

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:

Celal Tolga FAKI

Company Advisor:

Eng. Giovanni Guida

Company Co – Advisor:

Eng. Saverio Milo

October 2021

Summary

One of the biggest challenges of the automotive industry in recent history is the process of powering the vehicle by electricity which is called Vehicle Electrification.

Electrification in the automotive industry is the process of converting a vehicle from a traditional fuel source to one powered by electricity. Generally, vehicle electrification focuses on the electric powertrain and related auxiliary systems, like on-board and off-board charging systems, as well as wireless power transmission.

The major reasons for supporting the vehicle electrification are the reduction of carbon dioxide emission and environmental pollutants, transition to new intelligent transportation systems in the cities, and possibly the lack of fossil fuels soon. Traditional gasoline cars have an efficiency of 17% to 21%, while electric motors have an efficiency is approximately 85% to 90%. When it comes to maximizing efficiency, electrical systems are the most effective since they can be monitored and communicated with more efficiently than any other system. Consequently, a 100% electric vehicle (EV) will have high efficiency and zero pollutant emissions, reducing the total carbon footprint while still having appealing designs.

After Increasing the importance of Vehicle Electrification, the proposal for the “EVERGRIN” project was introduced by Brain Technologies and Politecnico di Torino.

EVERGRIN is an acronym that stands for – Device for Electric Vehicle ERGonomic for RINnovamento (renovation) and Revamping of Existing Vehicle with Traditional Engine.

The vehicles undergoing the transformation are vehicles that have been produced and released since 2000, far from being "vintage".

Production or replacement with new models for at least 5 years in the EURO 3 or EURO 4 emission class, characterized by a modest and not yet widespread presence of electronic components, but still fully functional and functional for a task, especially if it is an urban and suburban nature.

Main activities of the EVERGRIN project are designing the architecture on Fiat Panda 2nd Series, development of new components (e.g. ABS), VMU development, HMI System development,

Functional Architecture and Vehicle Network Integration, Test Bench Development and Development and Adaptation of Mechanical Parts.

Therefore, the main goal of this thesis is to develop a VMU (Vehicle Management Unit) and functional architecture and vehicle network integration that can allow for more efficient vehicle conversion and management of the electric propulsion system. In that case we worked with the HY-TTC 32S ECU (Electronic Control Unit) which is equipped with a Watchdog and a Main CPU. The watchdog can be used to bring the Electronic Control Unit to a safe state by interrupting the Main CPU and disabling safety switches with using the dedicated output. It is based on an Infineon XC22xx microcontroller, and the software is written in C language. Control unit has 28 I/Os which are freely configurable and two can buses, one is using for the engine and safety systems and another one for the body controls (BCM, ABS, On-Board Systems, HVAC Heater, HMI).

In conclusion, some high priority tasks of the VMU is implemented and the first task is focused on the pedal management (brake pedal and acceleration pedal) which has the most priority in the architecture of the car.

Second task is behavior of the battery of the vehicle. Third one is related to inverter control and the last one is controlling the ABS of the car. Additionally, a debouncer task is implemented and it takes the input value and returns the debounced value in the output of the VMU. It is a general debouncer which can be used for the several purposes (diagnostic test etc.).

The primary objective of the EVERGRIN project of Brain Technologies and Politecnico di Torino is vehicle electrification, or in our instance, the conversion of a diesel-powered FIAT PANDA into a completely electric vehicle. Thus, the objective is "Transformation (electrification) of a diesel-powered Fiat Panda car," i.e., converting it into a completely 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, the construction of a VMU's low-level firmware and basic software for converting a diesel Fiat Panda into an electric vehicle is described. Before considering how to construct our VMU, one of the first actions undertaken by Brain Technologies was selecting the VMU to be built and put in the automobile. Examining the structure of the tasks in order to undertake development was the primary activity. The development was accomplished using the C programming language.

Here are the tasks developed in this thesis work:

1. Acceleration and Brake Pedal Management.
2. Battery Key management.
3. Inverter Management
4. Pump Management Task
5. Debouncer

Acknowledgments

The purpose of the thesis is to contribute to the EVERGRIN project undertaken by Brain Technologies and the Polytechnic University of Turin.

Firstly, I am deeply grateful to my supervisor professor Stefano Carabelli for providing me the support to join the EVERGRIN project and constant guidance throughout this project. The project was not just helpful for my academic career but it was also help me to boost my professional career.

Secondly, I would like to extend my sincere thanks to Giovanni Guida for their assistance at every stage of the research project and helping to start my career as an embedded developer engineer.

Thirdly, I am deeply grateful to Saverio Milo for their unwavering support and belief in me with leading me in this project with sharing his experience in the field of software development.

Lastly, I would like to thank all my family for their encouragement when the times got rough are much more appreciated and duly noted. My heartfelt thanks.

Contents

1 Introduction.....	1
1.1 The Idea and Motivation Behind Electric Vehicles andVehicle 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 criticalsystems.....	25
2.3.2 Classic OSvs 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 TechnicalDetails and Architecture	43
3.2.1 Overview.....	43
3.2.2 Deeper into 32S and 30 family Technical 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	68
4.2 Acceleration and Brake Pedal Management Task.....	69
4.2.1 Task Architecture.....	69

4.2.2 Acceleration and Brake Pedal Management Code Implementation	71
4.2.4 Development and Implementation.....	76
4.3 Battery Key management Task	81
4.3.1 Task Architecture.....	81
4.3.2 Battery Key Management Code Implementation	83
4.3.4 Development and Implementation.....	86
4.4 Inverter Management Task.....	90
4.4.1 Task Architecture.....	90
4.4.2 Inverter Management Code Implementation	91
4.4.3 Development and Implementation.....	94
4.5 Pump Management Task	97
4.5.1 Task architecture	97
4.5.2 Pump Management Code Implementation.....	99
4.5.3 Development and implementation.....	103
4.6 Debouncer Task	104
4.6.1 Task architecture	104
4.6.2 Debouncer Code Implementation.....	105
4.6.3 Development and implementation.....	108
Bibliography.....	109

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.....	69
FIGURE 4. 2: BATTERY KEY MANAGEMENT TASK ARCHITECTURE FLOW CHART.....	81
FIGURE 4. 3: INVERTER MANAGEMENT TASK ARCHITECTURE FLOW CHART	90
FIGURE 4. 4: PUMP MANAGEMENT TASK ARCHITECTURE FLOW CHART	97

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. 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.

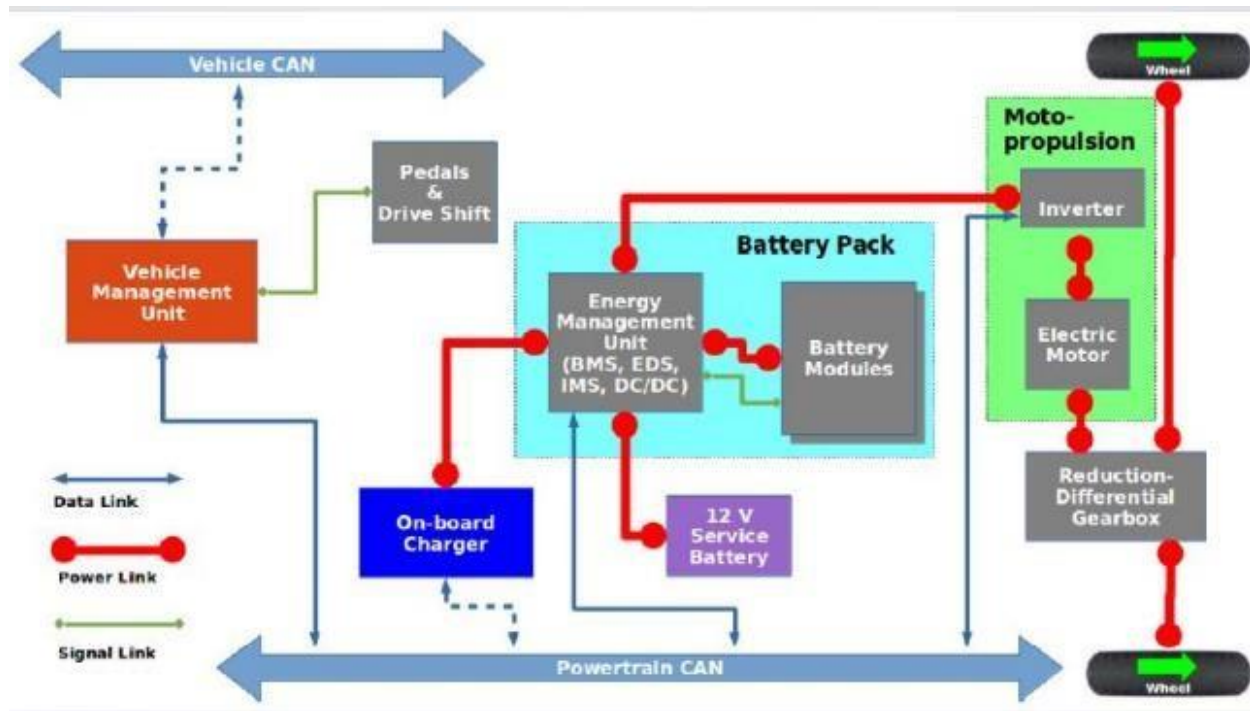


Figure 2. 1: Saver Kit

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 provide 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.

