

POLITECNICO DI TORINO

Master's degree course in Electronics Engineering



Master's degree Thesis

Development of a Retrofit Kit for the Transformation of an Electric Vehicle

Supervisor:
Prof. Stefano Carrabelli

Candidate:
Ahmed Bersy

Company Advisor:
Eng. Giovanni Guida

Company Co – Advisor:
Eng. Saverio Milo

October 2021

Summary

I would like to start by asking Why electric cars in the first place? The first and probably most important motive would be the environment. Studies proved that electric vehicles are better for the environment. They produce fewer greenhouse gases and pollutants than gasoline or diesel vehicles. This includes their production as well as the electricity generated to run them[1].

The main advantage of electric vehicles is the contribution they make to improving air quality in cities. Pure electric vehicles emit no carbon dioxide emissions while driving because they do not have a tailpipe. This significantly reduces air pollution. Simply put, electric vehicles provide cleaner streets, making our cities a better place. Over the course of a year, just one electric vehicle on the road can save an average of 1.5 million grams of CO₂. [2]

Another crucial reason why electric cars is that electric cars actually cost less than gas vehicles in the sense that while electric cars may have a higher sticker price than gas-burning ones at first glance, they will save drivers money in the long run because they require less maintenance, have fewer moving parts that can break or need to be changed, no oil to change and no gas to buy. Each year, electric cars are less expensive than conventional gas vehicles[3]. As the cost of electric cars falls to the same level or lower than that of existing vehicles, users will have no choice but to "go electric." A gasoline-powered car is significantly more expensive to operate and maintain than an electric vehicle. The vehicle does not require gas or oil changes, smog tests, or oil changes, and there are fewer moving parts that can break or wear out. In fact, many electric car owners go for years without having to pay for any repairs or service.

Another reason why electric cars is safety, Electric vehicles must pass same safety tests as gasoline-powered vehicles, ensuring that they are completely safe to drive [4]. In fact, many Electric vehicles must pass same safety tests as gasoline-powered vehicles, ensuring that they are completely safe to drive. outperform conventional vehicles in crash tests; the Tesla Model X, for example, received a perfect score. A common concern about the safety of electric vehicles is the risk of fire, but actually electric cars are actually much less likely to catch fire than gasoline burning cars. Gas cars catch fire at a rate of about one fire every 20 million miles driven on average. The rate of fire for electric vehicles is one per 120 million miles driven.

What about the electric cars market in Europe and the rest of the world? The answer to this question would be that the market for electric vehicles in Europe and the rest of the world is actually booming. 2019 saw a record-breaking number of electric car sales, topping 2018's record-breaking figure. After several years of over 40 percent annual electric car sales growth, sales in 2019 increased by 6 percent. Despite the fact that the global car market is contracting for the third consecutive year, the global electric car market share has reached a new high of 2.6%, up from 2.4 percent in 2018 and 1% in 2017. So, in 2019, the number of electric cars on the road grew by 40 percent compared to the previous year, indicating that the electric vehicle sector has been growing steadily since 2016, and that the 36 percent average annual stock growth needed to reach the SDS target by 1.06 million electric cars were sold in 2019 in China, followed by 560 000 in Europe and

326 000 in the United States; these three regions accounted for over 90% of all sales in the year. With a market share of 56 percent in 2019, Norway continues to dominate the sales market, closely followed by Iceland (23 percent) and the (15 percent) [5]. Nearly three-quarters of all electric car sales in 2019 were battery electric cars. As the power sector continues to decarbonize, the CO2 emissions reduction benefits of EVs will accelerate.

To conclude the electric vehicles market discussion, I must dedicate few words to China as it is without doubt the world leader country in electric cars industry, China is expected to produce 13 million battery electric vehicles (BEVs) and plug-in hybrid electric vehicles (PHEVs) by 2023. Other large markets, such as the United States, Germany, and Japan, are expected to fall short of China's estimated production level.

EVERGRIN project of Brain Technologies and Polytechnic University of Turin's main goal is vehicle electrification, so converting a diesel-powered car into a fully electric one, in our case the car is a FIAT PANDA. So put in short simple words the goal is "Transformation(electrification) of a diesel-powered fiat panda vehicle", so transforming it into a fully electric vehicle.

Here are the macro activities and where this thesis work is placed in it [6]:

Adaptation of the Saver KIT on Panda 2nd Series

- Development of new components (e.g., ABS)
- VMU development (This thesis work lies here)
- Power Box design
- HMI System development
- Functional Architecture and Vehicle Network Integration
- Development of test benches
- Development and Adaptation of Mechanical Parts

In this thesis work the development of the low-level firmware and Base line software of a VMU for the purpose of transforming a diesel fiat panda car into an electric car is presented. One of the main activities performed by brain technologies right before thinking about how we want to develop our VMU was actually choosing the VMU to be developed and installed into the car, the board chosen is the 32S. The main activity was studying the architecture of the tasks in order to perform the developing. The developing was performed in C programming language.

Here are the tasks developed in this thesis work:

1. Pedals management
2. Power socket verification.
3. Buzzer management.
4. Acceleration pedal verification.

Acknowledgments

The aim of the thesis is to contribute to the EVERGRIN project adopted by the Polytechnic University Of Turin and Brain Technologies.

Firstly, I would like to thank my supervisor professor Stefano Carabelli for giving me the opportunity to join the EVERGRIN project as it was definitely the perfect step for my career at that point. It was not only a useful experience on the academic and career level, but it was also a fun and exciting experience.

Secondly, I would like to thank my company advisor engineer Giovanni Guida for such an opportunity and for opening for me the 1st gate for my career as an embedded developer engineer. Working with him and learning from him was a fun and great experience.

Thirdly, I would like to thank engineer Saverio Milo for leading me along the way regarding the development and programming. His knowledge and experience as a software developer was surprising me on a daily basis.

Lastly, I would like to thank my family for supporting me in every way possible along my career. I would like to thank my friends who were there for me every single time and every single moment I felt like I need support, they are the ones who made my life abroad and away from home feel like home.

Contents

1 Introduction	1
1.1 The Idea and Motivation Behind Electric Vehicles and Vehicle Electrification	1
1.2 The idea and motivation behind EVERGRIN	2
1.3 EVERGRIN main goal	3
2 EVERGRIN PROJECT and SOFTWARE ARCHITECTURE	5
2.1 Electric Vehicle (EV) Conversion Process	5
2.1.1 Adaptation of the Saver Kit on Panda 2nd Series	5
2.2 SOFTWARE ARCHITECTURE	9
2.2.1 EVERGRIN SOFTWARE ARCHITECTURE	9
2.3 RTOS	25
2.3.1 RTOS features and key reasons why it is used of critical systems	25
2.3.2 Classic OS vs RTOS	25
2.3.4 RTOS architectures	26
2.4 DRIVERS	29
2.4.1 Board Stress Test and Driver Initialization	29
2.4.2 CAN FIFO Buffer	35
2.5 Automotive Standards (AUTOSAR)	37
2.5.1 AUTOSAR Architecture	38
3 The EVERGRIN VMU from TTcontrol	42
3.1 VMU Suppliers (TTcontrol) History	42
3.2 The HY-TTC32S and the HY-TTC30 Family Technical Details and Architecture	43
3.2.1 Overview	43
3.2.2 Deeper into 32S and 30 family Technihcal details	43
3.2.3 HY-TTC 32S Features and specifications	47
3.3 Why the HY_TTC32S for EVERGRIN	57
4 Tasks and Drivers Implementation	59
4.1 Main Function	59
4.1.1 Main Function Code Implementation	59
4.1.2 Functions and Drivers Initialized in the Main Function	72
4.2 Pedals Management Task	73
4.2.1 Task Architecture	73
4.2.2 Pedals Management Code Implementation	75

4.2.4 Development and Implementation.....	79
4.3 power Socket Verification Task	86
4.3.1 Task Architecture	86
4.3.2 Power Socket Verification Code Implementation.....	88
4.3.4 Development and Implementation.....	91
4.4 Buzzer Management Task	96
4.4.1 Task Architecture	96
4.4.2 Buzzer Management Code Implementation	97
4.4.4 Development and Implementation.....	103
4.5 Acceleration Pedal Verification Task.....	107
4.5.1 Task architecture.....	107
4.5.2 Acceleration Pedal Verification Code Implementation	109
4.5.3 Development and implementation.....	117
Bibliography	120

List of Figures:

FIGURE 2. 1: SAVER KIT.....	6
FIGURE 2. 2: THE EVERGRIN ARCHITECTURE	7
FIGURE 2. 3: PIN CONFIGURATION OF THE VMU	10
FIGURE 2. 4: ACTION INDICATORS.....	10
FIGURE 2. 5: SOFTWARE BLOCK DIAGRAM OF THE KEY-STOP POSITION.....	12
FIGURE 2. 6: SOFTWARE BLOCK DIAGRAM OF KEY-ON POSITION	14
FIGURE 2. 7: SOFTWARE BLOCK DIAGRAM OF THE KEY-CRANK MOMENTARY POSITION.....	16
FIGURE 2. 8: SOFTWARE BLOCK DIAGRAM OF CLICK AFTER CRANKING MODE PART1	18
FIGURE 2. 9: SOFTWARE BLOCK DIAGRAM OF CLICK AFTER CRANKING MODE PART2.....	19
FIGURE 2. 10: SOFTWARE BLOCK DIAGRAM OF THE CAR IN MOTION STATE.....	21
FIGURE 2. 11: SOFTWARE BLOCK DIAGRAM OF HV BATTERY MANAGEMENT.....	22
FIGURE 2. 12: SOFTWARE BLOCK DIAGRAM OF DCDC FAULT MANAGEMENT	23
FIGURE 2. 13: SOFTWARE BLOCK DIAGRAM OF PARKING MANAGEMENT	24
FIGURE 2. 14: MONOLITHIC OPERATING SYSTEM ARCHITECTURE DIAGRAM	27
FIGURE 2. 15: MICROKERNEL OPERATING SYSTEM ARCHITECTURE DIAGRAM.....	28
FIGURE 2. 16: TEST 1 IMPLEMENTATION	30
FIGURE 2. 17: TEST 2 IMPLEMENTATION PART1.....	32
FIGURE 2. 18: TEST 2 IMPLEMENTATION PART2.....	33
FIGURE 2. 19: TEST 3 IMPLEMENTATION	34
FIGURE 2. 20: TEST 4 IMPLEMENTATION	34
FIGURE 2. 21: CAN FIFO BUFFER IMPLEMENTATION.....	35
FIGURE 2. 22: BUFFER REPORT.....	36
FIGURE 2. 23: UART PRINTS (1).....	36
FIGURE 2. 24 UART PRINTS (2)	37
FIGURE 2. 25: AUTOSAR LAYER ARCHITECTURE.....	38
FIGURE 3. 1: HY-TTC 30 AND 50 FAMILIES	45
FIGURE 3. 2: HY-TTC 500 FAMILY.....	46
FIGURE 3. 3: 30-H BLOCK DIAGRAM.....	49
FIGURE 3. 4: 30-H MODEL CODE AND DIMENSIONS.....	50
FIGURE 3. 5: 30S-H BLOCK DIAGRAM	51
FIGURE 3. 6: 30S-H MODEL CODE AND DIMENSIONS.....	52
FIGURE 3. 7: 32 BLOCK DIAGRAM	53
FIGURE 3. 8: 32 MODEL CODE AND DIMENSIONS.....	54
FIGURE 3. 9: 32S BLOCK DIAGRAM	55
FIGURE 3. 10: 32S MODEL CODE AND DIMENSIONS.....	56
FIGURE 4. 1: PEDALS MANAGEMENT TASK ARCHITECTURE FLOW CHART	73
FIGURE 4. 2: POWER SOCKET VERIFICATION TASK ARCHITECTURE FLOW CHART	86
FIGURE 4. 3: BUZZER MANAGEMENT TASK ARCHITECTURE FLOW CHART	96

Chapter 1

1 Introduction

1.1 The Idea and Motivation Behind Electric Vehicles and Vehicle Electrification

climate change and air pollution are causing national and regional regulations to be tightened, and that's why most of the 1st world countries are doing everything to get rid of gas burning vehicles as soon as possible. To be able to reach this point they are contributing to the electric vehicles production industry and vehicle electrification which is converting a gas-burning vehicles into a fully electric one [7]. During pollution peaks, the city of Beijing in China, for example, has banned the use of the most polluting cars [8]. By 2025, several European cities will prohibit the circulation of diesel vehicles. Valeo, for example, a global leader in electrification, is already developing and offering cutting-edge technologies that emit less CO₂. Vehicle electrification is expected to increase by over 15 percent over the next decade, from just 2 percent today to 15 percent in the next decade. Increasing consumer demand for greener transportation options and regulations requiring carbon emission reductions and improved fuel efficiency are driving the rapid adoption of vehicle electrification. As a result of these twin pressures, every major automaker has announced plans to release at least one electric vehicle in the next years. [9]

Nowadays thanks to the efforts put in the development of fully electric vehicles and vehicles electrification we can say that we are one step closer towards a world free of gas burning vehicles, a healthier and more eco-friendly transportation. Of course, also thanks to the very advanced stage we are in regarding embedded development, nowadays even gas burning cars are fully electronic apart from the motor. All the safety critical systems like the ABS and the airbag are fully electronic systems which are very reliable because they are real time embedded. Now the purpose of the EVERGRIN project introduced by Brain technologies and Polytechnic University Of Turin is to transform a Fiat panda vehicle from a diesel vehicle to a Fully electric one.

The term "electrified vehicles" encompasses a variety of technologies that rely on electricity to propel a vehicle:

- HEV: Hybrid Electric Vehicles get all of their net propulsion energy from petroleum, but they use an electrical system to save money on gas.
- PHEV: Plug-in Hybrid Electric Vehicles (PHEVs) store energy from the electric grid and can run on both electricity and gasoline. The following are the two main variations:

- When the battery is charged, blended PHEVs use a combination of gasoline and electricity, then switch entirely to gasoline when the battery is depleted. Because the electrical system does not have to meet peak power demands on its own, blended operation has the advantage of being smaller.
- Extended Range Electric Vehicles (EREVs) are plug-in hybrid electric vehicles (PHEVs) that run solely on electricity when the battery is charged and switch to gasoline when the battery is depleted. The vehicle operates as a BEV for trips that are shorter than the battery's range.
- BEV: Battery Electric Vehicles have larger battery packs that can store more energy from the grid for a longer range. They don't have a gasoline backup engine. BEVs are also known as "pure-electric vehicles" or "all-electric vehicles" by some (AEVs).
- FCEV: Fuel Cell Electric Vehicles are hydrogen-powered vehicles that use a fuel cell to generate electricity. Fuel cell vehicles, or FCVs, are another name for FCEVs.
- PEV: Plug-in Electric Vehicle is a term that refers to all vehicles that use the electric power grid to charge (BEVs and PHEVs).
- EV: The term "electric vehicle" is a bit of a misnomer. Some people use the term "electric vehicle" to refer to BEVs only, while others use it to refer to PEVs, PEVs + FCEVs, or any electrified vehicle.

1.2 The idea and motivation behind EVERGRIN

The idea is that instead of only relying on producing electric cars, we can also start to electrify diesel and gas burning vehicles, because if we want to arrive to the point of having no diesel and gas burning vehicles on the streets which is the goal just by producing tons of full electric vehicles, this would mean that we will have to throw away most of the vehicles we have now, since most of the vehicles we have on the streets at this point are diesel and gas burning vehicles. And that would be a lot of waste of course.

Now despite the fact that Italy is a little bit lagging behind the industry leading countries like China and USA, it seems that the shift from combustion to electric vehicles is a must. It also seems that this process needs to be performed as fast as possible, and that's due to latest data which is pretty alarming which found that the nitrogen oxide levels in Lombardy and Emilia Romagna are exceeding the limits. [10]

1.3 EVERGRIN main goal

The main goal is the development of the VMU (vehicle management unit) which would allow the conversion of the vehicle into a FEV (full electric vehicle) providing the following functions:

- Management of the electric propulsion system
- Interface with existing electronic systems
- Additional commands such as gear selector, touch-screen display, other interfaces for smart devices.

The main idea is developing the VMU (and that's covered in this thesis work), and then the physical conversion would carry on by disassembling the components which are no longer necessary like the combustion engine and so on, and installing the parts necessary for the conversion to take place which would be (battery pack, electric motor, VMU with connections).

The introduction of the VMU allows for a significant upgrade in the car by leveraging its established technological base and adding features that are only available in the latest high-end electric vehicles

For the conversion to happen we need to create hardware and software which would allow the conversion into an electric vehicle with respect to the original electronic architecture, so we need to choose the VMU which would be compatible and would allow for a smooth installation, then comes the software part where it is necessary to develop the entire software from low level (drivers and communication) to service for high level software developed in model based design by polytechnic University of Turin.

Chapter 2

2 EVERGRIN PROJECT and SOFTWARE ARCHITECTURE

2.1 Electric Vehicle (EV) Conversion Process

The development of an electric vehicle (EV) conversion process could be performed without spending a lot of money and time even with using the actual components of the design architecture. Using model-based system design, the electric vehicle's propulsion and dynamic load are computed in a systematic way.

There are 2 main inputs for the simulation and the first one is vehicle specification and driving cycles. Consequently, the method could accurately predict Electric Vehicle features and design parameters, such as EV performance, range of driving, torque speed properties, power of the motor, and battery power charge/discharge, which are the required for size of the most important EV components and design.

2.1.1 Adaptation of the Saver Kit on Panda 2nd Series

The current system configuration includes functional ECU hardware, electric car models, and control area network (CAN) connectivity. The Electric Vehicle components and system models can be simulated virtually in real time and this current methodology can be employed as a quick design tool for software development and ECU design and validation.

For the EVERGRIN project the SAVER KIT, illustrated in figure 2.1 is adapted to use on the Panda 2. Series. Mainly function architecture and vehicle network integration are composed by VMU, Battery Pack, Moto-propulsion, 2 Can busses, Pedals & Drive Shift, On-board Charger, 12V Service Battery and Reduction- Differential Gearbox.

SAVER

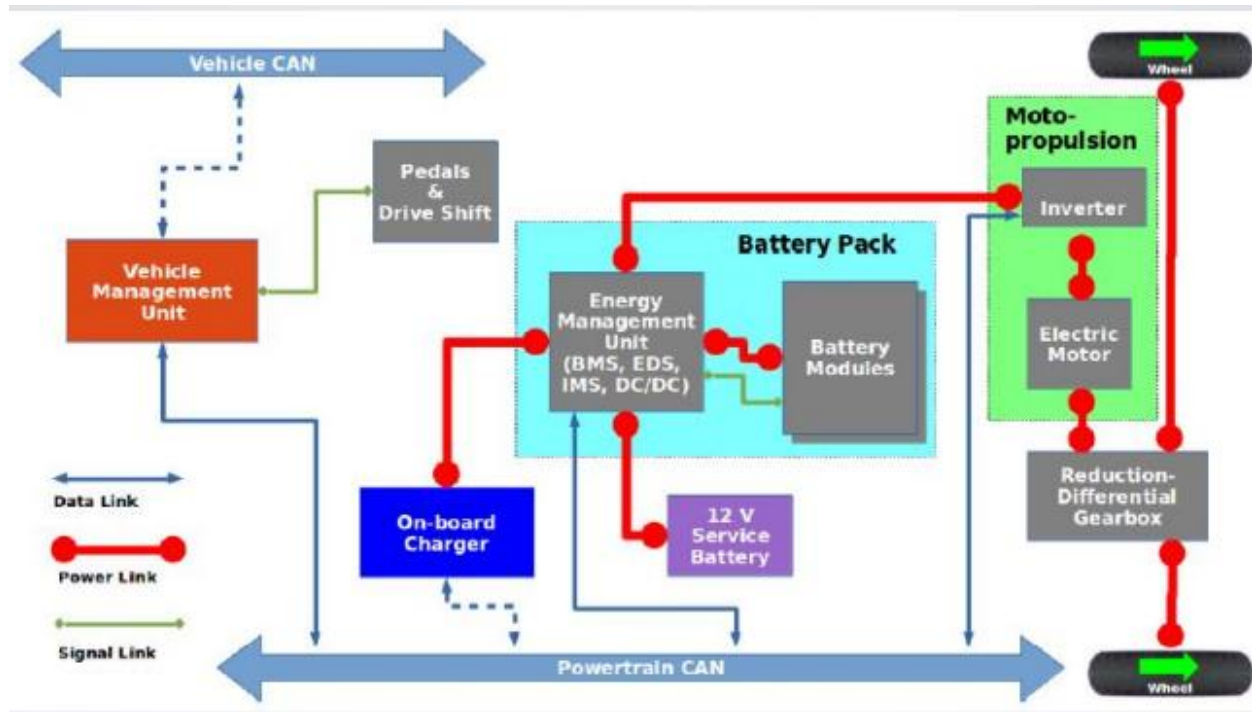


Figure 2. 1: Saver Kit

Vehicle management unit (VMU) has a connection with 2 Can bus which are Vehicle Can is responsible for the communication between VMU and the other Electronics Control Units and Powertrain Can which provides to communication between VMU and the other main power components. VMU has also direct connection with Pedals and Drive Shift components of the car.

Battery Pack is created with Energy Management Unit which could be BMS, EDS, IMS, DC/DC converter and Battery Modules. Energy Management Unit is connected to 12V battery, Battery Modules, On-Board Charger, Inverter with Power link and it has connection with powertrain Can bus.

Moto-propulsion part has 2 main components which are Inverter and Electric Motor. Inverter is connected to EMU with power link and Powertrain can bus with data link. Electric Motor is placed between Inverter and Reduction-Differential Gearbox and lastly 2 wheel is connected to Reduction-Differential Gearbox with power link.

The EVERGRIN architecture is illustrated in figure 2.2 which is developed and configured based on SAVER model and the car specifications.

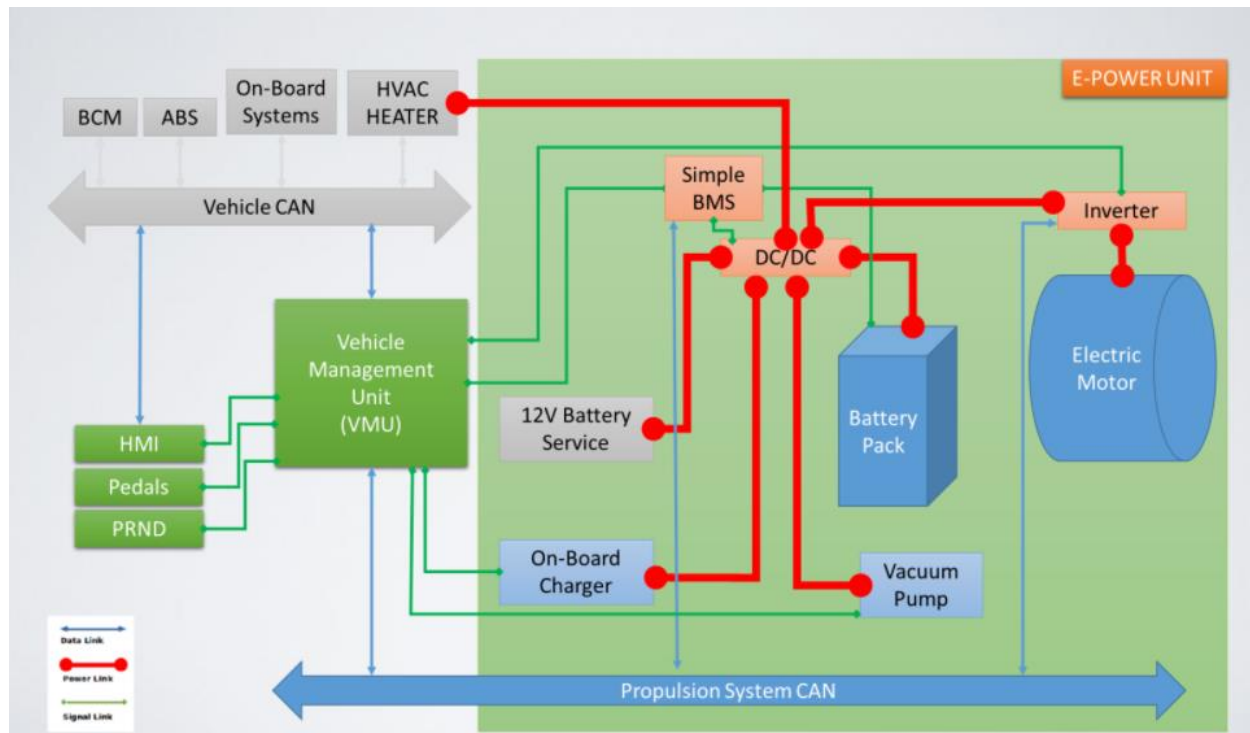


Figure 2. 2: The EVERGRIN architecture

In the EVERGRIN architecture, vehicle can bus is connected between the car electronic control units and VMU. These ECUs are BCM, ABS, On-Board Systems and HVAC Heater.

All these ECUs, which are integrated inside in the car, must also interact with one another.

The Main goal of the Body Control Module (BCM) is to manage and control this communication between the ECUs.

The function the Body Control Modules in automobiles connect with different ECUs in the car via the vehicle's bus system such as CAN, LIN, etc. in our case this communication provided by CAN protocol.

The BCM may be thought of as the brain of the ECUs with controlling different components of the body (different ECUs) by sending and receiving signals via the vehicle bus.

A BCM unit, which is also an ECU, acts as a gateway or hub in order to interact with different ECUs. This mitigates the need for cabled plug-in connection between ECUs within the vehicle. A

BCM unit, which is also an ECU, functions as a gateway or hub to communicate with other ECUs. This eliminates the requirement for a cabled plug-in connection between the vehicle's ECUs.

With the increased use of electronics in cars, the electronic body control module (BCM) plays an essential role in controlling the vehicle's body components. Some of them are, central door locking or key entry with remote way, front and back wash and wiper, lighting system of the car these contains head, hazard, park and break lamps.

Mirror control, horn control or if there is a siren system in the car and HVAC control which is Heating Ventilation and air conditioning control of the car.

BCM also enables warnings for the safety (e.g., hand brake warnings, seat belt display) and diagnostics (e.g., Lamp Mistakes), which increases driver safety and lowers maintenance costs.

The primary most important function of the anti-lock braking system is to prevent the vehicle from sliding uncontrollably by providing better traction when needed. Since the 1970s, ABS has been featured in cars.

First, anti-lock systems, like other modern automotive technologies, were just available in high-end luxury and performance vehicles. ABS has become a regular feature in many cars. The ABS mechanism is thought to be a mix of threshold and cadence braking techniques; however it is more efficient than either of those. Advanced methods, such as electronic stability systems, can be seen to be a development of the ABS.

The ABS system consists mostly of ABS sensors, a control module, and hydraulic valves, that all work together to keep the wheels from locking up. As a result, the technology is known as anti-lock braking.

ABS sensors or wheel speed sensors are placed in all four wheels of modern automobiles. The module constantly analyzes the velocities of all four wheels. If one wheel is moving slower than all the others, the control module activates the brake hydraulic valves to decrease the braking power supplied to that wheel. This causes them to run quicker, putting them back into synchronization with other wheels. On the other side, when the module discovers that one of the wheels is going faster than another three, it provides additional braking force to that wheel, that bringing its speed down to match the speeds of the other wheels. In this approach, The ABS control module, with the aid of sensors and valves, restores the vehicle's traction in any condition.

When the automobile is making a turn, the speeds of the wheels usually vary because the inner wheels spin slower than the outer wheels. The reason for that the outer wheels must go a larger distance than the inner wheels while turning. The ABS control unit is designed to tolerate disparities in wheel velocities up to a specific point, which covers differences in wheel speeds during turning or rounding curves. Usually, when the ABS system is active, the driver often feels a pulsation inside the brake pedal as a result of the valves rapidly opening and closing.

An on-board system constantly informs the driver of the vehicle's current or average fuel usage through specialized instrument cluster displays.

The on-board computer estimates the range based on that data and the amount of gasoline left in the tank.

Furthermore, additional details regarding the average speed or journey duration may be provided. With all of this data, the driver gains critical insight into the most efficient and fuel-efficient driving technique.

Heating, ventilation, and air conditioning (HVAC) technology is used to provide a comfort conditions and vehicle environment. By controlling the level of hot/cold within the cabin, the HVAC system helps in providing a comfortable temperature.

HVAC was initially introduced in cars with in 1960s and is now standard equipment in the number of major vehicles. It is a complicated system with switching devices and knobs inside the frontend.

The system's backend consists of one or even more blower motors, actuators that are providing fresh air circulation, air flow, and temperature control, and a refrigeration module, and several ducts that transport air to the cabin.

Due to the pressure differences, heat transfer occurs from a low-temperature area to a high-temperature area inside within the vehicle. Refrigeration is the term that describes this process of heat transfer.

2.2 SOFTWARE ARCHITECTURE

2.2.1 EVERGRIN SOFTWARE ARCHITECTURE

This part is focus on the software tasks architecture of the Vehicle Management Unit. Main software architecture is mainly divided based on the 2 sections first one is User Action and other one System States. Tasks based on the User actions are the Key positions which are STOP, position, ON position, CRANK MOMENTARY position and CAR in motion state.

The pin configuration of the VMU for the project and the action indicators are illustrated in figure 2.3 and figure 2.4.

