)
- **DLC**: 3 (TX req. command + 2 parameters)

- **Byte 0**: `0x3D` (TX Request Command)
- **Byte 1**: `0x40` (`STATUS_MAP` register)
- **Byte 2**: `0x64` (decimal: 100ms)

Example For sending a torque command instead, the message is formatted as a write command on a register:

- **ID**: `0x201` (default)
- **DLC**: 3 (REGID + 16 bits integer i.e. 2 bytes)
- **Byte 0**: `0x90` (TORQUE_CMD register)
- **Byte 1**: LSB of the torque value
- **Byte 2**: MSB of the torque value

3.4.7 Pedals Control Unit (PCU)

The PCU modules exposes functions for reading the last values received for the accelerator and brake pedals sensors, plus a function to scale this value to a range between 0 and 100 according to the calibration boundaries configured during the relative phase.

At the moment, the calibration process is implemented only partially since the PCU board doesn't yet respond to the relevant CAN messages. However, the DAS PCB contains an EEPROM chip intended to store the ADC ranges data.

```c
typedef struct {
    uint16_t accel1_adc_min;
    uint16_t accel1_adc_max;
    uint16_t accel2_adc_min;
    uint16_t accel2_adc_max;
} PCU_CalibValTypeDef;

PCU_CalibValTypeDef _PCU_read_calibration_values() {
    m95256_t heeprom;
    PCU_CalibValTypeDef calib_vals;

    m95256_init(&heeprom, EEPROM_HSPI, EEPROM_GPIO, EEPROM_CSPIN);
    m95256_ReadBuffer(heeprom, (uint8_t*)&calib_vals, 0x0, sizeof(
    PCU_CalibValTypeDef));
    m95256_deinit(&heeprom);

    return calib_vals;
}

void PCU_save_calibration_values(PCU_CalibValTypeDef calib) {
    m95256_t heeprom;

    m95256_init(&heeprom, EEPROM_HSPI, EEPROM_GPIO, EEPROM_CSPIN);
    m95256_WriteBuffer(heeprom, (uint8_t*)&calib, 0x0, sizeof(PCU_CalibValTypeDef))
    ;
    m95256_deinit(&heeprom);
}
```

Listing 3.8: Code sample from the PCU that loads and stores the ADC calibration values

3.4.8 Traction Control

The traction Control module is in draft state and ready to be expanded with CRC support as soon as the ECU will be able to transmit values over UART or SPI. At the moment it consists of an implementation optimized for SPI communication as slave device (since the ECU RPi can only act as master for firmware constrainsts).

Essentially, an SPI exchange consists in the master device activating a clock line (CLK) and transmitting a byte on the MOSI (Master-Out, Slave-In) line while at the same time reading one other byte on the MISO (Master-In, Slave-Out) line. Figure 3.18 displays the exchange of data that will happen between the ECU and the DAS. On the left-side, the view is abstract and shows the overall set of parameters and their direction, while on the right-side it is detailed at a lower level how a 4-byte float value (plus CRC byte) would be requested and read. Note that arrows are closer in pairs to represent that they happen simultaneously on the MOSI and MISO lines.

In the specific protocol being tested, and as can be inferred from the first entry in the second part of the image mentioned above, at each transmission the Raspberry Pi signals with a predefined reserved byte to the slave to prepare in memory a new value to be sent next, and reads the previously prepared one (if present, none otherwise), so that the throughput is maximized.

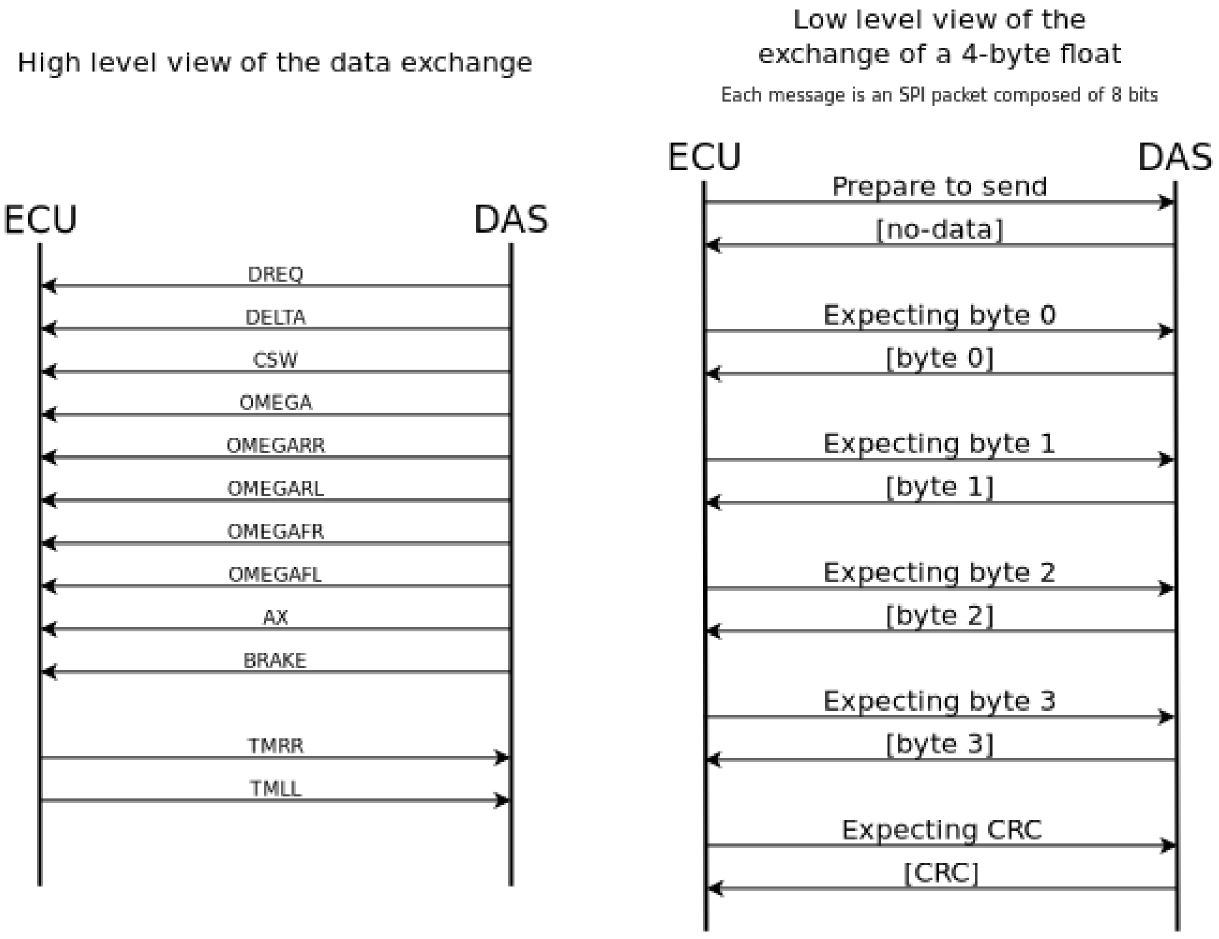
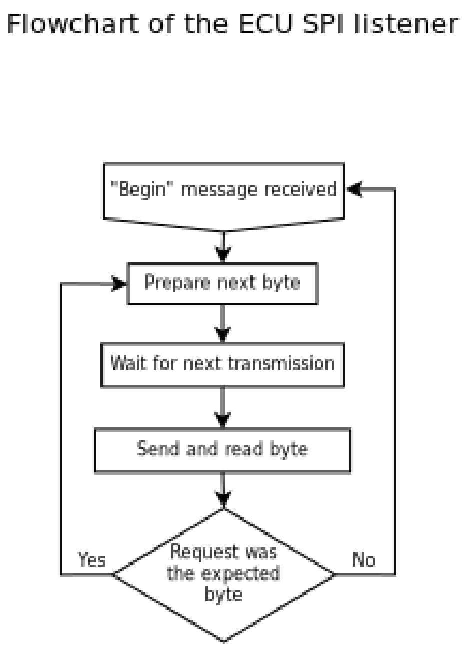
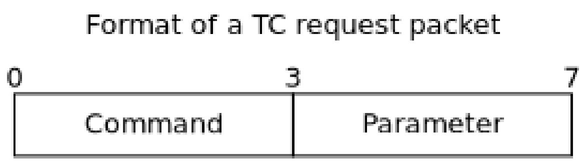


Figure 3.18: High-level and low-level representation of the Traction Control SPI protocol

Firmware-wise, Figure 3.19 shows on the left the state machine of the DAS when receiving, and on the right side a decoding table for all commands and parameters that can be encoded in one byte. For instance, if the ECU were to request the DAS to prepare to send the OMEGA-FL parameter, it would activate the CLK line and send, bit by bit on the MOSI line, the "prepare to send" command (`0010`) plus the "OMEGA-FL" parameter (`0111`), resulting in the transmitted byte `00100111` = DEC `39` = HEX `0x27`. Next, to receive the actual 4 bytes that compose the OMEGA-FL floating point value, the ECU would need to activate the clock line and, for consistency's sake, "acknowledge" every byte that it is expecting with the ”Expecting byte X” command (0100, 0101, 0110, 0111) each followed by, again, OMEGA-FL (0111).


Flowchart of the ECU SPI listener
"Begin" message received
Prepare next byte
Wait for next transmission
Send and read byte
Request was the expected byte
Yes
No



Command	Description
0000	Ping
0001	Pong
0010	Prepare to send data
0011	Prepare to receive data
0100	Expecting byte 0
0101	Expecting byte 1
0110	Expecting byte 2
0111	Expecting byte 3
1000	Expecting CRC byte

Parameter	Description
0000	DREQ
0001	DELTA
0010	CSW
0011	OMEGA
0100	OMEGARR
0101	OMEGARL
0110	OMEGAFR
0111	OMEGAFL
1000	AX
1001	BRAKE
1010	TMRR
1011	TMLL

Figure 3.19: State machine of the TC protocol and encoding table of commands and parameters

Instead of reporting the whole TC module, only the most ”representative” function follows:

```
/**
 * @brief     Treat the received byte as a command packet (4 bits of cmd and 4 bits
    of param) and prepare a reply byte
 *
 * @param     rx_byte The received byte
 * @param     tx_byte The next byte to send as a reply
 */
void _TC_process_cmd(uint8_t rx_byte, uint8_t *tx_byte) {
    /* Aliases for better readability */
    _TC_ProtocolStateTypeDef *state = &_TC_protocol_state;

    /* Unpack byte into command and parameter */
    uint8_t cmd = rx_byte >> 4;
    uint8_t param = rx_byte & 0x0F;

    /* Parse command */
    switch (cmd) {
        case TC_CMD_PREP_SEND:
            state->error_state = TC_PROTO_NO_ERROR;
            state->tx_buf_idx = 0;
            state->current_param = param;
            _TC_prepare_param_in_buffer();
            *tx_byte = 0x0;
            break;

        case TC_CMD_PREP_RECV:
            state->error_state = TC_PROTO_NO_ERROR;
            state->rx_buf_idx = 0;
            state->current_param = param;
            state->is_receiving_raw_bytes = true;
            *tx_byte = TC_CMD_EXP_B0;
            break;

        case TC_CMD_EXP_B0:
            /* We should have sent byte #0, prepare byte #1 */
            if (state->tx_buf_idx != 0)
                _TC_set_error(TC_PROTO_UNEXP_CMD);
            else
                state->tx_buf_idx++;
            break;

        case TC_CMD_EXP_B1:
            /* We should have sent byte #1, prepare byte #2 */
            if (state->tx_buf_idx != 1)
                _TC_set_error(TC_PROTO_UNEXP_CMD);
            else
                state->tx_buf_idx++;
            break;

        case TC_CMD_EXP_B2:
            /* We should have sent byte #2, prepare byte #3 */
            if (state->tx_buf_idx != 2)
                _TC_set_error(TC_PROTO_UNEXP_CMD);
            else
                state->tx_buf_idx++;
            break;

        case TC_CMD_EXP_B3:
            /* We should have sent byte #3, prepare CRC */
            if (state->tx_buf_idx != 3)
                _TC_set_error(TC_PROTO_UNEXP_CMD);
            else
                ; // TODO: Prepare CRC
            break;
        case TC_CMD_EXP_CRC:
            // TODO: Handle CRC
            break;

        default:
            _TC_set_error(TC_PROTO_UNEXP_CMD);
            break;
    }
}
```