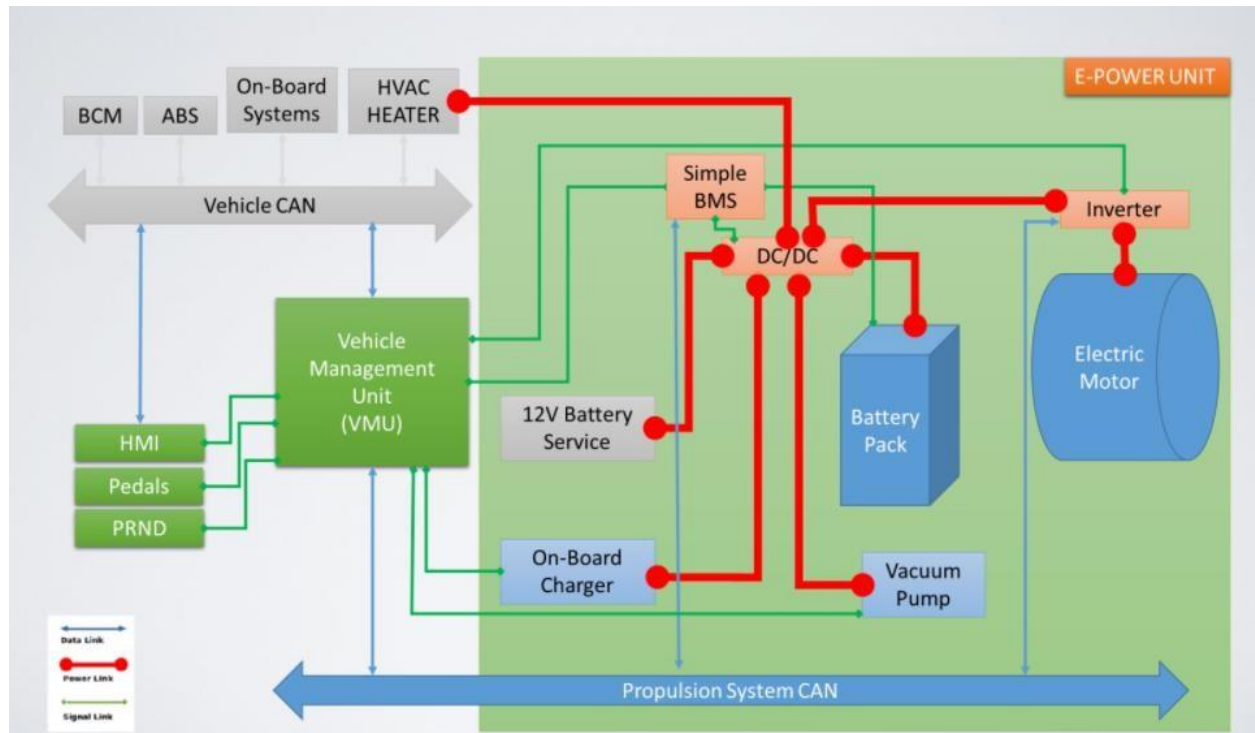


Figure 2. 2: The EVERGRIN architecture

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.

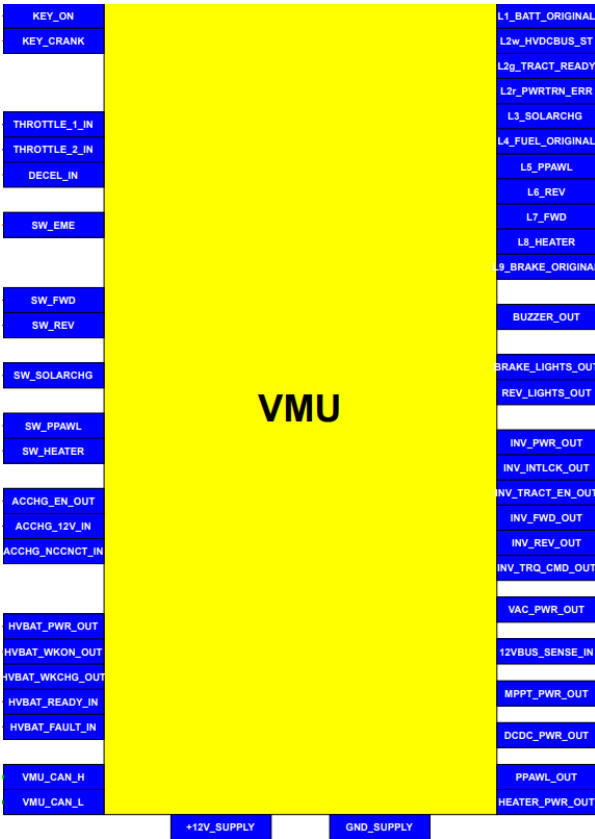


Figure 2. 3: pin configuration of the VMU

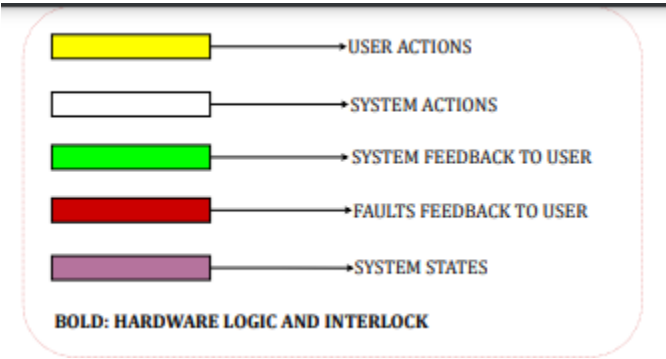


Figure 2. 4: action indicators

2.3.1.1 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.

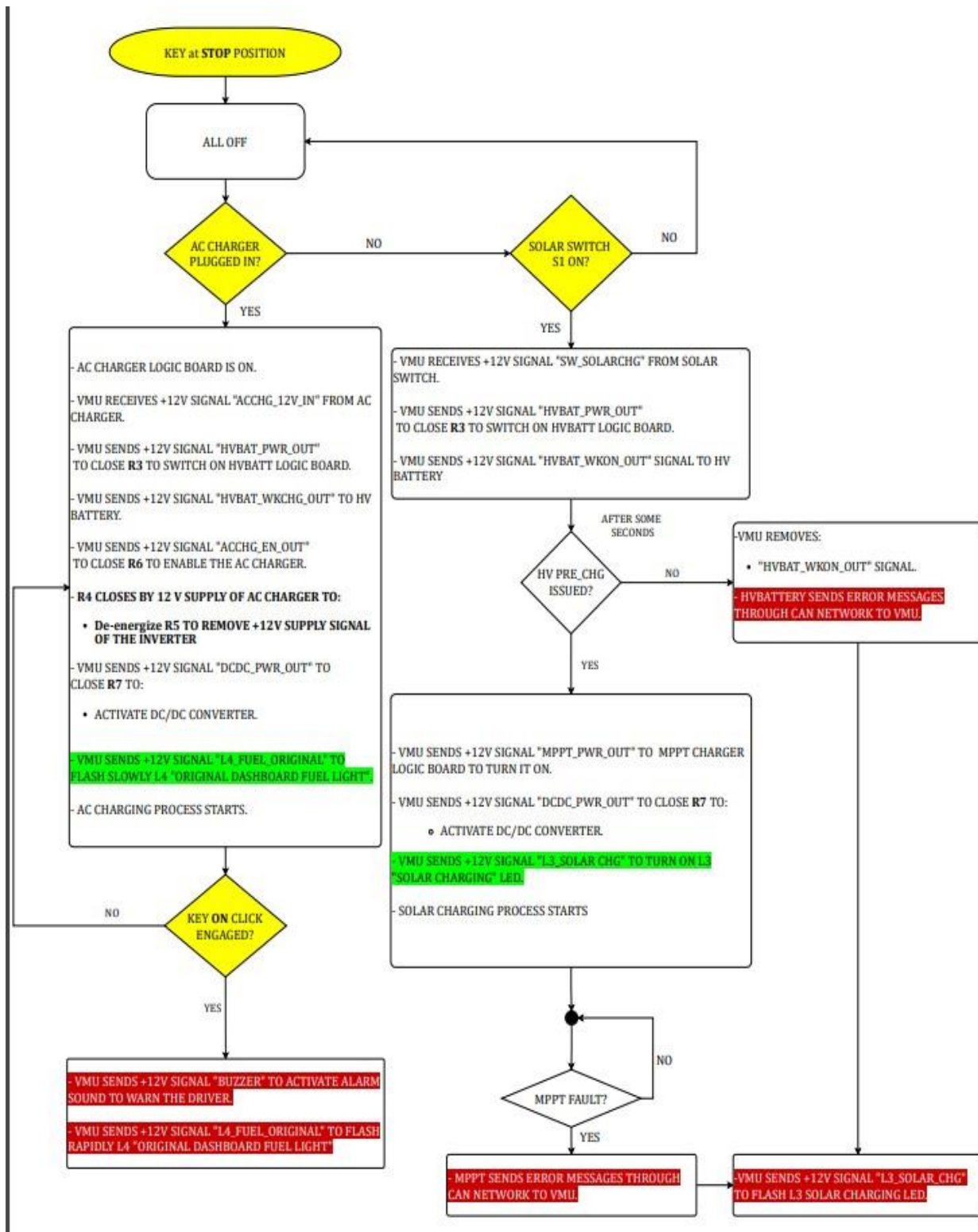


Figure 2. 5: software block diagram of the Key-STOP position

2.3.1.2 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 plugging 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.