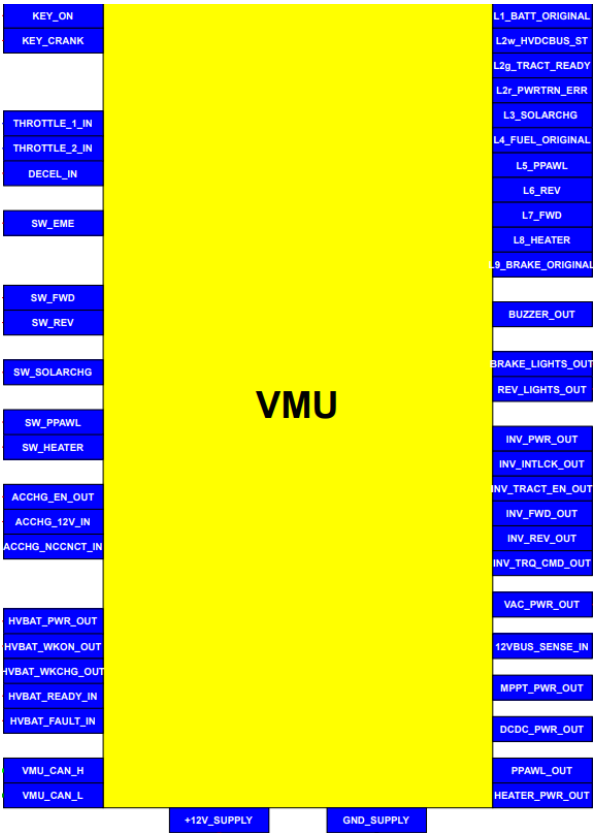


Figure 2. 3: pin configuration of the VMU

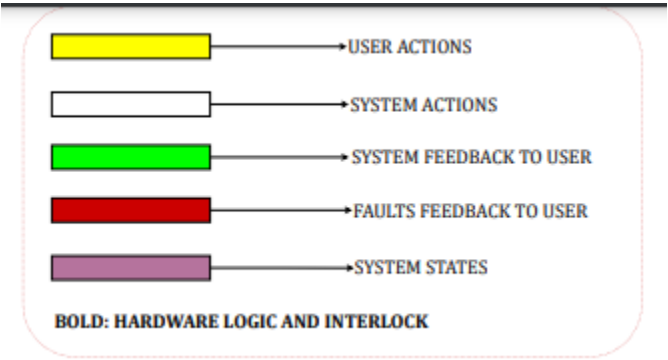


Figure 2. 4: action indicators

2.2.1.1 KEY at STOP POSITION

This task algorithm will be in process due to the user action on the car.

When the key is in the stop position the algorithm will go on to check if there will be any charging process during the key in the stop position. If one of the AC CHARGERS is plugged or SOLAR SWITCH is on the VMU is start the process of the charging the battery.

If the AC Charger is plugged in, VMU receives +12V signal from AC charger which is indicates that AC Charger logic board is on, then VMU sends 3 different signals to go on with the task.

Firstly, +12V signal from 'HVBAT_PWR_OUT' pin of the VMU and this signal closes the switch R3 which causes to activate HVBATT LOGIC BOARD. Second one is +12V signal from the 'HVBAT_WKCHG_OUT' pin to HV BATTERY. Lastly VMU sends +12V signal from the pin name 'ACCHG_EN_OUT' and this signal closes the R6 switch to activate the AC CHARGER, and this closes the switch R4 because of the 12V supply of the AC CHARGER and this de-energize the R5 switch to remove power supply signal of the inverter.

Thereafter VMU sends +12v signal from 'DCDC_PWR_OUT' pin to close R7 switch and activate DC/DC Converter. As a last step of the process and as system feedback, VMU sends 12v signal from 'L4_FUEL_ORIGINAL' pin to flash slowly L4 which is original dashboard fuel light to indicate that process is started. If the key on click engaged status, it will run the task for the starting the AC charging process and send a signal to the buzzer and this activates alarm sound to warn the driver.

If solar switch is on position, +12V signal "SW SOLARCHG" is received by VMU from the solar switch.

For a reaction of this input signal VMU sends 2 different signals.

For starters, VMU sends the +12v signal from the "HVBAT PWR OUT" pin to close R3 switch to turn on HVBATT LOGIC BOARD. The second one is +12V signal which is sent from the "HVBAT WKON OUT" pin to HV BATTERY. After some seconds VMU checks if the HV PRE_CHG issued or not. If it is not, then HVBATTERY sends an error message through can network to the VMU.

If everything is okey then VMU 3 other signals which are activates MPPT charger logic board, DC/DC Converter and as system feedback turns on the solar charging led, and this indicated that the solar charging process is started.

Below is also illustrated in figure 2.5 the block diagram of the how charging process is starts during the Key-STOP position.

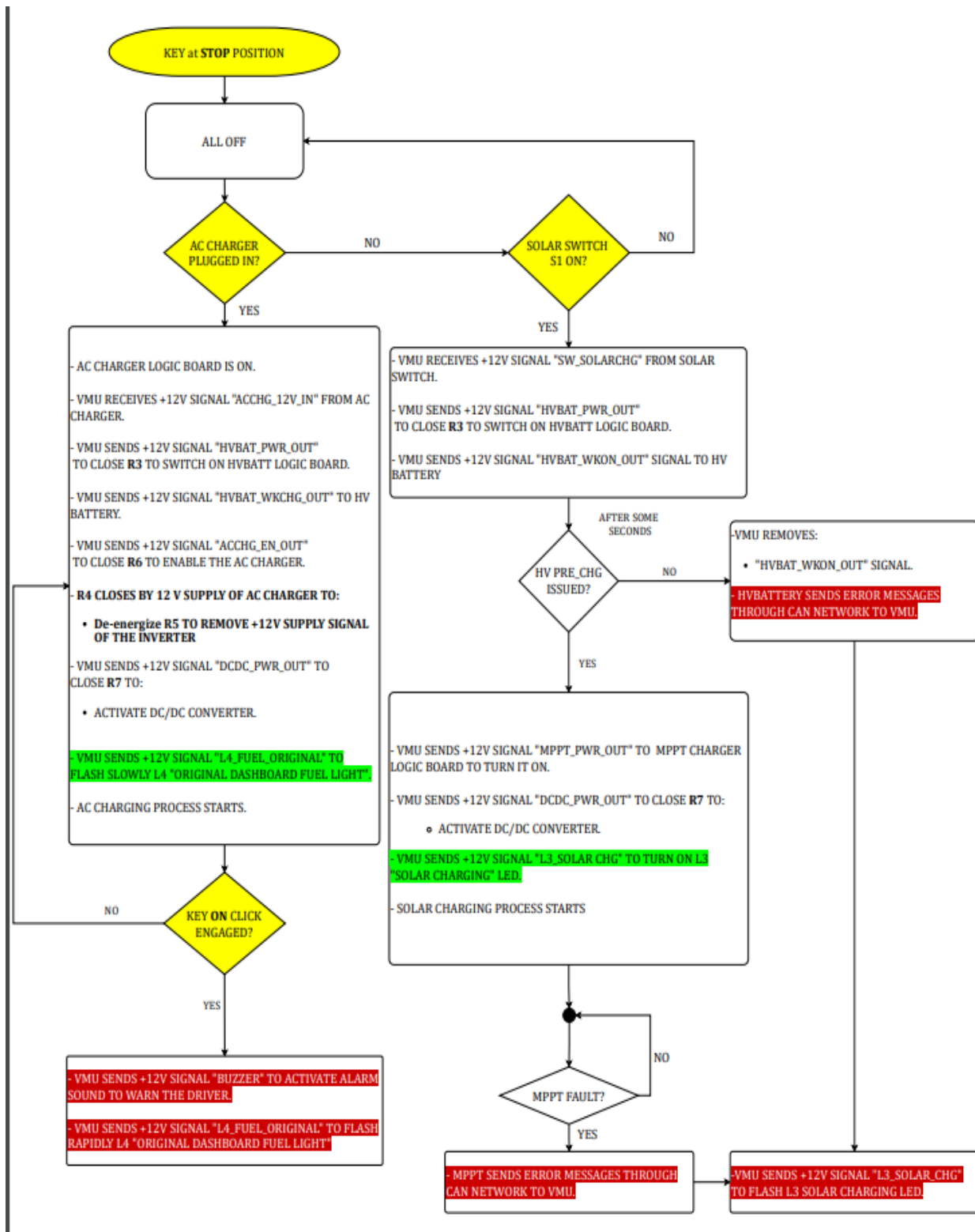


Figure 2. 5: software block diagram of the Key-STOP position

2.2.1.2 KEY at ON POSITION

Main goal of this task is focus on the pre-charge process, battery fault, heater switch status, AC Charger is plugged in or Solar charging status.

In the beginning when the key turns to on position, VMU receives the 'Key On' signal from the key switch. The vehicle's dashboard lights will be flashed when the VMU is activated, and a 12v signal "L1 BATT ORIGINAL" is sent by the VMU in order to turn on L1 "ORIGINAL DASHBOARD BATTERY LIGHT." Indicator.

Later that, R1 switch will be closed by clicking to activate the 12v dc bus for auxiliary power (Radio).

In order to turn on the HVBATT logic board, the VMU sends a +12V signal called "HVBAT PWR OUT" to close R3 switch.

A total of two signals are sent by the VMU: a +12V SIGNAL "HVBAT WKON OUT" signal to the high-voltage battery and a +12V SIGNAL "L2w HVDCBUS ST" signal to the flashing L2 YELLOW LED and signaling the beginning of the DC bus.

The Preparation for the pre-charge process is carried out by the battery's internal logic circuitry.

After the completion of this procedure task checks if HV PRE_CHG is issued or not. If it is not issued yet it also checks Battery fault timeout, if this is true then VMU sends errors to inform the user if this is not a timeout problem then the procedure starts from the beginning part again. If everything is fine and HW PRE_CHG is issued then procedure will continue with checking the heater switch status, Ac charger is plug status and solar switch status.

If Heater switch S2 is on position, then VMU receives a signal from the heater switch and sends a signal to enable heating and sends system feedback to turn on the 'L8_HEATER' on the dashboard.

If AC charger is plugged in, then VMU receives a signal from the AC Charger and sends 2 signals to warn the driver. First one is activation of the alarm sound and other one is flash the fuel light warning on the dashboard.

If Solar switch S1 is on position, , then VMU receives a signal from the Solar switch and sends a signal to charger logic board to turn it on and another signal to flashing the 'Solar Charging' led on the dashboard.

At the same time VMU receives HV battery is ready signal from the HV battery then VMU sends 3 signals to turn on the vacuum pump system, wake up the inverter logic board and activate the DC/DC converter and lastly sends another signal to dashboard to turn on 'HV DC BUS' led and turn off the 'Original Dashboard Battery light'.

Below is illustrated in figure 2.6 the block diagram of how the charging process starts during the Key-ON position.

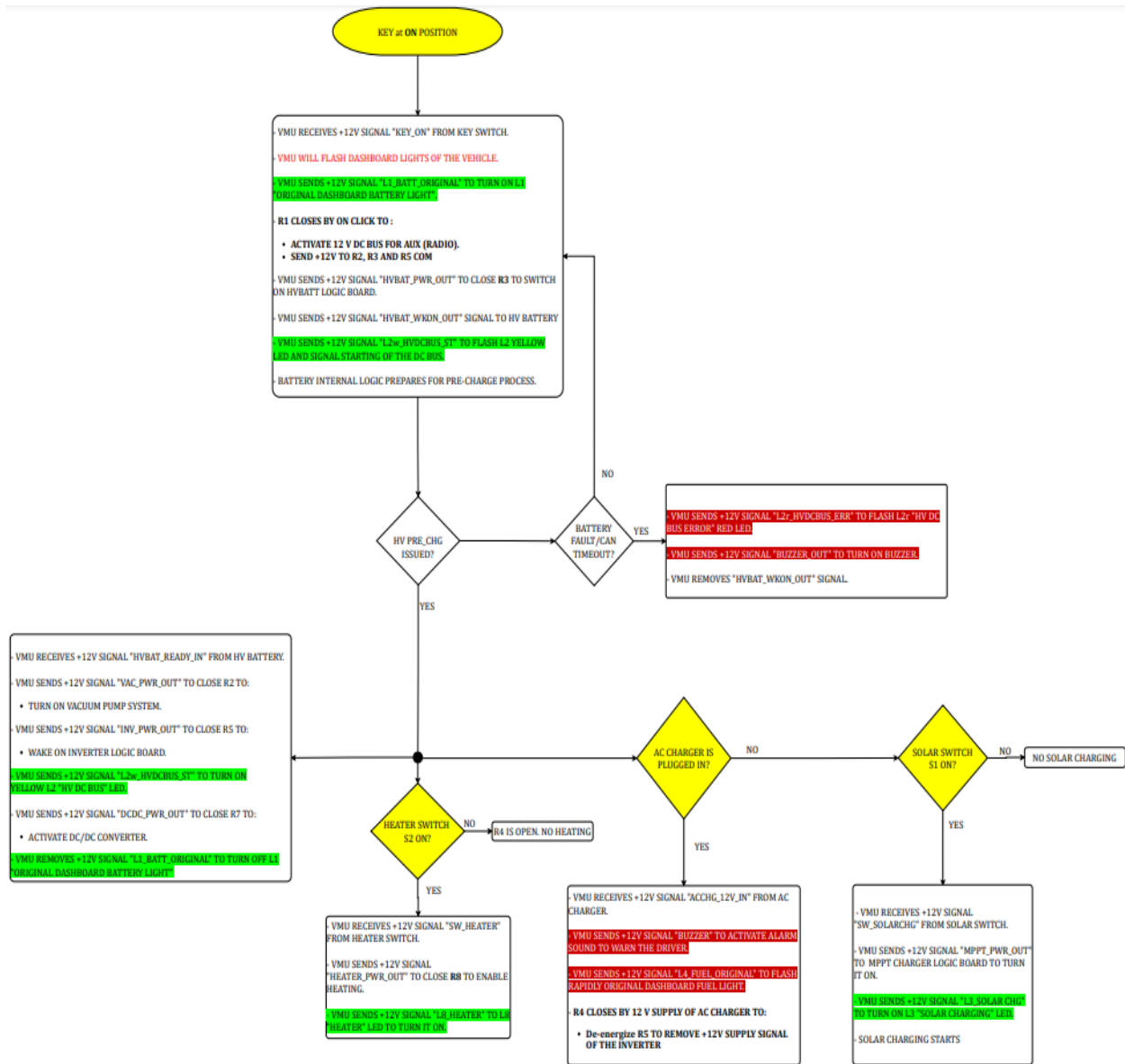


Figure 2. 6: software block diagram of Key-On position

2.2.1.3 KEY at CRANK MOMENTARY POSITION

Main goal of this task is focus on the make the car ready state. When the key at the Crank momentary position then VMU receives a signal from the key switch that indicates the key position.

And then VMU checks if the brake is pressed or not pressed.

If the brake is not pressed, then VMU send a signal to flash the brake light inside of the original dashboard and repeat the process until brake is pressed.

Whenever brake is pressed during the task VMU receives 12v signal from brake pedal and then VMU send signal to close main contactor and send another signal to activates the green traction ready light.

After 1 second, it continues with if process that controls inverter pre-charge process. If this process is not done yet, then inverter sends blocking message through can network to vehicle management unit.

And VMU sends back a signal that flash the L2 red HV DC bus error light and the process comes back to the starting point of when the brake is pressed.

Whenever inverter pre-charge process is finished then inverter sends signal to close the main contactor.

Later the task is checking whether if main contactor is closed or not. If it found out that main conductor is not closed, then inverter sends blocking message through can network to vehicle management unit and sends a signal to flash the L2 Red HV DC bus error light.

Whenever main conductor is closed then HV is connected to the inverter and inverter sends 'Main Conductor Closing' message with using can network and VMU sends +12v signal to turn on L2 green traction ready light.

Then at the end of this procedure it continues with starting the 2nd Click After Cranking State procedure.

Below is illustrated in figure 2.7 the block diagram of the how charging process is starts during the Key-CRANK MOMENTARY position.

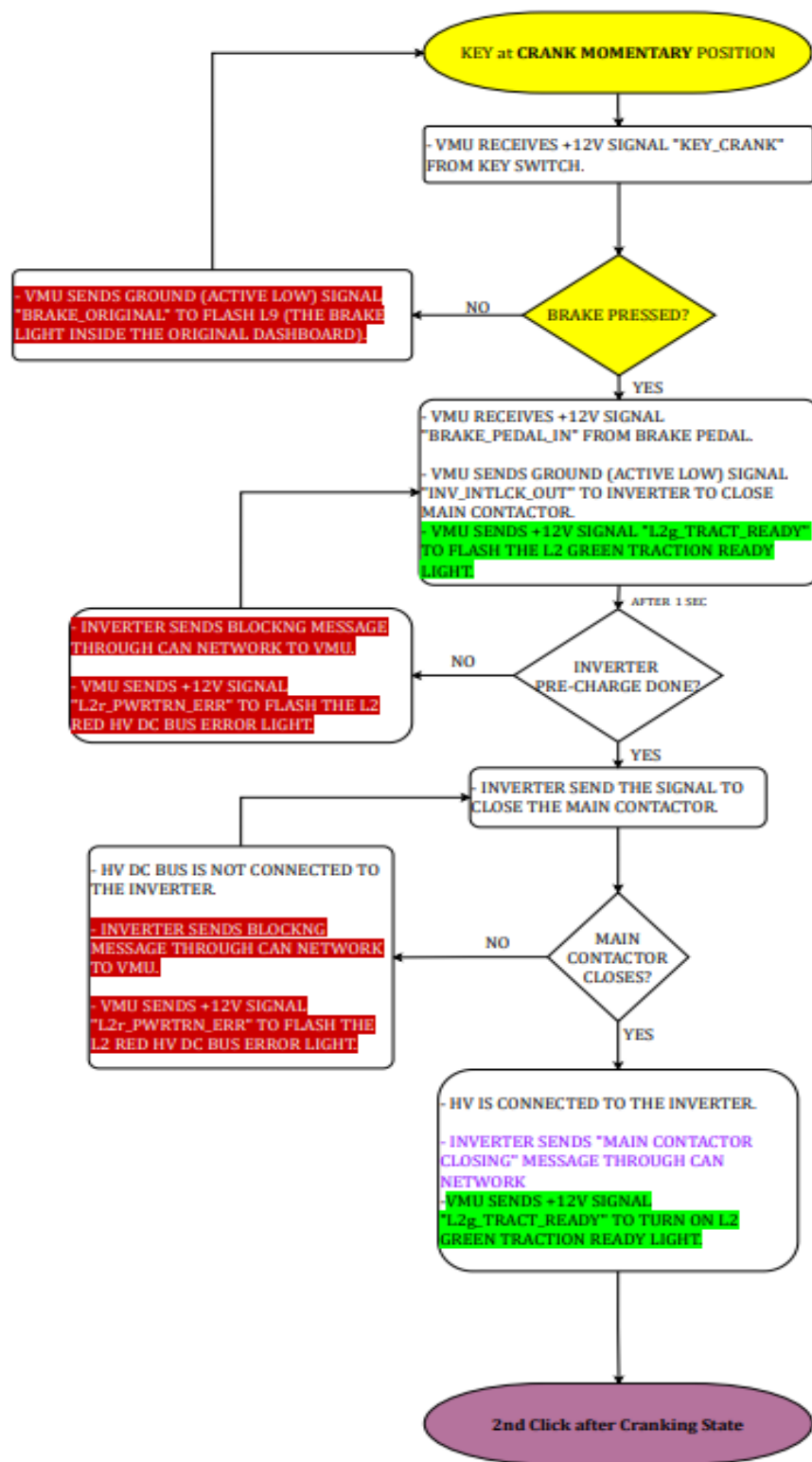


Figure 2. 7: software block diagram of the Key-CRANK MOMENTARY position

2.2.1.4 2nd Click after Cranking State

Main goal of this task is focus on the from the parking state to car in the motion state. When the key at the Crank momentary position and 2nd clicks after cranking state is a system state which is starting after key at the crank momentary procedure.

First, this procedure is starting with checking the status of the Parking Pawl Switch S3 status.

If this checking is on state then VMU receives signal from parking pawl switch and then, sends back 'PPAWL_OUT' to close the R11/11a switch and this will activate bistable brake and lock the vehicle. Also, another signal will be sent from VMU to flash L5 led 'Parking Pawl' and then traction will be disabled. Else, when the parking pawl switch is off state then procedure will continue to be checking the brake status.

If the brake is not pressed, then VMU removes the signal which is 'PPAWL_OUT' signal to disable bistable brake and let the vehicle free. VMU is also removes the 'L5_PPAWL' signal to turn of the L5 'Parking Pawl' light. Now traction is enabled and VMU sends enable traction signal to the inverter.

Now the procedure starts a loop with asking the VMU that brake status. If the brake is still pressed, then loop checks if one of the FWD or REV is selected or not. If they are not selected, then procedure returns to vehicle is neutral position and it comes back to asking if brake is still pressed which is beginning of the loop. If brake is not still pressed and FWD or REV is selected, then VMU sends active low (ground) warning signal to flash L9 which is 'The brake light inside the original dashboard' light and comes back to vehicle is in neutral position state. But if brake is still pressed and REV or FWD is selected at the same time then the task will go on with ne next steps.

When the FWD is selected then VMU receives +12v signal 'SW_FWD' from forward switch and sends back 'L7_FWD' to turn on the led L7. Lastly VMU sends forward signal to the inverter wait in the loop until the ACC pedal is pressed.

When the REV is selected then VMU receives +12v signal 'SW_REV' from forward switch and sends back 'L6_REV' to turn on the led L6. Lastly VMU sends reverse signal to the inverter and wait in the loop until the ACC pedal is pressed.

When ACC pedal is pressed then VMU receives 2 analog signals from the pedal which are 'THROTTLE_1_IN' and 'THROTTLE_2_IN' signals and now it checks if there is an ACC pedal fault or not.

If there is an ACC pedal fault then VMU removes enable traction signal which is sent to the inverter before and sends another signal to flash the L2 red HV DC bus error light.

If there won't be any ACC pedal fault then VMU sends signal to the inverter to set torque request and that is a command for the motion state of the vehicle.

Below is illustrated the software block diagram of Click After Cranking Mode in figure 2.8 and in figure 2.9.

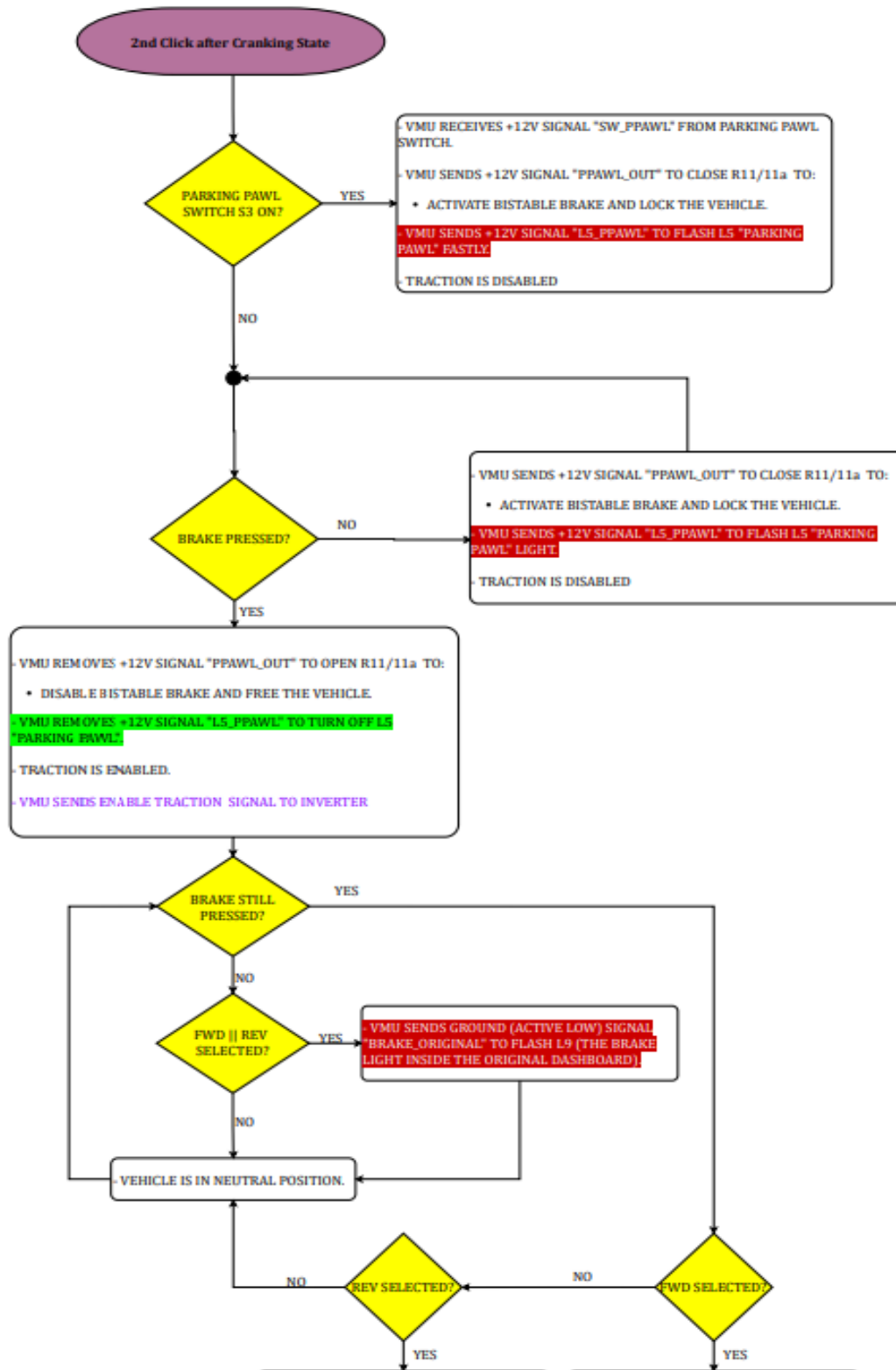


Figure 2. 8: software block diagram of Click After Cranking Mode Part1

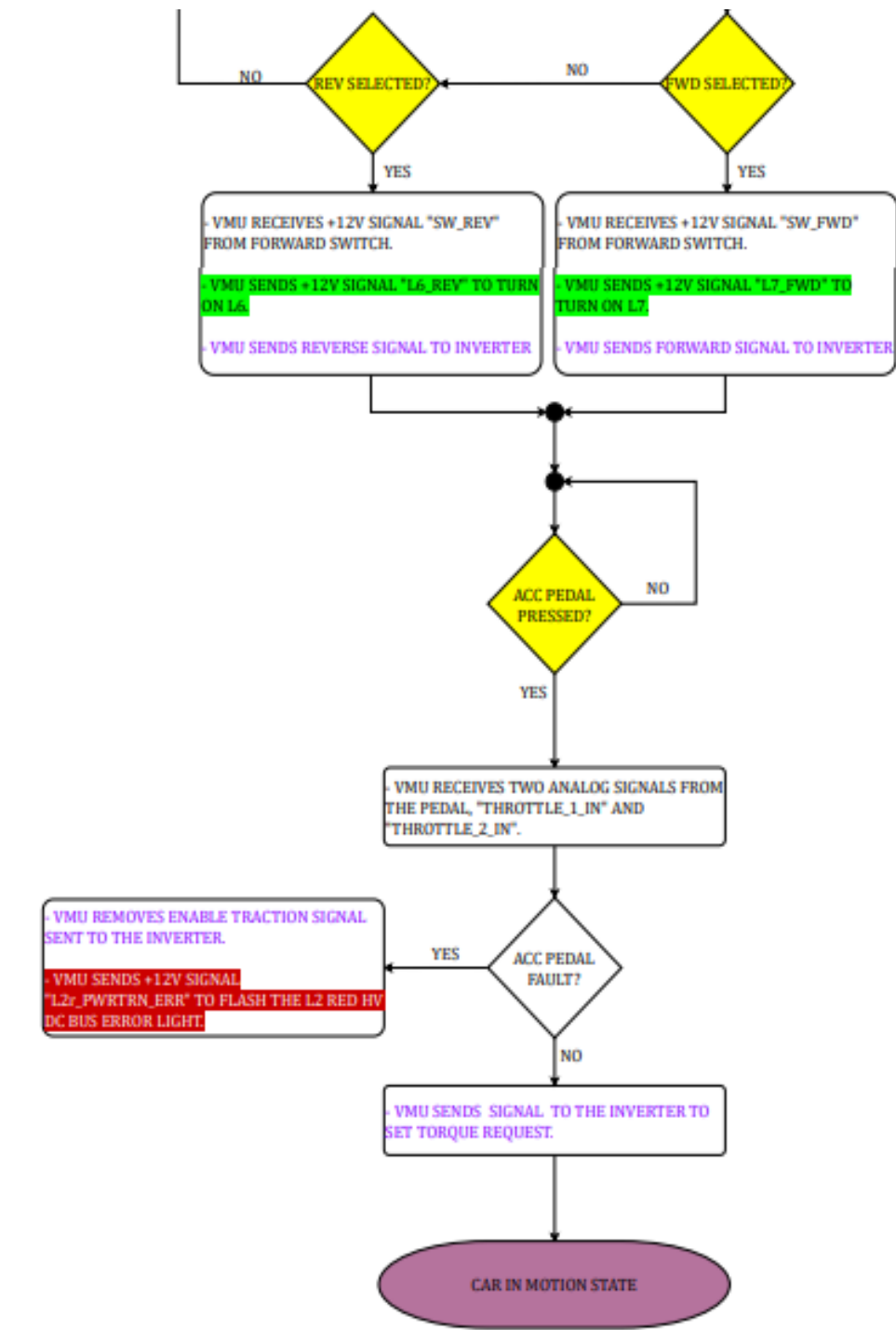


Figure 2. 9: software block diagram of Click After Cranking Mode Part2

2.2.1.5 CAR IN MOTION STATE

Main goal of this task is behavior of the VMU to the actions for the when the brake is pressed, FWD or REV selected, and the direction switch toggled when the current state of the car is in the motion state.

First, the algorithm will check that if the motor speed is above of the zero. If the motor speed is still zero and the direction switch is toggled and also the brake is pressed in the continuously then task going to check one of the REV and FWD is selected.

If REV is selected the VMU receives +12 'SW_REV' signal from forward switch and sends back 'L6_REV' signal to turn on L6 and sends reverse signal to the inverter.

If FWD is selected the VMU receives +12 'SW_FWD' signal from forward switch and sends back 'L7_FWD' signal to turn on L7 and sends reverse signal to the inverter.

After checking the forward and reverse switch it checks if acc pedal is pressed if yes then VMU receives two analog signals from the pedal which are called 'Throttle_1_in' and 'Throttle_2_in' signals and checks if there will be any pedal fault. In the case of without any error, VMU sends a signal to set the torque to the inverter.

Otherwise, if there will be any pedal error then VMU removes enable traction signal which is sent to the inverter before and send another error signal 'L2r_PWRTRN_ERR' to flash the L' red HV DC bus error light.

If motor speed is more than zero without considering situation of the direction switch, and also ACC Pedal is pressed then VMU receives the same 2 signals and checks if there is a ACC pedal fault and repeat the process as I explained above.

Below is illustrated in figure 2.10 the block diagram of the how charging process is starts during the Car In Motion state.