Listing 3.9: Traction Control function that parses an incoming command

3.4.9 Tractive System

This module commands the BMS-HV when to power-on and off, based on requests received by the steering wheel.

```
/**
 * @brief     Check and update the car status
 */
void _TS_update_TS_CAN_status() {
    CANMSG_SetCarStatus.info.is_new = false;
    if (CANMSG_SetCarStatus.data.car_status_set == Primary_Car_Status_Set_IDLE)
        _TS_request_poweroff();
    else if (CANMSG_SetCarStatus.data.car_status_set == Primary_Car_Status_Set_RUN)
        _TS_request_poweron();
    else
        LOG_write(LOGLEVEL_ERR, "[CAN] Received invalid SET_CAR_STATUS");
}

/**
 * @brief     Update the TS's CAN TX message to request TS-ON
 */
void _TS_request_poweron() {
    CANMSG_SetTSStatus.data.ts_status_set = Primary_Ts_Status_Set_ON;
    CANMSG_SetTSStatus.info.is_new = true;
    CANMSG_SetTSStatus.info.timestamp = HAL_GetTick();
}

/**
 * @brief     Update the TS's CAN TX message to request TS-OFF
 */
void _TS_request_poweroff() {
    CANMSG_SetTSStatus.data.ts_status_set = Primary_Ts_Status_Set_OFF;
    CANMSG_SetTSStatus.info.is_new = true;
    CANMSG_SetTSStatus.info.timestamp = HAL_GetTick();
}
```

Listing 3.10: Code sample illustrating how the TS module - upon request by steering - prepares a message to modify the state of the BMS-HV

Secondly, this module keeps track of the BMS's internal status by updating an enumeration each time an update via CAN is received.

```
/**
 * @brief     Read the last reported status from the BMS-HV
 */
void _TS_update_BMS_CAN_status() {
    if (!CANMSG_TSStatus.info.is_new) return;

    CANMSG_TSStatus.info.is_new = false;

    switch (CANMSG_TSStatus.data.ts_status) {
        case Primary_Ts_Status_OFF:
            _TS_status = TS_STATUS_OFF;
            break;
        case Primary_Ts_Status_PRECHARGE:
            _TS_status = TS_STATUS_PRECHARGE;
            break;
        case Primary_Ts_Status_ON:
            _TS_status = TS_STATUS_ON;
            break;
        case Primary_Ts_Status_FATAL:
            _TS_status = TS_STATUS_FATAL;
            break;
        default:
            LOG_write(LOGLEVEL_ERR, "[TS] Received invalid TS_STATUS");
            break;
    }
}
```

Listing 3.11: TS function periodically called to update the BMS status from CAN

3.4.10 Extra: Custom Logging Library

Since the number of modules that operate simultaneously on the MCU started to grow, and most of them need to output several debug messages to ease development and troubleshooting, a custom logging library was created that could provide these features:

- **No overhead**: there might be the need to invoke it inside an ISR and must be therefore be kept as quick as possible.
- **Different log levels**: often times, errors or warnings need to be the only events logged without additional clutter. Other times, problems might come from deeper zones of the code and a more verbose output is needed. Ideally, this change of policy should happen with the switch of a flag, and not manual intervention all over the code base.
- **Printf-like formatting**: most logging messages are much more useful if printed along with some parameters. Manually allocating a buffer each time and invoke formatting functions is very error-prone, cumbersome, and inconvenient. With the use of C's built-in variable arguments, the user can invoke the library with the same API as the standard printf functions.
- **Auxiliary information**: sometimes, additional help can come from knowing where exactly the message generated. For this reason, with the use of a macro, the library can be configured to prepend to each entry the source file and line number.
- **Redirectable output**: having no standard console, the best destination for logging messages is the serial interface. With the goal of making the library versatile and usable by other members of the team on other MCUs and with other hardware configurations, the "raw" print function is defined as `__weak` so that each user can implement it in the most suitable way for their use-case.

The addition of the information on which file and line generated a message happens through a macro:

```c
/**
 * @brief     Log a message with a corresponding informative level
 * @param     loglevel A ‘LOG_LogLevelTypeDef‘ defining the message type
 * @param     fmt A printf-style format string to create the message
 * @param     ... Format string parameters
 */
#define LOG_write(...) _LOG_write(__FILE__, __LINE__, __VA_ARGS__)
void _LOG_write(char*, uint32_t, LOG_LogLevelTypeDef, char*, ...);
```

The core functionality of the library is implemented in this function:

```c
/**
 * @brief     Write a log message to UART in printf-style
 *
 * @param     filename Internally macro-expanded to the caller's filename
 * @param     lineno Internally macro-expanded to the caller's line number
 * @param     loglevel Verbose level of the message for filtering
 * @param     fmt Format string
 * @param     ... Format string parameters
 */
void _LOG_write(char* filename, uint32_t lineno, LOG_LogLevelTypeDef loglevel, char
     *fmt, ...) {
    /* If not needed by current log level, return */
    if (loglevel < _LOG_curr_loglevel)
        return;

    /* Create buffer */
    uint16_t buf_size = 100;
    uint16_t offset = 0;
    char buffer[buf_size];
    if (_LOG_print_lineno) {
        /* Print source filename and line number */
        offset += snprintf(buffer, buf_size, "%s:%lu ", filename, lineno);
    }

    if (_LOG_print_prefix) {
        /* Print loglevel prefix */
        char* prefixes[] = {"[DBG] ", "[INF] ", "[WRN] ", "[ERR] "};
        offset += snprintf(
            buffer+offset, buf_size-offset,
            prefixes[loglevel]
        );
    }

    /* Print actual log message */
    va_list args;
    va_start(args, fmt);
    offset += vsnprintf(buffer+offset, buf_size-offset, fmt, args);
    va_end(args);

    /* Write message */
    _LOG_write_raw(buffer);

    if (offset >= buf_size && buf_size > 75) // If the error message fits without
    generating a recursive error
        LOG_write(LOGLEVEL_WARN, "[LOG] Previous log message was truncated due to
    buffer size (%ld chars)", buf_size);

}
```

4 Reliability of the System

Considered that the DAS is one of the most critical control units for the vehicle's correct operation, an important effort has been put into ensuring that the code-base is as solid as possible.

4.1 BARR-C Conventions

Barr Group's Embedded C Coding Standard was designed specifically to reduce the number of programming defects in embedded software. By following this coding standard, firmware developers not only reduce hazards to users and time spent in the debugging stage of their projects but also improve the maintainability and portability of their software. Together these outcomes can greatly lower the cost of developing high-reliability embedded software. This "BARR-C" coding standard is different from other coding standards. Rather than being based on the stylistic preferences of the authors, the rules in this standard were selected for their ability to minimize defects. When it was the case that one rule had the ability to prevent more defects from being made by programmers than an alternative rule for a similar aspect of coding, that more impactful rule was chosen. For example, the stylistic rules for when and where to place curly braces were selected on the basis of their ability to reduce bugs across a whole program. Other important reasons to adopt this coding standard include increased readability and portability of source code. Some of the guiding principles of this document are:

- For better or worse (well, mostly worse), the ISO C Programming Language "Standard" permits a considerable amount of variation between compilers. The ISO C Standard's implementation-defined, unspecified, and undefined behaviors, along with locale-specific options, mean that even programs compiled from identical source code but via different ISO C-compliant compilers may behave very differently at run-time. Such gray areas in the C language standard greatly reduce the portability of source code that is not carefully crafted.

- The reliability, readability, efficiency, and sometimes portability of source code is more important than programmer convenience.

[2]

The rest of the rules is divided into sections that concern the different aspects of the implementation process. Here, the most considered and valuable are reported.

4.1.1 Abbreviations

Abbreviations and acronyms should generally be avoided unless their meanings are widely and consistently understood in the engineering community. For instance, ADC, BMS, or GPIO might be considered well-understood acronyms in our team and field, while casually abbreviating function names by removing vowels or trimming nouns is to be avoided.

4.1.2 Modules

Naming

All module names shall consist entirely of lowercase letters, numbers, and underscores. No spaces shall appear within the module's header and source file names. No module's header file name shall share the name of a header file from the C Standard Library or C++ Standard Library.

Header Files

There shall always be precisely one header file for each source file and they shall always have the same root name. Each header file shall contain a preprocessor guard against multiple inclusion. The header file shall identify only the procedures, constants, and data types (via prototypes or macros, defines, and typedefs, respectively) about which it is strictly necessary for other modules to be informed.

Source Files

Each source file shall include only the behaviors appropriate to control one entity. Examples of entities include encapsulated data types, active objects, peripheral drivers (e.g., for a UART), and communication protocols or layers (e.g., ARP). Each source file shall be comprised of some or all of the following sections, in the order listed: comment block; include statements; data type, constant, and macro definitions; static data declarations; private function prototypes; public function bodies; then private function bodies. Each source file shall always include the header file of the same name (e.g., file adc.c should include adc.h), to allow the compiler to confirm that each public function and its prototype match. Absolute paths shall not be used in include file names. Each source file shall be free of unused include files. No source file shall include another source file.

4.1.3 Data Types

All new structures, unions, and enumerations shall be named via a typedef. The name of all public data types shall be prefixed with their module name and an underscore. Whenever the width, in bits or bytes, of an integer value matters in the program, one of the fixed width data types shall be used in place of char, short, int, long, or long long. The keywords short and long shall not be used. Use of the keyword char shall be restricted to the declaration of and operations concerning strings. Signed integers shall not be combined with unsigned integers in comparisons or expressions. In support of this, decimal constants meant to be unsigned should be declared with a 'u' at the end. Avoid the use of floating point constants and variables whenever possible. Fixed-point math may be an alternative. Never test for equality or inequality of floating point values. Non-Boolean values shall be converted to Boolean via use of relational operators (e.g., ¡ or !=), not via casts.

4.1.4 Procedures and Functions

No procedure shall have a name that is a keyword of any standard version of the C or C++ programming language or that overlaps a function in the C Standard Library. Each procedure's name shall be descriptive of its purpose. Note that procedures encapsulate the "actions" of a program and thus benefit from the use of verbs in their names (e.g., adc_read()); this "noun-verb" word ordering is recommended. Alternatively, procedures may be named according to the question they answer (e.g., led_is_on()). The names of all public functions shall be prefixed with their module name and an underscore (e.g., sensor_read()). All reasonable effort shall be taken to keep the length of each function limited to one printed page, or a maximum of 100 lines. *(In our case, we try to limit it even further, to about 30 lines at most)*. It is a preferred practice that all functions shall have just one exit point and it shall be via a return at the bottom of the function. A prototype shall be declared for each public function in the module header file.