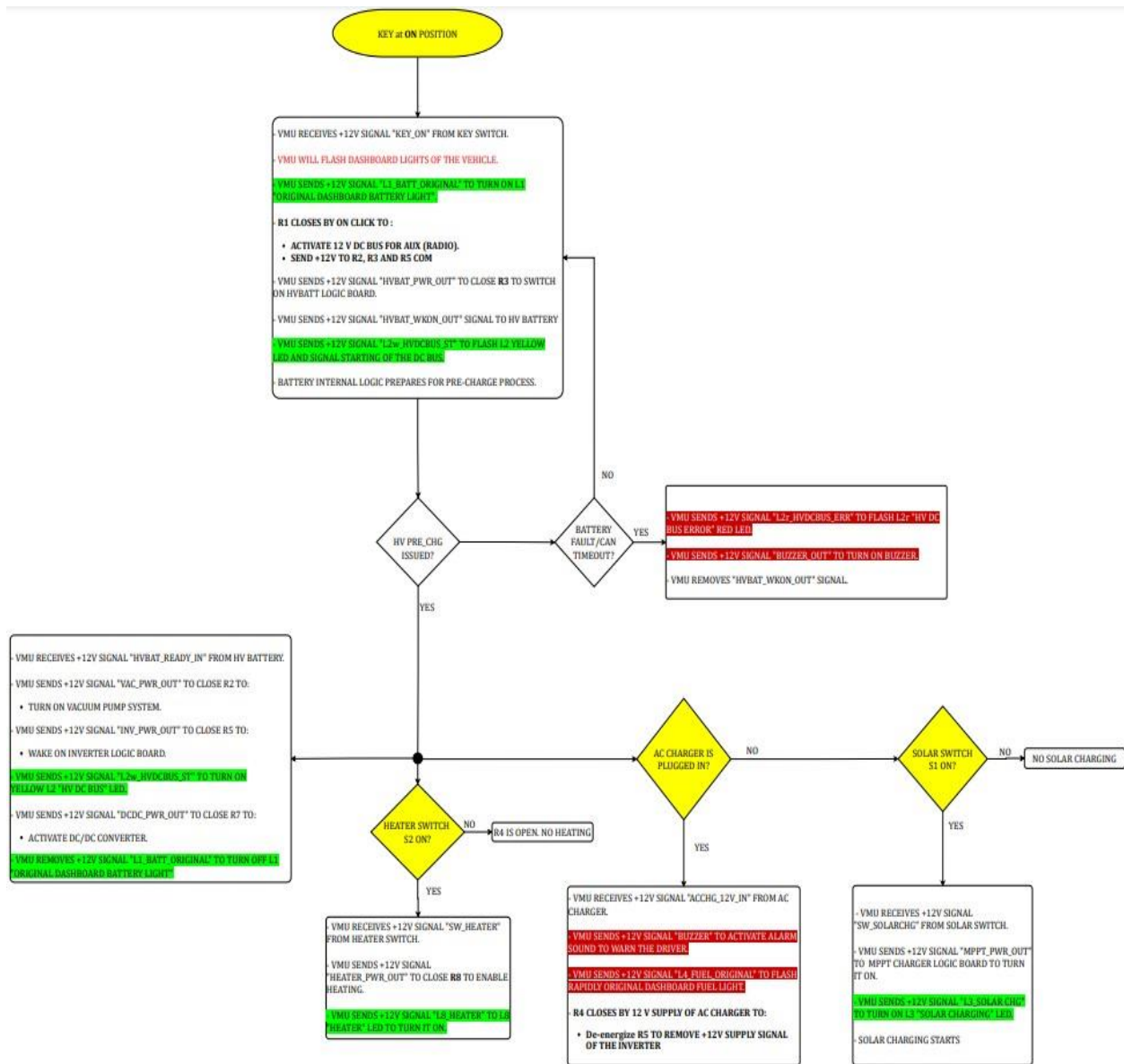


Figure 2. 6: software block diagram of Key-On position

2.3.1.3 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 contractor 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 contractor.

Later the task is checking whether if main contractor 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.

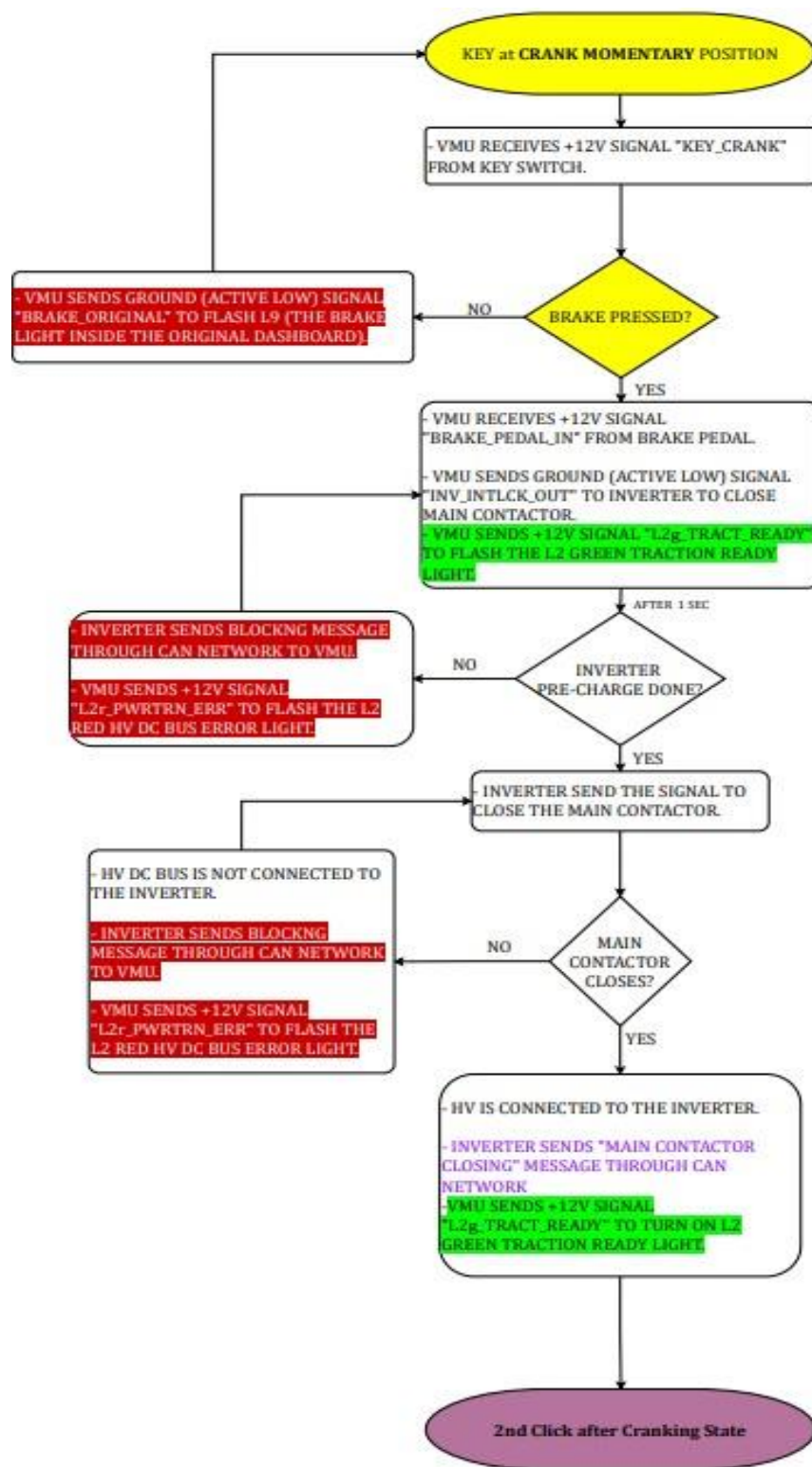


Figure 2. 7: software block diagram of the Key-CRANK MOMENTARY position

2.3.1.4 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.