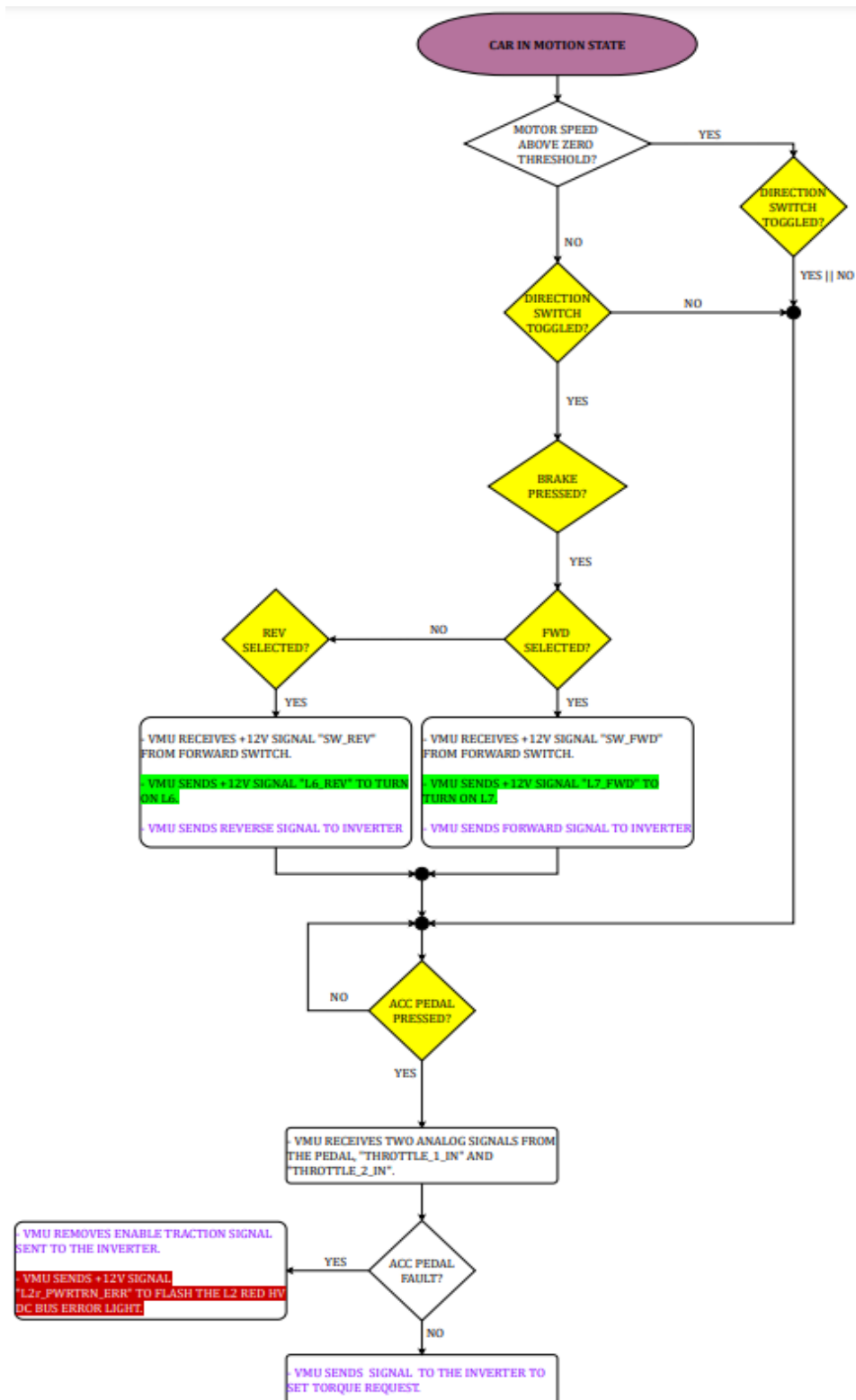


Figure 2. 10: software block diagram of the car in motion state

2.2.1.6 HV BATTERY FAULT MANAGEMENT

This fault loop is checking periodically HV battery fault, so whenever fault occurs then VMU receives 'HVBAT_FAULT_IN' signal and that activates the procedure with removing the 'INV_INTLCK_OUT' signal to open the main conductor. Additionally, VMU also removes 'HVBAT_WKON_OUT' signal to turn off HV DC bus and 'INV_TRACT_EN_OUT' signal to disable traction. Meanwhile VMU sends other 2 signals to turn on warning led and turn on the buzzer on the vehicle. Below is illustrated the block diagram of HV battery management shown in figure 2.11.

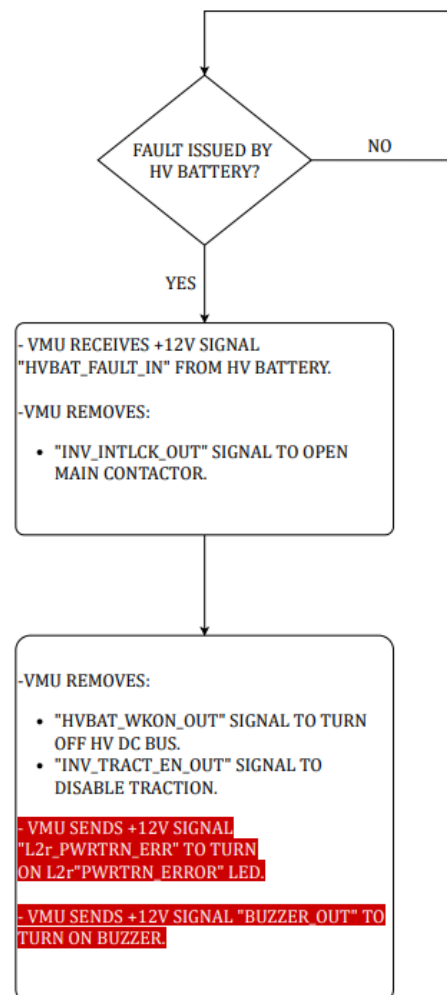


Figure 2. 11: software block diagram of HV battery management

2.2.1.7 DCDC FAULT MANAGEMENT

When the DCDC fault occurs, VMU disable the DC/DC converter with removing the 'DCDC_PWR_OUT' signal immediately to open R17 switch and also sends another signal to turn on the warning light immediately on the original dashboard battery light.

Below is illustrated the block diagram of DCDC fault management in figure 2.12.

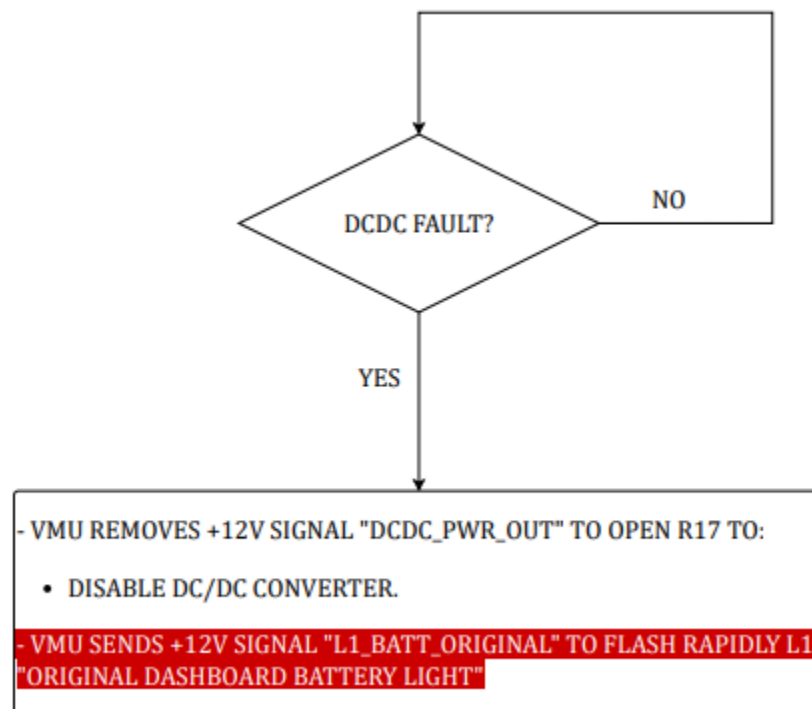


Figure 2. 12: software block diagram of DCDC fault management

2.2.1.8 PARKING MANAGEMENT

This task is starting the whenever user take the parking pawl switch S3 to on position. Later it checks if the brake is also pressed and if this condition is also true then VMU receives 'SW_PPAWL' signal from the parking pawl switch and sends back +12v 'PPAWL_OUT' signal to close R11/11a switch which is activating the bistable brake and lock the vehicle. And for the end of this condition VMU activates 'PARKING PAWL' led with sending another signal to the dashboard.

Another condition will start the part of the task when the parking pawl switch is closed, and brake is pressed at the same time. When this condition is active then VMU removes 'PPAWL_OUT' to open R11/11a switch to disable the bistable brake and this leaves the vehicle free position. At the end VMU sends another signal to deactivate the led of the parking pawl in the dashboard.

Below is illustrated the block diagram of parking management in figure 2.13.

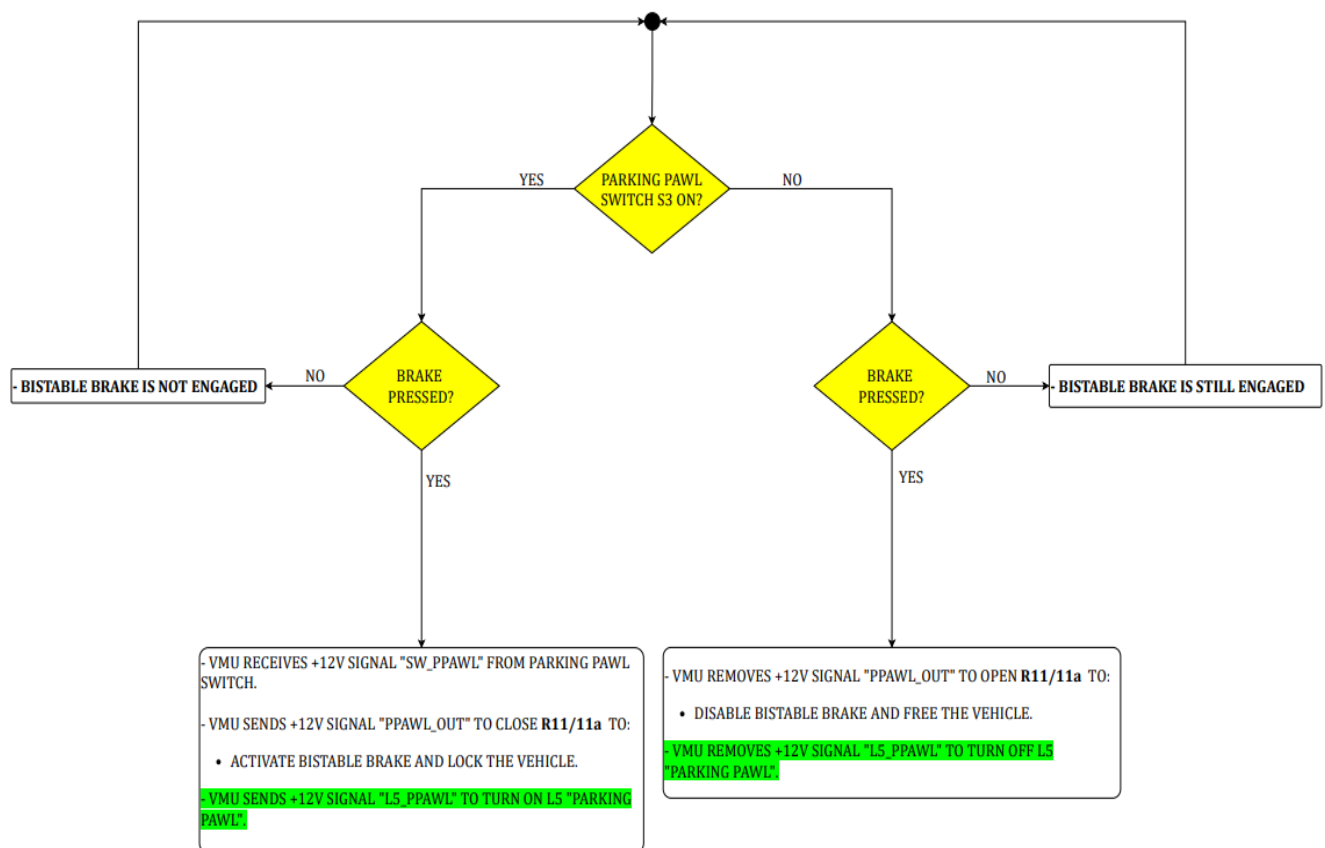


Figure 2. 13: software block diagram of parking management

2.3 RTOS

RTOS is a real time operating system. It is a software component which quickly switches among tasks, creating the impression that multiple programs are running on a single processing core at the same time.

RTOS has 2 key characteristics:

1. Predictability.
2. Determinism.

2.3.1 RTOS features and key reasons why it is used of critical systems

- Determinism: If you repeat an input, you'll get the same result [11].
- High performance: RTOS based systems are quick and efficient, often completing tasks in a fraction of the time required by a traditional operating system.
- Safety and security: RTOS is commonly used in critical systems, such as robotic systems or flight controllers, where breakdowns can have devastating results and consequences. They should have larger safety requirements and much more accurate and reliable safety features to prevent failures [12].
- Prioritized class schedules: means that high-priority tasks have to be completed first, then lower-priority tasks, which means that the highest priority tasks will always be executed by an RTOS [13]. For example, if a vehicle user is pressing the brakes and at the same time increasing the volume of the radio, for the user he will see that both tasks will be executed at the same time because it all happens very fast. But actually the brakes system will be executed by the RTOS because it is a critical task, while the radio is not.

2.3.2 Classic OS vs RTOS

The response time to external events differs between an OS (Operating System) such as Windows or Unix and an RTOS (Real Time Operating System) found in embedded systems. Operating systems usually provide a non-deterministic, non real-time response, in which there are no assurances as to when every task would be completed, but they will make every effort to remain responsive to the user. An RTOS differ significantly in that it provides a solid real-time response, meaning it reacts to external events quickly and predictably. [14] Comparing the editing of a

pdf document on a PC for example to the process of a motor control, highlights the difference between the two.

Free RTOS which is a class of RTOS is typically implemented for microcontroller and small microprocessor. It is the need operating system for applications which demand real time response as ABS of the car for example, the ABS is a function that cannot be delayed it has to be executed in real time, which means that when the vehicle crashes or hits an object hard enough that that requires the ABS to be executed it has to be executed with no delays.

FreeRTOS is an open source operating system that includes a kernel and a growing set of libraries that can be used in a variety of industries. FreeRTOS is designed with dependability, accessibility, and ease of use in mind.

FreeRTOS is a completely free operating system that can be utilized in commercial applications. There are a number of other factors that make FreeRTOS a good choice [15]:

- Has a small amount of ROM, RAM, and computing power. Its kernel binary image is typically in the 6K up to 12K byte range.
- Simple. The RTOS kernel core is comprised within just three C files.
- offers a unified and self-contained solution for a wide range of architectures and design tools.
- For each and every port, there is a pre-configured example. There's no need to learn how to set up a project; simply and straight forward download and compile.
- Has a fantastic, well-managed, and active free support forum.
- guaranteed that commercial assistance will be available if needed.
- Is quite scalable, straightforward, and simple to use.

2.3.4 RTOS architectures

1. *Monolithic RTOS:*

Monolithic refers to a single massive stone. A monolithic kernel continues to run all the operating system components in kernel space. The kernel space of a monolithic RTOS, for example, includes device drivers, file management, networking, and graphics stack. Even though, applications that run in the user space. Despite the fact that running user applications as memory-protected processes protects a monolithic kernel from errant user code, a single programming error in a file system, protocol stack, or driver can cause the system to crash. Furthermore, any change to a driver or system file necessitates an OS update and recompilation. [16] [17]

The monolithic RTOS architecture is illustrated in figure 2.14.

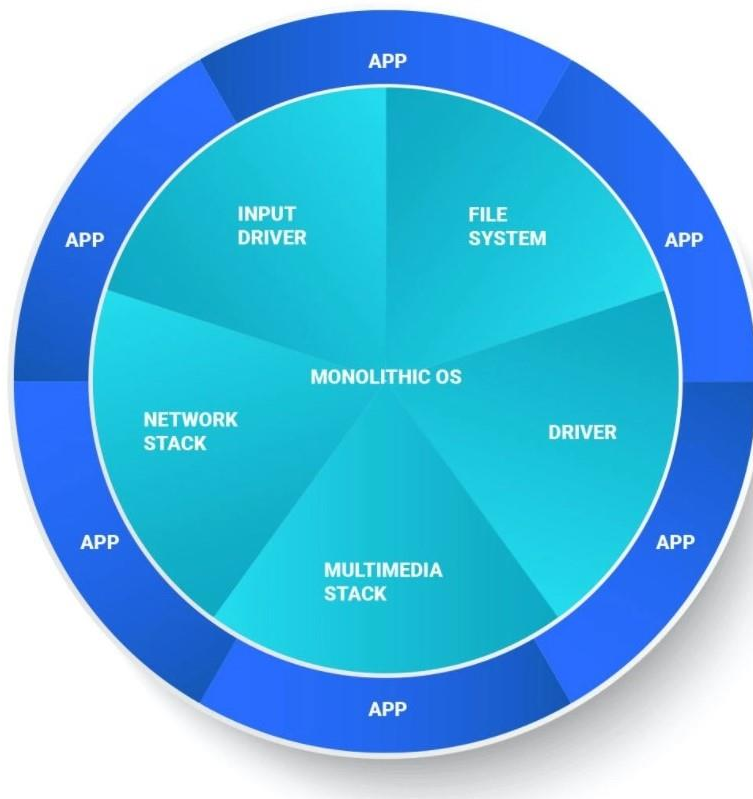


Figure 2. 14: Monolithic operating system architecture diagram

Monolithic RTOS advantages:

- Thread and process scheduling and file management are all run in the same address space as a single large process, which enhances performance.
- The full operating system is included within fixed binary file that runs faster and more accurately than dynamically linked libraries.

Monolithic RTOS disadvantages:

- Any service failure can cause the entire Operating system to crash.
- Modifying and recompiling the Operating system is required to add or remove a service.
- The kernel services of the operating system represent a large attack surface; if one service is compromised, the entire system is vulnerable.
- The footprint is quite large.
- It's difficult to debug and keep up with.

2. *microkernel RTOS:*

A microkernel RTOS is made up of a small kernel that offers only the most basic services. The microkernel collaborates with a group of optional collaborating processes that run outside kernel space, allowing for higher-level Operating system functionality. The microkernel itself is devoid of file systems and many other services that one would expect from an operating system. A microkernel RTOS embraces a fundamental shift in the way Operating system functionality is delivered: modularity is the key, and small size is a bonus.

Just the cornerstone RTOS kernel has access to the entire system in a microkernel, which enhances security and reliability. The microkernel provides task switching as well as memory protection and allocation for other processes. All other components, such as drivers and components of the system level, are isolated in their own process space.

The monolithic RTOS architecture is illustrated in figure 2.15.

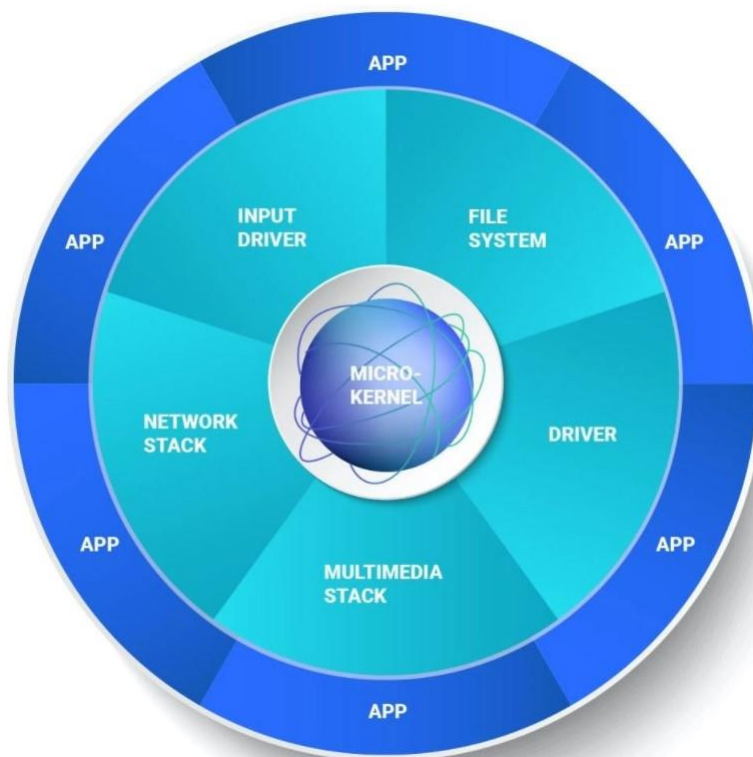


Figure 2. 15: Microkernel operating system architecture diagram

Microkernel RTOS advantages:

- Without affecting the kernel, dynamically restart a failed system service (no system reboot).
- Expansion is simple.
- Easy debug.
- Small footprint.

Microkernel RTOS advantages:

- Increased over head due to the fact that it requires more context switching.

2.4 DRIVERS

2.4.1 Board Stress Test and Driver Initialization

In this part it is consisted in carrying out performance analyzes of the HY-TTC 30 board produced by TTControl.

These analyzes were conducted to try to quantify the computation capacity of the board in a certain time interval, in order to obtain sufficient data to start developing the tasks that our VMU will have to perform.

Four tests were conducted in order to obtain:

1. The maximum number of multiplications that can be performed in a task lasting 100 milliseconds.
2. The time needed to initialize all the peripherals on the board.
3. The time required to activate and shut down all peripherals on the board.
4. The time it takes to read the value of an ADC pin.

2.4.1.1 Test 1: Obtain the maximum number of multiplications that can be performed in a 100ms task

This test was carried out to evaluate the computational capacity of the board.

The operation tested was multiplication. Since, this is generally the most expensive operation. So, finding a maximum limit of possible multiplications gives us an idea of what can be performed within a task.

The duration of the task has been set at 100ms and corresponds to our initial estimate of what ideally the tasks should last.

```
while (TRUE != Check_Task_End(task_test_loop_timestamp, (ubyte4)TASK_TL_CYCLE_TIME))
{
    test_mul = test_mul*(counter);
    counter++;

    if(counter == TEST_LIMIT){
        IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION_TL, 14, FALSE, string);

        while (IO_EEPROM_PreloadStatus () != IO_E_OK)
        {
            (void) IO_EEPROM_PreloadTask ();
        }
        UART_Printf (IO_UART, "\n\r Done !\n\r");
    }
}
```

Figure 2. 16: Test 1 implementation

It is illustrated in the code above in figure 2.16, the implementation of the task is very simple. It is used a while loop and within this a multiplication is performed and a variable is incremented counter.

At the end of each cycle, it is checked whether the task has reached its end. When the variable counter reaches the value TEST_LIMIT a writing is made on the EEPROM, in order to verify the achievement of the predetermined multiplication value.

This operation was made necessary by the unreliability of printf UARTs.

The TEST_LIMIT variable is the value that has been incremented in subsequent runs of the task.

The values that have been tested:

- ✓ 10
- ✓ 100
- ✓ 1000
- ✓ 10000
- ✓ 20000
- ✓ 30000
- ✓ 35000
- ✓ 37000
- ✓ 39000
- ✓ 40000
- ✓ 100000

The initial value was 10. Once it was verified (trivially) that it was possible to perform 10 multiplications, the value of TEST_LIMIT has been increased by an order of magnitude. Finally, verified that it was not possible perform 100000 multiplications, restarted from 10000 by increasing the value of TEST_LIMIT by 10000 to the time.

The maximum number of multiplications (to which, for each cycle, an addition and a "appears" are added to check the end of the task) is 39000.

2.4.1.2 Test 2: Find the initialization time of all devices

When starting the car, it is necessary to initiate all the necessary peripherals. The initialization of a peripheral is an even more expensive operation than multiplication, however it is only required once. Therefore, the initialization took place inside the main, before the while loop that deals with calling the tasks.

The implementation is illustrated in figure 2.17 below.

```

vout_rc = IO_VOut_Init (IO_VOUT_00, NULL, NULL );
vout_rc = IO_VOut_Init (IO_VOUT_01, NULL, NULL );
vout_rc = IO_VOut_Init (IO_VOUT_02, NULL, NULL );
vout_rc = IO_VOut_Init (IO_VOUT_03, NULL, NULL );
vout_rc = IO_VOut_Init (IO_VOUT_04, NULL, NULL );
vout_rc = IO_VOut_Init (IO_VOUT_05, NULL, NULL );

IO_ADC_ChannelInit(IO_ADC_00, IO_ADC_ABSOLUTE, IO_ADC_RANGE_10V, NULL);
IO_ADC_ChannelInit(IO_ADC_01, IO_ADC_ABSOLUTE, IO_ADC_RANGE_10V, NULL);

IO_POWER_Set (IO_INT_PIN_PVG_VOUT_0_ENABLE, IO_POWER_ON);
IO_POWER_Set (IO_INT_PIN_PVG_VOUT_1_ENABLE, IO_POWER_ON);

IO_CAN_Init (IO_CAN_CHANNEL_0, IO_CAN_BAUDRATE_500K );
IO_CAN_Init (IO_CAN_CHANNEL_1, IO_CAN_BAUDRATE_500K );

IO_DO_Init(IO_DO_00, 2500);
IO_DO_Init(IO_DO_01, 2500);
IO_DO_Init(IO_DO_02, 2500);
IO_DO_Init(IO_DO_03, 2500);
IO_DO_Init(IO_DO_04, 2500);
IO_DO_Init(IO_DO_05, 2500);
IO_DO_Init(IO_DO_06, 2500);
IO_DO_Init(IO_DO_07, 2500);
IO_DO_Init(IO_DO_10, 2500);

IO_DI_Init(IO_DI_02, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_03, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_04, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_05, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_06, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_07, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_10, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_11, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_12, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_13, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_14, IO_DI_PU, &limits);
IO_DI_Init(IO_DI_15, IO_DI_PU, &limits);
IO_POWER_Set (IO_INT_POWERSTAGE_ENABLE, IO_POWER_ON);

```

Figure 2. 17: Test 2 implementation part1

The initialized peripherals shown in the figure are for voltage output (IO_VOut_Init), for communication

via CAN on the 2 available channels (IO_CAN_CHANNEL), for the ADC (IO_ADC_ChannelInit) and for input / output digital (IO_DO_Init and IO_DI_Init).

The number of pins initialized for each function to perform depended on the fact that each pin is programmed to carry out different ones. For further information, please refer to the official documentation of the board (in particular, see IO_Pins.h).

As regards the activation of the peripherals, a task has been implemented, which will be deepened in the next section, which activates all the peripherals initialized in figure 2.18.

In order to measure the initialization times, a digital oscilloscope was used. Specifically, like shown in figure 3, the first pin that is raised is the voltage output pin IO_VOUT_00 (corresponding on the board to the IO_PIN_K2 pin). At this point, through an oscilloscope display software, a trigger was inserted that tripped when the IO_VOUT_00 pin rose, that is, at the end of the process initialization.

In this way it was possible to obtain the time we were looking for, which turned out to be equal to 220 milliseconds.

```
vout_er = IO_VOut_SetVoltage(IO_VOUT_00, (ubyte2) 32000, &voltage);

IO_CAN_ConfigMsg(&handle_w_can0, IO_CAN_CHANNEL_0, IO_CAN_MSG_WRITE, IO_CAN_STD_FRAME,0,0);
IO_CAN_ConfigMsg(&handle_w_can1, IO_CAN_CHANNEL_1, IO_CAN_MSG_WRITE, IO_CAN_STD_FRAME,0,0);

// assemble CAN0 frame:
can_frame_can0_w.id = 1;
can_frame_can0_w.id_format = IO_CAN_STD_FRAME;
can_frame_can0_w.length = 6;
can_frame_can0_w.data[0] = 1;
can_frame_can0_w.data[1] = 2;
can_frame_can0_w.data[2] = 3;
can_frame_can0_w.data[3] = 4;
can_frame_can0_w.data[4] = 0;
```

Figure 2. 18: Test 2 implementation part2

2.4.1.3 Test 3: Find the activation and shutdown times of all peripherals

The same method was used to find the activation and deactivation time of the peripherals used for test 2.

As mentioned above, a task has been implemented, called immediately after the initialization of peripherals, which activates and turns off all peripherals. The first pin to be raised is IO_VOUT_00, which corresponds to the last pin to be lowered, after the activation and deactivation of all devices initialized.

Therefore, a trigger has been inserted again when the IO_VOUT_00, pin is lowered in order to be able to obtain the time sought, which was 60 milliseconds.


```

IO_DO_Set(IO_DO_00,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_01,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_02,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_03,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_04,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_05,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_06,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_07,FALSE,&do_voltage_fb);
IO_DO_Set(IO_DO_10,FALSE,&do_voltage_fb);
vout_er = IO_VOut_SetVoltage(IO_VOUT_05, (ubyte2) 0, &voltage);
vout_er = IO_VOut_SetVoltage(IO_VOUT_04, (ubyte2) 0, &voltage);
vout_er = IO_VOut_SetVoltage(IO_VOUT_03, (ubyte2) 0, &voltage);
vout_er = IO_VOut_SetVoltage(IO_VOUT_02, (ubyte2) 0, &voltage);
vout_er = IO_VOut_SetVoltage(IO_VOUT_01, (ubyte2) 0, &voltage);
vout_er = IO_VOut_SetVoltage(IO_VOUT_00, (ubyte2) 0, &voltage);

driver_task_end_rc = IO_Driver_TaskEnd ();

```

Figure 2. 19: Test 3 implementation

2.4.1.4 Test 4: Find the time to read the value of an ADC pin

To find the activation time of an ADC pin, a specific task has been implemented; a fragment of it is illustrated in figure 2.20.

```

driver_task_begin_rc = IO_Driver_TaskBegin ();

adc_init_rc = IO_ADC_ChannelInit(IO_ADC_00, IO_ADC_ABSOLUTE, IO_ADC_RANGE_10V, NULL);
adc_init_rc = IO_ADC_ChannelInit(IO_ADC_01, IO_ADC_ABSOLUTE, IO_ADC_RANGE_10V, NULL);

vout_rc = IO_Vout_Init (IO_VOUT_00, NULL, NULL );
IO_POWER_Set (IO_INT_PIN_PVG_VOUT_0_ENABLE, IO_POWER_ON);
IO_POWER_Set (IO_INT_PIN_PVG_VOUT_1_ENABLE, IO_POWER_ON);

vout_rc = IO_VOut_SetVoltage(IO_VOUT_00, (ubyte2) 32000, &voltage);

adc_error_1 = IO_ADC_Get(IO_ADC_00, &adc_20, &adc_20_fresh);

vout_rc = IO_VOut_SetVoltage(IO_VOUT_00, (ubyte2) 0, &voltage);

driver_task_end_rc = IO_Driver_TaskEnd ();

```

Figure 2. 20: Test 4 implementation

The activated pin IO_ADC_00 corresponds on the board to the IO_PIN_J4 pin. To find the reading time of this pin it was necessary to raise and lower the IO_VOUT_00 pin. In this way it was possible to insert a trigger in the same way exactly as the documented tests previously.