4.1.5 Variables and Statements

All variables shall be initialized before use. It is preferable to define local variables as you need them, rather than all at the top of a function. If project- or file-global variables are used, their definitions shall be grouped together and placed at the top of a source code file. Any pointer variable lacking an initial address shall be initialized to NULL. All switch statements shall contain a default block. Any case designed to fall through to the next shall be commented to clearly explain the absence of the corresponding break. Magic numbers shall not be used as the initial value or in the endpoint test of a while, do...while, or for loop. Each loop with an empty body shall feature a set of braces enclosing a comment to explain why nothing needs to be done until after the loop terminates.

4.2 Conventions From the "Clean Code" Book

This book by Robert Martin presents a great number of practices and heuristics that I personally adopt for producing organized and easy to maintain code.

4.2.1 Names

Informative names

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

Avoiding Disinformation

Do not refer to a grouping of accounts as an *accountList* unless it's actually a List. If the container holding the accounts is not actually a List, it may lead to false conclusions. A truly awful example of disinformative names would be the use of lower-case L or uppercase O as variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively. Noise words are meaningless distinction. Imagine that you have a *Product* class. If you have another called *ProductInfo* or *ProductData*, you have made the names different without making them mean anything different. Info and Data are indistinct noise words like a, an, and the. Pick one word for one abstract concept and stick with it. For instance, it's confusing to have *fetch*, *retrieve*, and *get* as equivalent methods.

Pronounceable and Searchable Names

If you can't pronounce it, you probably cannot discuss it efficiently either. One might easily grep for *MAX_CLASSES_PER_STUDENT*, but the number 7 could be more troublesome. Likewise, the name *e* is a poor choice for any variable for which a programmer might need to search.

4.2.2 Functions

The first rule of functions is that they should be small.

Doing One Thing

To know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation. In order to make sure our functions are doing "one thing", we need to make sure that the statements within our function are all at the same level of abstraction. Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. We want the code to read like a top-down narrative. We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

Descriptive Names

Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name or a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.

Arguments

Arguments are hard. They take a lot of conceptual power. The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification and shouldn't be used anyway. Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there's one argument, it's not too hard. With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

4.2.3 Comments

Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation. Code changes and evolves. Chunks of it move from here to there. Those chunks bifurcate and reproduce and come together again to form chimeras. Unfortunately the comments don't always follow them - *can't* always follow them. And all too often the comments get separated from the code they describe and become orphaned blurbs of ever decreasing accuracy.

One of the more common motivations for writing comments is bad code. Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

It is sometimes useful to provide basic information with a comment, however. Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable. In general it is better to find a way to make that argument or return value clear in its own right; but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful. There is a substantial risk, of course, that a clarifying comment is incorrect.

It is sometimes reasonable to leave "To do" notes in the form of *//TODO* comments. TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem. It might be a request for someone else to think of a better name or a reminder to make a change that is dependent on a planned event. Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

4.2.4 Formatting

Vertical Formatting

mall files are usually easier to understand than large files are. Think of a well-written newspaper article. You read it vertically. At the top you expect a headline that will tell you what the story is about and allows you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutia. We would like a source file to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.

Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines. If openness separates concepts, then vertical density implies close association, so lines of code that are tightly related should appear vertically dense. Concepts that are closely related should be kept vertically close to each other. For those concepts that are so closely related that they belong in the same source file, their vertical separation should be a measure of how important each is to the understandability of the other. We want to avoid forcing our readers to hop around through our source files and classes.

Horizontal Formatting

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. For instance, assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

Aligning variable names in a list of declarations seems to emphasize the wrong things and leads the eye away from the true intent. For example, one is tempted to read down the list of variable names without looking at their types. Likewise, in a list of assignment statements one is tempted to look down the list of *rvalues* without ever seeing the assignment operator.

A source file is a hierarchy rather like an outline. Each level of this hierarchy is a scope into which names can be declared and in which declarations and executable statements are interpreted. To make this hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hierarchy.
[6]

4.3 Code Testing

Even though of crucial importance, test units are applicable with a certain degree of difficulty in this particular application. Most often, the code is tightly coupled with the underlying hardware if not for rare abstraction functions. Consequently, these decoupled functions are the main feasible target, to not add excessive complexity, for our testing procedures. The second most approachable target are instead the various external libraries we developed to accommodate specific needs, which lie at a certain level of intrinsic abstraction for obvious implementation reasons.

For instance, this is an extract from a suite of tests based on the *munit* testing framework[1] for a custom priority queue.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MUNIT_ENABLE_ASSERT_ALIASES
```

[1]https://nemequ.github.io/munit/

```c
#include "../../munit/munit.h"
#include "../priority_queue.h"
#include "tests.h"

#define PQ_SIZE 100

typedef struct msg {
    uint8_t n;
    char c[8];
} msg;

// [...]

MunitResult test_is_empty(const MunitParameter *params, void *data) {
    PQ_QueueTypeDef q;
    PQ_init(&q, PQ_SIZE, sizeof(msg), _cmp_op, _pop_op);

    assert_true(PQ_is_empty(q));

    msg m = {1, "abc"};
    PQ_insert(q, 1, &m);

    assert_false(PQ_is_empty(q));

    PQ_destroy(&q);
    return MUNIT_OK;
}

// [...]

MunitResult test_random_insert(const MunitParameter *params, void *data) {
    PQ_PriorityTypeDef IDs[PQ_SIZE];

    srand(42);
    for (int i = 0; i < PQ_SIZE; i++)
        IDs[i] = rand() % UINT8_MAX;

    PQ_QueueTypeDef q;
    PQ_init(&q, PQ_SIZE, sizeof(msg), _cmp_op, _pop_op);

    for (int i = 0; i < PQ_SIZE; i++) {
        msg m = {IDs[i], "abc"};
        PQ_insert(q, IDs[i], &m);
    }

    qsort(IDs, PQ_SIZE, sizeof(IDs[0]), _cmp_int);

    for (int i = 0; !PQ_is_empty(q); i++) {
        msg *m = (msg *)PQ_peek_highest(q);
        assert_uint16(IDs[i], ==, m->n);
        PQ_pop_highest(q, NULL);
    }

    PQ_destroy(&q);
    return MUNIT_OK;
}
```

4.4 FSM From a Graph File

GV_FSM[2] is a tool for creating stubs in C/C++ for implementing Finite State Machines. The advantage that made it suitable for the DAS is that the code it generates is based on an input file written in Graphviz format. In this way, the programmer can be entirely sure that the transition table

[2]https://github.com/pbosetti/gv_fsm

completely reflects the intended design and does not allow unexpected behavior to happen or miss unnoticed transitions. The conventions adopted for the input scheme are very straightforward: nodes represent possible states labeled with their names, and directed edges represent state transitions (with an optional transition function if a label is given).
In the specific case of the DAS, the graph in figure 3.17 resulted in the following header file:

```
/******************************************************************************
Finite State Machine
Project: VFSM
Description: vehicle_fsm

Generated by gv_fsm ruby gem, see https://rubygems.org/gems/gv_fsm
gv_fsm version 0.2.9
Generation date: 2021-11-16 16:08:09 +0100
Generated from: fsm.dot
The finite state machine has:
  9 states
  12 transition functions
Functions and types have been generated with prefix "VFSM_"
******************************************************************************/

#ifndef FSM_H
#define FSM_H
#include <stdlib.h>

// State data object
// By default set to void; override this typedef or load the proper
// header if you need
typedef void VFSM_state_data_t;

// NOTHING SHALL BE CHANGED AFTER THIS LINE!

// List of states
typedef enum {
  VFSM_STATE_INIT = 0,
  VFSM_STATE_IDLE,
  VFSM_STATE_PRECHARGE,
  VFSM_STATE_DISCHARGE,
  VFSM_STATE_INV_UPDATE_ACTIVATION,
  VFSM_STATE_INV_UPDATE_DEACTIVATION,
  VFSM_STATE_INV_DRIVE_ACTIVATION,
  VFSM_STATE_INV_DRIVE_DEACTIVATION,
  VFSM_STATE_RUN,
  VFSM_NUM_STATES,
  VFSM_NO_CHANGE
} VFSM_state_t;

// State human-readable names
extern const char *state_names[];

// State function and state transition prototypes
typedef VFSM_state_t state_func_t(VFSM_state_data_t *data);
typedef void transition_func_t(VFSM_state_data_t *data);

// State functions

// Function to be executed in state init
// valid return states: VFSM_STATE_IDLE
VFSM_state_t VFSM_do_init(VFSM_state_data_t *data);

// Function to be executed in state idle
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_IDLE, VFSM_STATE_PRECHARGE
VFSM_state_t VFSM_do_idle(VFSM_state_data_t *data);
```

```c
// Function to be executed in state precharge
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_PRECHARGE, VFSM_STATE_DISCHARGE,
     VFSM_STATE_INV_UPDATE_ACTIVATION
VFSM_state_t VFSM_do_precharge(VFSM_state_data_t *data);

// Function to be executed in state discharge
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_IDLE, VFSM_STATE_DISCHARGE
VFSM_state_t VFSM_do_discharge(VFSM_state_data_t *data);

// Function to be executed in state inv_update_activation
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_INV_UPDATE_ACTIVATION,
    VFSM_STATE_INV_UPDATE_DEACTIVATION, VFSM_STATE_INV_DRIVE_ACTIVATION
VFSM_state_t VFSM_do_inv_update_activation(VFSM_state_data_t *data);

// Function to be executed in state inv_update_deactivation
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_DISCHARGE,
    VFSM_STATE_INV_UPDATE_DEACTIVATION
VFSM_state_t VFSM_do_inv_update_deactivation(VFSM_state_data_t *data);

// Function to be executed in state inv_drive_activation
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_INV_DRIVE_ACTIVATION,
    VFSM_STATE_INV_DRIVE_DEACTIVATION, VFSM_STATE_RUN
VFSM_state_t VFSM_do_inv_drive_activation(VFSM_state_data_t *data);

// Function to be executed in state inv_drive_deactivation
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_INV_UPDATE_DEACTIVATION,
    VFSM_STATE_INV_DRIVE_DEACTIVATION
VFSM_state_t VFSM_do_inv_drive_deactivation(VFSM_state_data_t *data);

// Function to be executed in state run
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_INV_DRIVE_DEACTIVATION,
    VFSM_STATE_RUN
VFSM_state_t VFSM_do_run(VFSM_state_data_t *data);


// List of state functions
extern state_func_t *const VFSM_state_table[VFSM_NUM_STATES];


// Transition functions
void VFSM_ready(VFSM_state_data_t *data);
void VFSM_TS_ON_requested(VFSM_state_data_t *data);
void VFSM_precharge_abort(VFSM_state_data_t *data);
void VFSM_precharge_ok(VFSM_state_data_t *data);
void VFSM_discharge_ok(VFSM_state_data_t *data);
void VFSM_inv_update_activation_abort(VFSM_state_data_t *data);
void VFSM_inv_updates_activation_ok(VFSM_state_data_t *data);
void VFSM_inv_update_deactivation_ok(VFSM_state_data_t *data);
void VFSM_inv_drive_activation_abort(VFSM_state_data_t *data);
void VFSM_inv_drive_activation_ok(VFSM_state_data_t *data);
void VFSM_inv_drive_deactivation_ok(VFSM_state_data_t *data);
void VFSM_TS_OFF_requested(VFSM_state_data_t *data);

// Table of transition functions
extern transition_func_t *const VFSM_transition_table[VFSM_NUM_STATES][
    VFSM_NUM_STATES];

// state manager
VFSM_state_t VFSM_run_state(VFSM_state_t cur_state, VFSM_state_data_t *data);

#endif
```

Plus, the corresponding source file with function bodies already laid out to accept and reject the correct sets of states. For instance, the listing below shows the state function called when in *PRECHARGE*. The first switch statement was added to implement the correct logic, while the second one is a generated stub that ensures coherence in the FSM transitions.