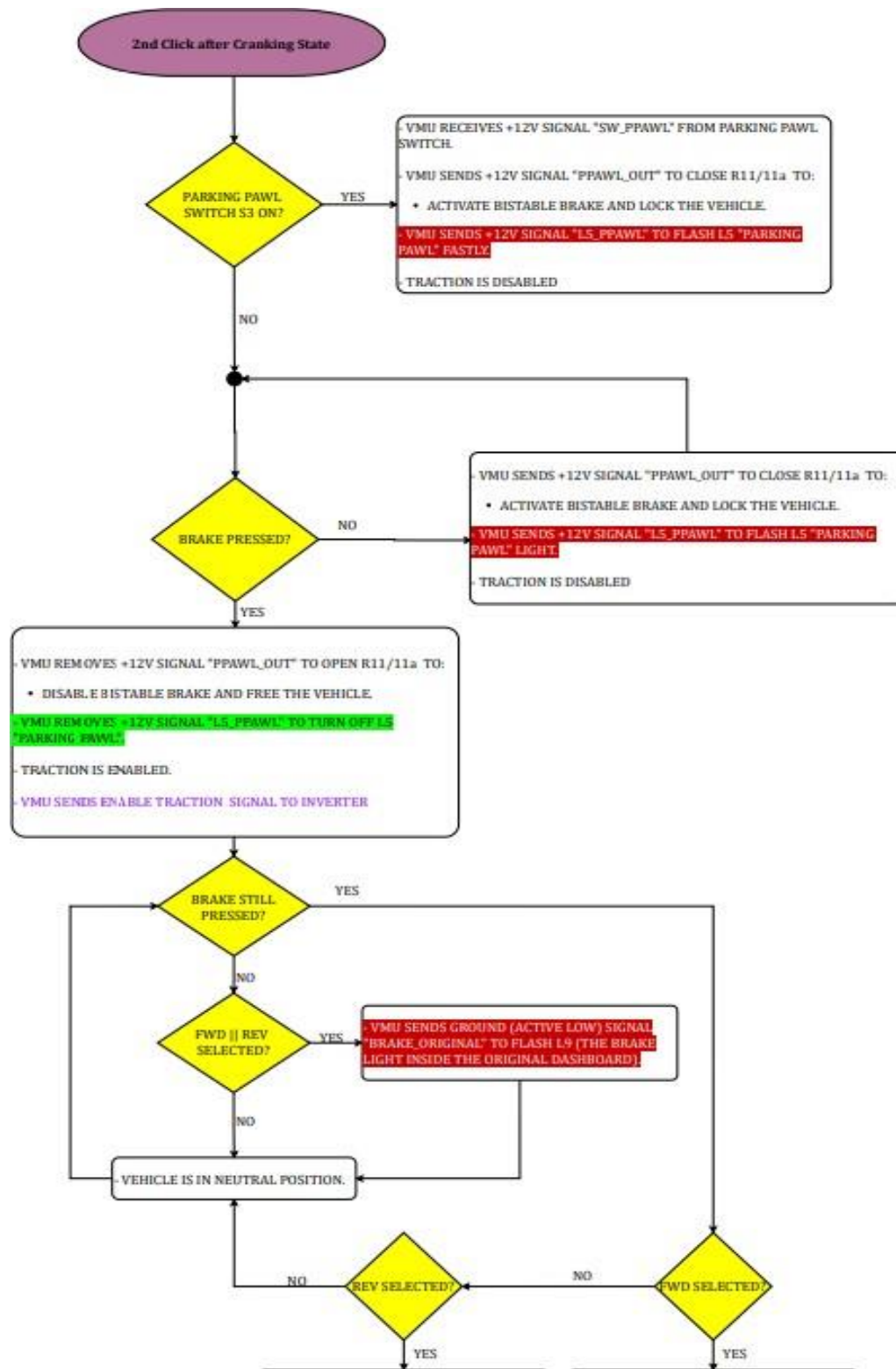


Figure 2. 8: software block diagram of Click After Cranking Mode Part1

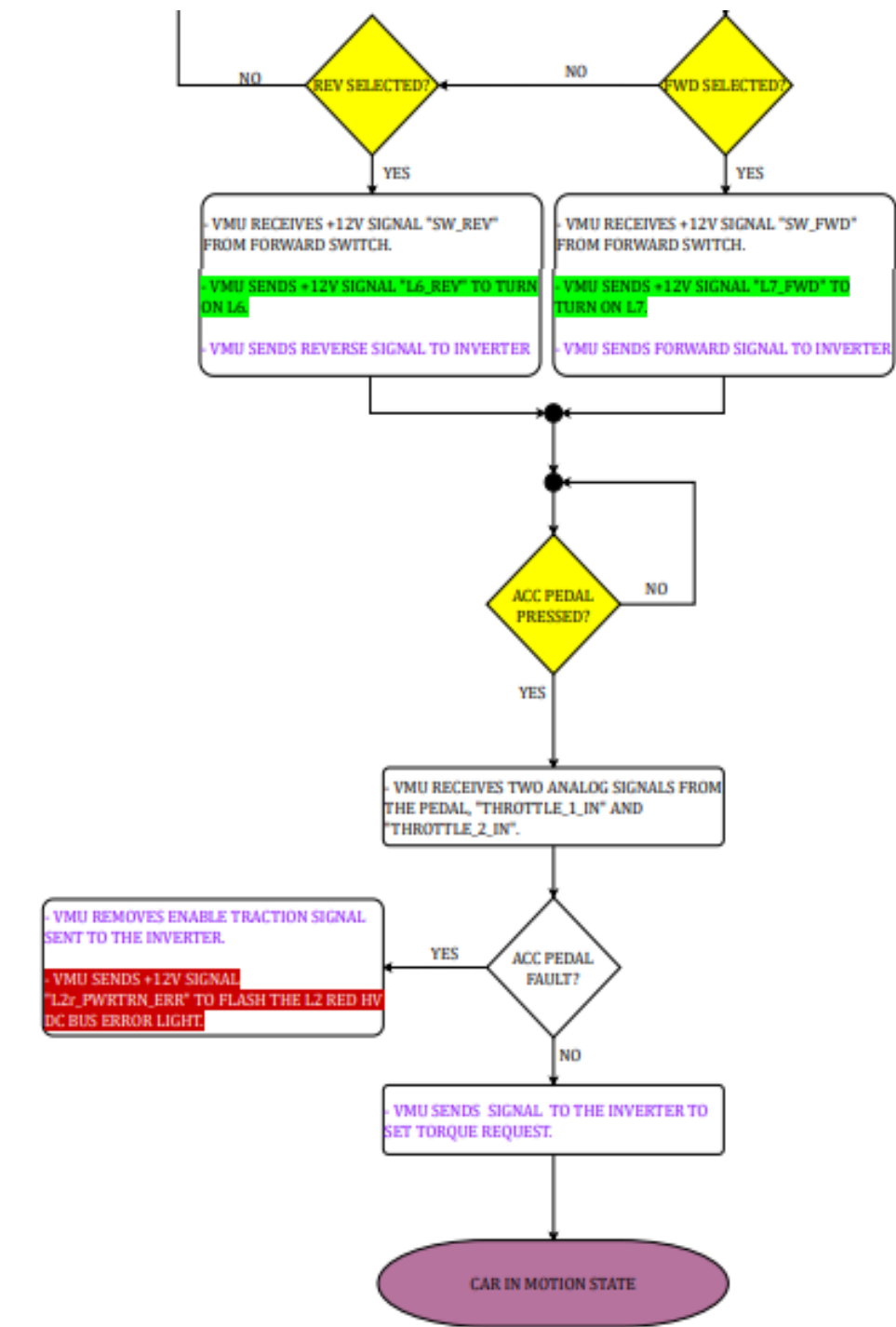


Figure 2. 9: software block diagram of Click After Cranking Mode Part2

2.3.1.5 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.