The time found was 2 milliseconds.

2.4.2 CAN FIFO Buffer

This activity reported the implementation of a function which puts several CAN messages in a FIFO buffer in order to use the data fields of the mentioned messages to set some variables. The test can function has been created only to test the feasibility of capturing a sequence of CAN frames into a buffer. The values of the data fields of these frames are then utilized to set variables.

The implementation is reported in figures 2.21 below.

```
void test_can(void) {
    uint8_t i = 0;
    uint8_t handle_can_msg;
    uint8_t frame_actually_copied;
    IO_CAN_DATA_FRAME can_frame[NUMBER_OF_MESSAGES];
    IO_CAN_DATA_FRAME can_frame_2;
    IO_ErrorType can_msg_conf;
    IO_ErrorType can_status;

    can_msg_conf = IO_CAN_ConfigFIFO(handle_can_msg,
                                     IO_CAN_CHANNEL_0,
                                     NUMBER_OF_MESSAGES,
                                     IO_CAN_MSG_READ,
                                     IO_CAN_STD_FRAME,
                                     1,
                                     0);

    UART_Printf (IO_UART, "\n\r CONFIG RETURN: %d \n\r", can_msg_conf);

    can_status=IO_CAN_FIFOStatus(handle_can_msg);

    if ((can_status == IO_E_OK) || (can_status == IO_E_CAN_OVERFLOW)) {
        can_status = IO_CAN_ReadFIFO(handle_can_msg, &can_frame,NUMBER_OF_MESSAGES,&frame_actually_copied);
        UART_Printf (IO_UART, "\n\r CAN return error: %d \n\r", can_status);
        UART_Printf (IO_UART, "\n\r Frame actually copied: %d \n\r", frame_actually_copied);
    }

    UART_Printf (IO_UART, "\n\r CAN STATUS %d \n\r", can_status);
}

// Figure 6 - test_can function (1/2)

while(i != NUMBER_OF_MESSAGES) {
    switch(can_frame[i].id) {
        case 0:
            can_variable1 = can_frame[i].data[0];
            can_variable2 = can_frame[i].data[1];
            can_variable3 = can_frame[i].data[2];
            can_variable4 = can_frame[i].data[3];
            break;
        default :
            can_variable1 = can_frame[i].data[0];
            can_variable2 = can_frame[i].data[1];
            can_variable3 = can_frame[i].data[2];
            can_variable4 = can_frame[i].data[3];
            break;
    }

    UART_Printf (IO_UART, "\n\r READ FIFO CAN FRAME ID: %d \n\r", can_frame[i].id);
    UART_Printf (IO_UART, "\n\r READ FIFO CAN FRAME DATA FIELD 1: %d \n\r", can_variable1);
    UART_Printf (IO_UART, "\n\r READ FIFO CAN FRAME DATA FIELD 2: %d \n\r", can_variable3);
    UART_Printf (IO_UART, "\n\r READ FIFO CAN FRAME DATA FIELD 3: %d \n\r", can_variable3);
    UART_Printf (IO_UART, "\n\r READ FIFO CAN FRAME DATA FIELD 4: %d \n\r", can_variable4);

    i++;
}
```

Figure 2. 21: Can FIFO Buffer Implementation

The number of items that may be stored into the buffer is equal to the constant NUMBER OF MESSAGES. Notice that the UART prints are reported solely for debug purpose, and they were removed once the validity of the method has been verified.

A demonstration is given in the following figures: the number of CAN frames transmitted is equal to 5, simply to give a brief look to the behavior of this implementation.

In the graphic below in figure 2.22 the messages sent and those are going to be captured into the buffer are reported.

CAN-ID	Type	Length	Data
001h		8	01 01 01 01 01 01 01 01
002h		8	02 02 02 02 02 02 02 02
003h		8	03 03 03 03 03 03 03 03
004h		8	04 04 04 04 04 04 04 04
005h		8	05 05 05 05 05 05 05 05

Figure 2. 22: Buffer Report

In the following figures (2.23, 2.24), the UART prints in the terminal are reported.

```

READ FIFO CAN FRAME ID: 3 !
READ FIFO CAN FRAME DATA FIELD 1: 3 !
READ FIFO CAN FRAME DATA FIELD 2: 3 !
READ FIFO CAN FRAME DATA FIELD 3: 3 !
READ FIFO CAN FRAME DATA FIELD 4: 3 !
READ FIFO CAN FRAME ID: 5 !
READ FIFO CAN FRAME DATA FIELD 1: 5 !
READ FIFO CAN FRAME DATA FIELD 2: 5 !
READ FIFO CAN FRAME DATA FIELD 3: 5 !
READ FIFO CAN FRAME DATA FIELD 4: 5 !
READ FIFO CAN FRAME ID: 2 !
READ FIFO CAN FRAME DATA FIELD 1: 2 !
READ FIFO CAN FRAME DATA FIELD 2: 2 !
READ FIFO CAN FRAME DATA FIELD 3: 2 !
READ FIFO CAN FRAME DATA FIELD 4: 2 !

```

Figure 2. 23: UART Prints (1)

```
READ FIFO CAN FRAME ID: 4 !  
READ FIFO CAN FRAME DATA FIELD 1: 4 !  
READ FIFO CAN FRAME DATA FIELD 2: 4 !  
READ FIFO CAN FRAME DATA FIELD 3: 4 !  
READ FIFO CAN FRAME DATA FIELD 4: 4 !  
READ FIFO CAN FRAME ID: 1 !  
READ FIFO CAN FRAME DATA FIELD 1: 1 !  
READ FIFO CAN FRAME DATA FIELD 2: 1 !  
READ FIFO CAN FRAME DATA FIELD 3: 1 !  
READ FIFO CAN FRAME DATA FIELD 4: 1 !
```

Figure 2. 24 UART prints (2)

2.5 Automotive Standards (AUTOSAR)

AUTOSAR (Automotive Open System Architecture) is a global development collaboration of automotive stakeholders established in 2003. AUTOSAR offers an open software architecture that is standardized for automobile ECUs.

Without an uniform model, such as AUTOSAR, Manufacturers developed ECU software on separate platforms. Tier 1 manufacturers and its distributors utilized a range of different software architectures to develop Electronic Control Unit software for OEMs. With this strategy, it was very difficult for OEMs to move to a new tier 1 supplier or vice versa.

The new provider previously had tremendous difficulties in comprehending the current software architecture, hardware platforms, and standards utilized in the creation of ECU software. The new supplier confronts important difficulties in restarting an ongoing project in the middle of its manufacturing life cycle. The following are some of the stated goals, primary difficulties, and proposed solutions by AUTOSAR, along with the associated advantages.

These are managing the system's increasing electrical/electronic complexity, freedom of implementation for product change, upgrading, and updating, increase the flexibility and cross-compatibility of software and services, increase the system quality of the software and dependability, allows error detection throughout the early stages of development. [18]

2.5.1 AUTOSAR Architecture

AUTOSAR is a standardized open-source software architecture for the automobile sector. The AUTOSAR architecture provides a common interface between application software and fundamental automotive operations. AUTOSAR is designed to help members by assisting companies in managing increasing complexity E/E in-vehicle settings.

Layered Software Architecture is the term used to describe the Autosar framework. Layered architecture explains the hierarchical organizational structure of AUTOSAR software from the top down. It connects the Fundamental Software Modules to software layers and illustrates their connection.

In new cars, the number of electronic/electric systems and their complexity are growing. AUTOSAR was developed in response to the growing complexities of the vehicle network. Every modern car has over a hundred ECUs. Each of them performs thousands of tasks. Without following to the guideline, it is quite probable that software development will have to be redone whenever the ECU design specification is modified. AUTOSAR enables hardware-independent program creation, therefore standard software is much more transferrable. This enables software to also be readily shared across various vehicle systems, generally regardless of the system's hardware resources, which AUTOSAR enhanced via component interaction standardization.

AUTOSAR's application scope is limited to vehicle ECUs. These ECUs do have following features. These are strong connection with hardware, connectivity to automotive networks such as CAN, LIN, Ethernet, and microcontrollers with restricted computation and memory capabilities. System that is time-critical and executes real-time programs from internal storage. [19] [20]

LAYERS OF AUTOSAR ARCHITECTURE:

As illustrated in figure 2.25 below the AUTOSAR Structure differentiates different software levels at the highest abstraction level. These are Application Layer, Runtime Environment and Basic Software which is running on the microcontroller. [20]

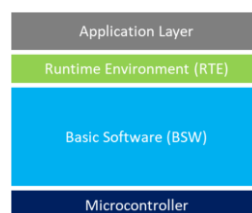


Figure 2. 25: AUTOSAR Layer Architecture

2.5.1.1 Application level

The application level is indeed the uppermost level of the AUTOSAR architecture and therefore is responsible for the implementation of bespoke functions. This layer is comprised of numerous software products and apps, which are each composed of a collection of linked AUTOSAR Software Components that executes particular duties in response to commands. Every AUTOSAR System Software contains a subset of the application's operations.

AUTOSAR makes no specification regarding the size of AUTOSAR Software Applications. Based on the application's needs, an AUTOSAR System Software may consist of a tiny, reusable portion of functionality like line of traffic support, wiper management, and automatic door unlocking. By the use of a virtualized Functional Bus, communication across software modules is facilitated through particular ports. Additionally, these ports enable communication among software components and the AUTOSAR Basic Software. [20]

2.5.1.2 Runtime environment

The Runtime Environment level communicates with the software modules, which may include AUTOSAR Subprograms and AUTOSAR Sensor/Actuator Elements. RTE level enables software application modules with ECU-independent application interfaces. The application level is composed of numerous SWC that do not correspond to the layered architectural style but instead to the component approach. RTE is used to interact with the Software Modules and other elements (inter and intra ECU). The SWC Interfaces are fully isolated from the ECU. It separates AUTOSAR Software Modules from of the mapping to a particular ECU. Communications between SWCs is mostly accomplished through two types of ports. These are Client-Server port and Sender-Receiver ports. Client/Server ports are those in which the server is the provider and indeed the client is just the service user. Sender/Receiver ports are used when a sender delivers data to one or more recipients. [21]

2.5.1.3 AUTOSAR Basic Software (BSW)

This layer is further divided into different 3 layers which are Service Layer, ECU abstraction Layer, Microcontroller Abstraction and Complex Device Drivers layer. Each one of the three levels is composed of a few distinct functional groupings. Each one of these tasks may be divided into distinct modules.

To create a packetized software control structure, each functional group interacts with a specific software module located in the subsequent tier.

Because the Microcontroller Layer is the bottom level of the Fundamental Software, MCAL units have full access to the hardware resources. Internal drivers are program units that provide full access to the CPU and inner peripherals in MCAL. As the title indicates, the MCAL level isolates the higher layers from the Hardware (MCU).

The Electronic Control Unit Abstraction Layer communicates with the Mcu Abstract Layer's drivers (MCAL). Additionally, it includes drivers for external hardware included inside the ECU and acts as a layer of abstraction for different peripherals.

It offers interfaces for accessing all of an ECU's capabilities, including as communication, memory, and I/O, regardless of whether these capabilities are integrated within the microprocessor or are provided by peripheral devices.

First from underlying hardware level to the RTE, Complex Device Drivers (CDD) Layer is present.

CDD satisfies the unique functionality and timing constraints associated with the operation of sophisticated sensors and actuators.

Allow for the integration of specialized functions.

This level comprises drivers for items which were not defined in AUTOSAR and are subject to very strict time constraints.

The Service Level is the outermost level of the Basic Software (BSW), and it also has application software implications. It offers a separate api for application software to communicate with a microcontroller (MCU) as well as ECU hardware.

Services Level provides the following properties:

- Capability of the OS.
- Solutions for automotive network connection and administration.
- Service related to memories diagnostics (UDS).
- Control of the ECU's status and mode Controlling the logical and chronological flow of a program.
- Task Provides fundamental capabilities for programs, remote terminal emulators, and fundamental software components.
- Each AUTOSAR level is composed of a collection of recognized software components.
- Each module is responsible for the interfaces between its neighbors.

Chapter 3

3 The EVERGRIN VMU from TTcontrol

3.1 VMU Suppliers (TTcontrol) History

The VMU (vehicle management unit) to be installed in the vehicle for the successful transformation of the vehicle is the HY-TTC32S. The family is developed and sold by TTcontrol which is a joint venture between TTTech and HYDAC International which is located in Brixen, Vienna and Austria. They offer controller interfaces and Control systems for heavy vehicles and mobile machinery. Equipment manufacturers can quickly and affordably develop highly reliable electronic control systems using their software and hardware platforms, which are recognized as industry leaders in functional safety.

For over 20 years, TTControl has been involved with commercial production projects in the field of electronic control systems for heavy vehicles (off-highway vehicles) such as cranes, forklifts and snow groomers, which rely on their equipment to function properly even in the most adverse conditions. They offer Electromechanical Control Units (ECUs) for high-pressure environments, I/O slave modules based on the CANopen protocol, functional safety, and Operator interfaces that are extremely durable. And they have a wide variety of applications such as construction, warehousing and distribution, agriculture, municipal and special vehicles. This is besides their collaboration with TTTech group on a number of research and development projects, which is majorly focused on determinism, real time triggered protocols and real time performance in applications involving safety-relevant data communication in mixed critically environments, basically their goal is to seamlessly integrate and combine such metrics with existing communication methods that are currently used in many industrial domains including, , space, aerospace, automotive off-highway, railroad, energy, and many others.

As mentioned above by TTcontrol is a joint venture between TTTech and HYDAC International. TTTech group includes several companies which are TTTech Industrial automation AG, TTTech auto AG, and TTControl gmbH which are are high-tech enterprises with a global focus that run under the roof of the TTTech Group. The solutions provided by TTTech Group, which include real-time networking platforms and safety controls, contribute to enhance the reliability and performance of electronic systems in the automotive segment, as well as to contribute to making the Iot and automated driving a reality in the near future. The companies provide products and services that are based on extremely creative software technology combined with a thorough understanding of the digital transformation process and its implications. They are involved in automotive (TTTech auto), aerospace, space, off-highway, manufacturing, railway, and energy.

3.2 The HY-TTC32S and the HY-TTC30 Family Technical Details and Architecture

3.2.1 Overview

32S is a powerful, yet cost-effective, electronic control unit. It is a safety-certified derivative of the HY-TTC 32 controller, which is available for purchase separately. It is equipped with the same processor and number of I/Os as the non-safety counterpart.

The 32S is a small control unit designed for applications with limited budgets or smaller machines. The device is made up of an Infineon XC22xx microcontroller that can be programmed in C. It can be controlled by a variety of sensors and actuators thanks to its 28 freely configurable Inputs outputs. Now this is the case of all the HY-TTC30 family in which the HY-TTC32S falls under, but what makes the HY-TTC32S an innovated and upgraded version of the HY-TTC30 family is the control unit with 2 can interfaces. As a result, it is perfectly suited for applications involving heterogeneous CAN networks (like, CANopen and J1939). And the fact that it has 2 can buses is actually one of the main reasons why we choose this ECU for EVERGRIN. The HYDAC controllers can be classified into three series, each of which is based on one of two powerful platforms: a 16-bit or a 32-bit processor, depending on the application. When a small compact design is required and high control choices are required, the HY-TTC 30 family is the ideal choice. Which is exactly the case of EVERGRIN.

3.2.2 Deeper into 32S and 30 family Technical details

The HY-TTC32S is without doubts one of the best compact ECUs and this can be for example because of its ability to control 3 hydraulic axes because it has 6 channels for pulse width modulation output with current measurement and other 2 channels for normal pulse width modulation which is actually the case for the other previously developed family members. While what makes the HY-TTC32 and the HY-TTC32S stand out is the can interface upgrade as they happen to have 2 CAN interfaces, while the HY-TTC30-H and the HY-TTC30S-H have only 1 CAN interface. While the 4 family members controllers are equipped with the same processor, the Infineon XC 22xx microcontroller running at 80MHz only the HY-TTC30S-H and the HY-TTC32S have watchdog which by interrupting the CPU and deactivating the safety switches via a dedicated output, can bring the ECU to a safe state. A watchdog processor is a compact simple coprocessor that monitors a system's behavior and detects faults. Basically, it's a hardware that checks and monitors the code execution for the purpose of resetting the processor in the case that

the software crashes. And it is one of the most powerful innovations in the embedded world. The 4 members of the family have the same memory, obviously the same flash since they are all equipped with the same processor, which is a 768 KBs of flash memory, 82 KBs of RAM, and 8 KBs of EEPROM. The 4 family members can have ISOBUS on request which is a CAN based standard protocol manages communication between tractors, software, and equipment from major manufacturers by allowing the exchange of data and information in a universal language through the use of a single control console located inside the tractor cab. ISOBUS was developed by the International Society of Tractor Manufacturers (ISTM). An agreement amongst the major agricultural machinery and equipment manufacturers to overcome compatibility issues by standardizing communication among different implements, regardless of the manufacturer, resulted in the Isobus protocol, which was developed to solve these issues. Because of Isobus, the cab is transformed into an authentic on-board computer that can operate the tools and implement, thereby enabling the transmission of data. This is not the case of EVERGRIN since Isobus is for off highway vehicles, mainly tractors and agriculture vehicles. All family members have 30 inputs and outputs: 8 pulse width modulation, 6 of those has current measurement. 10 analogue inputs, 6 analogue outputs. 4 timer inputs, and 2 digital outputs. The S versions (HY-TTC30S-H and HY-TTC32S) are functional safety certified. All family members are programmed in C programming language.

TUV NORD has certified the 32S, which was developed in accordance with the international standard EN ISO 13849. It complies with the Functional Safety requirements of Performance Level (PL) d.

The 32S version is ideal for proportional function control in safety applications. Six of the eight PWM outputs have integrated current measurement, allowing current control of up to three hydraulic axes which is quite impressive

The HY-TTC 32S was designed specifically for vehicles and machines that operate in harsh environments and at high temperatures. A proven, robust, and compact housing, specifically designed for the off-highway industry, protects the device.


							
	16-bit Controllers						
Type	HY-TTC 30-H	Functional safety PL c HY-TTC 30S-H	HY-TTC 32	Functional safety PL c HY-TTC 32S	HY-TTC 50	HY-TTC 60	Functional safety PL d HY-TTC 94
Processor	Infineon XC 22xx Microcontroller 80 MHz	Infineon XC 22xx Microcontroller 80 MHz Watchdog	Infineon XC 22xx Microcontroller 80 MHz	Infineon XC 22xx Microcontroller 80 MHz Watchdog	16-bit Infineon XC 2287 80 MHz		16-bit Infineon XC 2287 M 80 MHz Watchdog CPU
Memory	768 kB Flash 82 kB RAM 8 kB EEPROM				768 kB Flash 82 kB RAM 8 kB EEPROM	768 kB Flash 82 kB RAM 512 kB ext. RAM 8 kB EEPROM	832 kB Flash 50 kB RAM 512 kB ext. RAM 8 kB EEPROM
Interfaces	1 x CAN		2 x CAN		2 x CAN 1 x RS232 1 x LIN		4 x CAN 1 x RS232 1 x LIN
ISOBUS	On request		On request		On request		On request
Inputs and outputs ¹⁾ (Example configuration)	30 Total: 8 PWM (6 with current measurement) 10 Analogue IN 4 Timer IN 6 Analogue OUT (ratiometric) 2 Digital OUT				40 Total: 8 PWM 4 current meas. 8 Analogue IN 4 Timer IN 8 Digital IN 8 Digital OUT	48 Total: 8 PWM 4 current meas. 16 Analogue IN 4 Timer IN 8 Digital IN 8 Digital OUT	48 Total: 8 PWM 4 current meas. 16 Analogue IN 4 Timer IN 8 Digital IN 8 Digital OUT
Functional Safety (certified by TÜV Nord)		EN 13849 PL c		EN 13849 PL c			EN 13849 PL d
Programming	C		CODESYS V2.3 C	C	CODESYS V2.3 C		

Figure 3. 1: HY-TTC 30 and 50 families [22]


					
16-bit Controllers			32-bit μ -Controller Platform		
Specially for 12 V vehicle voltage			Functional safety PL d SIL 2	Functional safety PL d SIL 2	Functional safety PL d SIL 2
HY-TTC 71	HY-TTC 77		HY-TTC 510	HY-TTC 540	HY-TTC 580
16-bit Infineon XC 2288 H 80 MHz		32-bit TI TMS 570 Dual-core lockstep CPU 180 MHz Safety Companion CPU			
Watchdog CPU					
1.6 MB int. Flash	1.6 MB int. Flash	3 MB Flash	3 MB Flash	3 MB Flash	3 MB Flash 8 MB ext. Flash
138 kB RAM	138 kB RAM	256 kB RAM 2 MB ext. RAM	256 kB RAM 2 MB ext. RAM	256 kB RAM 2 MB ext. RAM	256 kB RAM 2 MB ext. RAM
32 kB EEPROM	32 kB EEPROM	64 kB EEPROM	64 kB EEPROM	64 kB EEPROM	64 kB EEPROM
1 x CAN	2 x CAN	3 x CAN 1 x LIN	4 x CAN	4 x CAN	7 x CAN 1 x RS232 1 x LIN 1 x RTC 1 x Ethernet
On request	On request	On request	On request	On request	On request
43 Total: 18 Digital OUT (6 with flyback diode) 24 Analogue IN 1 Timer IN	65 Total: 18 PWM 30 Analogue IN 2 Timer IN 7 Digital IN 8 Digital OUT	84 Total: 16 PWM (16 with current control) 24 Analogue IN 20 Timer IN 16 Digital OUT 8 multipurpose I/O	96 Total: 28 PWM (28 with current control) 32 Analogue IN 20 Timer IN 16 Digital OUT	96 Total: 36 PWM (36 with current control) 24 Analogue IN 12 Timer IN 16 Digital OUT 8 multipurpose I/O	
		IEC 61508 SIL 2 EN 13849 PL d			
C		CODESYS V3 CODESYS Safety SIL2 C			

Figure 3. 2: HY-TTC 500 family [22]

3.2.3 HY-TTC 32S Features and specifications

This section is based on the data sheet of the HY_TTC32S product [23].

- The Electronic Control Unit dimensions are $14 \times 92 \times 38$ mms.
- The ECU Weights 330 grams which is pretty light and won't cause installation problems.
- The Dimensions for Minimum Connector Release Clearance are $208 \times 92 \times 38$.
- The connector has 48 pins.
- The ECU operating temperature is -40 to $+85$ °C which is a more than enough range taking into account the presence of the vehicle almost anywhere in the world from the coldest places to the hottest places.
- The operating altitude is in the range of 0 to 4000 meters, which is again a more than enough range unless the user is using the vehicle to climb mountain Everest or so which is not a possible case.
- Supply voltage: 8 to 32 volts.
- Peak Supply Voltage: 40 volts.
- Idle current: up to 120 mA.
- Standby current: up to 1 mA.
- Total load current: 24 A.
- Standards:

Functional Safety	EN ISO 13849
CE-MARK	2014/30/EU 2006/42/EC
E-MARK	ECE-R10
EMC	EN 13309 ISO 14982 CISPR 25 EN 61000-6-2/-4
ESD	ISO 10605
Electrical	ISO 16750-2 ISO 7637-2,-3 Limited to 40V by external load bump protection.
Ingress protection	EN 60529 IP67 ISO 20653 IP6K9K
Climatic	ISO 16750-4
mechanical	ISO 16750-3

- The HY-TTC 32S is equipped with the infenion XC22xx which happens to be a 16/32 bit CPU which operates at 80 MHz. It has an integrated flash, integrated ram and a 8 Kbyte integrated EEprom.

- Interfaced with 2 can channels which operates on 125Kbit per second up to 1 mbit per second, and a CAN channel termination with connector pins that can be customized.
- A 5 volts sensor channel which operates at 100 mA.
- temperature, sensor supply, K15 input, and battery voltage internal monitoring.
- **Inputs:**
 1. 4 channels which can be configured as digital timer input which is able to operate from 0.1 HZ up to 10 KHZ, performance level d if used in pairs, could be also configured as an analogue input capable of operating at 0 to 32 volts, could also be configured as a rotatory encoder, or as a digital input Pull up/ pull down which can be configured if desired.
 2. 4 channels which are configured as analogue inputs which are software configurable whose input functions are performance level d if used in pairs, operating at an input voltage of 0-5 volts up to a maximum of 10 volts and an input current of 0 to 25 mA with an input resistance of 0 up to 65 K Ω s.
 3. 2 analogue input channels which are software configurable whose input functions are performance level d if used in pairs, digital input Pull up/ pull down which can be configured.
- **Outputs:**
 1. 6 channels which can be configured as a pulse width modulator outputs, or can also be configured as digital outputs, can go up till 3 amperes, high side switch, detection of overload and open load, performance level d capability. Could be also configured as a digital timer input capable of operating at 10 Hz up to 10 KHz with integrated pull up. Could also be configured as analogue inputs operating at a range of 0 up to 32 volts with pull up.
 2. 2 channels which can be configured as a pulse width modulator outputs, or can also be configured as digital outputs, can go up till 3 amperes, high side switch with detection of overload and open load, performance level d capability. Could be also configured as a digital timer input capable of operating at 10 Hz up to 10 KHz with integrated pull up. Could also be configured as analogue inputs operating at a range of 0 up to 32 volts with pull up.
 3. 2 digital output channels, , can go up till 3 amperes, low side switch For high-side pulse width modulator outputs, it's used as a redundant switch-off path.
 4. 6 channels configurable as PVG, can also be configured as voltage out
- Short-circuit protection is provided for all I/O ports and interfaces, which can be set up via software.
- analog inputs use 10-bit resolution.
- PL d inputs of the same type must be used in parallel to provide redundancy in case of a failure for safety functions.
- High side outputs have dedicated power supply pins.