```
// Function to be executed in state precharge
// valid return states: VFSM_NO_CHANGE, VFSM_STATE_PRECHARGE, VFSM_STATE_DISCHARGE,
     VFSM_STATE_INV_UPDATE_ACTIVATION
VFSM_state_t VFSM_do_precharge(VFSM_state_data_t *data) {
    LOG_write(LOGLEVEL_DEBUG, "[FSM] In state precharge");

    VFSM_state_t next_state = VFSM_NO_CHANGE;

    switch (TS_get_status()) {
        case TS_STATUS_OFF:
        case TS_STATUS_FATAL:
            next_state = VFSM_STATE_DISCHARGE;
            break;
        case TS_STATUS_ON:
            next_state = VFSM_STATE_INV_UPDATE_ACTIVATION;
            break;
    }

    switch (next_state) {
        case VFSM_NO_CHANGE:
        case VFSM_STATE_PRECHARGE:
        case VFSM_STATE_DISCHARGE:
        case VFSM_STATE_INV_UPDATE_ACTIVATION:
            break;
        default:
            LOG_write(LOGLEVEL_WARN, "[FSM] Cannot pass from precharge to %s,
    remaining in this state", state_names[next_state]);
            next_state = VFSM_NO_CHANGE;
    }
    return next_state;
}
```

4.5 CAN CICD

The purpose of this project is to manage any CAN network, the data flowing through it, its serialization/deserialization and the interactions with the bus itself. CAN CICD was born to solve the many issues regarding the CAN network, such as the following:

- Lack of a single source of truth for the CAN network description (hassle-free alternative to DBC files)
- Human error during the development (deserialization errors, wrong bitrate, etc.)
- Desynchronization between different nodes (ie. a message id has changed and the developer forgot to update the code)
- Message loss due to conflicting ids or wrong id priority
- Lack of visual representation of message definitions, useful during the development process

To accomplish such goals, it features the following functionalities:

- Network description format
  - JSON-based, simple, hassle-free and powerful
  - Single source of truth

- Automatic message IDs generation
  - Accounts for priority assigning **lower ids** to **higher priority** messages
  - IDs can be filtered by masking the least significant bits and filtering by topic
  - Generation of masks and filters based on the concept of topic, filter by the topics that your device has interest in
- Automatic source code generation for C, C++, Python and Javascript
  - Serialization/deserialization code generation with zero-copy zero-overhead
  - Header files with constants and types
  - Header files with CAN configuration such as bitrate, frequency, etc.
- Spreadsheet generation
  - Visual representation of the CAN networks with messages payloads, ids, topics, etc.

The configuration of a network consists in structuring a JSON file such as the following:

```
{
    "network_version": decimal,
    "max_payload_size": integer,
    "custom_types": {
        "type1_name": {
            "base_type": string,
            ...
        },
        "type2_name": {
            "base_type": string,
            ...
        },
        ...
    }
    "messages": [
    {
        "name": string,
        "topic": string,
        "priority": integer,
        "sending": [ string ],
        "receiving": [ string ],
        "contents": {
            "field_name": "type",
            "field2_name": "type",
            ...
        }
    }
    , ... ]
}
```

5 Additional Materials Developed

During my work time in the E-Agle Team, I faced a great number of issues with the CAN bus both at hardware and software level. When debugging any given problem, my colleagues often seemed to have already encountered similar issues but not remember the solution found at the time, or even recall to have seen related documentation but have no clue on its location. For this reason, at one point I decided to dedicate part of my time and efforts to create a thorough configuration and troubleshooting guide in order for new members to have quick and easy access to all needed information, from the initial setup not to miss any crucial parameter, to fine-grained interpretation of runtime errors. In the two next sections, this wiki is reported exactly as-is, therefore often using second-person sentences and a slightly less formal register.

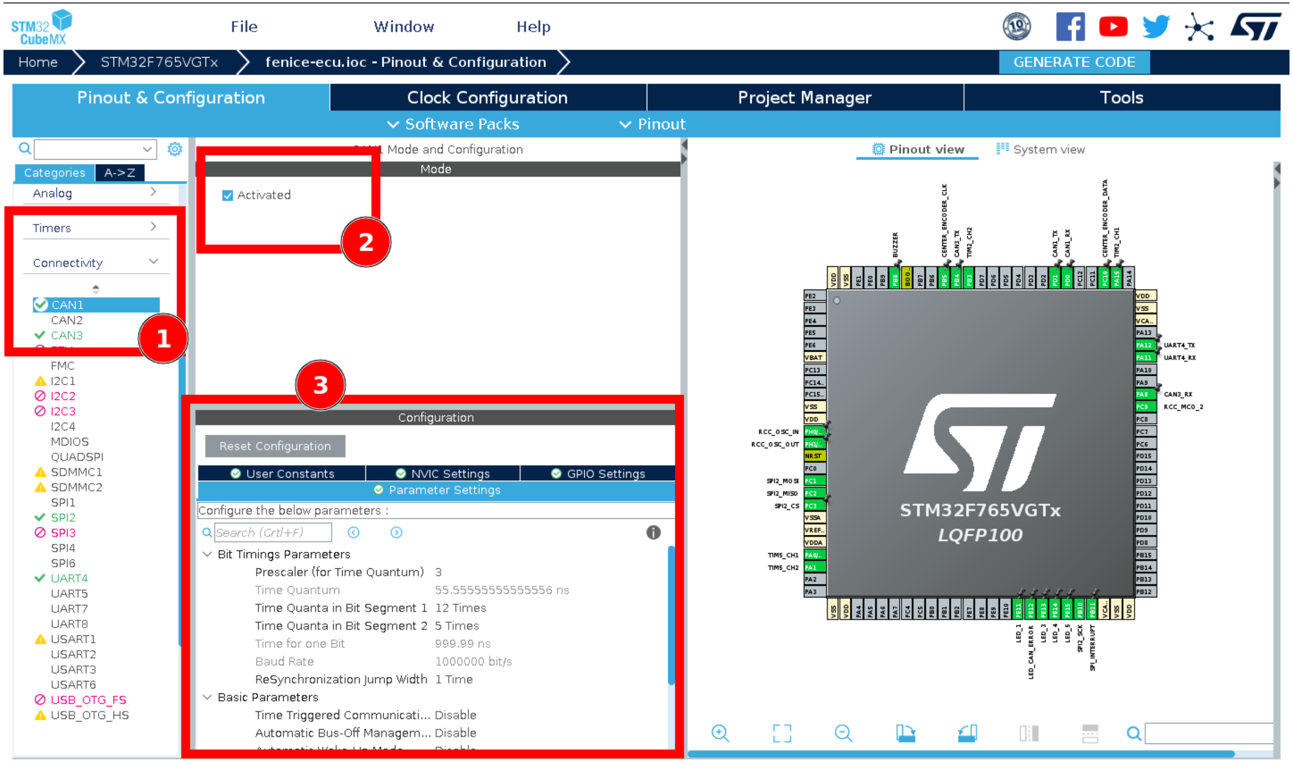
5.1 CAN-Bus Configuration on ST Microcontrollers

To configure the CAN controller on an STM MCU, several steps need to be taken: one first part in CubeMX to generate the low-level stubs that prepare the hardware, and a second part in the code to handle and define the different behaviors.

5.1.1 In CubeMX

Activate the Peripheral

In CubeMX, expand the *Connectivity* tab on the left pane and click on the CAN interface you wish to activate. A configuration pane will appear on the right; tick the *Activated* checkbox.



✓ Activated

Configuration

Reset Configuration

User Constants | NVIC Settings | GPIO Settings

Parameter Settings

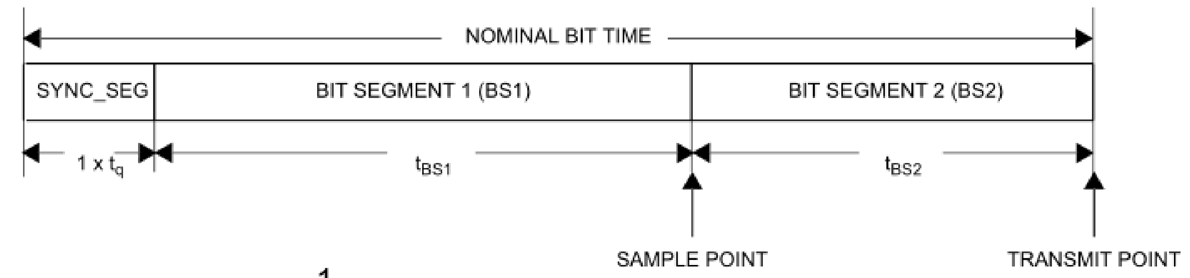
Configure the below parameters :