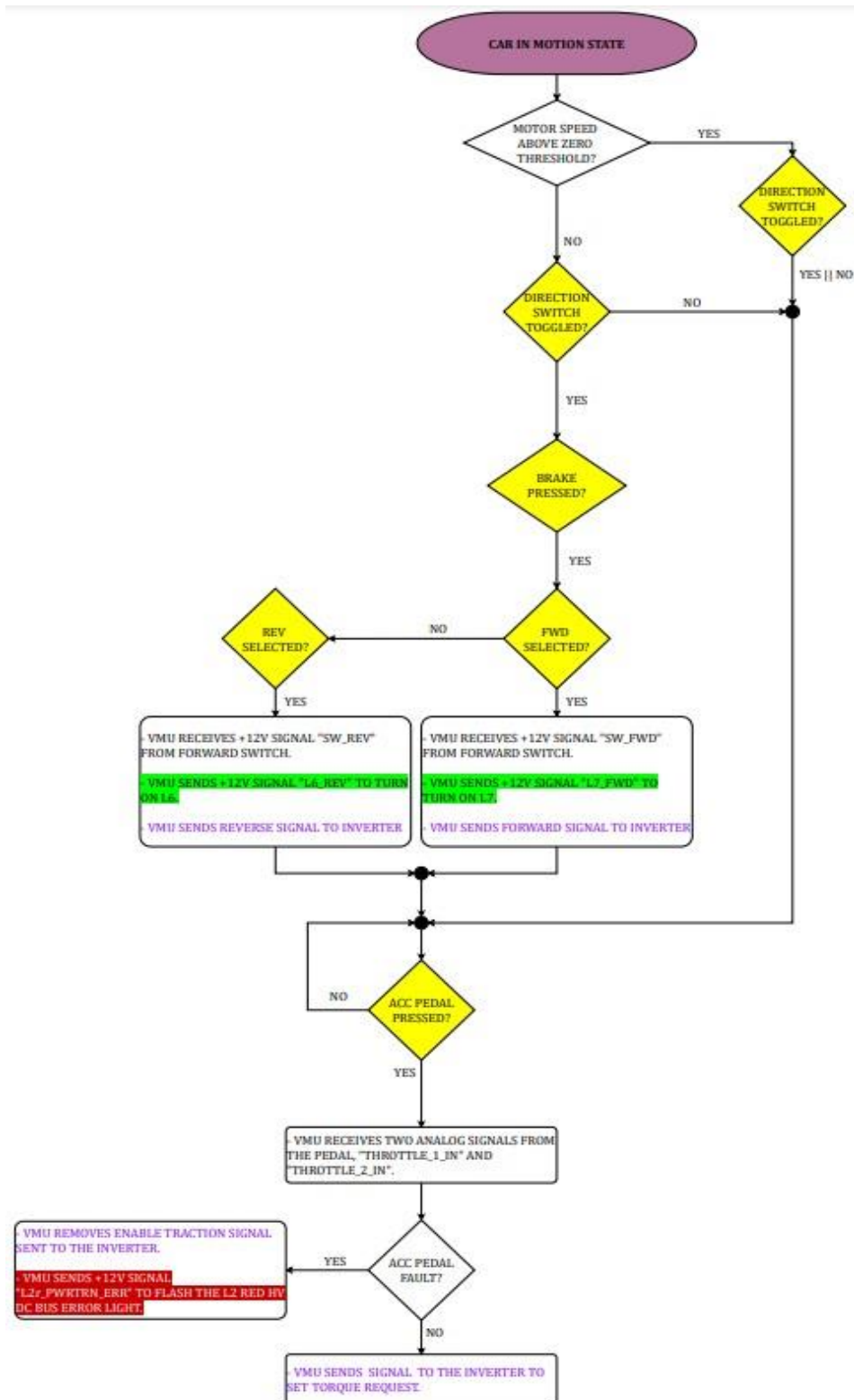


Figure 2. 10: software block diagram of the car in motion state

2.3.1.6 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.

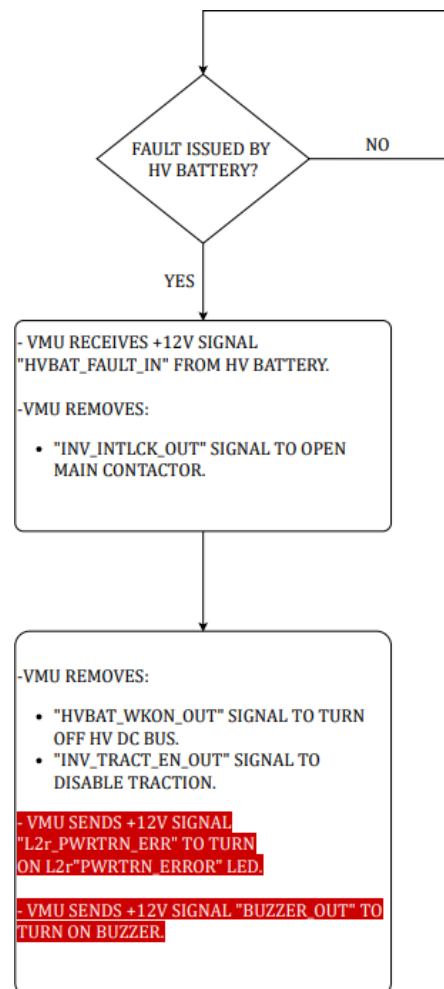


Figure 2. 11: software block diagram of HV battery management

2.3.1.7 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 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.

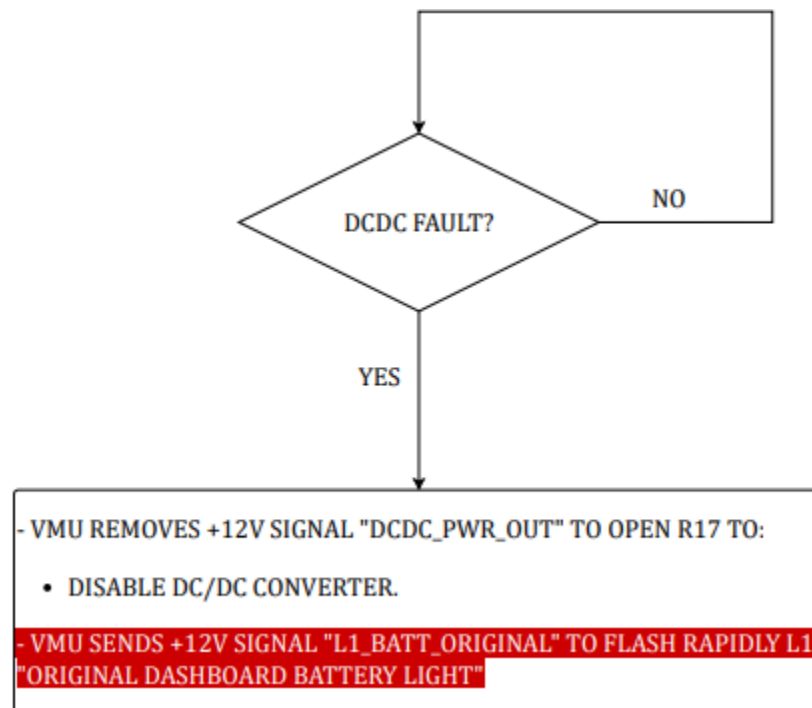


Figure 2. 12: software block diagram of DCDC fault management

2.3.1.8 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.

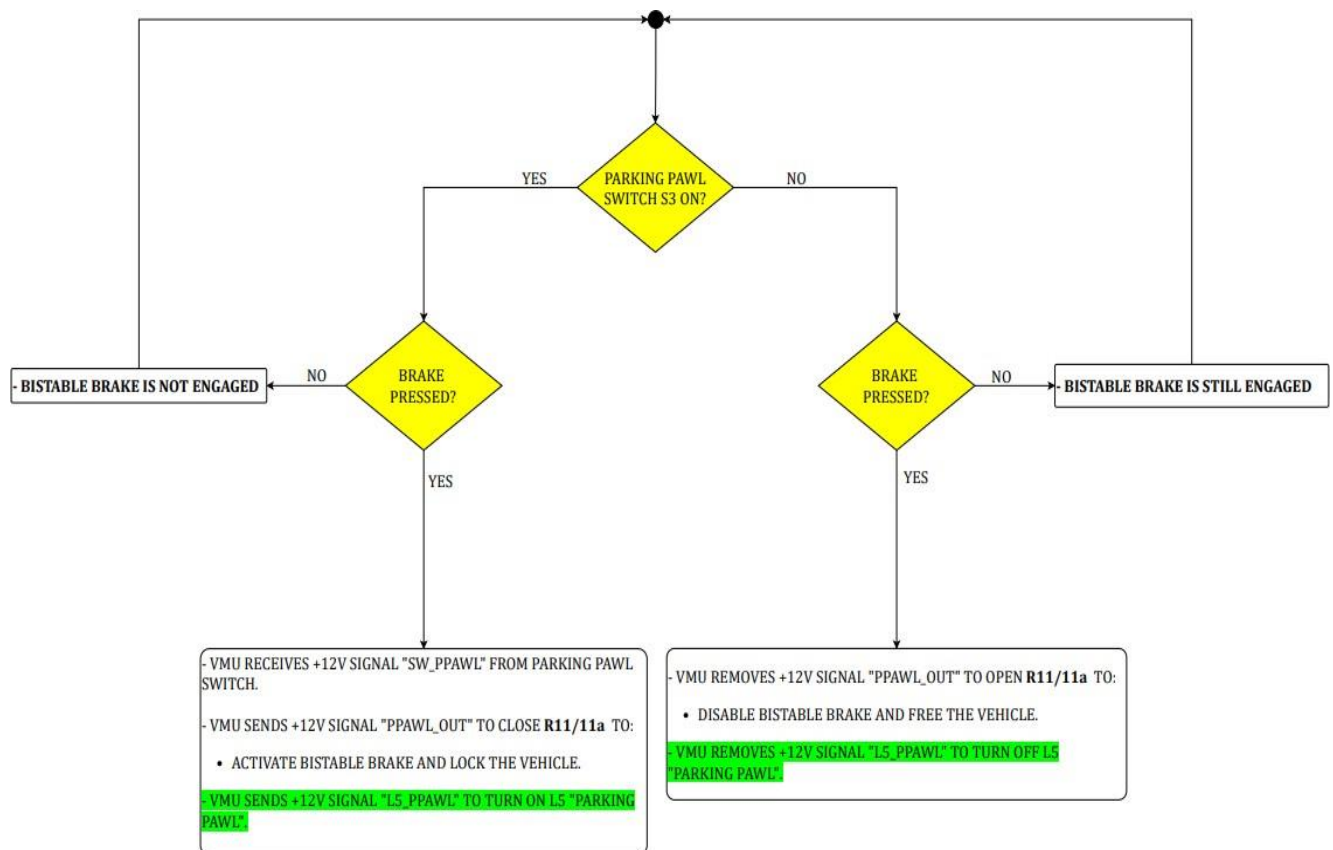


Figure 2. 13: software block diagram of parking management

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 differs 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 must 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 few 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 port, there is a pre-configured example. There's no need to learn howto 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.