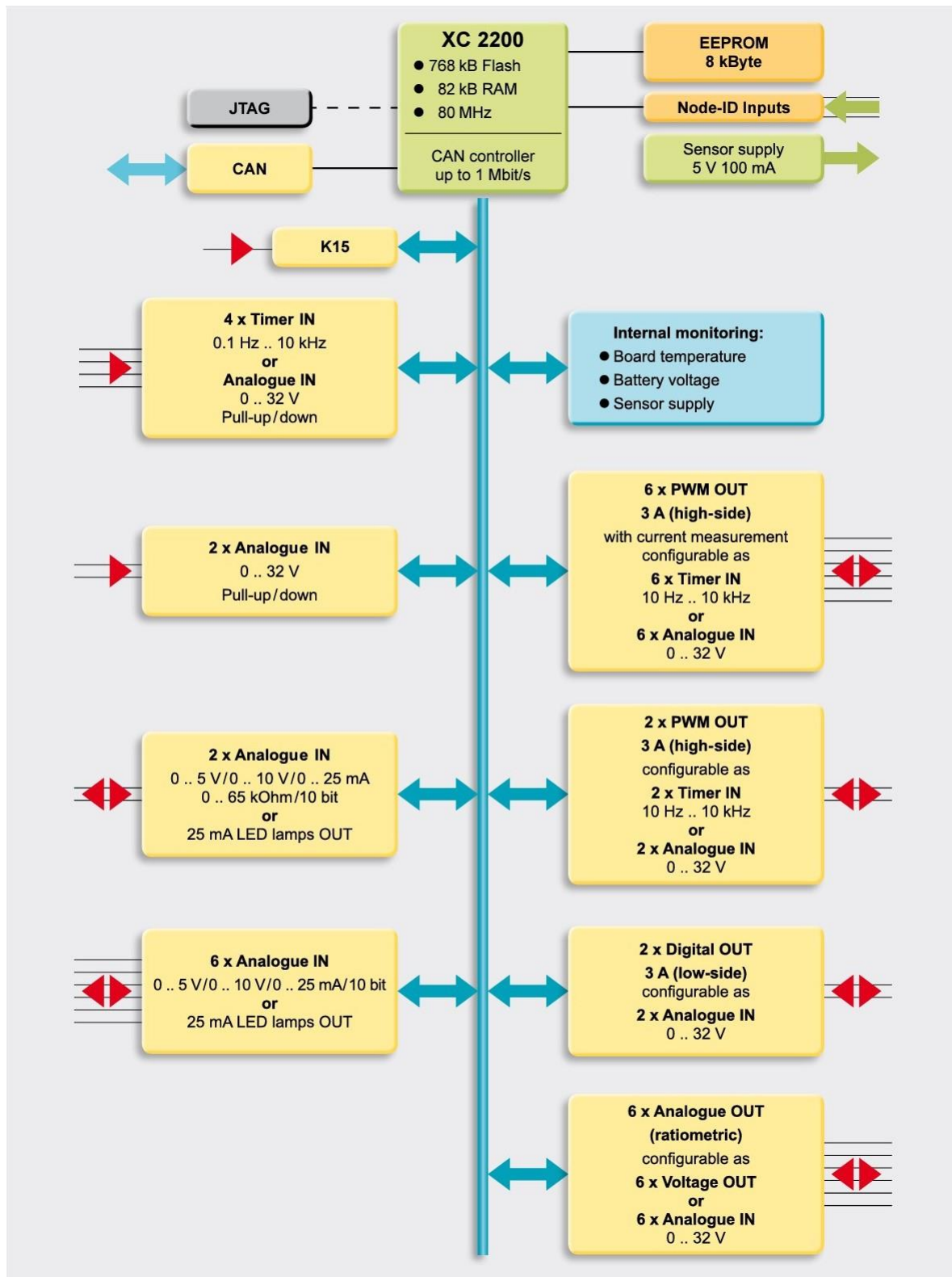


Figure 3. 3: 30-H Block Diagram [22]

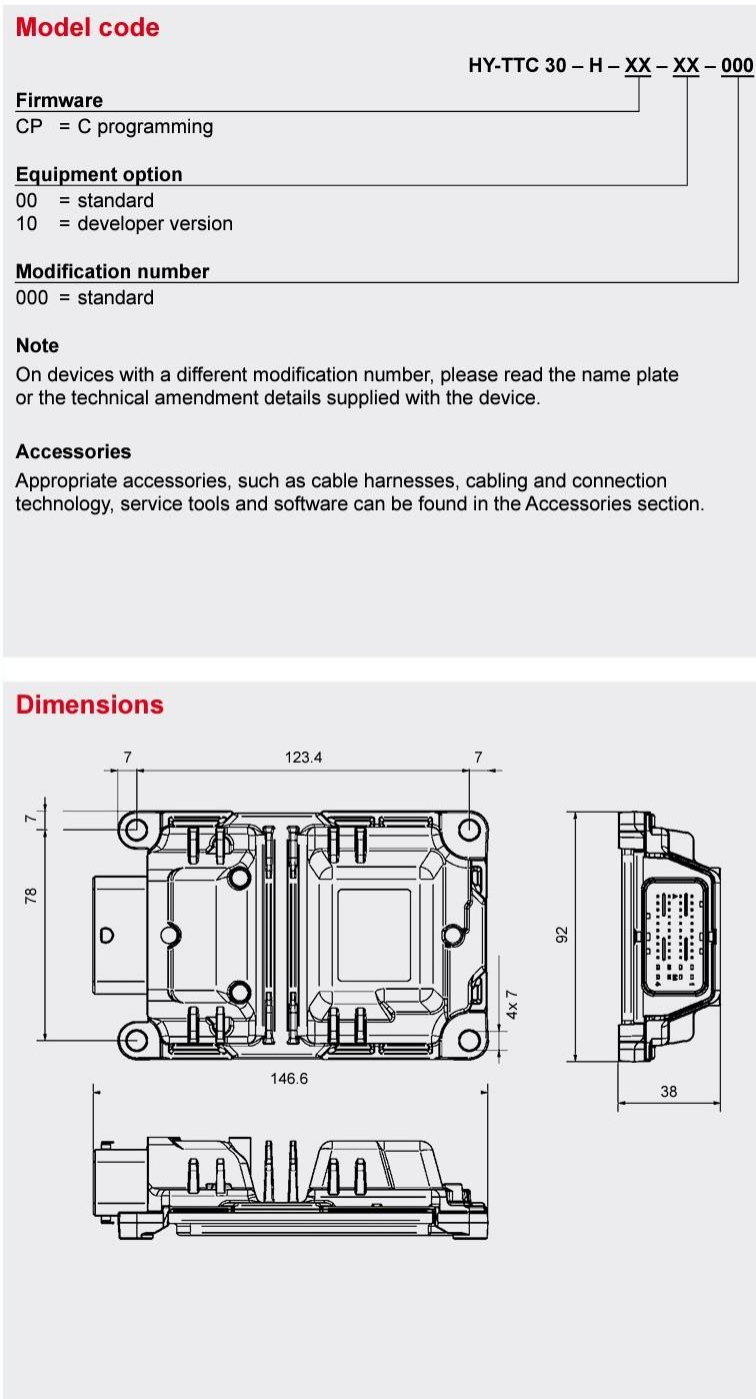


Figure 3. 4: 30-H Model code and Dimensions [22]

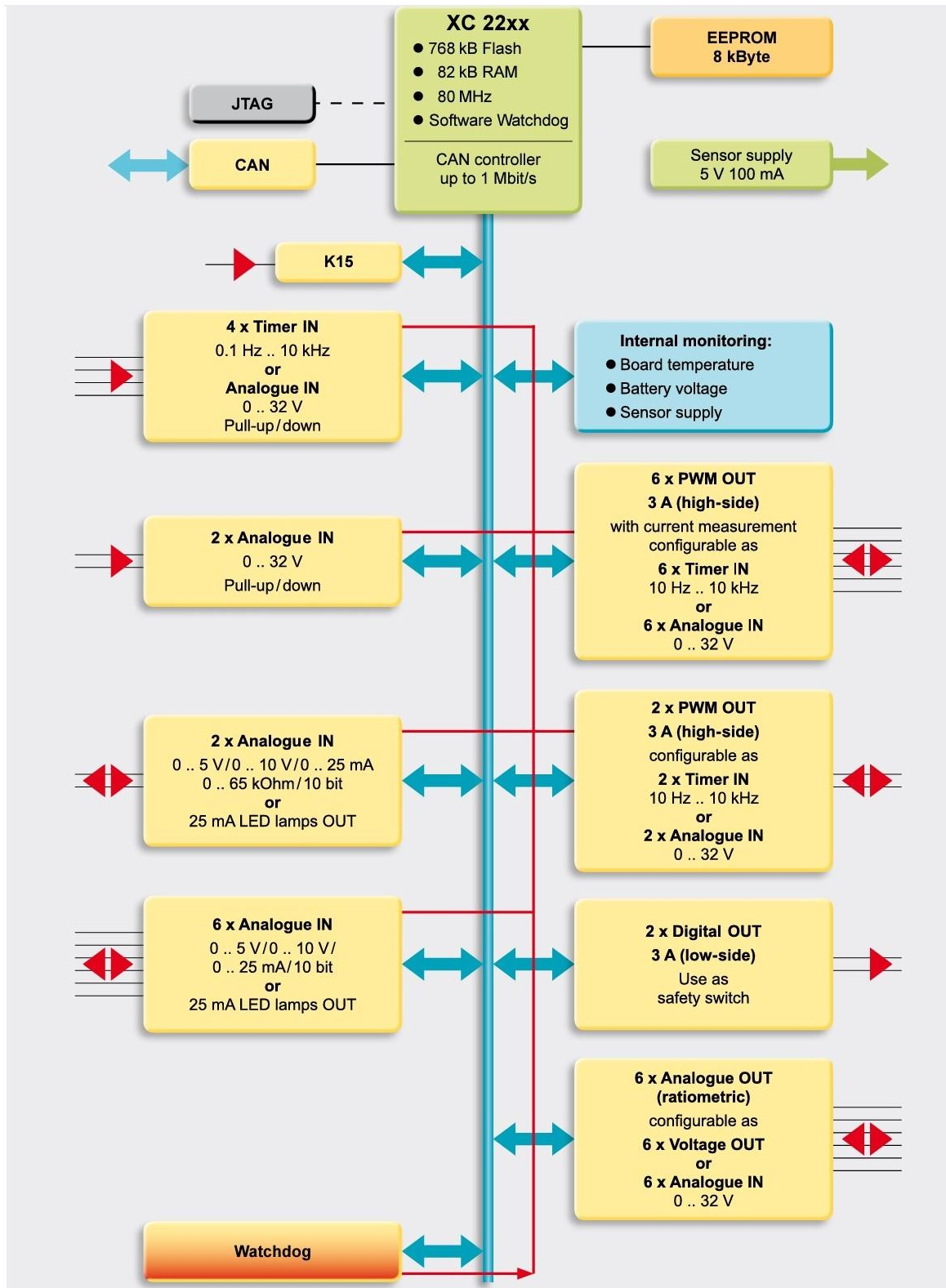


Figure 3. 5: 30S-H Block Diagram [22]

Model code

HY-TTC 30S – H – XX – XX – Pc – 000

Firmware

CP = C programming

Equipment option

00 = standard

10 = developer version

Functional safety

Pc = requirements for PL c

Modification number

000 = standard

Note

On devices with a different modification number, please read the name plate or the technical amendment details supplied with the device.

Accessories

Appropriate accessories, such as cable harnesses, cabling and connection technology, service tools and software can be found in the Accessories section.

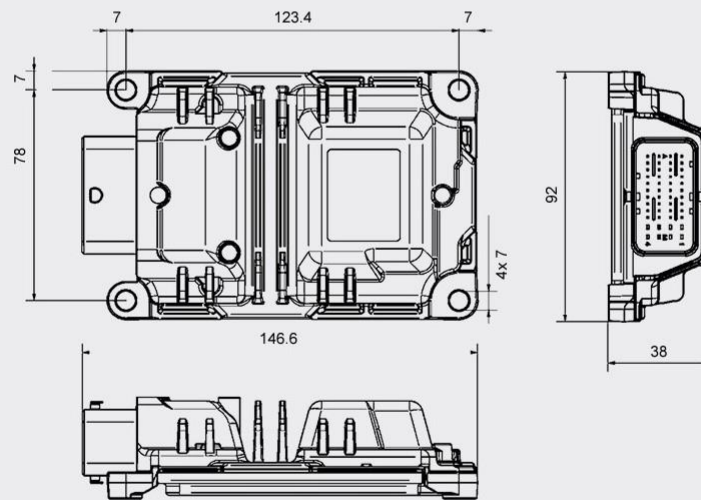
Dimensions

Figure 3. 6: 30S-H Model Code and Dimensions [22]

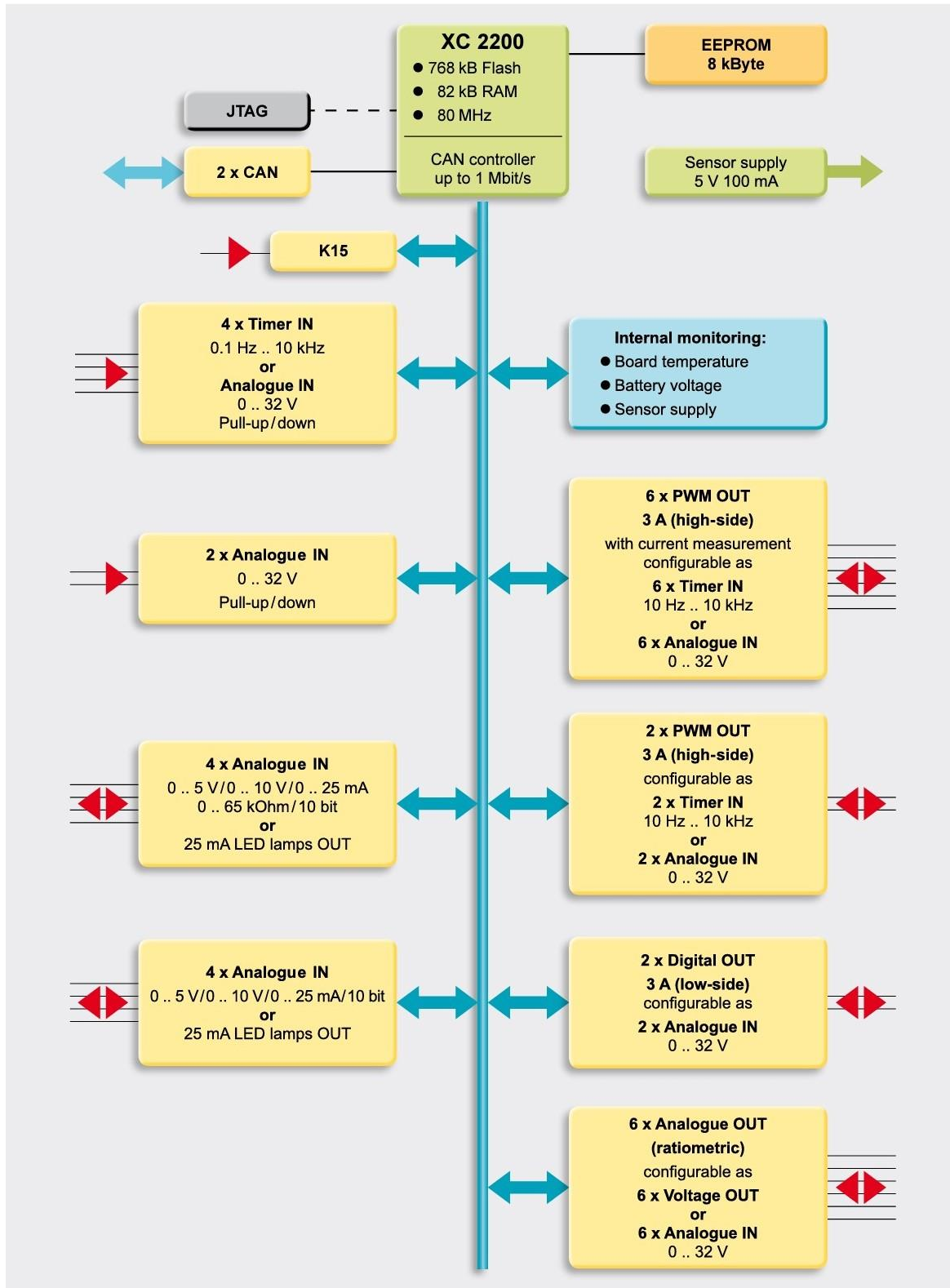


Figure 3. 7: 32 Block Diagram [22]

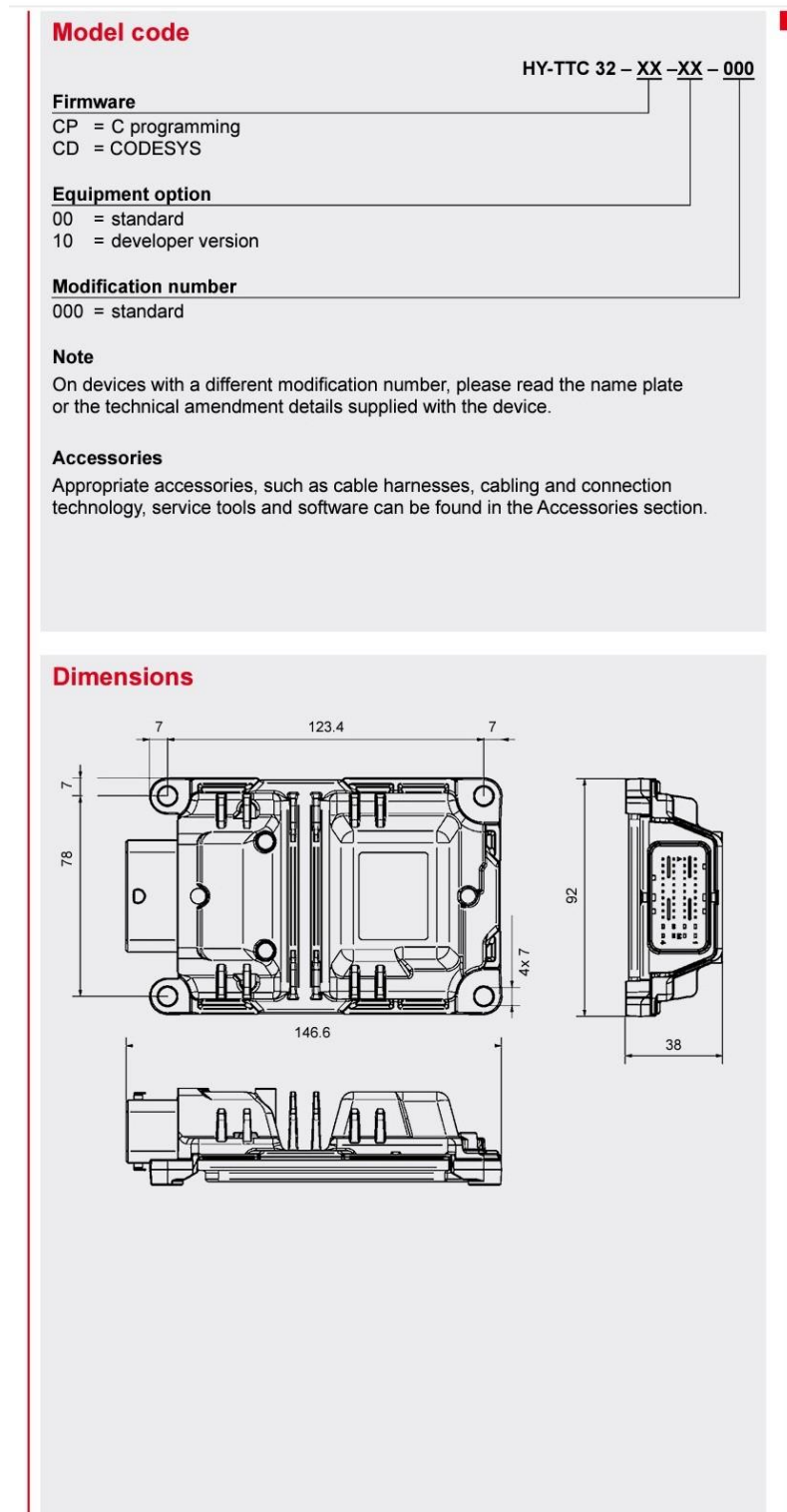


Figure 3. 8: 32 Model Code and Dimensions [22]

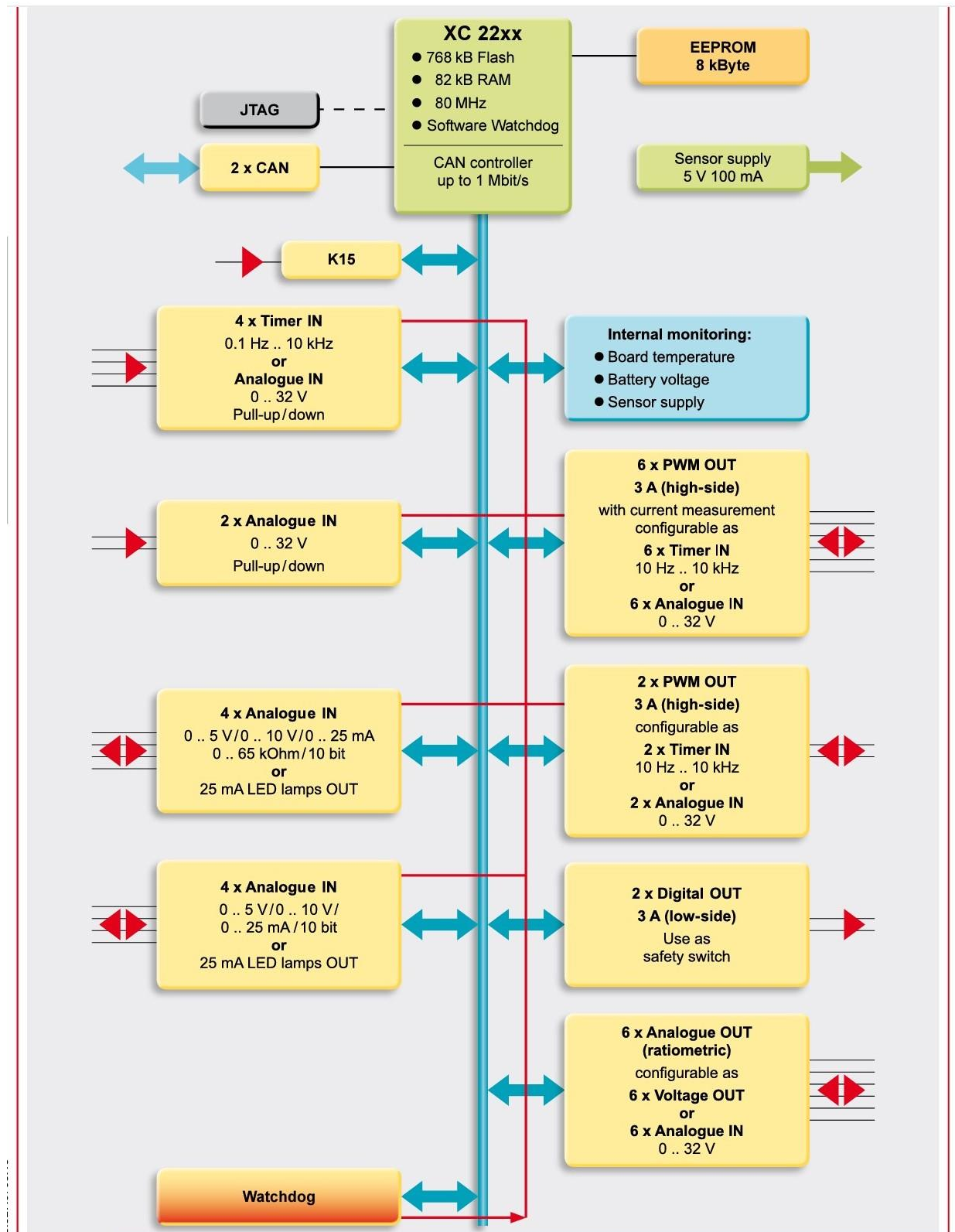


Figure 3. 9: 32S Block Diagram [22]

Model code

HY-TTC 32S – XX – XX – Pc – 000

Firmware

CP = C programming

Equipment option

00 = standard

10 = developer version

Functional safety

Pc = requirements for PL c

Modification number

000 = standard

Note

On devices with a different modification number, please read the name plate or the technical amendment details supplied with the device.

Accessories

Appropriate accessories, such as cable harnesses, cabling and connection technology, service tools and software can be found in the Accessories section.

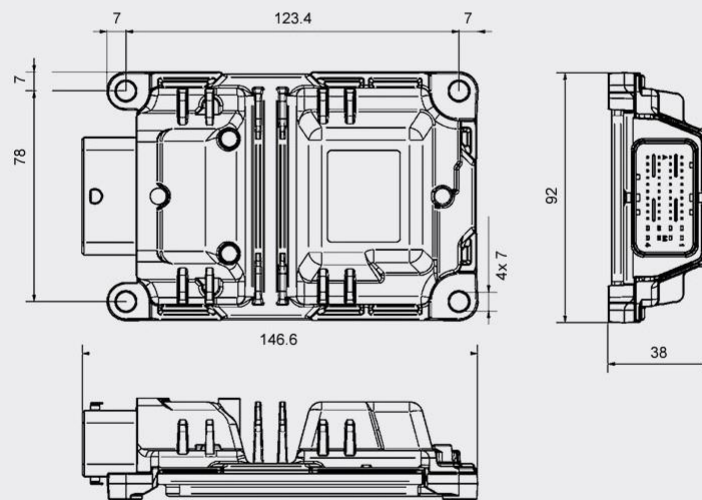
Dimensions

Figure 3. 10: 32S Model Code and Dimensions [22]

3.3 Why the HY_TTC32S for EVERGRIN

The HY-TTC32S board was chosen because it is easily integrated, it is very powerful yet low cost ECU. The HY-TTC32S is simply one of the best, if not the best compact electronic control units that can be used for automobile and heavy vehicle applications nowadays, thanks to its channels for pulse width modulation output with current measurement, it was possible to manage the pedals of the vehicle for evergrin, actually it was more than enough. Each and every pin of the board is configurable and flexible, it has been designed this way on purpose. All output pins are configurable as input pins if desired. It supports a CANopen, it is ASIL C, and with its powerful Infineon CPU it is a proper choice for automotive applications. Its Automotive style housing is suited for rough operating conditions. And last but not least is its special features, specially the fact that it has 2 CAN buses.

Here is a brief of the 32S special features and benefits:

- 30 I/Os.
- 10 analogue inputs.
- 4 timer inputs.
- 9 pulse width modulator outputs.
- 2 digital outputs
- 6 ratiometric voltage outputs.
- Robust.
- Water proof.
- 2 CAN buses.
- PLC certified according to EN ISO 13849
- Small form factor
- CANopen Safety compliant

Chapter 4

4 Tasks and Drivers Implementation

This chapter is dedicated to the implementation, so the programming and development of the Tasks (Architecture, Algorithm, and Code) of the vehicle management unit performed in C programming language.

4.1 Main Function

4.1.1 Main Function Code Implementation

```

/*****
 *
 *  HY-TTC 30
 *  UART example using printf-like method
 *
 *  Implements function UART_Printf that outputs format strings to RS2
32
 *  with printf-like syntax.
 *
 *  NOTICE: the return-
values of functions may not be sufficiently checked
 *  in the examples. an actual application shall evaluate the return
 *  values for proper error handling
 *
 *****/

#include "DIAG_Functions.h"
#include "DIAG_Functions.h"

```

```
#include "DIAG_Constants.h"
#include "IO_PWM.h"
#include "utility.h"
#include "IO_RTC.h"
#include "IO_Driver.h"
#include "IO_WDTimer.h"
#include "IO_UART.h"
#include "Apdb.h"
#include "ApdbCfg.h"
#include "IO_ADC.h"
#include "IO_PWM.h"
#include "IO_DIO.h"

#include "task_1_gestione_pedali.h"
#include "task_2_gestione_chiave_batteria.h"
#include "task_3_gestione_inverter.h"
#include "task_4_gestione_pompe.h"
#include "task_5_gestione_carica.h"
#include "task_6_gestione_buzzer.h"
#include "task_7_gestione_interrupt.h"

/*
 * use these fields to define the software version in three levels
 * major.minor.patch. this version is accessible from the APDB and
 * can therefore be read using the ttc-downloader
 */

#define SW_VERSION_MAJOR 1 // 8-bit
#define SW_VERSION_MINOR 0 // 8-bit
#define SW_VERSION_PATCH 0 // 16-bit

/*
 * use these definitions (or replace content in appl_db) to define the
 * node-nr and baud-rate used by the device
 */

#define NODE_NR 1
#define CAN_BAUDRATE 500 // kbps
```

```

/*
 * use these definitions to be able to identify the application software
 * using the TTC-Downloader (vendor and application ID)
 */

#define APDB_MANUF_ID ((ubyte1)0x00)
#define APDB_APP_ID ((ubyte1)0x00)

/*
 * Duty Cycle
 */

#define duty_cycle_inv_management 0x8000

/*
 * application database
 * needed by TTC-Downloader
 */

LOCATE_APDB appl_db = {
    APDB_VERSION // ubyte4 versionAPDB
    ,
    {0} // BL_T_DATE flashDate
        // BL_T_DATE buildDate
    ,
    {(ubyte4)((((ubyte4)RTS_TTC_FLASH_DATE_YEAR) & 0x0FFF) << 0) |
        (((ubyte4)RTS_TTC_FLASH_DATE_MONTH) & 0x000F) << 12) |
        (((ubyte4)RTS_TTC_FLASH_DATE_DAY) & 0x001F) << 16) |
        (((ubyte4)RTS_TTC_FLASH_DATE_HOUR) & 0x001F) << 21) |
        (((ubyte4)RTS_TTC_FLASH_DATE_MINUTE) & 0x003F) << 26)}}
    ,
    0 // ubyte4 nodeType
    ,
    0 // ubyte4 startAddress
    ,
    0 // ubyte4 codeSize
    ,
    0 // ubyte4 legacyAppCRC

```

```

,
0 // ubyte4 appCRC
,
NODE_NR // ubyte1 nodeNr
,
0 // ubyte4 CRCInit
,
0 // ubyte4 flags
,
0 // ubyte4 hook1
,
0 // ubyte4 hook2
,
0 // ubyte4 hook3
,
APPL_START // ubyte4 mainAddress
,
{0, 1} // BL_T_CAN_ID canDownloadID
,
{0, 2} // BL_T_CAN_ID canUploadID
,
0 // ubyte4 legacyHeaderCRC
  // ubyte4 version; (8bit.8bit.16bit)
,
{((ubyte4)(((ubyte4)((ubyte1)(SW_VERSION_MAJOR & 0x00FF)) << 24) |
  ((ubyte4)((ubyte1)(SW_VERSION_MINOR & 0x00FF)) << 16) |
  ((ubyte4)((ubyte2)(SW_VERSION_PATCH & 0xFFFF)))))),
CAN_BAUDRATE // ubyte2 canBaudrate
,
0 // ubyte1 canChannel
,
0 // ubyte4 password
,
0 // ubyte4 magicSeed
,
{0, 0, 0, 0} // ubyte1 targetIPAddress[4]
,
{0, 0, 0, 0} // ubyte1 subnetmask[4]
,

```

```

    {0, 0, 0, 0} // ubyte1 dIMulticastIP[4]
    ,
    0 // ubyte4 debugKey;
    ,
    3 // timeout value for ABRD in calm mode. [seconds]
    ,
    APDB_MANUF_ID // ubyte1 manufacturerID
    ,
    APDB_APP_ID // ubyte1 application ID
    ,
    {0} // reserved for future use (should be 0)
    ,
    0 // ubyte4 headerCRC
};

#define DEBUG

static IO_ErrorType driver_init_rc = IO_E_BUSY; //task1
static IO_ErrorType adc_init_rc;

static IO_ErrorType pwm_init_00; // task1 // variable storing the return of the initialization function of pwm for acceleration pedal.
static IO_ErrorType pwm_init_01; // task1 //variable storing the return of the initialization of pwm for brakes pedal.
static IO_ErrorType pwm_init_02;
static IO_ErrorType pwm_init_03;
static IO_ErrorType pwm_init_04;
static IO_ErrorType pwm_init_05;

const ubyte4 watchdog_timeout = 300000;
const ubyte2 ctrl_fun_timeout = 300000;

/* task5 */
IO_ErrorType DI_00_init;
IO_ErrorType DI_01_init;
IO_ErrorType DO_00_init;
IO_ErrorType DO_01_init;

```

```
IO_ErrorType buzzer_activation_pin_initialization;
IO_ErrorType can_initialization;

//Task1

// Acceleration pedal

static ubyte2 pwm_duty_cycle_acce;
static ubyte4 pwm_duty_cycle_fb_val_acce_here; // variable to store the
returned measured pulse-
width of the PWM signal in the last round in us (the pressure on the p
edal i guess).

// Brakes pedal

static ubyte2 pwm_duty_cycle_brakes;
static ubyte4 pwm_duty_cycle_fb_val_brakes_here; // variable to store
the returned measured pulse-
width of the PWM signal in the last round in us (the pressure on the p
edal i guess).

IO_DRIVER_DI_LIMITS digital_input_limits = {0, 3000, 3000, 32000}; //d
igital input limits, needed for the digital input function initializat
ion.

//task3
static ubyte2 pwm_duty_cycle_inv;
static ubyte4 pwm_duty_cycle_fb_inv;

static ubyte2 duty_cycle_set;
static ubyte4 duty_cycle_rb;
static ubyte4 duty_cycle_fb;

duty_cycle_set = duty_cycle_inv_management;

pwm_duty_cycle_inv = 0;
pwm_current_fb_inv = 0;
```

```

pwm_duty_cycle_acce = 0;
pwm_duty_cycle_freno = 0;

//debouncer
static IO_ErrorType pwm_init_rc_deb;
static IO_ErrorType pwm_set_rc_duty_deb;

IO_DRIVER_DI_LIMITS limits = {0, 3000, 3000, 32000};

const ubyte4 watchdog_timeout = 300000;
const ubyte2 ctrl_fun_timeout = 300000;
void main(void)
{

    ubyte4 pwm_duty_cycle_fb_val_acce;    //task1
    ubyte4 pwm_duty_cycle_fb_val_brakes; //task1
    ubyte4 pwm_duty_fb_val_inv;

    ubyte4 timestamp;
    /* driver initialitaion goes here */
    /* driver initialitiation for watchdog */

    driver_init_rc = IO_Driver_Init(IO_DRIVER_MODE_DEFAULT, NULL);
    /* set watchdog timeout */
#ifdef DEBUG
    IO_WDTimer_Init(watchdog_timeout);
#endif
    /* driver RTC initialization */
    while (IO_E_OK != IO_RTC_Init())
    {
    }
    /* driver EEPROM initialitiation */
    while (IO_E_OK != IO_EEPROM_PreloadInit())
    {
    }

    /* initialize the UART with a baudrate of 115200 baud/s */
    (void)IO_UART_Init(IO_UART, 115200);

```



```

/*
 * driver ADC initialization
 */

// initialize the ADC measurement of the battery voltage, task2
(void)IO_ADC_ChannelInit(
    IO_ADC_UBAT, IO_ADC_ABSOLUTE, IO_ADC_RANGE_10V, NULL);

/*
 * PWM Driver Initializations
 */

IO_POWER_Set(IO_INT_POWERSTAGE_ENABLE, IO_POWER_ON); // activate the
power stages for the PWM output.

//Safety configuration for the PWM outputs.

IO_PWM_SAFETY_CONF acc_pwm_safety_conf = // IO_PWM_SAFETY_CONF is ty
pedefed as struct of type _io_pwm_safety_conf.
{
    IO_SAFETY_SWITCH_0, // IO pin safety switch.
    80,                  // margin_lower_lim in microseconds for t
he lower minimum pulse.
    100,                 // margin_upper_lim in microseconds for t
he upper minimum pulse pause.
    200                  // duty_cycle_tolerance.

};

IO_PWM_SAFETY_CONF brakes_pwm_safety_conf =
{
    IO_SAFETY_SWITCH_1, // IO pin safety switch.
    80,                  // margin_lower_lim in microseconds for t
he lower minimum pulse.
    100,                 // margin_upper_lim in microseconds for t
he upper minimum pulse pause.
    200                  // duty_cycle_tolerance.