```
Bit Timings Parameters
    Prescaler (for Time Quantum)      3
    Time Quantum                      55.5555555555556 ns
    Time Quanta in Bit Segment 1      12 Times
    Time Quanta in Bit Segment 2      5 Times
    Time for one Bit                  999.99 ns
    Baud Rate                         1000000 bit/s
    ReSynchronization Jump Width      1 Time
Basic Parameters
    Time Triggered Communicati...     Disable
    Automatic Bus-Off Managem...      Disable
```

Figure 5.1: CubeMx Window

Set the Bit Timing

The sampling of the signal is dictated by a number of parameters that set time constraints for the reading and transmitting instants. The duration of a bit (nominal bit time) is split into four segments: a synchronization segment (`SYNC_SEG`) to sync all nodes on the network, the first Bit Segment (`BS1`) to define the location of the sampling point, the second Bit Segment to define the transmit point (`BS2`), and a resynchronization Jump Width (`SJW`) that indicates by how much segments can be lengthened or shortened to compensate for timing errors or delays. The length of these segments is defined in terms of number of Time Quanta (t_q). For a visual representation of the subdivision and the mapping of the values to the registers refer to Figure 5.2.



$$\text{Baud Rate} = \frac{1}{\text{NominalBitTime}}$$

$$\text{NominalBitTime} = 1 \times t_q + t_{BS1} + t_{BS2}$$

with:

$t_{BS1} = t_q \times (\text{TS1}[3:0] + 1)$,

$t_{BS2} = t_q \times (\text{TS2}[2:0] + 1)$,

$t_q = (\text{BRP}[9:0] + 1) \times t_{PCLK}$

where t_q refers to the Time quantum

t_{PCLK} = time period of the APB clock,

BRP[9:0], TS1[3:0] and TS2[2:0] are defined in the CAN_BTR Register.

MS31001V2

Figure 5.2: Subdivision of the nominal bit time into segments

To define the duration of a time quanta and the length of the various segments, you need to input your parameters into the "Configuration" panel that appeared in CubeMx from the previous step (below the "Activate" checkbox). If your parameters are still to be defined, use an online calculator such as **bittiming.can-wiki.info**[1] and fill the form; note that selecting the platform is only needed if you're interested in the register mapping, but in any case this website provides a choice for *ST Microelectronics bxCAN* which is our case. Input the clock frequency after checking your Clock Tree in CubeMX for how the relevant bus is configured (in the example below, the bxCAN peripheral is connected to the **APB1** bus which is at 54 MHz) and of course that the frequency is high enough to operate at the target link speed.

Next, input the desired sample point (the default 87.5% is fine), the SJW duration (the default 1 is fine as well), and the desired bitrate **in kbit/s**. The calculator will generate a table with valid combinations: copy over the values into CubeMX for the Prescaler, BS1, and BS2 (SYNC_SEG is always of length 1 t_q) and check that the resulting baud rate is correct.

The last operation left to perform in CubeMX is to activate all CAN interrupts in the NVIC array: in the same panel as above, switch to the *NVIC* tab and tick all checkboxes.

[1]http://www.bittiming.can-wiki.info/

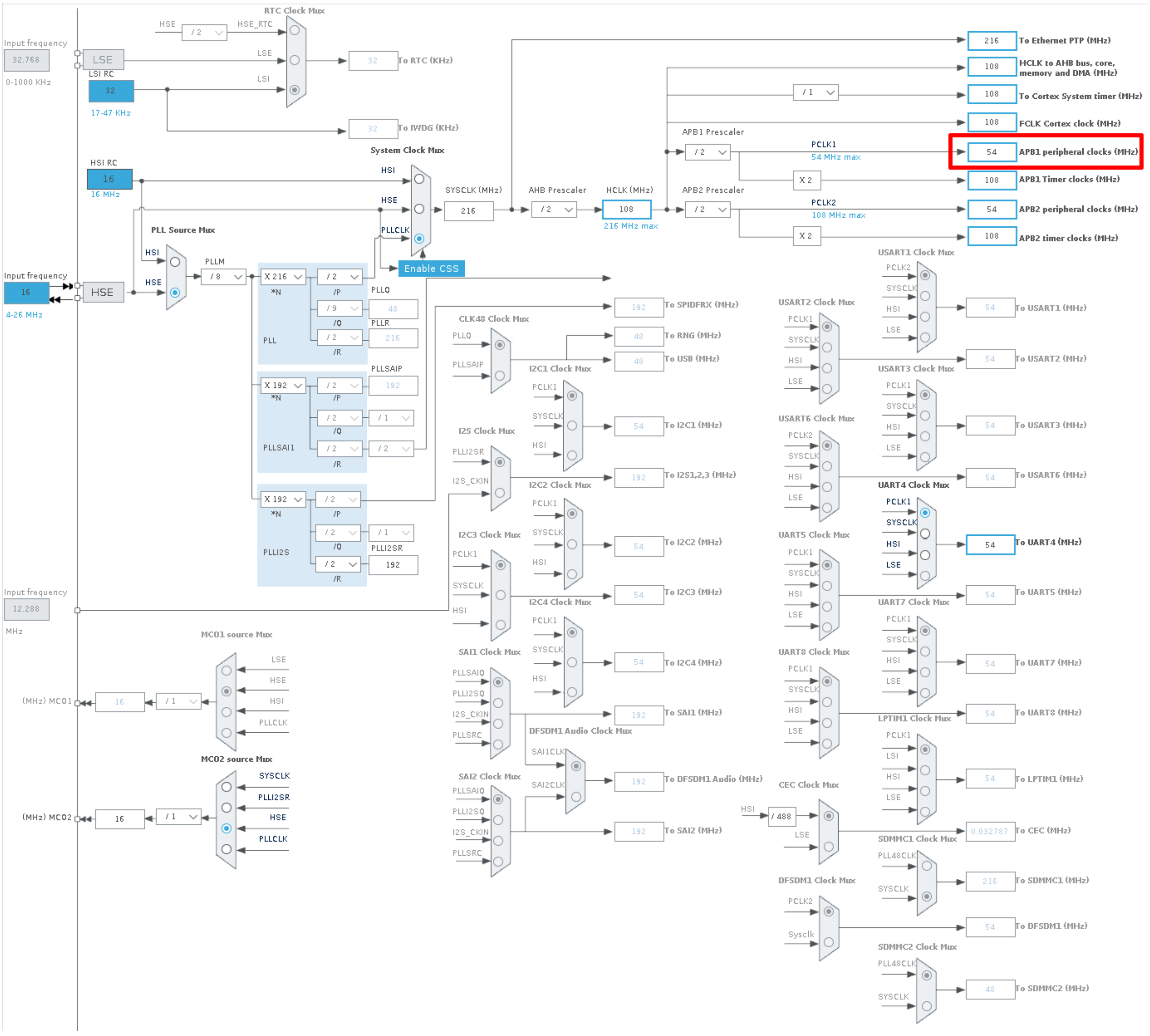


Figure 5.3: Clock Tree of a CubeMX project

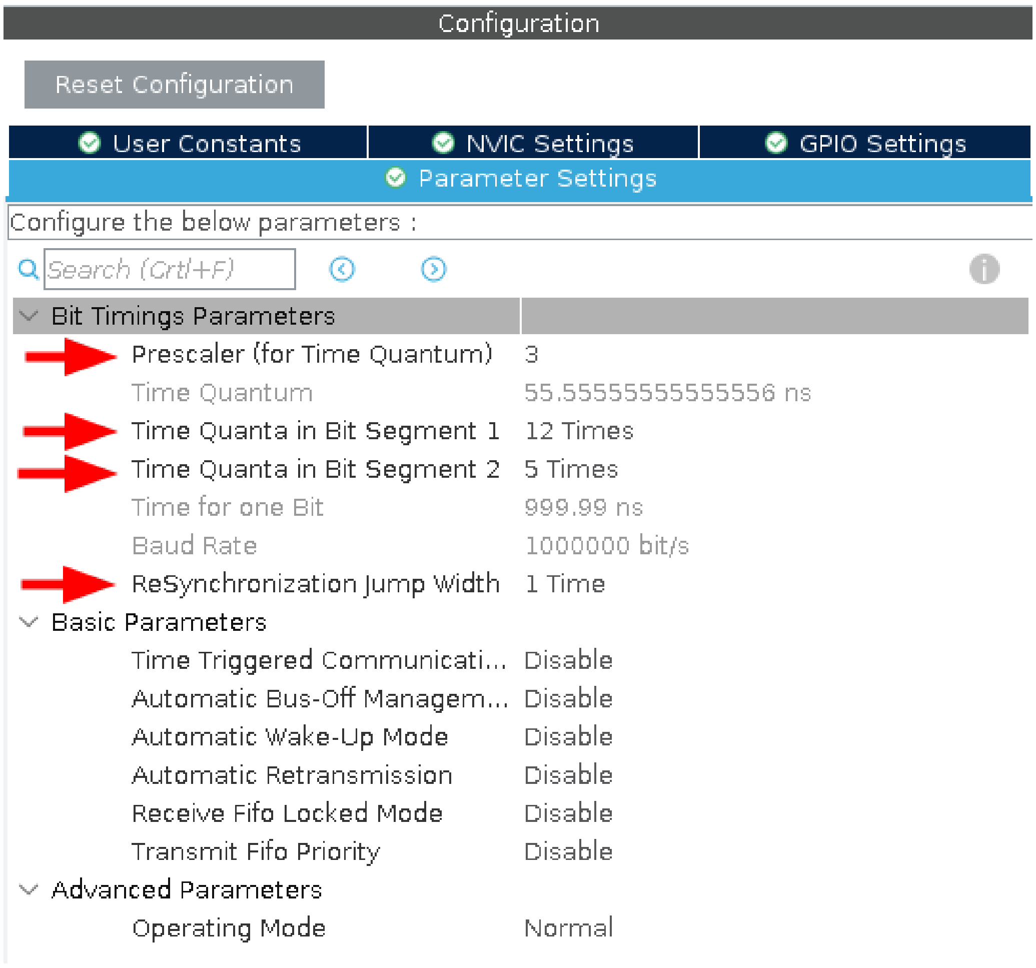
5.1.2 In Your Code

Most of the information that follows is available in the source documentation of the `stm32fXxx_hal_can.c` driver file, so check there should you need any additional detail. However, much of the process outlined there is handled automatically by CubeMX, so this document should simplify your setup by only describing what has effectively to be done, while also using a less schematic and cryptic approach.

Create the Callback Functions

To register callbacks you need to create the relative functions with the proper names (in the form of `HAL_CAN_xxxCallback(...)`, detailed in Table 5.1).

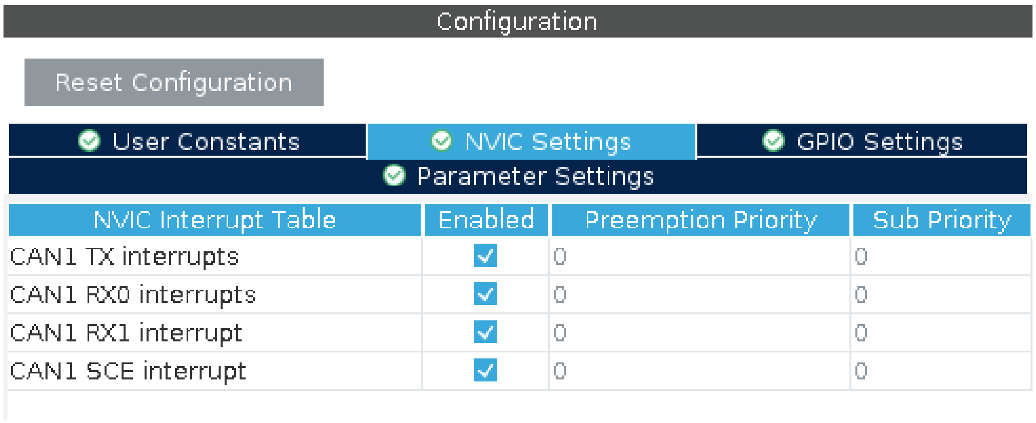
Example For your convenience, here is a prepared Error Callback that verbosely logs any error bit that is high in the error code. `CAN_error_handler(...)` is a custom sample function that prints the message over UART and lights up a red LED.



Configure the below parameters :

Bit Timings Parameters	
Prescaler (for Time Quantum)	3
Time Quantum	55.5555555555556 ns
Time Quanta in Bit Segment 1	12 Times
Time Quanta in Bit Segment 2	5 Times
Time for one Bit	999.99 ns
Baud Rate	1000000 bit/s
ReSynchronization Jump Width	1 Time
Basic Parameters	
Time Triggered Communicati...	Disable
Automatic Bus-Off Managem...	Disable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Disable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Disable
Advanced Parameters	
Operating Mode	Normal

Figure 5.4: Configuration parameters in CubeMX



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
CAN1 TX interrupts	✓	0	0
CAN1 RX0 interrupts	✓	0	0
CAN1 RX1 interrupt	✓	0	0
CAN1 SCE interrupt	✓	0	0

Figure 5.5: NVIC activation in CubeMX

Callback signature	**Action upon which it is called**
void HAL_CAN_RxFifo0MsgPendingCallback (CAN_HandleTypeDef *)	A message has been received into FIFO 0 and is ready to be read.
void HAL_CAN_RxFifo1MsgPendingCallback (CAN_HandleTypeDef *)	A message has been received into FIFO 1 and is ready to be read.
void HAL_CAN_RxFifo0FullCallback (CAN_HandleTypeDef *)	FIFO 0 has reached its full capacity. The next RX will cause an overrun error.
void HAL_CAN_RxFifo1FullCallback (CAN_HandleTypeDef *)	FIFO 1 has reached its full capacity. The next RX will cause an overrun error.
void HAL_CAN_TxMailbox0CompleteCallback (CAN_HandleTypeDef *)	Mailbox 0 is now clear (a pending TX has ended).
void HAL_CAN_TxMailbox1CompleteCallback (CAN_HandleTypeDef *)	Mailbox 1 is now clear (a pending TX has ended).
void HAL_CAN_TxMailbox2CompleteCallback (CAN_HandleTypeDef *)	Mailbox 2 is now clear (a pending TX has ended).
void HAL_CAN_ErrorCallback (CAN_HandleTypeDef *)	An error in the peripheral has occurred. **Make sure to properly log the raised error code.**

Table 5.1: Callbacks invoked by CAN interrupts