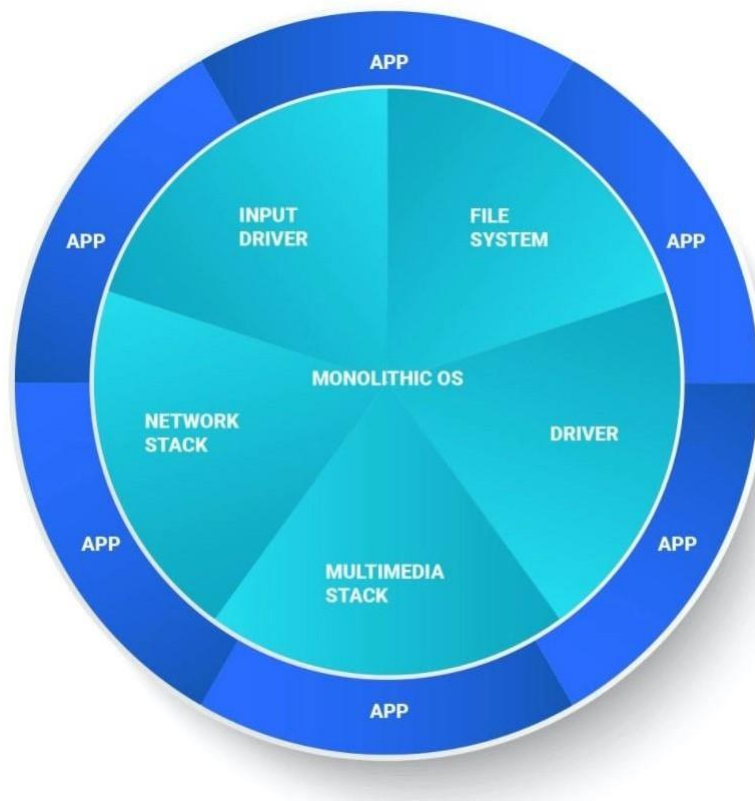


Figure 2. 14: Monolithic operating system architecture diagram

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.

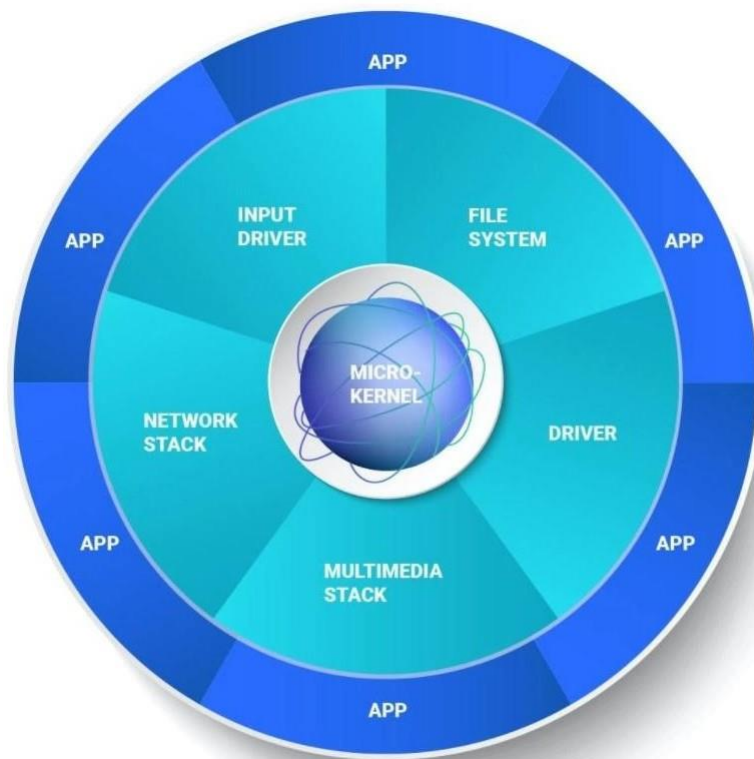


Figure 2. 15: Microkernel operating system architecture diagram

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 because 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);

    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_variable2);
        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 6 - test_can function (1/2)

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 a 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]

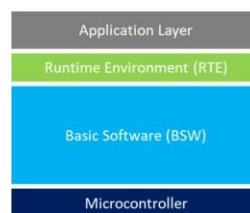


Figure 2. 25: AUTOSAR Layer Architecture

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 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 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 soon. 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 infenion 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 using 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		Watchdog CPU	32-bit TI TMS 570 Dual-core lockstep CPU 180 MHz Safety Companion CPU		
1.6 MB int. Flash 138 kB RAM 32 kB EEPROM	1.6 MB int. Flash 138 kB RAM 32 kB EEPROM		3 MB Flash 256 kB RAM 2 MB ext. RAM 64 kB EEPROM	3 MB Flash 256 kB RAM 2 MB ext. RAM 64 kB EEPROM	3 MB Flash 8 MB ext. Flash 256 kB RAM 2 MB ext. RAM 64 kB EEPROM
1 x CAN	2 x CAN		3 x CAN 1 x LIN	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
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 an 8 Kbyte integrated EEprom.

- Interfaced with 2 can channels which operates on 125Kbit per second up to 1 mbit persecond, 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 output, 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 output, 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.

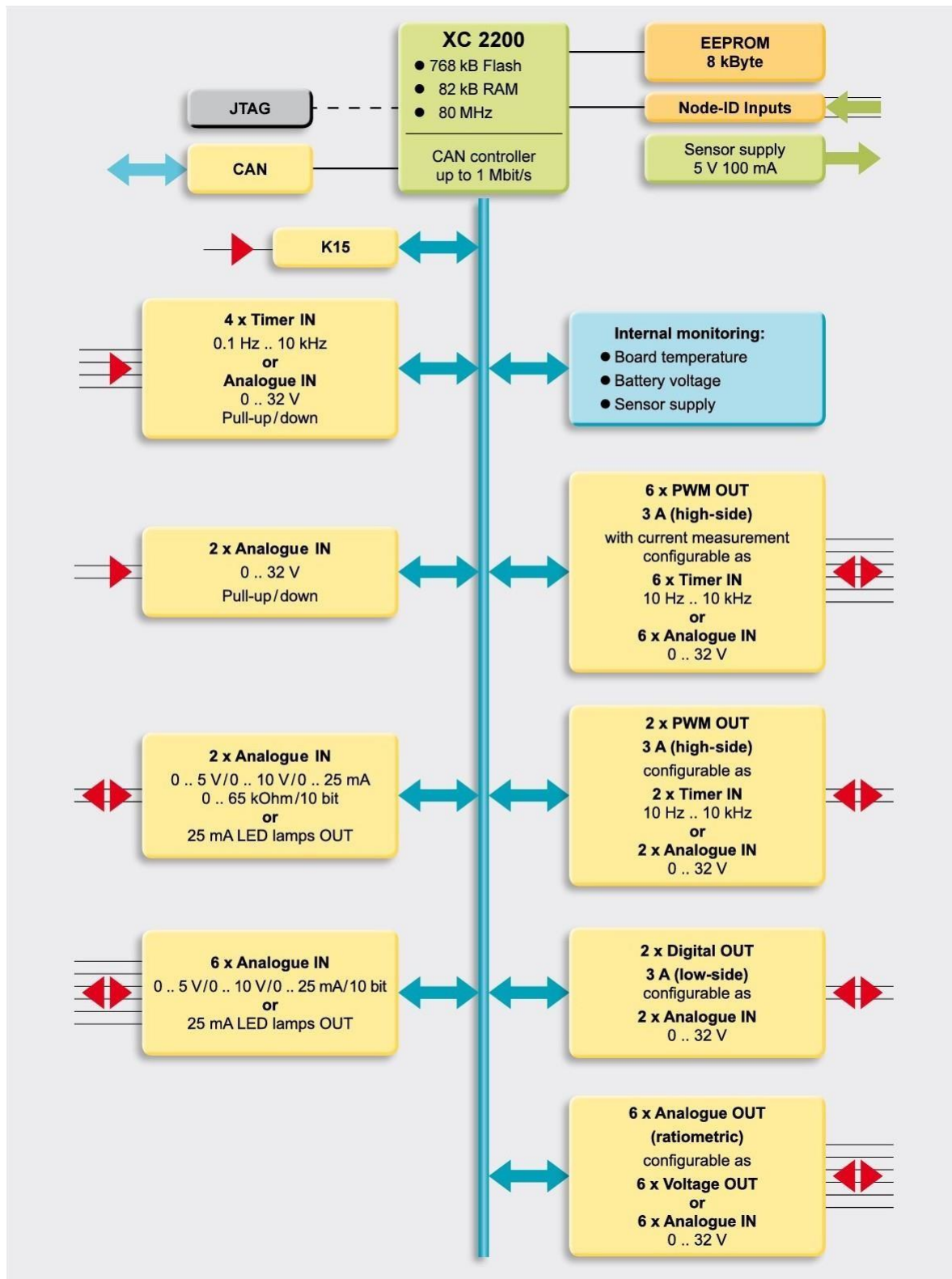


Figure 3. 3: 30-H Block Diagram [22]