```

```

};

// initialize PWM output as safety-critical -TASK1 (Acceleration)

pwm_init_00 = IO_PWM_Init(
    IO_PWM_PIN_ACCE,      // pwm channel for acceleration pedal.
    30,                   // frequency in [HZ].
    TRUE,                  // polarity : high time is variable.
    TRUE,                  // enable diagnostic margin
    3000,                  // user overload limit in [mA]
    &acc_pwm_safety_conf); // saftey configuration

// initialize PWM output as safety-critical -TASK1 (Brakes)

pwm_init_01 = IO_PWM_Init(
    IO_PWM_PIN_BRAKES,    // pwm channel for brakes pedal.
    30,                   // frequency in [HZ].
    TRUE,                  // polarity : high time is variable.
    TRUE,                  // enable diagnostic margin
    3000,                  // user overload limit in [mA]
    &brakes_pwm_safety_conf); // saftey configuration

// initialize PWM output as safety-critical -
// TASK3 inverter pedal read
pwm_init_02 = IO_PWM_Init(
    IO_PWM_02 // PWM channel
    ,
    200 // frequency in [Hz]
    ,
    TRUE // polarity : high time is variable
    ,
    TRUE // enable diagnostic margin
    ,
    3000 // user overload limit in [mA]);
    ,
    &pwm_safety_cfg);

// initialize PWM output as safety-critical -
// TASK3 inverter management

```

```
pwm_init_03 = IO_PWM_Init(IO_PWM_03, 100, TRUE, TRUE, 2000, &_io_pwm_
safety_conf);
```

```
// initialize PWM output as safety-critical -TASK34 knob read
```

```
pwm_init_04 = IO_PWM_Init(
    IO_PWM_04 // PWM channel
    ,
    200 // frequency in [Hz]
    ,
    TRUE // polarity : high time is variable
    ,
    TRUE // enable diagnostic margin
    ,
    3000 // user overload limit in [mA]);
    &pwm_safety_cfg);
```

```
// initialize PWM output as safety-critical -
TASK3 inverter management
```

```
pwm_init_05 = IO_PWM_Init(IO_PWM_05, 100, TRUE, TRUE, 2000, &_io_pwm_
safety_conf);
```

```
/*
    * PWM DUTY CYCLE
    */
```

```
//TASK1
```

```
pwm_set_duty_acc = IO_PWM_SetDuty( //Acceleration
    IO_PWM_PIN_ACCE,
    pwm_duty_cycle_acce,
    &pwm_duty_cycle_fb_val_acce);
if (pwm_duty_cycle_fb_val_acce != 0) // if the value is zero the meas
urement is not yet finished
{
    pwm_duty_cycle_fb_val_acce_here = pwm_duty_cycle_fb_val_acce;
}
```

```
pwm_set_duty_brakes = IO_PWM_SetDuty( //Brakes
    IO_PWM_PIN BRAKES,
```

```

    pwm_duty_cycle_brakes,
    &pwm_duty_cycle_fb_val_brakes);
    if (pwm_duty_cycle_fb_val_brakes != 0) // if the value is zero the me
asurement is not yet finished
    {
        pwm_duty_cycle_fb_val_brakes_here = pwm_duty_cycle_fb_val_acce;
    }

//TASK3

pwm_set_rc_inv = IO_PWM_SetDuty(
    IO_PWM_PIN_INV, pwm_duty_cycle_inv, &pwm_duty_fb_val_inv);

// if the value is zero the measurement is not yet finished
if (pwm_duty_fb_val_inv != 0)
{
    pwm_duty_cycle_fb_inv = pwm_duty_fb_val_inv;
}

pwm_set_rc_inv_man = IO_PWM_SetDuty(
    IO_PWM_PIN_INV_Man, duty_cycle_set, &duty_cycle_rb);

/*
 * DEBOUNCER INITIALIZATIONS
 */

pwm_init_rc_deb = IO_PWM_Init(
    IO_PWM_PIN_INPUT // PWM channel
    ,
    200 // frequency in [Hz]
    ,
    TRUE // polarity : high time is variable
    ,
    TRUE // enable diagnostic margin
    ,
    3000 // user overload limit in [mA]);
    ,
    &pwm_safety_cfg);

```

```

pwm_set_rc_duty_deb = IO_PWM_SetDuty(
    IO_PWM_PIN_INPUT_debouncer, pwm_duty_cycle, &pwm_duty_cycle_fb);

/*
 * DIO initializations
 */

IO_DI_Init(IO_DI_00, IO_DI_PU, &digital_input_limits); // Digital input initialization for the magnetic brakes

IO_DO_Init(IO_DO_01, 2500); //task2
IO_DI_Init(IO_DI_00, IO_DI_PU, &limits); //task2
IO_DI_Init(IO_DI_01, IO_DI_PU, &limits); //task2

IO_DI_Init(IO_DI_02, IO_DI_PU, &limits); //task3
IO_DI_Init(IO_DI_02, 2500); //task3

IO_DO_Init(IO_DO_02, 2500); //task4

/* task 5 */

IO_DRIVER_DI_LIMITS limits = {0, 3000, 3000, 32000};
high_voltage_input_pin_init = IO_DI_Init(high_voltage_input_pin, IO_DI_PU, &limits); //High voltage input pin initialization
low_voltage_input_pin_init = IO_DI_Init(low_voltage_input_pin, IO_DI_PU, &limits); // low voltage input pin initialization
high_voltage_output_pin_init = IO_DO_Init(high_voltage_output_pin, 2500); // High voltage output pin initialization
low_voltage_output_pin_init = IO_DO_Init(low_voltage_output_pin, 2500); //low voltage output pin initialization

buzzer_activation_pin_initialization = IO_DO_Init(buzzer_activation_pin, 2500); //task6 DO initialization.
can_initialization = IO_CAN_Init(can_channel_for_buzzer, CAN_BAUDRATE); //task6_can_initialization

```

```
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r Main task !\n\r");
#endif

/* activate interrupt function to check */
/* everything is ok */
IO_RTC_PeriodicInit(ctrl_fun_timeout, task_7_gestione_interrupt);

while (1)
{
    UART_Printf(IO_UART, "\n\r sono nel main !\n\r");
    IO_RTC_StartTime(&timestamp);

    task_1_gestione_pedali();

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 1 terminating !\n\r");
#endif

    task_2_gestione_chiave_batteria();

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 2 terminating !\n\r");
#endif

    task_3_gestione_inverter();

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 3 terminating !\n\r");
#endif

    task_4_gestione_pompe();

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 4 terminating !\n\r");
#endif

    task_5_gestione_carica();
```

```

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 5 terminating !\n\r");
#endif

    task_6_gestione_buzzer();

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r task 6 terminating !\n\r");
#endif

    while (IO_RTC_GetTimeUS(timestamp) < 50000)
        ;
    }
}

```

4.1.2 Functions and Drivers Initialized in the Main Function

Pulse Width Modulation (PWM) driver

Driver for digital inputs and outputs (DIO)

Analog to Digital Converter (ADC) driver

Pulse Width Modulation (PWM) driver

Real Time Clock (RTC) driver

Controller Area Network (CAN) driver

Universal Asynchronous Receiver Transmitter (UART) driver

EEPROM driver

EEPROM preload functions

Driver for ECU power functions

```

IO_Driver_Init           //watch dog driver initialization
IO_EEPROM_PreloadInit    // EEprom diver initialization
IO_UART_Init             // UART driver initialization
IO_ADC_Channellnit       //ADC driver initialization
IO_PWM_Init              // PWM driver initialization

```

4.2 Pedals Management Task

4.2.1 Task Architecture

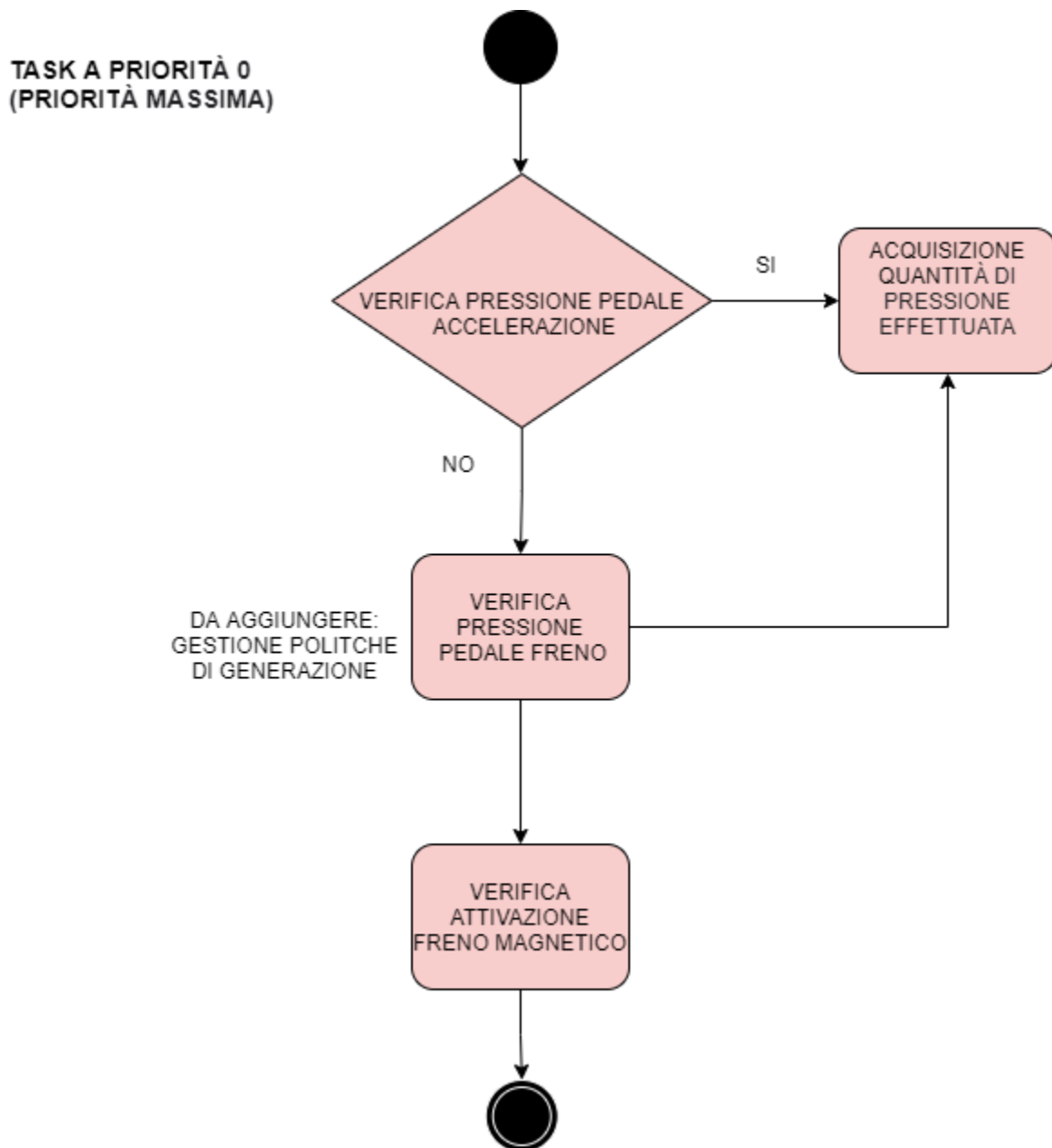


Figure 4. 1: pedals management task architecture flow chart

The purpose of the task is to manage the pedals (the acceleration pedal and the brakes pedal) and the magnetic brakes.

The task is a maximum priority. The architecture and algorithm to program and develop the task is as follows:

1. Check the acceleration pedal:
 - if there is a non-zero current value returned (current value returned $\neq 0$), this indicates that there is some pressure on the acceleration pedal then read the pressure on the pedal.
 - If there is no current returned (current value returned = 0) from the acceleration pedal, this indicates that there is no pressure on the acceleration pedal (the pedal is not pressed) then check the brakes pedal.
2. Check the brakes pedal:
 - if there is a non-zero current value returned (current value returned $\neq 0$), this indicates that there is some pressure on the brakes pedal then read the pressure on the pedal.
 - If there is no current returned (current value returned = 0) from the brakes pedal, this indicates that there is no pressure on the acceleration pedal (the pedal is not pressed).
3. check (verify) the magnetic brakes (hand brakes) using DIO driver.

The Pulse width modulator driver is utilized to read the state of the pedal.

The technique used to read the pressure on the pedals (acceleration pedal and brakes pedal):

- The pedals return an electrical signal which indicates the pressure on the pedal, if there is any, and we read it using **pulse width modulator**, as 6 of the 8 channels of our HY_TTC32S are with current measurement which is perfect because this way we can measure the current on the pedals which would indicate if there is any pressure on the pedal, and if there is, how much pressure exactly.

The task flow chart is illustrated in figure 4.1 above.

The implementation in terms of code and a detailed documentation of the implementation is mentioned below.

4.2.2 Pedals Management Code Implementation

4.2.2.1 Pedals Managment .c Code

```
/*
 * task_1_gestione_pedali.c
 *
 * Created on: 07/july/2021
 *
 */

#include "task_1_gestione_pedali.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_Driver.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK1_ID[1] = {1};
static IO_ErrorType pwm_get_rc_acce;
static IO_ErrorType pwm_get_rc_freno;

static ubyte2 pwm_current_fb_val_acce;
static bool pwm_fresh_acce;
static ubyte2 pwm_current_fb_acce_pedal_pressure_value;

static ubyte2 pwm_current_fb_val_freno;
static bool pwm_fresh_freno;
static ubyte2 pwm_current_fb_freno_pedal_pressure_value;

static IO_ErrorType di_get_rc;
static bool value_magnetic_brakes;

void task_1_gestione_pedali()
{
    driver_task_begin_rc = IO_Driver_TaskBegin();
```

```

IO_ErrorType eeeprom_error = 512;
ubyte4 task_1_timestamp;
(void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LENGTH, FALSE, TASK1_ID);
while (IO_EEPROM_PreloadStatus() != IO_E_OK)
{
    (void)IO_EEPROM_PreloadTask();
}
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeeprom write ok !\n\r");
#endif

/* VERIFICA PRESSIONE PEDALE ACCELERAZIONE */

do
{
    // read the current feedback value
    pwm_get_rc_acce = IO_PWM_GetCur(
        IO_PWM_PIN_ACCE, &pwm_current_fb_val_acce, &pwm_fresh_acce);

    // only update the current feedback value if fresh

    if ((pwm_get_rc_acce == IO_E_OK) && (pwm_fresh_acce))
    {
        pwm_current_fb_acce_pedal_pressure_value = pwm_current_fb_val_acce;
#ifdef DEBUG
            UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r",
                pwm_current_fb_acce_pedal_pressure_value);
#endif
        }
    } while (pwm_current_fb_acce_pedal_pressure_value != 0);

/* VERIFICA PRESSIONE PEDALE ACCELERAZIONE */

do
{
    // read the current feedback value
    pwm_get_rc_freno = IO_PWM_GetCur(
        IO_PWM_PIN BRAKES, &pwm_current_fb_val_freno, &pwm_fresh_freno);

```

```

// only update the current feedback value if fresh

if ((pwm_get_rc_freno == IO_E_OK) && (pwm_fresh_freno))
{
    pwm_current_fb_freno_pedal_pressure_value = pwm_current_fb_val_acce
;
#ifdef DEBUG
    UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r",
pwm_current_fb_acce_pedal_pressure_value);
#endif
}
} while (pwm_current_fb_freno_pedal_pressure_value != 0);

//VERIFICA ATTIVAZIONE FRENO MAGNETICO

di_rc_get = IO_DI_Get(IO_DI_PIN_MAGNETIC_BREAK, &value_magnetic_brake
s);

if (di_get_rc == IO_E_OK)
{
#ifdef DEBUG
    UART_Printf(IO_UART, "Magnetic Break value: %u mV\n\r", value_magnet
ic_brakes);
#endif
}

(void)IO_RTC_StartTime(&task_1_timestamp);
while (TRUE != Check_Task_End(task_1_timestamp, (ubyte4)TASK1_CYCLE_T
IME))
{
    IO_RTC_PeriodicDeInit();
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 1 terminating %u !\n\r", ta
sk_1_timestamp);
    /*serve watchdog */
#endif
    /*serve watchdog */
#endif
}

```

```

    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifndef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
    driver_task_end_rc = IO_Driver_TaskEnd();
}

```

4.2.2.2 Pedals Management .h Code

```

/*
 * task_1_gestione_pedali.h
 *
 * Created on: 07/july/2021
 *
 */

#ifndef TASK_1_GESTIONE_PEDALI_H_
#define TASK_1_GESTIONE_PEDALI_H_
#include "ptypes_xe167.h"
#define DEBUG
#define TASK1_CYCLE_TIME 100000u
void task_1_gestione_pedali(void);

#define IO_PWM_PIN_ACCE IO_PWM_00
#define IO_PWM_PIN_BRAKES IO_PWM_01

#define IO_DI_PIN_MAGNETIC_BREAK IO_DI_00

#endif /* TASK_1_GESTIONE_PEDALI_H_ */

```

4.2.4 Development and Implementation

4.2.4.1 Functions and Drivers Utilized

IO_Driver_TaskBegin

IO_Driver_TaskEnd

IO_RTC_StartTime

IO_EEPROM_PreloadWrite

IO_EEPROM_PreloadStatus

IO_EEPROM_PreloadTask

IO_WDTimer_Service

IO_PWM_Init (PWM driver function)

IO_PWM_SetDuty (PWM driver function)

IO_POWER_Set (PWM driver function)

IO_PWM_GetCur (PWM driver function)

IO_DI_Init (DIO driver function)

IO_DI_GET (DIO driver function)

EEPROM driver

Real Time Clock (RTC) driver

Window Watchdog driver

Digital inputs and outputs (DIO) driver

Pulse Width Modulation (PWM) driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.2.4.2 Development and Implementation Documentation

First things first we understood that we need to use the PWM (pulse width modulator) driver and so the PWM driver functions to manage the acceleration and brakes pedals, because we need to read the pressure on the pedals, so the current on the acceleration and brakes pedals and this is possible thanks to the fact that 6 of the 8 channels of the pulse width modulation of our HY-TTC32S have current measurement.

proceeding with the implementation, PWM driver was initialized and setup utilizing the proper PWM initialization functions:

- IO_POWER_Set to enable the PWM outputs, otherwise the outputs continue to be disabled.
- IO_PWM_SetDuty to set the duty cycle for the chosen PWM channels.
- IO_PWM_Init to initialize the PWM channels.

2 PWM channels were initialized, one for the acceleration pedal and one for the brakes pedal, the PWM channels chosen to manage the acceleration pedal and brakes pedal have current measurement IO_PWM_00 and IO_PWM_01 as the precise current measurement is configured for the channels IO_PWM_00 to IO_PWM_05 only.

Passing the proper parameters to the IO_PWM_Init functions:

1. The channel (pin) was defined as IO_PWM_PIN_ACCE for the acceleration pedal and IO_PWM_PIN_BRAKES for the brakes pedal, which was defined in the header file of the pedals management task for the ease of changing the pin if desired and for the ease of the readability of the code.
2. A proper frequency in the range of 15Hz to 1 KHz to be sent to the function, TRUE polarity, and TRUE for the diagnostic margin to enable it,
3. The overload limit.
4. The address of the safety configuration IO_PWM_SAFETY_CONF stored at acc_pwm_safety_conf for the acceleration pedal and brakes_pwm_safety_conf for the brakes pedal. IO_PWM_SAFETY_CONF is typedefed as a struct of type _io_pwm_safety_conf which is Safety configuration for the PWM outputs. It Stores all relevant safety configuration parameters for the PWM outputs. It is included in the PWM header. Its data fields are:
 - IO pin safety switch which is a Connected safety switch IO_SAFETY_SWITCH_0 or IO_SAFETY_SWITCH_1, other wise if neither safety switch is connected then it is IO_SAFETY_SWITCH_NONE.
 - margin_lower_lim in microseconds for the lower minimum pulse. Which is a Limit in microseconds for the lower minimum pulse. The limit can be modified between 50us to 100us.
 - margin_upper_lim in microseconds for the upper minimum pulse. Which is a Limit in microseconds for the lower minimum pulse. The limit can be modified between 50us to 100us.
 - duty_cycle_tolerance to set the duty cycle, the tolerance is measured in microseconds. This limit can be changed between 100 and 200 milliseconds.

The return (output) of the the IO_PWM_Init is stored at the pwm_init_00 for the acceleration pedal and pwm_init_01 for the brakes pedal which are declared as static variables of type IO_ErrorType.

Proceeding with the implementation the duty cycle for the PWM was set utilizing the IO_PWM_SetDuty function for the acceleration and brakes pedals.

sending the proper parameters to the IO_PWM_SetDuty:

1. The PWM channel (pin), defined as IO_PWM_PIN_ACCE for the acceleration pedal and IO_PWM_PIN_BRAKES for the brakes pedal, which is the same pin used for the functions IO_PWM_Init which was also defined in the header file of the task for the ease of changing the pin if desired and for the ease of the readability of the code.
2. The proper duty cycle value pwm_duty_cycle_acce which would range from 0 to 65535 stored at pwm_duty_cycle_acce for the acceleration pedal and pwm_duty_cycle_brakes for the brakes pedal which are declared as static ubyte2 (2 bytes unsigned static).
3. The address of the variable holding the duty cycle feed back which is pwm_duty_cycle_fb_val_acce for the acceleration pedal and pwm_duty_cycle_fb_val_brakes for the brakes pedal which are declared as ubyte4 (4 bytes unsigned). The duty cycle feed back has to be always checked (verified) and the value is to be updated only if the result from the verification is different from zero, because if the value is zero it means that the duty cycle measurement is not finished yet. After checking (verifying) that they hold a value different from zero, the value is copied to the declared static ubyte4 (static 4 byte unsigned integer) pwm_duty_cycle_fb_val_acce_here for the acceleration pedal and the declared static ubyte4 pwm_duty_cycle_fb_val_brakes_here for the brakes pedal which are variables to store the returned measured pulse-width of the PWM signal in the last round in us, the pressure on the pedal .

The return (output) of the IO_PWM_SetDuty is stored at the pwm_set_duty_acc for the acceleration pedal and pwm_set_duty_brakes for the brakes pedal which are declared as static variables of type IO_ErrorType.

The last function implemented for the initialization of the PWM driver is the IO_POWER_Set to activate the power stages for the PWM output.

Sending the IO_POWER_Set the proper parameters:

1. The pin for which the mode shall be set, the chosen one is the IO_INT_POWERSTAGE_ENABLE which is an Internal Pin for enabling power stages.
2. The IO_POWER_ON to set the mode, which is defined 1 to switch on the power stages – activate the respective power function.

After making sure that the PWM driver is properly initialized with the proper functions and parameters, the core implementation of the pedals was proceeded by implementing acceleration pedal.

Acceleration Pedal Implementation:

The first step done was implementing the IO_PWM_GetCur function which is the function responsible for returning the measured current of a certain channel, which is the goal of the task, since the goal is measuring the pressure on the pedals, so the current on the pedals.

The proper parameters were sent to the IO_PWM_GetCur:

1. The PWM channel (pin), defined as IO_PWM_PIN_ACCE for the acceleration pedal , which is the same pin used for the functions IO_PWM_Init which was also defined in the header file of the task for the ease of changing the pin if desired and for the ease of the readability and reusability of the code.
2. The address of the variable to hold (store) and return the measured current range which would be in the range of 0mA up to 7575mA, the variable is named pwm_current_fb_val_acce and it is declared as a static unsigned 2 bytes integer.
3. The address of the variable to store a TRUE or a FALSE based on if there is or there isn't new values since the last call, so basically this is the variable to check to know if there a new current measured value or there isn't, by there isn't I mean there is no new value which means that the variable holding the measured current value (pwm_current_fb_val_acce in our case, and in the case of the acceleration pedal, but the same concept holds for the brakes pedal which should be checked later) doesn't have a new value, it has a value of a previous measurement and in this case I will not update the current feedback value. The variable is named pwm_fresh_acce and is declared as a static bool as it holds a true or a false.

The return (output) of the IO_PWM_GetCur function for the acceleration pedal is returned and stored in the pwm_get_rc_acce variable which is declared as as a static variable of type io_errortype. After implementing the IO_PWM_GetCur function, I proceeded by checking 2 conditions, I check if the pwm_get_rc_acce returns IO_E_OK and that pwm_fresh_acce returns ture:

1. pwm_get_rc_acce:

If pwm_get_rc_acce is IO_E_OK means that the io_pwm_getcur returned IO_E_OK which indicates that everything is fine, more precisely it means:

- The PWM channel is correctly configured.
- The channel ID is the there (exists).
- There were no ADC errors.
- Current measurement accuracy hasn't been reduced.
- The specified channel is a PWM channel and it has a current measurement.
- No null pointer has been sent to the function.

2. pwm_fresh_acce:

I check if the pwm_fresh_acce is true which would indicate that there is a new value available in the pwm_current_fb_val_acce, other wise in the case that the pwm_fresh_acce is false it indicates that there isn't a new value available in the pwm_current_fb_val_acce which means the current value available in the pwm_current_fb_val_acce is a value of a previous measurement and in that case there is no point to update the current value.

Basically I update the current feedback value if and only if the pwm_get_rc_acce is IO_E_OK AND the pwm_fresh_acce is true. Other wise (if any of those 2 conditions fail) I don't update the feedback current value.

On the top of the get current function implementation and the checking of the `pwm_get_rc_acce` and the `pwm_fresh_acce` I need to check that the feed back current value is a value different from zero, Because if the feedback current value is equal to zero, this would indicate that there is zero current returned from the acceleration pedal, so there is no pressure on the acceleration pedal. And this is exactly when I go to check the brakes pedal according to the task architecture. And then I go implement the brakes pedal the same exact way I implemented the acceleration pedal.

Brakes Pedal Implementation:

Implemeting the `IO_PWM_GetCur` function.

The proper parameters were passed to the `IO_PWM_GetCur`:

1. The PWM channel (pin), defined as `IO_PWM_PIN BRAKES` for the acceleration pedal , which is the same pin used for the functions `IO_PWM_Init` .
2. The address of the variable to hold (store) and return the measured current range, the variable is named `pwm_current_fb_val_freno` and it is declared as a static unsigned 2 bytes integer.
3. The address of the variable to store a TRUE or a FALSE based on if there is or there isn't new values since the last call, so basically this is the variable to check to know if there is a new current measured value or there isn't, by there isn't I mean there is no new value which means that the variable holding the measured current value (`pwm_current_fb_val_freno` in our case, and in the case of the brakes pedal, in the case that the brakes pedal) doesn't have a new value, it has a value of a previous measurement and in this case I will not update the current feedback value. The variable is named `pwm_fresh_freno` and is declared as a static bool as it holds a true or a false.

The return (output) of the `IO_PWM_GetCur` function of the brakes pedal is returned and stored in the `pwm_get_rc_freno` variable which is declared as as a static variable of type `IO_ERRORTYPE`. After implementing the `IO_PWM_GetCur` function, I proceeded by checking 2 conditions, I check if the `pwm_get_rc_freno` is `IO_E_OK` and that `pwm_fresh_freno` is ture.:

1. `pwm_get_rc_freno`:

If `pwm_get_rc_freno` is `IO_E_OK` means that the `IO_PWM_GetCur` returned `IO_E_OK` which indicates that everything is fine, more precisely it means:

- The PWM channel is correctly configured.
- The channel ID is the there (exists).
- There were no ADC errors.
- Current measurement accuracy hasn't been reduced.
- The specified channel is a PWM channel and it has a current measurement.
- No null pointer has been sent to the function.

2. *pwm_fresh_freno:*

I check if the `pwm_fresh_freno` is true which would indicate that there is a new value available in the `pwm_current_fb_val_freno`, other wise in the case that the `pwm_fresh_freno` is false it indicates that there isn't a new value available in the `pwm_current_fb_val_freno` which means the current value available in the `pwm_current_fb_val_freno` is a value of a previous measurement and in that case there is no point to update the current value.

Basically I update the current feedback value if and only if the `pwm_get_rc_freno` is `IO_E_OK` AND the `pwm_fresh_freno` is true. Other wise (if any of those 2 conditions fail) I don't update the feedback current value.

On the top of the get current function implementation and the checking of the `pwm_get_rc_freno` and the `pwm_fresh_freno` I need to check that the feed back current value is a value different from zero. Because if the feedback current value is equal to zero, this would indicate that there is zero current returned from the brakes pedal, so there is no pressure on the brakes pedal.