```c
void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan) {
    uint32_t e = hcan->ErrorCode;

    if (e & HAL_CAN_ERROR_EWG)
        CAN_error_handler("Protocol Error Warning");
    if (e & HAL_CAN_ERROR_EPV)
        CAN_error_handler("Error Passive");
    if (e & HAL_CAN_ERROR_BOF)
        CAN_error_handler("Bus-off Error");
    if (e & HAL_CAN_ERROR_STF)
        CAN_error_handler("Stuff Error");
    if (e & HAL_CAN_ERROR_FOR)
        CAN_error_handler("Form Error");
    if (e & HAL_CAN_ERROR_ACK)
        CAN_error_handler("ACK Error");
    if (e & HAL_CAN_ERROR_BR)
        CAN_error_handler("Bit Recessive Error");
    if (e & HAL_CAN_ERROR_BD)
        CAN_error_handler("Bit Dominant Error");
    if (e & HAL_CAN_ERROR_CRC)
        CAN_error_handler("CRC Error");
    if (e & HAL_CAN_ERROR_RX_FOV0)
        CAN_error_handler("FIFO0 Overrun");
    if (e & HAL_CAN_ERROR_RX_FOV1)
        CAN_error_handler("FIFO1 Overrun");
    if (e & HAL_CAN_ERROR_TX_ALST0)
        CAN_error_handler("Mailbox 0 TX failure (arbitration lost)");
    if (e & HAL_CAN_ERROR_TX_TERR0)
        CAN_error_handler("Mailbox 0 TX failure (tx error)");
    if (e & HAL_CAN_ERROR_TX_ALST1)
        CAN_error_handler("Mailbox 1 TX failure (arbitration lost)");
    if (e & HAL_CAN_ERROR_TX_TERR1)
        CAN_error_handler("Mailbox 1 TX failure (tx error)");
    if (e & HAL_CAN_ERROR_TX_ALST2)
        CAN_error_handler("Mailbox 2 TX failure (arbitration lost)");
    if (e & HAL_CAN_ERROR_TX_TERR2)
        CAN_error_handler("Mailbox 2 TX failure (tx error)");
    if (e & HAL_CAN_ERROR_TIMEOUT)
        CAN_error_handler("Timeout Error");
    if (e & HAL_CAN_ERROR_NOT_INITIALIZED)
        CAN_error_handler("Peripheral not initialized");
    if (e & HAL_CAN_ERROR_NOT_READY)
        CAN_error_handler("Peripheral not ready");
    if (e & HAL_CAN_ERROR_NOT_STARTED)
        CAN_error_handler("Peripheral not started");
    if (e & HAL_CAN_ERROR_PARAM)
        CAN_error_handler("Parameter Error");
}
```

Listing 5.1: Example error callback that verbosely logs any error

5.1.3 Activate the IRQ Handlers ("Notifications")

At this point, in order for the code to actually invoke your callbacks, you need to activate the relative so-called Notifications with `HAL_CAN_ActivateNotification(CAN_HandleTypeDef *, uint32_t)`. Possible values for the IRQ number are, as defined in the driver file:

```
/* Transmit Interrupt */
#define CAN_IT_TX_MAILBOX_EMPTY     ((uint32_t)CAN_IER_TMEIE)   /*!< Transmit
    mailbox empty interrupt */


/* Receive Interrupts */
#define CAN_IT_RX_FIFO0_MSG_PENDING ((uint32_t)CAN_IER_FMPIE0)  /*!< FIFO 0 message
     pending interrupt */
#define CAN_IT_RX_FIFO0_FULL        ((uint32_t)CAN_IER_FFIE0)   /*!< FIFO 0 full
    interrupt            */
#define CAN_IT_RX_FIFO0_OVERRUN     ((uint32_t)CAN_IER_FOVIE0)  /*!< FIFO 0 overrun
     interrupt         */
#define CAN_IT_RX_FIFO1_MSG_PENDING ((uint32_t)CAN_IER_FMPIE1)  /*!< FIFO 1 message
     pending interrupt */
#define CAN_IT_RX_FIFO1_FULL        ((uint32_t)CAN_IER_FFIE1)   /*!< FIFO 1 full
    interrupt            */
#define CAN_IT_RX_FIFO1_OVERRUN     ((uint32_t)CAN_IER_FOVIE1)  /*!< FIFO 1 overrun
     interrupt         */


/* Operating Mode Interrupts */
#define CAN_IT_WAKEUP               ((uint32_t)CAN_IER_WKUIE)   /*!< Wake-up
    interrupt                  */
#define CAN_IT_SLEEP_ACK            ((uint32_t)CAN_IER_SLKIE)   /*!< Sleep
    acknowledge interrupt       */


/* Error Interrupts */
#define CAN_IT_ERROR_WARNING        ((uint32_t)CAN_IER_EWGIE)   /*!< Error warning
    interrupt         */
#define CAN_IT_ERROR_PASSIVE        ((uint32_t)CAN_IER_EPVIE)   /*!< Error passive
    interrupt         */
#define CAN_IT_BUSOFF               ((uint32_t)CAN_IER_BOFIE)   /*!< Bus-off
    interrupt                  */
#define CAN_IT_LAST_ERROR_CODE      ((uint32_t)CAN_IER_LECIE)   /*!< Last error
    code interrupt           */
#define CAN_IT_ERROR                ((uint32_t)CAN_IER_ERRIE)   /*!< Error
    Interrupt                     */
```

Listing 5.2: CAN IRQ numbers

Note that IRQ numbers can be OR-ed together and be activated many at once. For instance:

```
HAL_CAN_ActivateNotification(hcan,
    CAN_IT_ERROR_WARNING |
    CAN_IT_ERROR_PASSIVE |
    CAN_IT_BUSOFF |
    CAN_IT_LAST_ERROR_CODE |
    CAN_IT_ERROR
);
```

Listing 5.3: Activating multiple notifications at once

5.1.4 Activate and Configure the Hardware Filters

When a new message is received, CAN nodes can use hardware filters to decide if the message must be copied into SRAM and trigger an IRQ, or if it has to be discarded. The advantage of hardware filters is that unwanted messages don't waste any CPU cycles thus saving potentially a lot of resources, especially on heavy traffic networks with many nodes.

STMs are equipped with a number of filter banks, each of which is composed by two 32-bit registers. Each bank can operate in **ID Mask Mode** or in **ID List Mode**: in the former, one register acts as the mask and the second as the ID, while in the latter mode, both registers represent IDs of the list. Filter banks can also be configured to operate in 16-bit mode, therefore doubling their capacity (four 16-bit registers instead of two 32-bit registers, which are split in two). In ID Mask mode this yields two pairs of Mask+ID, while in ID List mode, of course, it yields 4 ID registers.

For a better understanding of the layout of a filter bank in relation to hardware registers, refer to Figure 5.6.

ID Mask Mode

In this mode the filtering is based on a mask register which tells which bits of the ID register need to be compared with the incoming message ID. For instance, with a mask of `11110000000` and an ID of **01110000000**, only messages whose ID begins with `0111` (`0x3`) will be accepted. Some more examples follow:

- **Mask**: `0b11111111111` (`0x03FF`)
  ID: `0b11111111111` (`0x03FF`)
  Accepted: only ID == `0x03FF`
- **Mask**: `0b11111111110` (`0x03FE`)
  ID: `0b11111111110` (`0x03FE`)
  Accepted: IDs `0x03FE`, `0x03FF` (as bit 0 of the filter is disabled)
- **Mask**: `0b11111111000` (`0x03F8`)
  ID: `0b00000000000` (`0x0000`)
  Accepted: IDs from `0x000` to `0x007` (as bits 0, 1, 2 of the filter are disabled)

ID List Mode

In ID List Mode, incoming messages' IDs are simply compared for equality with every element of the list.

Code

To configure a filter, you need to allocate and populate a `CAN_FilterTypeDef` struct and pass it by reference to `HAL_CAN_ConfigFilter(...)`. Its fields are the following:

- `FilterMode`: either `CAN_FILTERMODE_IDMASK` or `CAN_FILTERMODE_IDLIST`, for the respective modes

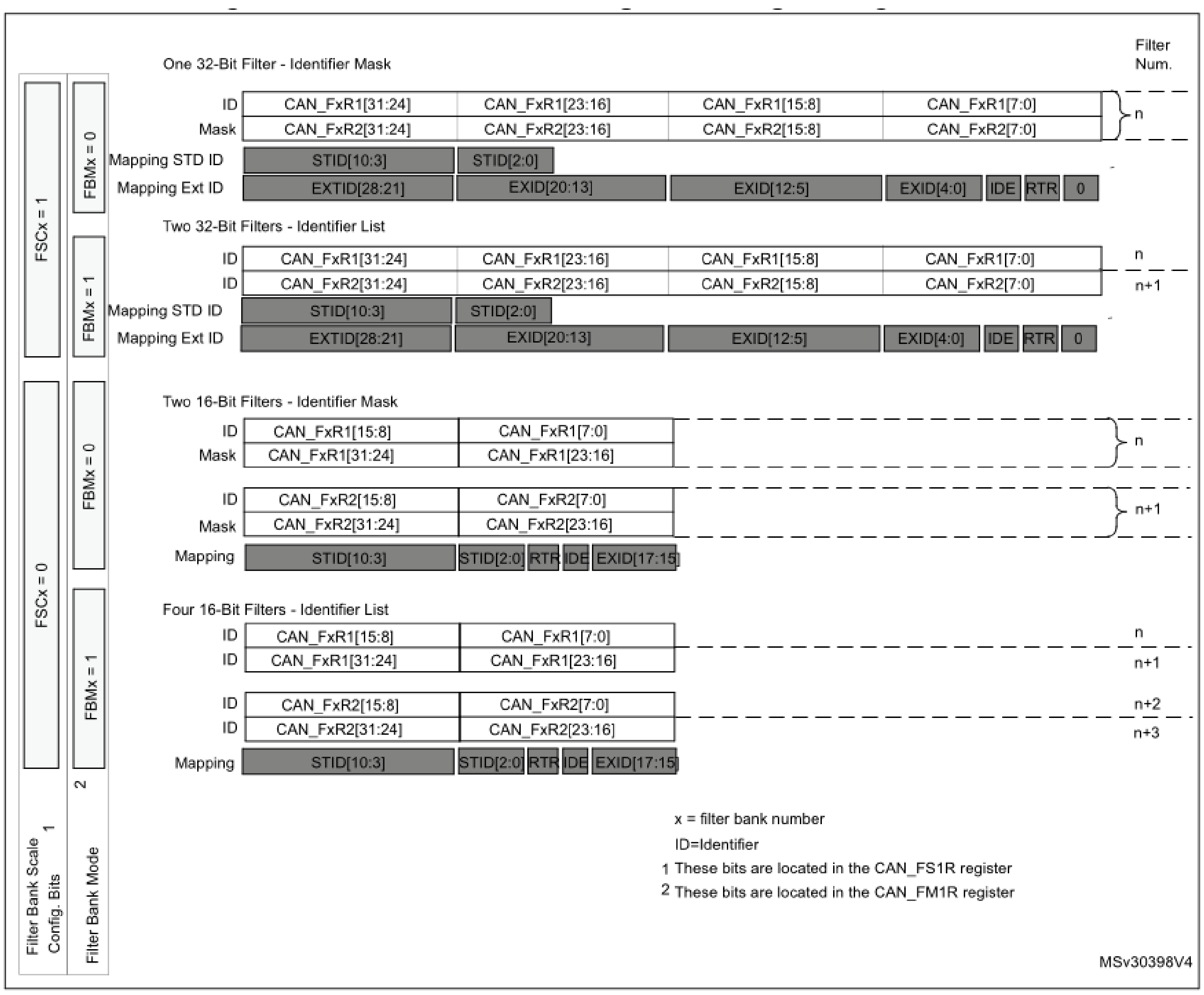


Figure 5.6: CAN filter registers

- `FilterIdHigh`: MSBs of the ID register (or first one of the two 16-bit registers)
- `FilterIdLow`: LSBs of the ID register (or second one of the two 16-bit registers)
- `FilterMaskIdHigh`: MSBs of the Mask register (or first one of the two 16-bit registers)
- `FilterMaskIdLow`: LSBs of the Mask register (or second one of the two 16-bit registers)
- `FilterFIFOAssignment`: the queue in which to store messages (a value of `CAN_filter_FIFO`, e.g. `CAN_FILTER_FIFO0`)
- `FilterBank`: index of the filter bank to be initialized
- `FilterScale`: `CAN_FILTERSCALE_16BIT` or `CAN_FILTERSCALE_32BIT`, for the respective register scale
- `FilterActivation`: `ENABLE` or `DISABLE`

Example A filter that accepts everything would look like this:

```
CAN_FilterTypeDef f;
f.FilterMode = CAN_FILTERMODE_IDMASK;
f.FilterIdLow = 0x0;  // Meaningless, mask is all 0s
f.FilterIdHigh = 0x0; // Meaningless, mask is all 0s
```

```
f.FilterMaskIdHigh = 0x0;
f.FilterMaskIdLow = 0x0;
f.FilterFIFOAssignment = CAN_FILTER_FIFO0;
f.FilterBank = 0;
f.FilterScale = CAN_FILTERSCALE_32BIT;
f.FilterActivation = ENABLE;

HAL_CAN_ConfigFilter(hcan, &f);
```

Listing 5.4: Example of a CAN filter that accepts all IDs

5.1.5 Final Operations

The last step is to actually start the peripheral by calling `HAL_CAN_Start(...)`. To use the bus:

- Send messages with `HAL_CAN_AddTxMessage(...)`
- Check for free mailboxes with `HAL_CAN_GetTxMailboxesFreeLevel(...)`
- Read received messages with `HAL_CAN_GetRxMessage(...)`

Should anything not work properly, refer to the next document, **CAN-bus Troubleshooting**.

5.2 CAN-Bus Troubleshooting on ST Microcontrollers

5.2.1 Step 1: Check Your Configuration

Check that your CubeMX configuration and your code precisely match what is detailed in the **CAN-Bus Configuration** document.

Warning

Be sure to activate all error interrupts and log them properly, otherwise you'll have no way of knowing what is failing and how to fix it.

5.2.2 Step 2: Follow This Flowchart

Find your problem in the flowchart of Figure 5.7 and more details in the next section.

5.2.3 Step 3: Find More Details From the Flowchart

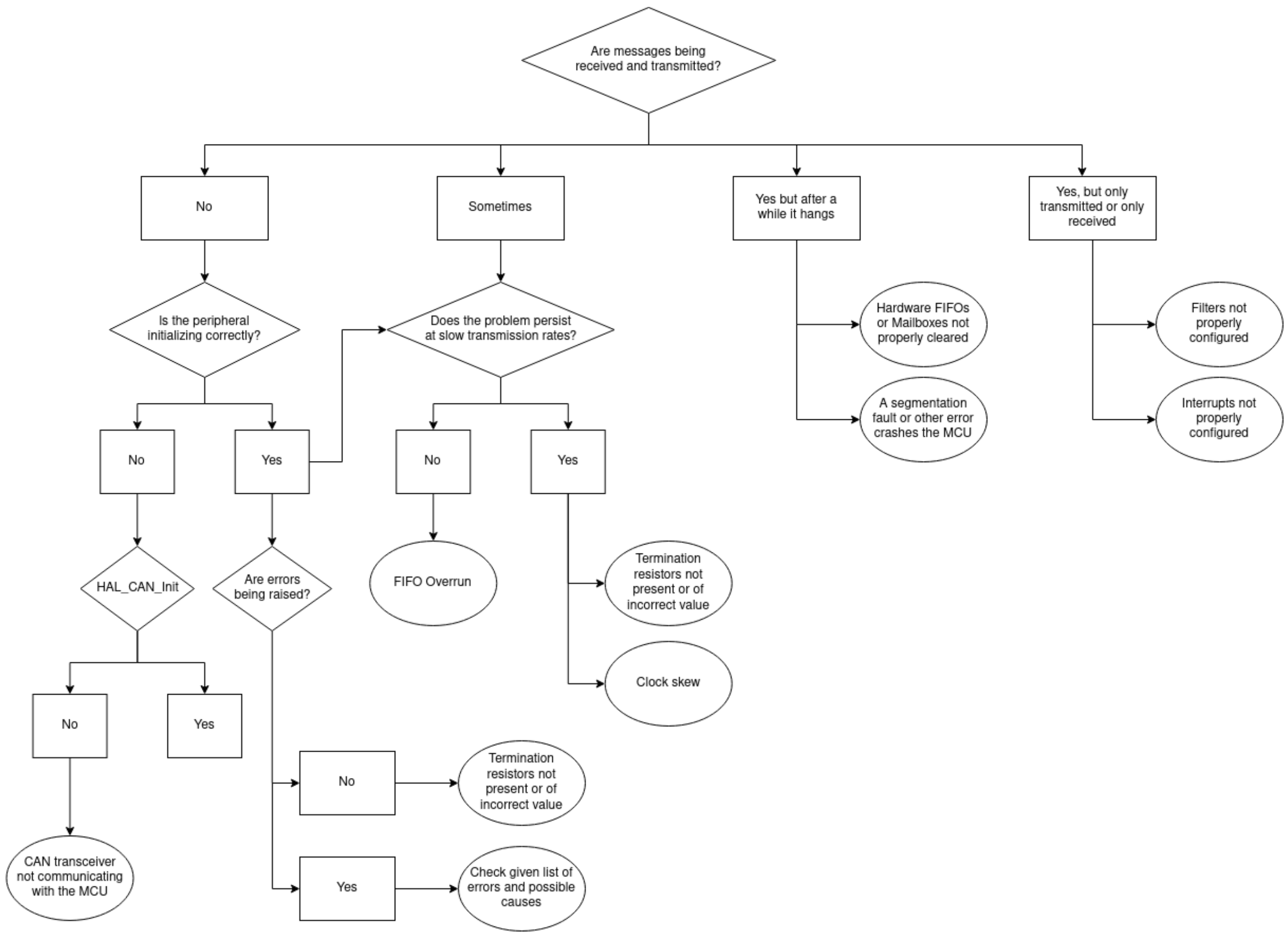
Look for the paragraph titled as the flowchart block that interests you and read through. If the problem is not solved by following the instructions in the green boxes, use the specific components or registers mentioned there to formulate a web search that better describes your condition. Remember to also refer to the official documents such as the STM Reference Manual and the HAL Documentation.

CAN Transceiver not Communicating With the MCU

While the CAN bus operates on two differential lines, the MCU uses a serial protocol, for which a transceiver is needed (for instance, an MCP2562) that translates the CAN_H and CAN_L signals into CAN_TX and CAN_RX. Absence of communication with this device causes the `HAL_CAN_Start` call to fail with a timeout error.

Check

Check continuity between all relevant pins of the transceiver, the CAN socket, and the MCU pins. Try swapping the transceiver with a new one.



Are messages being received and transmitted?
No
Sometimes
Yes but after a while it hangs
Yes, but only transmitted or only received
Is the peripheral initializing correctly?
Does the problem persist at slow transmission rates?
Hardware FIFOs or Mailboxes not properly cleared
A segmentation fault or other error crashes the MCU
Filters not properly configured
Interrupts not properly configured
No
Yes
No
Yes
HAL_CAN_Init
Are errors being raised?
FIFO Overrun
Termination resistors not present or of incorrect value
Clock skew
No
Yes
No
Termination resistors not present or of incorrect value
Yes
Check given list of errors and possible causes
CAN transceiver not communicating with the MCU

Figure 5.7: CAN Troubleshooting Flowchart

Termination Resistors not Present or of Incorrect Value

On a CAN network it is very important that the bus is terminated on both sides by a 120 Ω resistor (interesting article on why[2]). Also note that some devices might already incorporate a termination resistor on their end, so pay attention to how many you put.

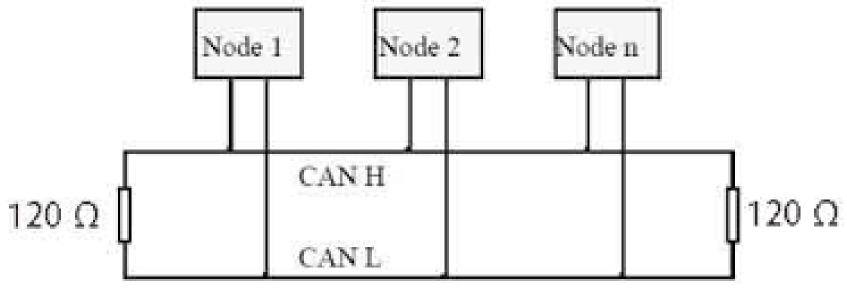


Figure 5.8: Can bus layout

Check

Verify that resistors are properly connected to the bus and are of the correct value. Check schematics and datasheets for already-present termination resistors on the connected nodes. Use an oscilloscope to ensure that the quality of the signal is good.

[2]https://e2e.ti.com/blogs_/b/industrial_strength/posts/the-importance-of-termination-networks-in-can-transceivers

List of Errors Raised by the HAL Library

CAN errors are defined in the driver file `stm32f7xx_hal_can.h` as follows:

```
#define HAL_CAN_ERROR_NONE            (0x00000000U)  /*!< No error */
#define HAL_CAN_ERROR_EWG             (0x00000001U)  /*!< Protocol Error Warning */
#define HAL_CAN_ERROR_EPV             (0x00000002U)  /*!< Error Passive */
#define HAL_CAN_ERROR_BOF             (0x00000004U)  /*!< Bus-off error */
#define HAL_CAN_ERROR_STF             (0x00000008U)  /*!< Stuff error */
#define HAL_CAN_ERROR_FOR             (0x00000010U)  /*!< Form error */
#define HAL_CAN_ERROR_ACK             (0x00000020U)  /*!< Acknowledgment error */
#define HAL_CAN_ERROR_BR              (0x00000040U)  /*!< Bit recessive error */
#define HAL_CAN_ERROR_BD              (0x00000080U)  /*!< Bit dominant error */
#define HAL_CAN_ERROR_CRC             (0x00000100U)  /*!< CRC error */
#define HAL_CAN_ERROR_RX_FOV0         (0x00000200U)  /*!< Rx FIFO0 overrun error */
#define HAL_CAN_ERROR_RX_FOV1         (0x00000400U)  /*!< Rx FIFO1 overrun error */
#define HAL_CAN_ERROR_TX_ALST0        (0x00000800U)  /*!< TxMailbox 0 transmit
    failure due to arbitration lost */
#define HAL_CAN_ERROR_TX_TERR0        (0x00001000U)  /*!< TxMailbox 0 transmit
    failure due to transmit error */
#define HAL_CAN_ERROR_TX_ALST1        (0x00002000U)  /*!< TxMailbox 1 transmit
    failure due to arbitration lost */
#define HAL_CAN_ERROR_TX_TERR1        (0x00004000U)  /*!< TxMailbox 1 transmit
    failure due to transmit error */
#define HAL_CAN_ERROR_TX_ALST2        (0x00008000U)  /*!< TxMailbox 2 transmit
    failure due to arbitration lost */
#define HAL_CAN_ERROR_TX_TERR2        (0x00010000U)  /*!< TxMailbox 2 transmit
    failure due to transmit error */
#define HAL_CAN_ERROR_TIMEOUT         (0x00020000U)  /*!< Timeout error */
#define HAL_CAN_ERROR_NOT_INITIALIZED (0x00040000U)  /*!< Peripheral not
    initialized */
#define HAL_CAN_ERROR_NOT_READY       (0x00080000U)  /*!< Peripheral not ready */
#define HAL_CAN_ERROR_NOT_STARTED     (0x00100000U)  /*!< Peripheral not started */
#define HAL_CAN_ERROR_PARAM           (0x00200000U)  /*!< Parameter error */
```

Listing 5.5: CAN errors defined by the HAL

In general, bus-related and protocol-related errors (protocol error warning, error passive, bus-off, stuff error, form error, ACK error, bit dominant, bit recessive, CRC error) indicate either an electric problem with the wiring, or an underlying issue with the clock precision or frequency.

Check

Double check with a multimeter that the bus is wired properly and that any jumpers make good contact. Use an oscilloscope to ensure that the signal is clear from interference and edges are sharp. Connect a logic analyzer to better understand the communication issue at binary level. See the **Clock Skew** section and double-check that the configured frequencies match the oscillators.

Firmware-related errors such as overruns or transmit failures may instead indicate an overly congested network or a mistake by the developer when invoking low-level API calls.

Check

Test your device with less traffic to ensure that the performance of the MCU is suitable for the application. If you are using only two devices, try connecting a third one to better understand where and how the errors generate. Carefully compare your code with the HAL and ST application notes.

Clock Skew

ST's MCUs incorporate an internal low-cost 16MHz crystal (called HSI in the clock tree - *High Speed Internal*) which is factory calibrated with a precision up to 1%. However, due to its low accuracy especially with temperature variations of the chip, peripherals can often encounter timing problems, in particular if running at higher frequencies generated by the PLL.

Check

Set-up an external crystal (HSE) and configure the MCU for the correct frequency. Check at runtime that the HSI is not being used as a fallback after a configuration error with a condition like `if (__HAL_RCC_GET_SYSCLK_SOURCE() == RCC_SYSCLKSOURCE_HSE)` (or equivalent if running through the PLL).

Hardware FIFOs or Mailboxes not Properly Cleared

For MCUs with transmissions managed via mailboxes, the developer enqueues a new message to send by calling `HAL_CAN_AddTxMessage(...)` and providing as a last argument the address of a `uint32_t` variable in which the function will store the ID of the first free mailbox found. Errors may arise in two occasions: either the function is being called so frequently that the bus has no time to send and empty a slot, or no one is listening on the bus and the "Automatic Retransmission" option is set, making the MCU hang in a loop where it unsuccessfully tries to send messages over and over again.

Conversely, reception queues might as well fill up and cause overruns if they are not read often enough or the network is so congested that the MCU cannot keep up.

Warning

When a CAN message arrives and its ISR is triggered, remember to call `HAL_CAN_GetRxMessage(...)` to clear the reception slot, otherwise an overrun will occur after few messages.

Check

If errors happen when transmitting, make sure you are not sending too frequently and either have free mailboxes available (check with `HAL_CAN_GetTxMailboxesFreeLevel(...)`) or a software queue to temporarily store your pending packets.
If instead they happen upon reception, test your code with as little incoming traffic as possible and try to understand if your MCU and your software architecture match the intended network performance. Always remember to also properly configure your hardware filters in order to only process what's necessary and not waste resources and CPU cycles.

A Segmentation Fault or Other Error Crashes the MCU

The ST drivers and HAL library are generally very solid, so I advise to thoroughly check your code for memory leaks (if the MCU runs out of memory it simply stops) or possible sources of segmentation faults such as handling of pointers or calls to low-level functions like `memcpy` and such. Remember that if the firmware crashes you won't see any error message printed over serial, the processor will just freeze and stop.

Check

Print debug statements over UART or blink on-board LEDs to make sure whether your code keeps executing or if, in fact, stops after a while. Use your IDE built-in debugging tools to interface with OpenOCD and find the exact instruction on which the crash happens.

Filters not Configured Properly

If a message is discarded by a hardware filter you will not receive interrupts of any kind.

Warning

Disabling filters will cause every message to be discarded, contrary to what you might expect.

Check

Carefully read the filter configuration section in the **CAN-Bus Configuration** document and ensure your filters are behaving as you intend.

Interrupts not Configured Properly

For interrupts to be raised by the MCU, they need to be correctly configured both from CubeMX and in your code.

Check

Verify that you followed the correct procedure detailed in the Interrupt section in the **CAN-Bus Configuration** document.

6 Future Upgrades and Expansions

Developing a complex prototype such as Fenice requires many iterations and redesigns for almost every component. Consequently, not all features are immediately implemented, but only those that allow a fast evaluation of the correctness of the current setup. Once this is phase is terminated, either the boards and software are redesigned, or the rest of features is implemented. Either way, new issues are likely to come up and call for yet another iteration. For this reason, the current state of the DAS is only close to be complete, and minor functionalities are still missing.

6.1 Implement And Test Remaining Features

As anticipated, the majority of the effort has been put into implementing the most critical features as to assess that basic functionality can happen. During the next months, my job inside the team will be to terminate the DAS code base by integrating those minor features that are still missing, such as PWM signals to the brake lights, reading the pedals' potentiometers from the ADCs instead of the CAN bus, and working out a complete and solid communication protocol with the ECU.

6.2 Extend Testing Coverage

One other aspect that would greatly benefit from expansion is the code testing suite. As of now, only few procedures and libraries are under a Continuous Integration pipeline. Preferably, some time should be dedicated to decoupling more efficiently firmware and hardware so that functions could be tested without developing or integrating complex abstraction frameworks.

6.3 Take Full Advantage of The Hardware

Despite already having a more than acceptable performance, the architecture of the DAS can be further improved by taking advantage of some hardware features that, by slightly increasing the complexity of the code, allow to lay out a more efficient software architecture that relies less on interrupts and is therefore more solid.

Most notably, STMs provide two DMA controllers (one for memory access and one for peripheral access) with 8 streams each. Each stream, in turn, has 32 four-word deep FIFOs and 8 channels (scheme in Figure 6.1.

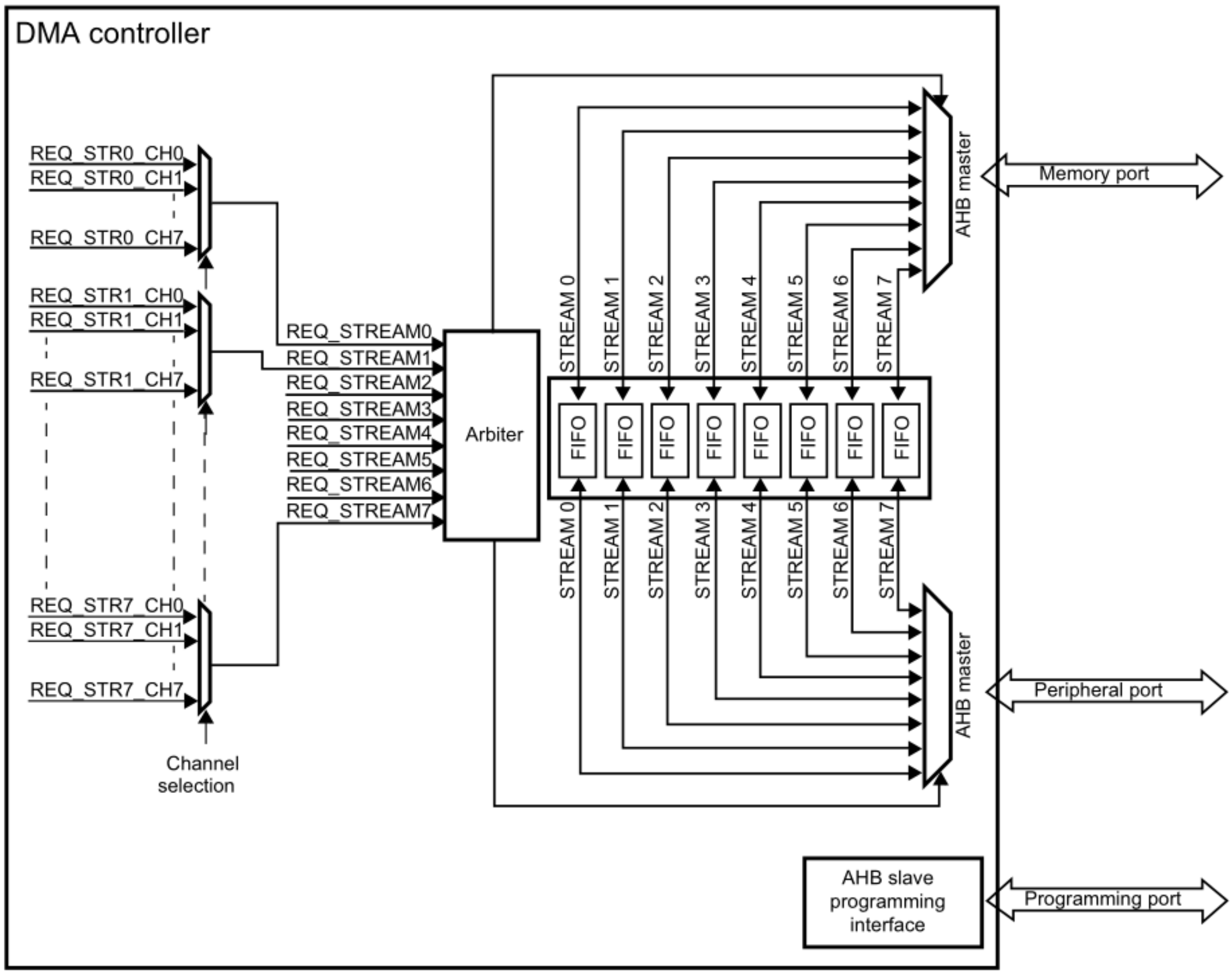


Figure 6.1: Block diagram of a DMA controller

This allows for great flexibility and considerable performance gains by delegating communications to and from peripherals to the DMA controller, which will map their data streams to memory areas with direct access. In this way, the CPU will only need to transparently elaborate the required data that will be promptly prepared in memory without caring about bus transfers that would otherwise occupy processor time and resources.

Bibliography

[1] Formula student rules 2022. https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf. Last accessed 10/02/22.

[2] Michael Barr. *Embedded C Coding Standards*. Barr Group, Germantown, MD 20874, 2018.

[3] Matteo Bonora. *Battery Management System Development*. University of Trento, 2021.

[4] UniTek Industrie Elektronik. *CAN - BUS for Servo Amplifiers DS 2xx / DS 4xx / DPCxx / BAMOCAR / BAMOBIL / BAMOBIL Dxx*. UniTek Industrie Elektronik, 2017.

[5] UniTek Industrie Elektronik. *Digital Battery Servo Amplifier - BAMOCAR PG-D3-700-100/160 - Datasheet*. UniTek Industrie Elektronik, 2021.

[6] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[7] ST Microelectronics. *STM32F446xx Reference Manual*. ST, 2021.

[8] Izze Racing. *Infrared Temperature Sensor - Datasheet*. Izze Racing, 2018.

[9] Izze Racing. *CAN 6-DOF IMU - Datasheet*. Izze Racing, 2019.

[10] RLS. *LM13 Incremental Magnetic Encoder System - Datasheet*. Renishaw, May 2018.

[11] RLS. *RM44 Rotary Magnetic Encoders - Datasheet*. Renishaw, September 2020.