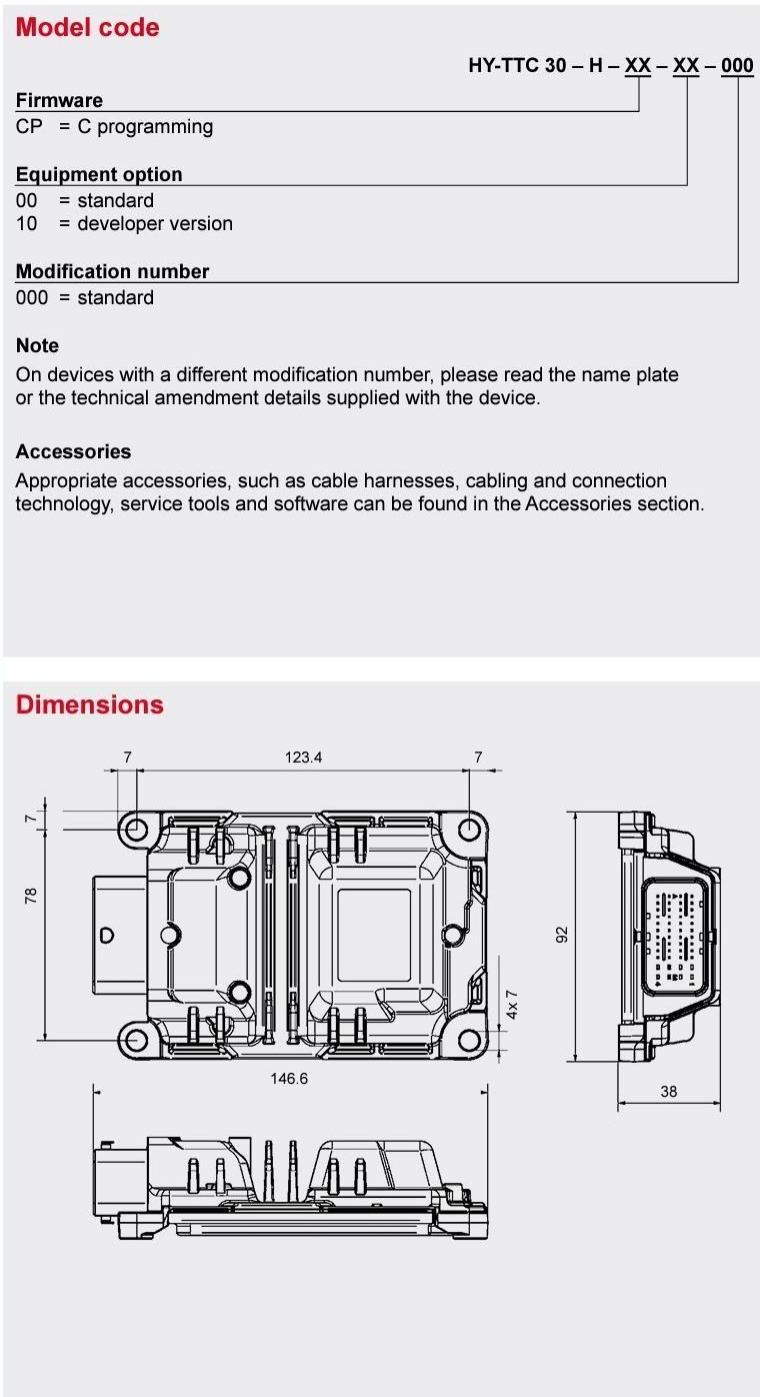


Figure 3. 4: 30-H Model code and Dimensions [22]

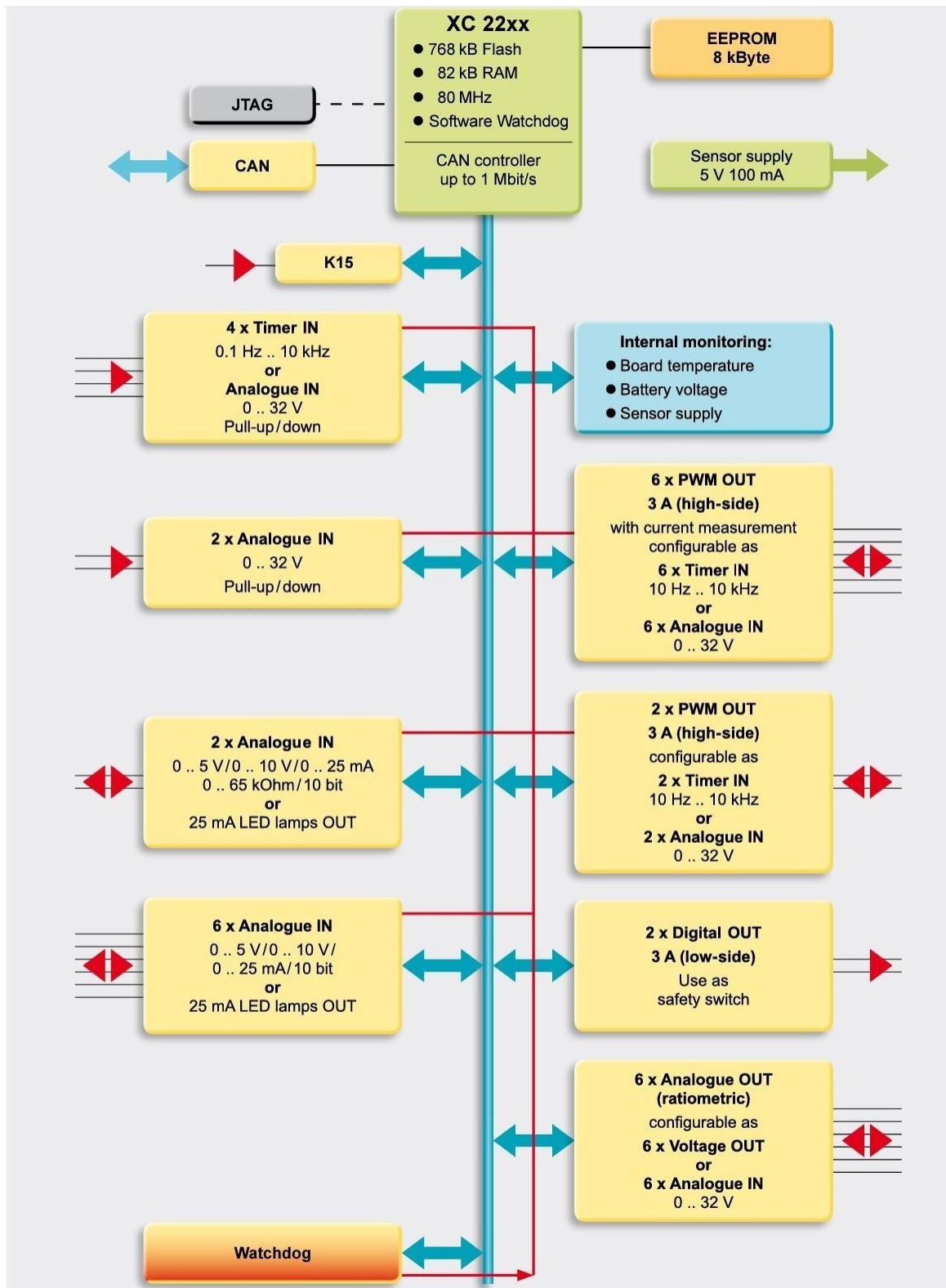


Figure 3. 5: 30S-H Block Diagram [22]