Magnetic Brakes Implementation:

After verifying successfully the acceleration and pedal brakes, this is the point where the magnetic brakes are checked (verified), which was implemented through the DIO driver using the digital input since I need to check if the user has pressed (used) the magnetic brakes. The function implemented is the `IO_DI_Get` which is a IO Driver functions for Digital Input/Output. To be able to successfully implement the `IO_DI_Get` function I needed to initialize the DIO driver, which was performed by implementing the proper initialization function to setup the digital inputs, the `IO_DI_Init` which was implemented in the project main function. The implementation of the `IO_DI_Init` goes as follows:

The proper parameters were passed to the `IO_DI_Init` function:

1. The digital input channel (pin), the channel (pin) chosen is the `IO_DI_00` which was then defined in the header file by the name `IO_DI_PIN_MAGNETIC_BREAK` for the ease of changing the pin later if desired and for the ease of readability and reusability of the code.
2. The `pupd` which is the Pull-up/down configuration which pulls the pin up configuring the pull-up resistor or down configuring the pull-down resistor, `IO_DI_PU` was passed to the function so the pin is pulled up.
3. The limits which is the voltage limits for low/high-levels. So the address of a variable called `digital_input_limits` is passed which is a struct of type `IO_DRIVER_DI_LIMITS` which is a typedefed struct of type `_io_driver_di_limits` which Contains the thresholds for valid low- and high-levels for digital inputs.

After successfully initializing and setting up the digital input of the DIO driver by successfully implementing the `IO_DI_Init` function. This is the point where the `IO_DI_Get` function was implemented which is a DIO driver function used for Returning the state of the Digital Input.

The IO_DI_Get function implementation was performed by passing the proper parameters:

1. The DI channel (pin)], the IO_DI_00 which is the same channel (pin) used for the initialization function of the DIO driver (IO_DI_Init) defined by the name IO_DI_PIN_MAGNETIC_BREAK.
2. The value (digital input value) which TRUE for high level and FALSE for low level and it is passed as a pointer, so the address of the variable to return the pin value was passed which was named value_magnetic_brakes and it is declared as a static bool since it returns the state of the pin, either TRUE or false.

The output (return) of the IO_DI_Get function is of type IO_ErrorType and it is returned in the variable with the name di_rc_get.

After the implementation of the IO_DI_Get, The return of the function is checked using an if condition by checking if the di_rc_get returns IO_E_OK. If it does return IO_E_OK, it means the function was implemented successfully and then the value of the magnetic brake pin is printed using UART, so the value in the variable value_magnetic_brakes is printed, which is the variable with which the address was passed to return the magnetic brakes value.

If the IO_DI_Get function doesn't return IO_E_OK, so if the if condition doesn't hold true the pin value is not printed because the fact that the IO_DI_Get function doesn't return IO_E_OK indicates that an error occurred.

The IO_DI_Get not returning IO_E_OK would indicate one of the following:

- The channel id does not exist.
- The specified channel is not a digital input.
- null pointer passed as argument.
- the specified channel is not configured.
- ADC measurement error.
- Detected short to ground.
- Detected short to battery.
- Detected open-load situation.
- voltage measured not in the range of the configured voltage limits.
- Detected open-load or short to GND.

4.3 power Socket Verification Task

4.3.1 Task Architecture

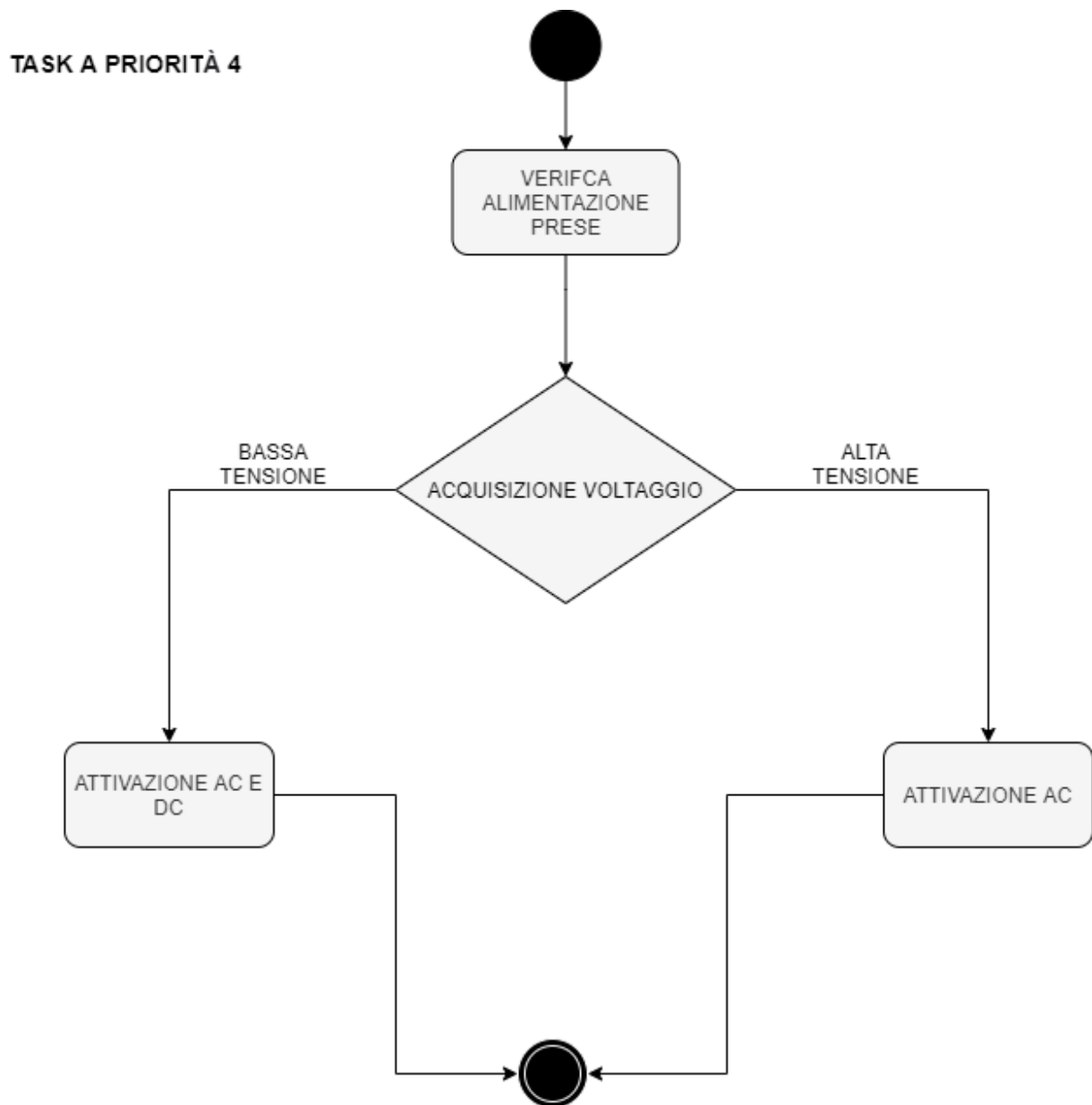


Figure 4. 2: Power socket verification task architecture flow chart

The main goal of the power socket verification task is to check (verify) the power socket for the purpose of the activation of either high voltage output, or both high and low voltage outputs. So basically the task can be managed from a high level of abstraction as a voltage acquisition implementation and then based on that voltage acquisition is to be implemented one of the following possibilities:

1. High voltage output activation.
2. Both High and low voltage outputs activation.

Thinking of a way to implement the voltage acquisition we thought of the possibility of checking (verifying) the power socket (implement the voltage acquisition) using the ADC (analogue to digital convertor) driver and functions so utilizing 2 of the analogue to digital convertor channels present, which is probably possible, but we came to the conclusion that the best way to perform this task is using the digital input functionalities as we need to read the pins and based on the return of the pins reading one of the possibilities mentioned above is to be implemented.

In simple words:

- 2 digital input pins to be configured, then read, check (verify) the pins. one pin is a high voltage input pin (is supplied with high voltage) and the other is a low voltage input pin (supplied with low voltage).
- 2 digital output pins to be configured, one indicates high voltage and the other indicates low voltage.

Based on the voltage acquisition of the power socket input pins the power socket outputs are activated.

The task architecture is implemented as follows:

1. Voltage acquisition of power socket input pins
2. Output voltage activation of power socket output pins
 - High voltage output activation. The high voltage output is activated if only the input high voltage pin is high, in other words in the case that the high voltage input pin is checked, and the return is high and the low voltage input pin is checked and the return is low; the high voltage output pin is to be activated.
 - Both High and low voltage outputs activation. Both High and low voltage outputs are activated if both input digital pins are high, if the high voltage input pin and the low voltage input pin are both returned to be high, in other words only in the case that both the high voltage input pin and the low voltage input pin are both checked and the return is high for both of them; then both the high voltage output pin and the low voltage input pin are to be activated.
 - Do nothing (don't activate any of the power socket output pins). This is in the case that the voltage acquisition returns that only the power socket low voltage input pin is activated.

4.3.2 Power Socket Verification Code Implementation

4.3.2.1 Power Socket Verification .c Code

```

/*
 * task_5_gestione_carica.c
 *
 * Created on: 12/jun/2021
 *
 */

#include "task_5_gestione_carica.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK5_ID[1] = {5};
bool high_volatge_input_pin_value;
bool low_volatge_input_pin_value;

ubyte2 const high_volatage_output_pin_voltage_feedback_in_mv;
ubyte2 const low_volatage_output_pin_voltage_feedback_in_mv;
void task_5_gestione_carica()
{
    ubyte4 task_5_timestamp;
    driver_task_begin_rc = IO_Driver_TaskBegin();
    (void)IO_RTC_StartTime(&task_5_timestamp);

    (void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LEN
GHT, FALSE, TASK5_ID);
    while (IO_EEPROM_PreloadStatus() != IO_E_OK)
    {
        (void)IO_EEPROM_PreloadTask();
    }
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeprom write ok !\n\r");

```

```

#endif

while (1)
{
    if (high_voltage_input_pin_init == IO_E_OK && low_voltage_input_pin_
init == IO_E_OK) //if both input pins(high voltage input pin and low v
oltage input pin) are initialized correctly
    {
        IO_DI_Get(high_voltage_input_pin, &high_volatge_input_pin_value); /
/ read high voltage input pin
        IO_DI_Get(low_voltage_input_pin, &low_volatge_input_pin_value);    /
/ read low voltage input pin

        if (high_volatge_input_pin_value == TRUE && low_volatge_input_pin_v
alue == FALSE)
        {
            IO_DO_Set(high_voltage_output_pin, TRUE, &high_volatage_output_pin
_voltage_feedback_in_mv); //activate the high voltage output pin
        }
        else if (high_volatge_input_pin_value == TRUE && low_volatge_input_
pin_value == TRUE)
        {
            /*activate both pins, high and low voltage out put pins*/
            IO_DO_Set(high_voltage_output_pin, TRUE, &high_volatage_output_pin
_voltage_feedback_in_mv);
            IO_DO_Set(low_voltage_output_pin, TRUE, &low_volatage_output_pin_v
oltage_feedback_in_mv);
        }
    }
}

while (TRUE != Check_Task_End(task_5_timestamp, (ubyte4)TASK5_CYCLE_T
IME))
{
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 5 terminating %u !\n\r", ta
sk_5_timestamp);
#endif
/*serve watchdog */

```



```

#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
driver_task_end_rc = IO_Driver_TaskEnd();
}

```

4.3.2.2 Power Socket Verification .h Code

```

/*
 * task_5_gestione_carica.h
 *
 * Created on: 6/jun/2021
 *
 */

#ifdef TASK_5_GESTIONE_CARICA
#define TASK_5_GESTIONE_CARICA
#define DEBUG
#define TASK5_CYCLE_TIME 100000u
void task_5_gestione_carica(void);

#define high_voltage_input_pin IO_DI_00
#define low_voltage_input_pin IO_DI_01
#define high_voltage_output_pin IO_DO_02
#define low_voltage_output_pin IO_DO_03
#endif

```

4.3.4 Development and Implementation

4.3.4.1 Functions and Drivers Utilized

IO_Driver_TaskBegin

IO_Driver_TaskEnd

IO_RTC_StartTime

IO_EEPROM_PreloadWrite

IO_EEPROM_PreloadStatus

IO_EEPROM_PreloadTask

IO_WDTimer_Service

IO_DI_Init (DIO driver function)

IO_DO_Init (DIO driver function)

IO_DI_Get (DIO driver function)

IO_DO_Set (DIO driver function)

EEPROM driver

Real Time Clock (RTC) driver

Window Watchdog driver

Digital inputs and outputs (DIO) driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.3.4.2 Development and Implementation Documentation

After studying and understanding the Power socket verification task architecture, The 1st step of the implementation was initializing and setting up the DIO driver, so initializing the digital inputs and digital outputs, in other words setting up the digital inputs and digital outputs to be able to implement and make use of the *IO_DI_Get* and *IO_DO_Set* functions. Having the task architecture and algorithm clear, I started to initialize (setup) 2 digital inputs (the power socket inputs), and 2 digital outputs (the power socket outputs), so the inputs: High voltage input pin and the low voltage input pin, and the outputs : High voltage output pin and the low voltage output pin.

For the high voltage input pin and the low voltage input pin, the function utilized for initialization and setup is the *IO_DI_Init* which is the DIO driver function used to Setup the Digital Inputs. The function was implemented in the main function of the project.

The proper parameters were passed to the IO_DI_Init function:

1. The di_channel (IO PIN), IO_DI_00 channel (pin) was used (chosen) for the power socket high voltage input pin and it was defined by the name high_voltage_input_pin in the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired. On the other hand IO_DI_01 channel (pin) was used (chosen) for the power socket low voltage input pin and it was define by the name low_voltage_input_pin in the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired.
2. The Pull-up/ pull-down configuration, which is the parameter responsible for configuring the pin with a pull up resistor (pull up the pin through the built in pull up resistor), or configure the pin with a oull down resistor (pull down the pin through the built in pull down resistor), IO_DI_PU was passed to the function to pull up the pin and this is the case for both of the IO_DI_Init functions, for the high voltage input pin and for the low voltage input pin (for both of the power socket input pins).
3. The limits (voltage limits for low/high-levels) and it is to be passed as a pointer, so the address of a variable called limits was passed which is a struct of type IO_DRIVER_DI_LIMITS which is a typedefed strcut of type _io_driver_di_limits which Contains the thresholds for valid low- and high-levels for digital inputs. And this is the case of both of the power socket inputs, so the case of the the IO_DI_Init functions (the one of the power socket input and the one of the power socket output).
The output (return) of the IO_DI_Init function is stored a variable by the name high_voltage_input_pin_init for the high voltage power socket input pin and low_voltage_input_pin_init init for the low voltage power socket input pin which are of type IO_ErrorType.

After initializing the power socket inputs, I started to the initialize and setup the power socket outputs to be able to utilize and make use of the IO_DO_Set function, which is the function that would be used for the activation of the different voltage level pins (either high voltage pin, or both high and low voltage pins).

For the high voltage output pin and low output voltage pin the function utilized for initialization and setup is IO_DO_Init which is the function used to setup digital outputs. The initialization function was implemented for both the High voltage output pin and the low voltage output pin (both of the power socket output pins).

The proper parameters were passed to both of the IO_DO_Init functions:

1. The do_channel (IO PIN), IO_DO_00 channel was chosen for the high voltage output pin, and IO_DO_01 was chosen for the low voltage output pin which are Digital High-Side Outputs with analog feedback, and they were defined by the names high_voltage_output_pin and low_voltage_output_pin respectevily, they were defined in the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired.

2. The `overload_limit` which is a Configurable limit in mA above which `IO_E_PROT_USER_OVERLOAD` is reported which is defined to be 140, so above this limit An output will detect a situation that specifies as overload by the user. In our case, the case of digital outputs If the measured current rises above the `overload_limit` specified upon initialization with `IO_DO_Init`, the driver task-function will return this error code.

The output (return) of the `IO_DO_Init` function is stored a variable by the name `high_voltage_output_pin_init` for the high voltage power socket output pin and `low_voltage_output_pin_init` for the low voltage power socket output pin which are of type `IO_ErrorType`.

After setting up and initializing the power sockets inputs and outputs, I started to implement the core of the task according to the architecture described previously.

First thing I started to implement the reading of the power socket input pins, but before doing so, I had to verify the initialization, make sure that the power socket input pins were initialized successfully, in other words, I had to verify that the initialization functions (`IO_DI_Init`) of both of the power socket input pins return `IO_E_OK`. If the initialization functions return `IO_E_OK` indicates that every thing is fine, no errors.

The initialization functions return `IO_E_OK` means the following:

- The channel id exists.
- The digital output channel is free to be used by this function, in other words the digital output channel is not currently used by another function.
- parameter is in range.
- No group conflict.
- The DI capability of this channel has been successfully activated.
- The given voltage limits are valid.
- The common driver init function `IO_Driver_Init()` has been called successfully.

After checking that the power socket input pins I started implementing the `IO_DI_Get` function for the purpose of reading the power socket inputs, it is the function to be utilized to return the state of digital inputs, and since i need to return the state of 2 digital inputs (The power socket inputs).

I implemented 2 `IO_DI_Get`:

1. `IO_DI_Get` for the high voltage input pin.
2. `IO_DI_Get` for the low voltage input pin.

The proper parameters were passed to the power socket inputs `IO_DI_Get` functions:

1. `di_channel` (IO pin), the digital input pin basically which is `IO_DI_00` for the high voltage power socket input pin, and `IO_DI_01` for the low voltage power socket input pin , The same channels (pins) used for the `IO_DI_Init` functions (Power socket input pins initialization and setup functions) defined with the variable name `high_voltage_input_pin`

for the power socket high voltage input pin and the variable name `low_voltage_input_pin` for the power socket low voltage input pin, they were defined in the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired.

2. `di_value`, The digital Input value, so the digital input value to be read on the pin, the digital input value to be returned. This parameter is to be passed as a pointer, so i passed the address of the variables in which I want the return values of the power socket digital input pins to be returned).

The variables are called:

- `high_volatge_input_pin_value` for the high voltage power socket input pin.
- `low_volatge_input_pin_value` for the low voltage power socket input pin.

This means that the variable to be checked later for the value of the power socket high voltage digital input pin is `high_volatge_input_pin_value`, and the variable to be checked for the value of the power socket low voltage digital input pin is `low_volatge_input_pin_value`.

After that I started implementing the activation of the power socket high voltage output pin, so the condition for which I should activate the power socket high voltage output pin.

The condition to activate the power socket high voltage output pin was implemented according to the following rules:

1. `high_volatge_input_pin_value` which is the variable returning the value of the power socket high voltage input pin to return TRUE which means that the pin is on a high level (The pin is high).
2. `low_volatge_input_pin_value` which is the variable returning the value of the power socket low voltage input pin to return FALSE which means that the pin is on a low level (The pin is low).

To activate the power socket high voltage output pin those 2 conditions have to hold, for any other possibility the power socket high voltage output pin shouldn't be activated. For example if `low_volatge_input_pin_value` (the variable returning the value of the power socket low voltage input) pin to returns TRUE at any point the power socket high voltage output pin shouldn't be activated. Both conditions should hold. It was implemented using an if condition and a logical AND.

After the implementation of the checking on the power socket input pins, I directly implemented the power socket high voltage output pin activation utilizing the `IO_DO_Set` function which is the function used for Setting the state of a Digital Output which is exactly what I need to do.

I started to send the proper parameters to the **`IO_DO_Set`** function:

1. `do_channel` (IO pin), the digital output pin basically which is `IO_DI_02` for the high voltage power socket output pin, and `IO_DI_03` for the low voltage power socket output pin , The same channels (pins) used for the `IO_DI_Init` functions (Power socket output pins initialization and setup functions) defined with the variable name `high_voltage_output_pin` for the power socket high voltage output pin and the variable name `low_voltage_output_pin` for the power socket low voltage output pin, they were defined in

the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired.

2. `do_value`, the output value to be set which is one of 2 possibilities:
 - `TRUE` for High Level, for setting the pin high.
 - `FALSE`, for Low Level, for setting the pin low.

And for this parameter I passed `TRUE` to activate the high voltage power socket output pin, under the condition that `high_volatge_input_pin_value` returns `TRUE` AND `low_volatge_input_pin_value` reutrns `FALSE`.
3. `voltage_fb` (voltage feedback), ADC voltage value in mV of the feedback signal. It is to be passed as a pointer, so I passed the adress of the variable with the name `high_volatage_output_pin_voltage_feedback_in_mv` which I declared as a global `ubyte2` const (global 2 byets unsigned constant). `high_volatage_output_pin_voltage_feedback_in_mv` is the variable in which I will return the voltage value of the pin, it is the variable to check later for the pin voltage value. The voltage value is in the Range: 0 up to 32780 (0mV up to 32780mV).

After implementing the activation of the power socket high voltage output pin, I started the implementation for both power socket outputs activation, The high voltage output pin AND the low voltage output pin activation.

The condition to activate both power socket output pins was implemented according to the following rules:

1. `high_volatge_input_pin_value` which is the variable returning the value of the power socket high voltage input pin to return `TRUE` which means that the pin is on a high level (The pin is high).
2. `low_volatge_input_pin_value` which is the variable returning the value of the power socket low voltage input pin to return `TRUE` which means that the pin is on a highlevel (The pin is high).

To activate both of the power socket output pins (high voltage output pin and low voltage output pin) those 2 conditions have to hold.

After the implementation of the checking on the power socket input pins, I directly implemented the power socket high voltage output pin activation utilizing 2 `IO_DO_Set` functions.

Parameters passed to the IO_DO_Set functions for the implemetaion of the activation of both power socket output pins:

1. Output channels (IO pins configured as output pins).
 - `IO_DO_02` for the high voltage output pin defined with the variable name `high_voltage_output_pin`.
 - `IO_DO_03` for the low voltage output pin defined as `low_voltage_output_pin`.
2. `do_value`. `TRUE` was passed for both of the `IO_DO_Set` functions (`IO_DO_Set` of the high voltage output pin and `IO_DO_Set` of the low voltage output pin) for the purpose of the activation of both of the power socket outputs.
3. `voltage_fb` (voltage feedback).

- The address of the variable `high_volatage_output_pin_voltage_feedback_in_mv` was passed for the `IO_DO_Set` of the power socket high voltage output pin.
- The adress of the variable `low_volatage_output_pin_voltage_feedback_in_mv` was passed for the `IO_DO_Set` of the power socket high voltage output pin.

4.4 Buzzer Management Task

4.4.1 Task Architecture

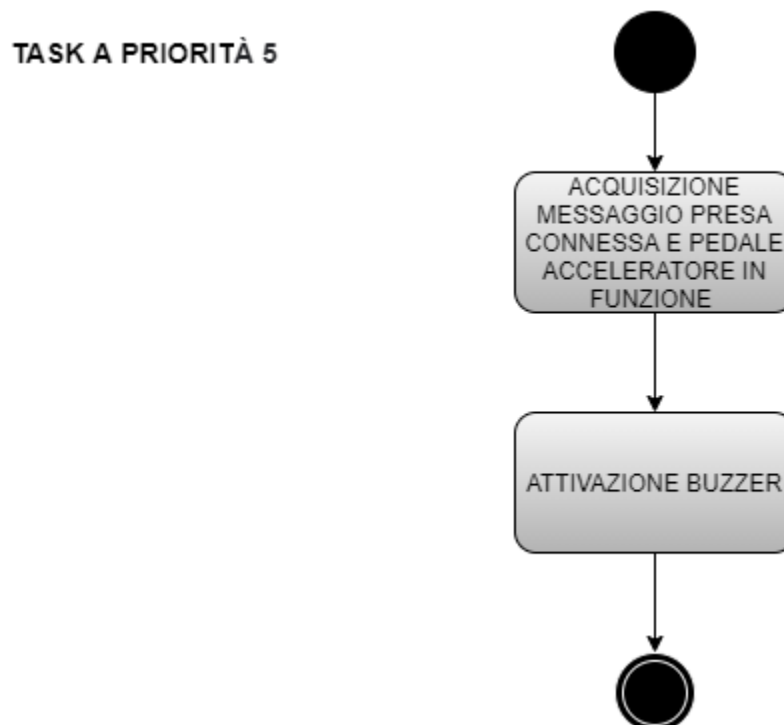


Figure 4. 3: Buzzer management task architecture flow chart

The purpose of the buzzer management task architecture is to activate the buzzer after reading (returning) the following:

1. The acceleration pedal PWM pin (value returned on the PWM pin, current value which indicates the pressure on the pedal) from the pedals management task.

2. Power socket input pins (power socket high voltage input pin and power socket low voltage input pin).

The buzzer is to be activated any way regardless of the values returned from the acceleration pedal and the power socket high and low voltage input pins.

We thought of 2 approaches to implement this task:

1. Wait for a CAN message, and then implement the buzzer activation.
2. Read the PWM pin of the acceleration pedal from the pedals management task and the power socket high and low voltage input pins from the power socket management task, and then implement the buzzer activation.

Since we are not sure which approach to implement at this phase I implemented both of them.

4.4.2 Buzzer Management Code Implementation

4.4.2.1 Buzzer Management .c Code

4.4.2.1.1 Buzzer Management_CAN.c

```

/*
 * task_6_gestione_buzzer.c
 *
 * Created on: 13/jun/2021
 *
 */

#include "task_6_gestione_buzzer.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK6_ID[1] = {6};
ubyte2 const buzzer_pin_voltage_fb;
IO_ErrorType buzzer_activation;
ubyte1 handle;

```



```

IO_CAN_DATA_FRAME const data_frame;
static IO_ErrorType can_return;

void task_6_gestione_buzzer()
{
    ubyte4 task_6_timestamp;
    driver_task_begin_rc = IO_Driver_TaskBegin();
    (void)IO_RTC_StartTime(&task_6_timestamp);
    (void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LEN
GHT, FALSE, TASK6_ID);
    while (IO_EEPROM_PreloadStatus() != IO_E_OK)
    {
        (void)IO_EEPROM_PreloadTask();
    }
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeprom write ok !\n\r");
#endif
    if (can_initialization == IO_E_OK)
    {
        can_return = IO_CAN_ReadMsg(handle, &data_frame);
        if (can_return == IO_E_OK)
        {
            if (buzzer_activation_pin_initialization == IO_E_OK)
            {
                buzzer_activation = IO_DO_Set(buzzer_activation_pin, TRUE, &buzzer
_pin_voltage_fb);
            }
            if (buzzer_activation == IO_E_OK)
            {
#ifdef DEBUG

#endif
                UART_Printf(IO_UART, "buzzer activated successfully \n\r");
            }
        }
    }

    while (TRUE != Check_Task_End(task_6_timestamp, (ubyte4)TASK6_CYCLE_T
IME))

```

```

{
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 6 terminating %u !\n\r", ta
sk_6_timestamp);
#endif
/*serve watchdog */
#ifndef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifndef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
    driver_task_end_rc = IO_Driver_TaskEnd();
}

```

4.4.2.1.2 Buzzer management_DIO.c

```

/*
 * task_6_gestione_buzzer.c
 *
 * Created on: 13/jun/2021
 *
 */

#include "task_6_gestione_buzzer.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
#include "task_5_gestione_carica.h"
static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;

```

```

const ubyte1 TASK6_ID[1] = {6};
ubyte2 const buzzer_pin_voltage_fb;
IO_ErrorType buzzer_activation;
bool high_volatge_input_pin_value;
bool low_volatge_input_pin_value;
bool acceleration_and_power_socket_pins_variable = FALSE;
void task_6_gestione_buzzer()
{
    ubyte4 task_6_timestamp;
    driver_task_begin_rc = IO_Driver_TaskBegin();
    (void)IO_RTC_StartTime(&task_6_timestamp);
    (void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LEN
GHT, FALSE, TASK6_ID);
    while (IO_EEPROM_PreloadStatus() != IO_E_OK)
    {
        (void)IO_EEPROM_PreloadTask();
    }

#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeprom write ok !\n\r");
#endif

/*code_here*/ //IO_PWM_PIN_ACCE is the acceleration pedal pin
    pwm_get_rc_acce = IO_PWM_GetCur(
        IO_PWM_PIN_ACCE, &pwm_current_fb_val_acce, &pwm_fresh_acce);
    // only update the current feedback value if fresh
    if ((pwm_get_rc_acce == IO_E_OK) && (pwm_fresh_acce))
    {
        pwm_current_fb_acce = pwm_current_fb_val_acce;
    }
    IO_DI_Get(high_voltage_input_pin, &high_volatge_input_pin_value);
    IO_DI_Get(low_voltage_input_pin, &low_volatge_input_pin_value);
    if ((pwm_get_rc_acce > 0) && (high_volatge_input_pin_value > 0) && (l
ow_volatge_input_pin_value > 0))
    {
        acceleration_and_power_socket_pins_variable = TRUE;
    }
    else
    {

```

```
    acceleration_and_power_socket_pins_variable = FALSE;
}

if (acceleration_and_power_socket_pins_variable == TRUE)
{
    buzzer_activation = IO_DO_Set(buzzer_activation_pin, TRUE, &buzzer_p
in_voltage_fb); // activate buzzer.
}
if (buzzer_activation == IO_E_OK)
{
#ifdef DEBUG

#endif
    UART_Printf(IO_UART, "buzzer activated successfully \n\r");
}

while (TRUE != Check_Task_End(task_6_timestamp, (ubyte4)TASK6_CYCLE_T
IME))
{
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 6 terminating %u !\n\r", ta
sk_6_timestamp);
#endif
    /*serve watchdog */
#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service());
#endif
    driver_task_end_rc = IO_Driver_TaskEnd();
}
```

4.4.2.2 Buzzer Management .h Code

4.4.2.2.1 Buzzer Management_CAN.h:

```
/*
 * task_6_gestione_buzzer.c
 *
 * Created on: 13/jun/2021
 *
 */

#ifndef TASK_6_GESTIONE_BUZZER
#define TASK_6_GESTIONE_BUZZER
#define DEBUG
#define TASK6_CYCLE_TIME 100000u
void task_6_gestione_buzzer(void);
#define buzzer_activation_pin IO_DO_07
#define can_channel_for_buzzer IO_CAN_CHANNEL_0
#define CAN_BAUDRATE IO_CAN_BAUDRATE_50K

#endif
```

4.4.2.2.2 Buzzer Management_DIO.h:

```
/*
 * task_6_gestione_buzzer.c
 *
 * Created on: 13/jun/2021
 *
 */

#ifndef TASK_6_GESTIONE_BUZZER
#define TASK_6_GESTIONE_BUZZER
#define DEBUG
#define TASK6_CYCLE_TIME 100000u
void task_6_gestione_buzzer(void);
#define buzzer_activation_pin IO_DO_07
```

```
#endif
```

4.4.4 Development and Implementation

4.4.4.1 Functions and Drivers Utilized

IO_Driver_TaskBegin

IO_Driver_TaskEnd

IO_RTC_StartTime

IO_EEPROM_PreloadWrite

IO_EEPROM_PreloadStatus

IO_EEPROM_PreloadTask

IO_WDTimer_Service

IO_PWM_Init

IO_PWM_SetDuty

IO_PWM_GetCur

IO_DI_Init

IO_DO_Init

IO_DI_Get

IO_DO_Set

IO_CAN_Init

IO_CAN_ReadMsg

Digital inputs and outputs (DIO) driver

Pulse Width Modulation (PWM) driver

Real Time Clock (RTC) driver

Controller Area Network (CAN) driver

Universal Asynchronous Receiver Transmitter (UART) driver

EEPROM driver

EEPROM preload functions

4.4.4.2 Development and implementation documentation

1st approach (Wait for a CAN message):

I started the implementation initializing the CAN driver by implementing the `IO_CAN_Init` in the main function which is function responsible for the Initialization of the CAN communication driver.

The IO_CAN_Init functionalities:

1. Allows the module to function (enables it).
2. Allows the module clock to be set to 40MHz.
3. Sets the bit timing for the specified baudrate automatically.

Implementing the `IO_CAN_Init`, I started passing the proper functions:

1. CAN channel (IO pin). The channel (IO pin) chosen and passed to the function as 1st parameter is the `IO_CAN_CHANNEL_0` which was defined in the header file of the task as `can_channel_for_buzzer` for the readability and reusability of the code and the ease of modifying the channel (pin) in the future if desired.
There are 2 CAN interfaces available, the `IO_CAN_CHANNEL_0` and the `IO_CAN_CHANNEL_1`. The `IO_CAN_CHANNEL_1` is available only for the HY-TTC 32 and HY-TTC 32S, and since our VMU is the HY-TTC 32S, it is available for us to use. In other words the CAN channel can be modified in the future from `IO_CAN_CHANNEL_0` to `IO_CAN_CHANNEL_1` without problems as long as we have the same VMU, the (HY-TTC 32S). If the VMU changes gets changed to a VMU different from the HY-TTC 32 and HY-TTC 32S then it will not be possible to use the `IO_CAN_CHANNEL_1` channel.
2. The 2nd parameter is the Baud rate. The baud rate chosen is the `IO_CAN_BAUDRATE_50K` which is defined in the header file of the task with the variable name `CAN_BAUDRATE`, which is defined and set in the main function to 500 Kbps.

The settings of the 500 kbps (IO_CAN_BAUDRATE_50K):

- BRP = 50.
- TSEG1 = 13.
- TSEG2 = 2.
- SJW = 2.
- DIV8 = FALSE (Sampling Point 85%).

After initializing the CAN driver I started to implement the `IO_CAN_ReadMsg` which is the function responsible for Returning the data of a message object for the CAN driver, it reads a message from a specified message object, and returns whether the message is new or not. I started implementing the `IO_CAN_Read_Msg` after I verified that the can driver was initialized correctly (`IO_CAN_Init` return `IO_E_OK`).

The can initialization function (IO_CAN_Init) return IO_E_OK indicates that the CAN driver was initialized correctly and successfully, more precisely it indicates the following:

- everything fine.
- There isn't any invalid parameter passed.
- The passed channel ID is valid.
- channel has not been initialized before, this is the 1st time this channel is being initialized, so it is perfectly fine to initialize and utilize it.
- The specified channel supports the desired functionality.

Passing the proper parameters to the IO_CAN_ReadMsg:

1. The 1st parameter passed to the IO_CAN_ReadMsg is the handle, the CAN message object handle which is retrieved from the IO_CAN_ConfigMsg() and declared as 1 byte unsigned integer.
2. The 2nd parameter passed to the IO_CAN_ReadMsg is the buffer which is to be passed as a Pointer to data buffer structure. The received frame is to be stored there. So I passed the address of the variable to return the received data frame, the variable name is data_frame and it is of type IO_CAN_DATA_FRAME which is a typedefed struct of type _io_can_data_frame which is a struct that Stores a data frame for the CAN communication.

After implementing the CAN driver I started to implement the 2nd part of the task which is the activation of the buzzer. To do so, I started initializing the digital input output driver, more precisely I need to initialize the digital output. To do so I started to implement the IO_DO_Init which is a digital inout output driver initialization function responsible for setting up the digital outputs. The IO_DO_Init was implemented in the main function.

Passing the proper parameters to the IO_DO_Init DIO driver initialization function:

1. The 1st parameter passed is the digital output channel (IO pin). The chosen channel (IO pin) is the IO_DO_07 which is a Digital High-Side Output pin with analog feedback. The chosen channel (IO_DO_07) is defined as buzzer_activation_pin in the header file of the task for the ease of the readability and reusability of the code and the ease of modifying the channel (pin) in the future if desired.
2. The 2nd parameter passed to the IO_DO_Init is the overload_limit which is a Configurable limit in mA above which IO_E_PROT_USER_OVERLOAD is reported which is defined to be 140, so above this limit An output will detect a situation that specifies as overload by the user. In our case, the case of digital outputs If the measured current rises above the overload_limit specified upon initialization with IO_DO_Init, the driver task-function will return this error code.

After initializing the DIO driver for the digital output pin of the buzzer I checked 2 conditions and then directly after implemented the buzzer activation.

The conditions are:

1. `IO_CAN_ReadMsg` (function Returning the data of a message object for the CAN driver) returns `IO_E_OK` which indicates that the `IO_CAN_ReadMsg` function was implemented successfully (everything is fine), more precisely it indicates the following:
 - No overflow of message object.
 - New data has been received successfully since the last read.
 - No invalid handle has been passed.
 - No null pointer has been passed.
 - the specified handle was configured successfully.
 - The received data is valid, No need to read again.
2. `IO_DO_Init` returns `IO_E_OK` which indicates that the `IO_DO_Init` function was implemented successfully (everything is fine), more precisely it indicates the following:
 - the channel id exists.
 - The digital output channel is free to be used by this function.
 - No Internal software error occurred.
 - The DO capability of this channel been successfully activated.
 - There are free slots to setup task function.
 - The common driver init function `IO_Driver_Init()` has been called successfully.

Then I implemented the buzzer activation If and only if both of the previously mentioned conditions hold. The 2nd condition mentioned previously is the one that indicates that the DIO driver is successfully initialized, so I directly started implementing the `IO_DO_Set` function.

I started to send the proper parameters to the IO_DO_Set function:

1. Digital output channel (IO pin), which is `IO_DI_07`, The same channel (pins) used for the `IO_DI_Init` function (DIO driver initialization function) defined with the variable name `buzzer_activation_pin`, it was defined in the header file of the task for the ease, readability and reusability of the code, and for the purpose of easing the process of modifying the pin later on if desired.
2. `do_value`, the output value to be set which is one of 2 possibilities:
 - `TRUE` for High Level, for setting the pin high.
 - `FALSE`, for Low Level, for setting the pin low.

For this parameter I passed `TRUE` to activate the buzzer output pin.

3. voltage feedback: ADC voltage value in mV of the feedback signal. It is to be passed as a pointer, so I passed the adress of the variable with the name `buzzer_pin_voltage_fb` which I declared as a global `ubyte2` const (global 2 byets unsigned constant). `buzzer_pin_voltage_fb` is the variable in which I will return the voltage value of the pin, it is the variable to check later for the pin voltage value. The voltage value is in the Range: 0 up to 32780 (0mV up to 32780mV).

2nd approach (Read the PWM pin of the acceleration pedal from the pedals management task and the power socket high and low voltage input pins from the power socket management task):

For this part please refer to the pedals management task and the power management task mentioned previously as the implementation is performed in the same exact way with the same variable names, definition and everything.

So here I read the acceleration pedal PWM pin through the PWM driver exactly as done in the the pedals management task, and the power socket input pins through the DIO driver exactly as performed in the power socket management task, and then I checked the following conditions:

1. The PWM driver and PWM driver functions for the acceleration pedal were implemented successfully by checking:
 1. The PWM driver initialization function `IO_PWM_Init` returns `IO_E_OK`.
 2. The `IO_PWM_GetCur` function returns `IO_E_OK` and it has a new current value.
 3. The variable returning the feed back current has a value greater than zero.
2. The DIO driver and DIO driver functions for the power socket inputs (high voltage input pin and low voltage input pin) were implemented successfully by checking:
 1. The variable returning the power socket high voltage input pin returns a value greater than zero.
 2. The variable returning the power socket low voltage input pin returns a value greater than zero.

After checking on those conditions I implemented the buzzer activation as the same way performed in the 1st approach mentioned previously in the condition that all the conditions hold.

4.5 Acceleration Pedal Verification Task

4.5.1 Task architecture

The purpose of the pedals comparison task is to make a function that reads (returns) the value of two pedal inputs and check that they are the same. Its is not to be implemented through a simple comparison. The comparison has to be implemented with in a custom range.

The return values from the two pedals is to be subtracted, for the purpose of obtaining the difference. If the difference of the return values of the 2 pedals is in the custom defined range, the function should return true, other wise it should return false.

The range have to be -limit and +limit.

The 2 pedals to be read, pressure value (current value) returned and then compared are actually the same acceleration pedal, but from 2 different pins, so I have 2 pins that give the same acceleration. It is to check for safety.

So, the main goal of the task is to check and verify the acceleration pedal through 2 different pins and compare them to make sure if their difference is in a certain custom range which is to be defined. If their difference is in the custom range, the function is to return true, other wise it is to return false.

To be clear I am not checking 2 different pedals. I am checking the same pedal, the acceleration pedal through 2 different pins for safety purposes.

Core Implementation steps:

1. Check the pressure on the acceleration pedal_1, which means to check the return current from the acceleration pedal_1. In other words check the acceleration pedal from a certain pin which is different from the pin from which I will check the acceleration again.
2. Check the pressure on the acceleration pedal_2 (redundant acceleration pedal), which means to check the return current from the acceleration pedal_2 (redundant acceleration pedal). In other words check the acceleration pedal from a different pin than the one I checked the acceleration pedal from previously.
3. update the current feedback values for both pedals (acceleration pedal from the 2 different pins) if the conditions to do so hold.
4. Calculate the difference of the the returned pressure (returned current value) from the 2 pedals (acceleration pedal from the 2 different pins).
5. Check if the difference of the returned value from the 2 different pins is with in the custom range. I implemented the logic of this step in 2 different ways (2 different algorithms).
6. Return true if the difference of the returned value from the 2 different pins is with in the custom range. Return false if the difference of the returned value from the 2 different pins is out of the custom range.

4.5.2 Acceleration Pedal Verification Code Implementation

4.5.2.3 Acceleration Pedal Verification .c Code

4.5.2.3.1 Acceleration Pedal Verification (algo_1) .c code:

```
/*
 * pedals_compare.c
 *
 * Created on: Jul 13, 2021
 *
 */

#include "task_1_gestione_pedali.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_Driver.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"

#define min_range_value -10
#define max_range_value 10

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK1_ID[1] = {1};
static IO_ErrorType pwm_get_rc_acce;
static IO_ErrorType pwm_get_rc_acce_red;

static ubyte2 pwm_current_fb_val_acce;
static bool pwm_fresh_acce;
static ubyte2 pwm_current_fb_acce_pedal_pressure_value;

static ubyte2 pwm_current_fb_val_acce_red;
static bool pwm_fresh_acce_red;
static ubyte2 pwm_current_fb_freno_pedal_pressure_value;
```

```

static ubyte2 res;
static ubyte2 pwm_current_fb_another_acce_pedal_pressure_value;

bool pedals_compare1()
{
    driver_task_begin_rc = IO_Driver_TaskBegin();
    IO_ErrorType eeprom_error = 512;
    ubyte4 task_1_timestamp;
    (void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LEN
GHT, FALSE, TASK1_ID);
    while (IO_EEPROM_PreloadStatus() != IO_E_OK)
    {
        (void)IO_EEPROM_PreloadTask();
    }
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeprom write ok !\n\r");
#endif

    /* function that read the value from the two pedal input and check th
ey are the same.
You don't have to make a simple comparison, but there have to be a cus
tom range.
When difference of that value goes in that range function return true
otherwise false */

    /* VERIFICA PRESSIONE PEDALE ACCELERAZIONE */

    // read the current feedback value
    pwm_get_rc_acce = IO_PWM_GetCur(
        IO_PWM_PIN_ACCE, &pwm_current_fb_val_acce, &pwm_fresh_acce);

    // only update the current feedback value if fresh

    if ((pwm_get_rc_acce == IO_E_OK) && (pwm_fresh_acce))
    {
        pwm_current_fb_acce_pedal_pressure_value = pwm_current_fb_val_acce;
#ifdef DEBUG
        UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r", p
wm_current_fb_acce_pedal_pressure_value);

```

```

#endif
}

/* VERIFICA PRESSIONE PEDALE ACCELERAZIONE REDUNDANT */

// read the current feedback value
pwm_get_rc_acc_red = IO_PWM_GetCur(
    IO_PWM_PIN_ACCE_RED, &pwm_current_fb_val_acce_red, &pwm_fresh_acce_red);

// only update the current feedback value if fresh

if ((pwm_get_rc_acc_red == IO_E_OK) && (pwm_fresh_acce_red))
{
    pwm_current_fb_acce_red_pedal_pressure_value = pwm_current_fb_val_acce_red;
#ifdef DEBUG
    UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r", pwm_current_fb_acce_red_pedal_pressure_value);
#endif
}

res = pwm_current_fb_acce_pedal_pressure_value - pwm_current_fb_acce_red_pedal_pressure_value;

if ((res >= min_range_value) && (res <= max_range_value))
{
    return TRUE;
}

else
{
    return FALSE;
}

(void)IO_RTC_StartTime(&task_1_timestamp);
while (TRUE != Check_Task_End(task_1_timestamp, (ubyte4)TASK1_CYCLE_TIME))
{

```

```
IO_RTC_PeriodicDeInit();
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 1 terminating %u !\n\r", ta
sk_1_timestamp);
    /*serve watchdog */
#endif
/*serve watchdog */
#ifndef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifndef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
driver_task_end_rc = IO_Driver_TaskEnd();
}
```

4.5.2.3.2 Acceleration Pedal Verification (algo_2) .c code:

```
/*
 * pedals_compare2.c
 *
 * Created on: Jul 13, 2021
 *
 */

#include "task_1_gestione_pedali.h"
#include "utility.h"
```

```

#include "eeprom_address.h"
#include "IO_Driver.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"

#define min_range_value -10
#define max_range_value 10

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK1_ID[1] = {1};
static IO_ErrorType pwm_get_rc_acce;
static IO_ErrorType pwm_get_rc_freno;

static ubyte2 pwm_current_fb_val_acce;
static bool pwm_fresh_acce;
static ubyte2 pwm_current_fb_acce_pedal_pressure_value;

static ubyte2 pwm_current_fb_val_freno;
static bool pwm_fresh_freno;
static ubyte2 pwm_current_fb_freno_pedal_pressure_value;

static ubyte2 res;
static ubyte2 pwm_current_fb_another_acce_pedal_pressure_value;

bool pedals_compare2()
{
    driver_task_begin_rc = IO_Driver_TaskBegin();
    IO_ErrorType eeprom_error = 512;
    ubyte4 task_1_timestamp;
    (void)IO_EEPROM_PreloadWrite(ID_TASK_EXECUTION, ID_TASK_EXECUTION_LEN
GHT, FALSE, TASK1_ID);
    while (IO_EEPROM_PreloadStatus() != IO_E_OK)
    {
        (void)IO_EEPROM_PreloadTask();
    }
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r eeprom write ok !\n\r");
#endif
}

```


/* function that read the value from the two pedal input and check they are the same.

You don't have to make a simple comparison, but there have to be a custom range.

When difference of that value goes in that range function return true otherwise false */

```
/* VERIFICA PRESSIONE PEDALE ACCELERAZIONE */
```

```
// read the current feedback value
```

```
pwm_get_rc_acce = IO_PWM_GetCur(
    IO_PWM_PIN_ACCE, &pwm_current_fb_val_acce, &pwm_fresh_acce);
```

```
// only update the current feedback value if fresh
```

```
if ((pwm_get_rc_acce == IO_E_OK) && (pwm_fresh_acce))
{
    pwm_current_fb_acce_pedal_pressure_value = pwm_current_fb_val_acce;
#ifdef DEBUG
    UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r", p
wm_current_fb_acce_pedal_pressure_value);
#endif
}
```

```
/* VERIFICA PRESSIONE PEDALE ACCELERAZIONE REDUNDANT */
```

```
// read the current feedback value
```

```
pwm_get_rc_acc_red = IO_PWM_GetCur(
    IO_PWM_PIN_ACCE_RED, &pwm_current_fb_val_acce_red, &pwm_fresh_acc
e_red);
```

```
// only update the current feedback value if fresh
```

```
if ((pwm_get_rc_acc_red == IO_E_OK) && (pwm_fresh_acce_red))
{
    pwm_current_fb_acce_red_pedal_pressure_value = pwm_current_fb_val_ac
ce_red;
#ifdef DEBUG
```

```
    UART_Printf(IO_UART, "Acceleration Pedal pressure value: %u \n\r", p
wm_current_fb_acce_red_pedal_pressure_value);
#endif
}

res = pwm_current_fb_acce_pedal_pressure_value - pwm_current_fb_acce_
red_pedal_pressure_value;

if ((res - min_range_value) * (res - max_range_value) <= 0)
{
    return TRUE;
}
else
{
    return FALSE;
}

(void)IO_RTC_StartTime(&task_1_timestamp);
while (TRUE != Check_Task_End(task_1_timestamp, (ubyte4)TASK1_CYCLE_T
IME))
{
    IO_RTC_PeriodicDeInit();
#ifdef DEBUG
    UART_Printf(IO_UART, "\n\r wait for task 1 terminating %u !\n\r", ta
sk_1_timestamp);
    /*serve watchdog */
#endif
    /*serve watchdog */
#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
}
/*serve watchdog */
#ifdef DEBUG
    while (IO_E_OK != IO_WDTimer_Service())
        ;
#endif
driver_task_end_rc = IO_Driver_TaskEnd();
```

```
}
```

4.5.2.4 pedals comparison .h code

```
/*
 * pedals_compare1.h
 *
 * Created on: Jul 14, 2021
 * Author: Ahmed
 */

//This .h goes for all pedals_compare.c files

#ifndef SRC_PEDALS_COMPARE1_H_
#define SRC_PEDALS_COMPARE1_H_

#define IO_PWM_PIN_ACCE IO_PWM_00
#define IO_PWM_PIN_ACCE_RED IO_PWM_02

#endif /* SRC_PEDALS_COMPARE1_H_ */
```

4.5.3 Development and implementation

4.5.3.1 functions and drivers utilized

IO_Driver_TaskBegin

IO_Driver_TaskEnd

IO_RTC_StartTime

IO_EEPROM_PreloadWrite

IO_EEPROM_PreloadStatus

IO_EEPROM_PreloadTask

IO_WDTimer_Service

IO_PWM_Init

IO_PWM_SetDuty

IO_POWER_Set

IO_PWM_GetCur

Pulse Width Modulation (PWM) driver

EEPROM driver

Real Time Clock (RTC) driver

Window Watchdog driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.5.3.2 Development and implementation documentation

The initialization and power set functions of the Pulse Width Modulation (PWM) driver (*IO_PWM_Init*, *IO_PWM_SetDuty*, *IO_POWER_Set*) were already implemented previously in the main function of the project for the pedals management task, so I was ready to directly implement the *IO_PWM_GetCur* function without problems.

I started the implementation by directly checking and verifying the pressure on the acceleration pedal through the 1st pin which is the same pin used for the pedals management task.

The first step done was implementing the **IO_PWM_GetCur** function which is the function responsible for returning the measured current of a certain channel.

The proper parameters were sent to the channel:

1. The PWM channel (pin) [1st parameter], defined as IO_PWM_PIN_ACCE for the acceleration pedal, which is the same pin used for the functions IO_PWM_Init which was also defined in the header file of the task for the ease of changing the pin if desired and for the ease of the readability and reusability of the code.
2. Then as a [2nd parameter] the address of the variable to hold (store) and return the measured current range which would be in the range of 0mA up to 7575mA, the variable is named pwm_current_fb_val_acce and it is declared as a static unsigned 2 bytes integer.
3. The last parameter [3rd parameter] sent to the IO_PWM_GetCur function is the address of the variable to store a TRUE or a FALSE based on if there is or there isn't new values since the last call, so basically this is the variable to check to know if there a new current measured value or there isn't, by there isn't I mean there is no new value which means that the variable holding the measured current value (pwm_current_fb_val_acce in our case, , doesn't have a new value, it has a value of a previous measurement and in this case I will not update the current feedback value. The variable is named pwm_fresh_acce and is declared as a static bool as it holds a true or a false. The return (output) of the IO_PWM_GetCur function for the acceleration pedal is returned and stored in the pwm_get_rc_acce variable which is declared as as a static variable of type io_errortype.

After implementing the IO_PWM_GetCur function, I proceeded by checking 2 conditions, ***I check if the pwm_get_rc_acce returns IO_E_OK and that pwm_fresh_acce returns TRUE:***

1. pwm_get_rc_acce

If pwm_get_rc_acce is IO_E_OK means that the io_pwm_getcur returned IO_E_OK which indicates that everything is fine, more precisely it means:

- The PWM channel is correctly configured.
- The channel ID is there (exists).
- There were no ADC errors.
- Current measurement accuracy hasn't been reduced.
- The specified channel is a PWM channel and it has a current measurement.
- No null pointer has been sent to the function.

2. pwm_fresh_acce

I check if the pwm_fresh_acce is true which would indicate that there is a new value available in the pwm_current_fb_val_acce, other wise in the case that the pwm_fresh_acce is false it indicates that there isn't a new value available in the pwm_current_fb_val_acce which means the current value available in the pwm_current_fb_val_acce is a value of a previous measurement and in that case there is no point to update the current value.

So Basically I update the current feedback value if and only if the pwm_get_rc_acce is IO_E_OK AND the pwm_fresh_acce is true. Other wise (if any of those 2 conditions fail) I don't update the feedback current value.

On the top of the get current function implementation and the checking of the `pwm_get_rc_acce` and the `pwm_fresh_acce` I need to check that the feed back current value is a value different from zero. Because if the feedback current value is equal to zero, this would indicate that there is zero current returned from the acceleration pedal, so there is no pressure on the acceleration pedal. And this is exactly when I go to check the brakes pedal according to the task architecture. And then I go implement the brakes pedal the same exact way I implemented the acceleration pedal.

After implementing the verification of the acceleration pedal through the 1st defined and chosen pin, I implemented the verification of the redundant acceleration pin (the verification of the acceleration pedal through the other pin).

The implementation is performed in the same exact way as for the verification pedal through the 1st PWM pin with minor modifications.

The only differences are:

- The PWM channel (PWM pin) passed as [1st parameter], the channel `IO_PWM_GetCur` function implemented for the redundant acceleration pedal verification is `IO_PWM_02` and is defined as `IO_PWM_PIN_ACCE_RED`.
- The variable for the pointer passed to return the measured current value [2nd parameter] has a different name which is `pwm_current_fb_val_acce_red`.
- The variable for the pointer passed to return if the value stored in the `pwm_current_fb_val_acce_red` (the variable returning the measured current value) [2nd parameter] has a different variable name which is `pwm_fresh_acce_red` [3rd parameter].

And then the checking `pwm_get_rc_acc_red` and `pwm_fresh_acce_red` to check if to update the current feedback value is implemented in the same way as the acceleration pedal verification through the 1st classic PWM pin.

After verifying the acceleration pedal through both of the PWM pins utilized for the classic and redundant acceleration, I implemented the calculation operation of the difference of the returned current (which indicates the returned pressure) of both acceleration pedals (the classic and the redundant).

The implementation of this part was simple and straight forward as I just subtracted the variable that returns and holds the current (pressure) value for the classic acceleration pedal from the variable that returns and holds the current (pressure) value for the redundant acceleration pedal. The result of the operation was stored in a variable with the name `res`, which was declared as a static unsigned 2 byte integer.

At this point the only thing left was to implement the checking of if `res` (the variable holding the result of the difference of the feedback current from the classic acceleration pedal and the redundant acceleration pedal operation) is within the custom range (-limit, +limit) which were defined as `min_range_value` and `max_range_value`.

The implementation of this step was done in 2 different ways (logics):

1. `res = pwm_current_fb_acce_pedal_pressure_value - pwm_current_fb_acc_red_pedal_pressure_value;`

```
if ((res >= min_range_value) && (res <= max_range_value))
{
    return TRUE;
}
```

```
else
{
    return FALSE;
}
```

2. `res = pwm_current_fb_acce_pedal_pressure_value - pwm_current_fb_acc_red_pedal_pressure_value;`

```
if ((res - min_range_value) * (res - max_range_value) <= 0)
{
    return TRUE;
}
else
{
    return FALSE;
}
```

Bibliography

[1] Knobloch, F., Hanssen, S., Lam, A. et al. Net emission reductions from electric cars and heat pumps in 59 world regions over time. *Nat Sustain* 3, 437–447 (2020).
<https://doi.org/10.1038/s41893-020-0488-7>

- [2] Hall, Dale and Lutsey, Nic. Effects of battery manufacturing on electric vehicle life-cycle greenhouse gas emissions. icct (THE INTERNATIONAL COUNCIL ON CLEAN TRANSPORTATION), FEBRUARY 2018
- [3] Carlsson, Fredrik; Johansson-Stenman, Olof. Costs and Benefits of Electric Vehicles. *Journal of Transport Economics and Policy*, January 1, 2003.
- [4] Jiuyu Du, Ye Liu, Xinying Mo, Yalun Li, Jianqiu Li, Xiaogang Wu, Minggao Ouyang,. Impact of high-power charging on the durability and safety of lithium batteries used in long-range battery electric vehicles. *Applied Energy*, 2019.
- [5] Bjart Holtsmark, Anders Skonhoft. The Norwegian support and subsidy policy of electric cars. Should it be adopted by other countries?. *Environmental Science & Policy*. 2014.
- [6] Brain technologies. EVERGRIN project Preliminary Tailoring. 17 November 2020.
- [7] A. G. Boulanger, A. C. Chu, S. Maxx and D. L. Waltz, "Vehicle Electrification: Status and Issues," in *Proceedings of the IEEE*, vol. 99, no. 6, pp. 1116-1138, June 2011, doi: 10.1109/JPROC.2011.2112750.
- [8] Cong Sun, Siqi Zheng, Rui Wang. Restricting driving for better traffic and clearer skies: Did it work in Beijing?. *Transport Policy*. 2014. Available: <https://doi.org/10.1016/j.tranpol.2013.12.010>.
- [9] Fedewa, E. and Chesbrough, C., "Sustainable Mobility: The Business Case for Global Vehicle Electrification," *SAE Technical Paper* 2010-01-2311, 2010, <https://doi.org/10.4271/2010-01-2311>.
- [10] Brain Technologies EVERGRIN ver 1.12 reduced.
- [11] M. Gotz, F. Dittmann and C. E. Pereira, "Deterministic Mechanism for Run-time Reconfiguration Activities in an RTOS," 2006 4th IEEE International Conference on Industrial Informatics, 2006, pp. 693-698, doi: 10.1109/INDIN.2006.275645.

- [12] Luna, R., Islam, S.A. Security and Reliability of Safety-Critical RTOS. SN COMPUT. SCI. 2, 356 (2021). <https://doi.org/10.1007/s42979-021-00753-y>
- [13] Haobo Yu, Andreas Gerstlauer, and Daniel Gajski. 2003. RTOS scheduling in transaction level models. In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03). Association for Computing Machinery, New York, NY, USA, 31–36. DOI:<https://doi.org/10.1145/944645.944653>
- [14] Gliwa P. (2021) Operating Systems. In: Embedded Software Timing. Springer, Cham. https://doi.org/10.1007/978-3-030-64144-3_3
- [15] Abdelazim Mansour, Abdelazim Mansour. Optimization of SPI Communication in Precise Farming. Politecnico Di Torino. March, 2020.
- [16] M. H. A. Abdelsamea, M. Zorkany and N. Abdelkader, "Real Time Operating Systems for the Internet of Things, Vision, Architecture and Research Directions," 2016 World Symposium on Computer Applications & Research (WSCAR), 2016, pp. 72-77, doi: 10.1109/WSCAR.2016.21.
- [17] Stankov, Ivan and Spasov, Grisha. Discussion of Microkernel and Monolithic Kernel Approaches. Technical University – Sofia, branch Plovdiv, Faculty of Electronics and Automation, Plovdiv, Bulgaria. 2006.
- [18] Simon Fürst, BMW Group, Jürgen Mössinger, Bosch, Stefan Bunzel, Continental, Thomas Weber, Daimler, Frank Kirschke-Biller, Ford Motor Company, Peter Heitkämper, General Motors, Gerulf Kinkelin, Peugeot Citroën Automobiles, Kenji Nishikawa, Toyota Motor Corporation, Klaus Lange, Volkswagen AG. AUTOSAR – A Worldwide Standard is on the Road.
- [19] Warschofsky, Robert. AUTOSAR Software Architecture. Hasso-Plattner-Institute für Softwaresystemtechnik.

[20] Dandotiya, Dharmendra. Software Architecture & AUTOSAR for Automotive Embedded system. PATHPARTNER. June 29, 2020.

[21] Naumann, Nico. AUTOSAR Runtime Environment and Virtual Function Bus. Department for System Analysis and Modeling Hasso-Plattner Institute for IT-Systems Engineering.

[22] HYDAC INTERNATIONAL. TTControl HYDAC INTERNATIONAL. Available: chrome-extension://efaidnbmnnnibpcajpcgiclfndmkaj/viewer.html?pdfurl=https%3A%2F%2Fwww.hydac-na.com%2Fwp-content%2Fuploads%2FTT-Control-Technology.pdf&cflen=14173069&chunk=true

[23] TTControl HYDAC INTERNATIONAL. TTControl-HY-TTC-32S-Datasheet. Available: chrome-extension://efaidnbmnnnibpcajpcgiclfndmkaj/viewer.html?pdfurl=https%3A%2F%2Fwww.ttccontrol.com%2Fwp-content%2Fuploads%2FTTControl-HY-TTC-32S-Datasheet.pdf&cflen=262721&chunk=true