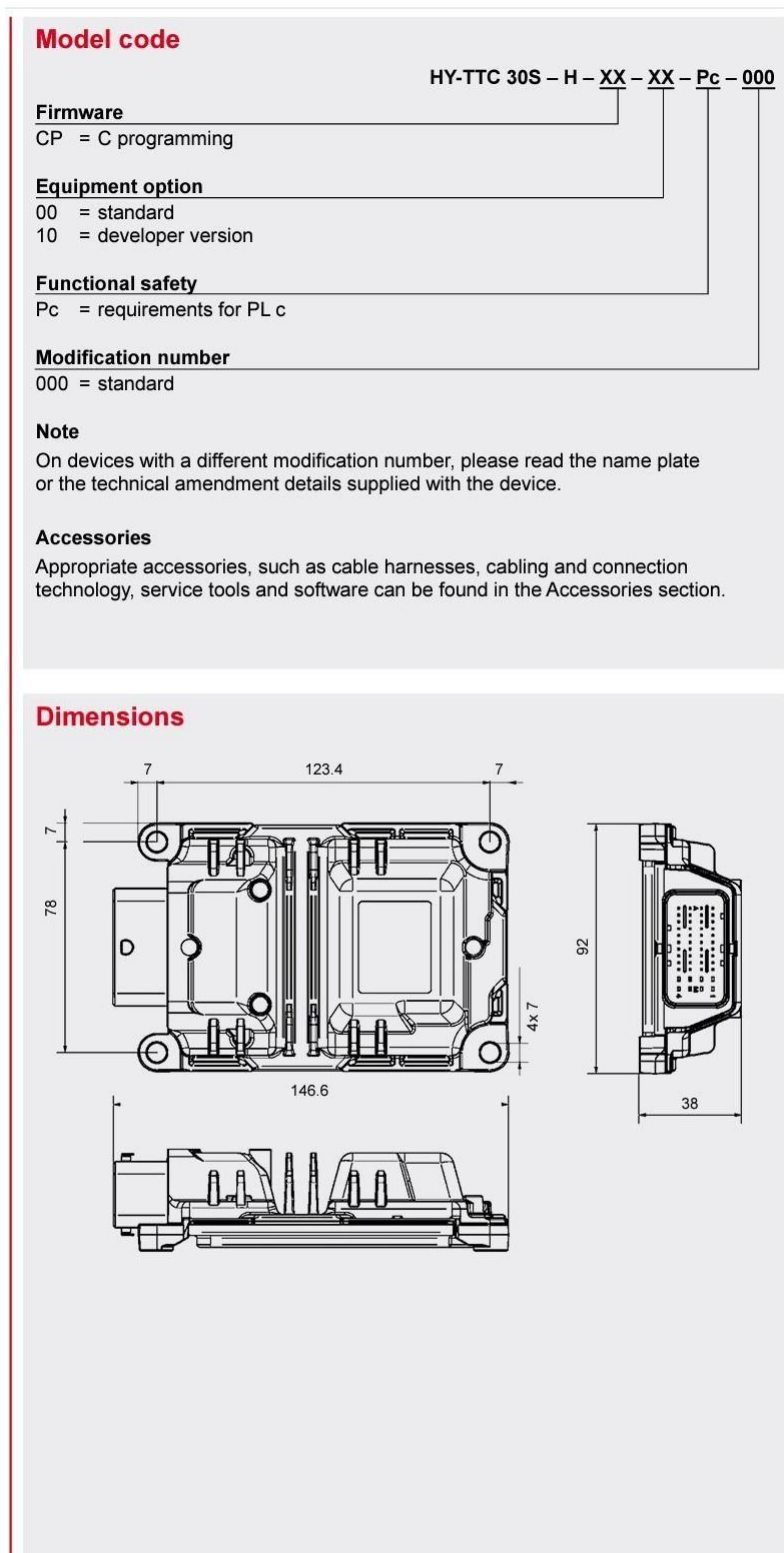


Figure 3. 6: 30S-H Model Code and Dimensions [22]

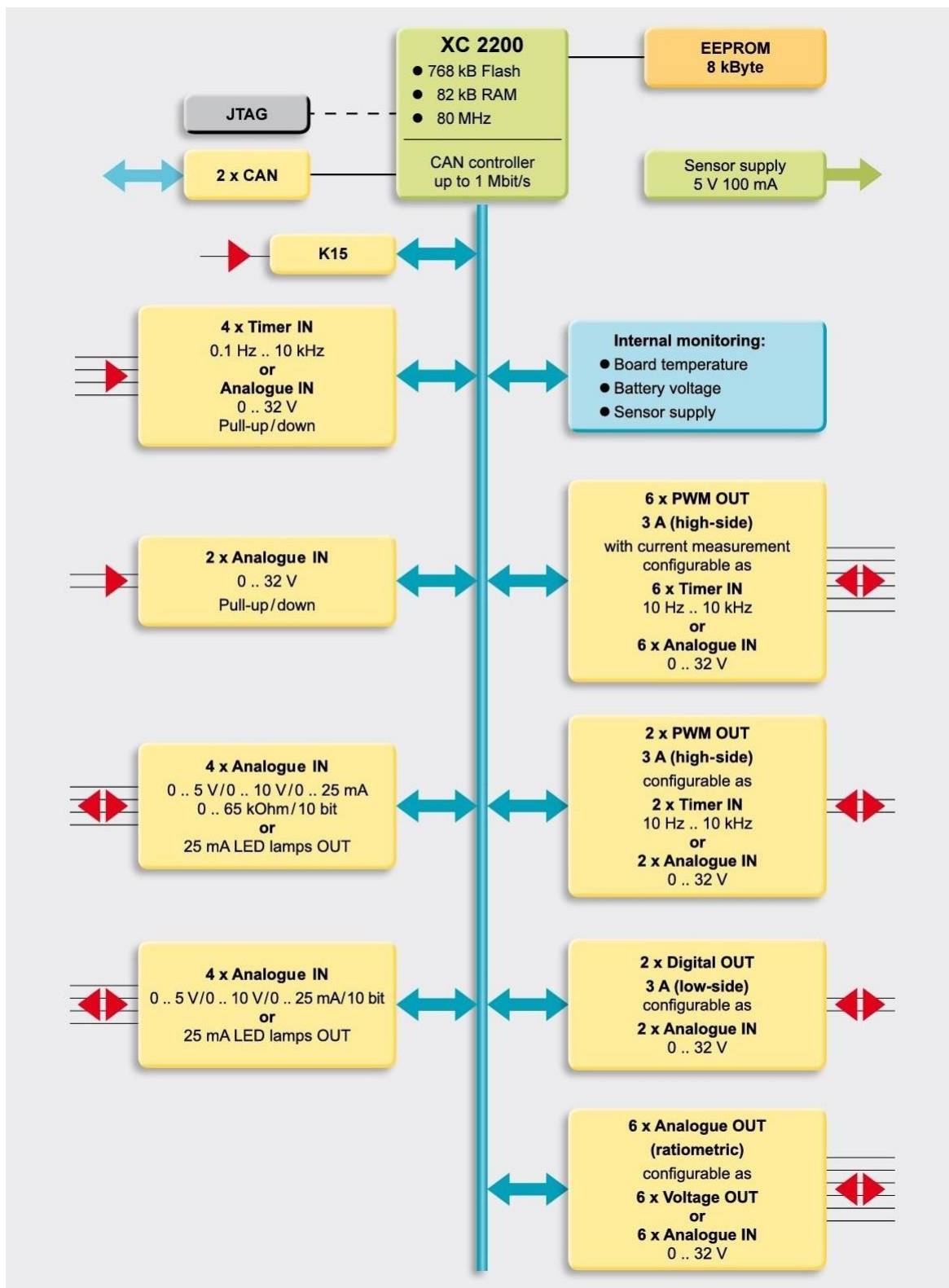


Figure 3. 7: 32 Block Diagram [22]

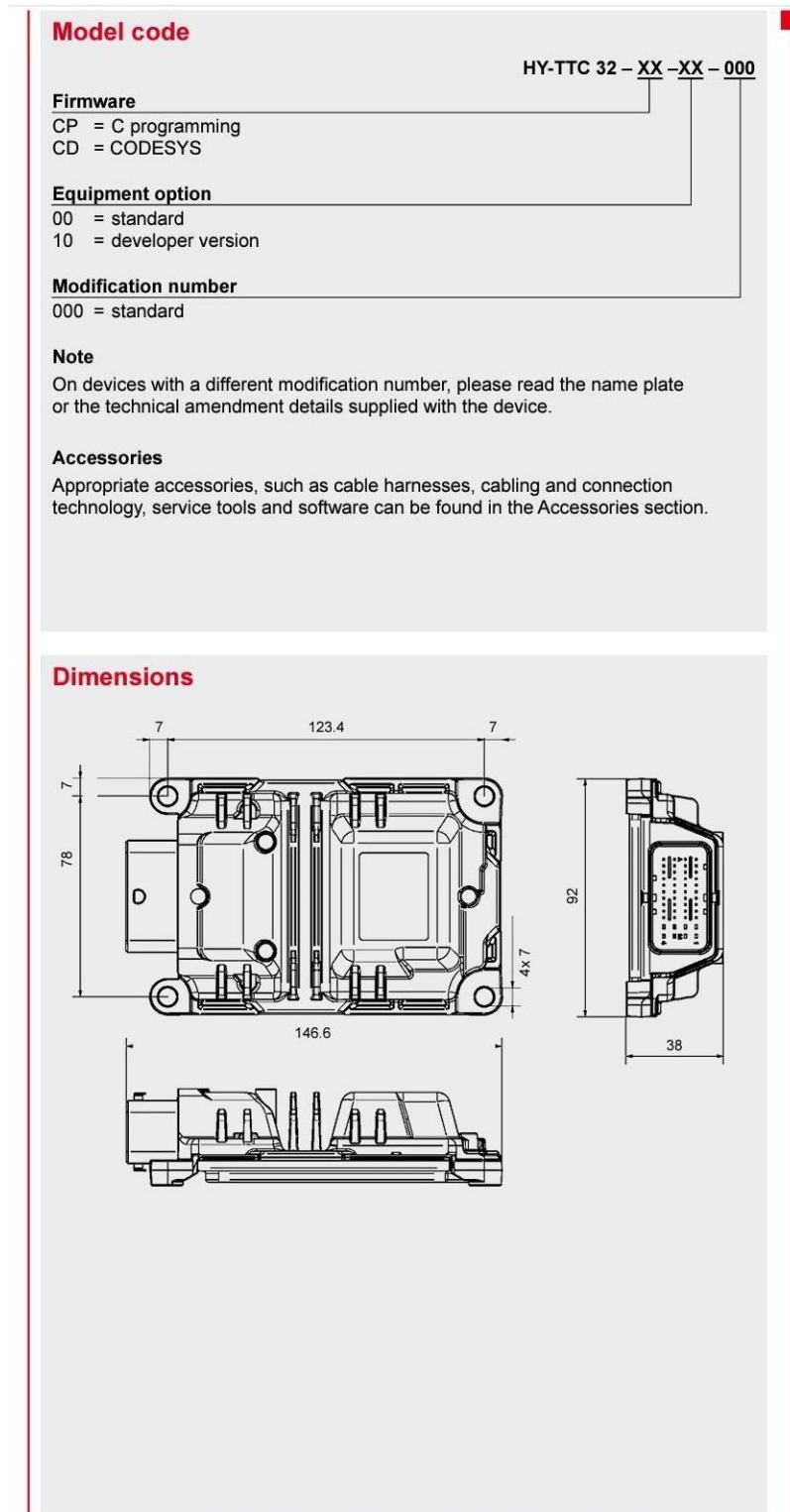


Figure 3. 8: 32 Model Code and Dimensions [22]

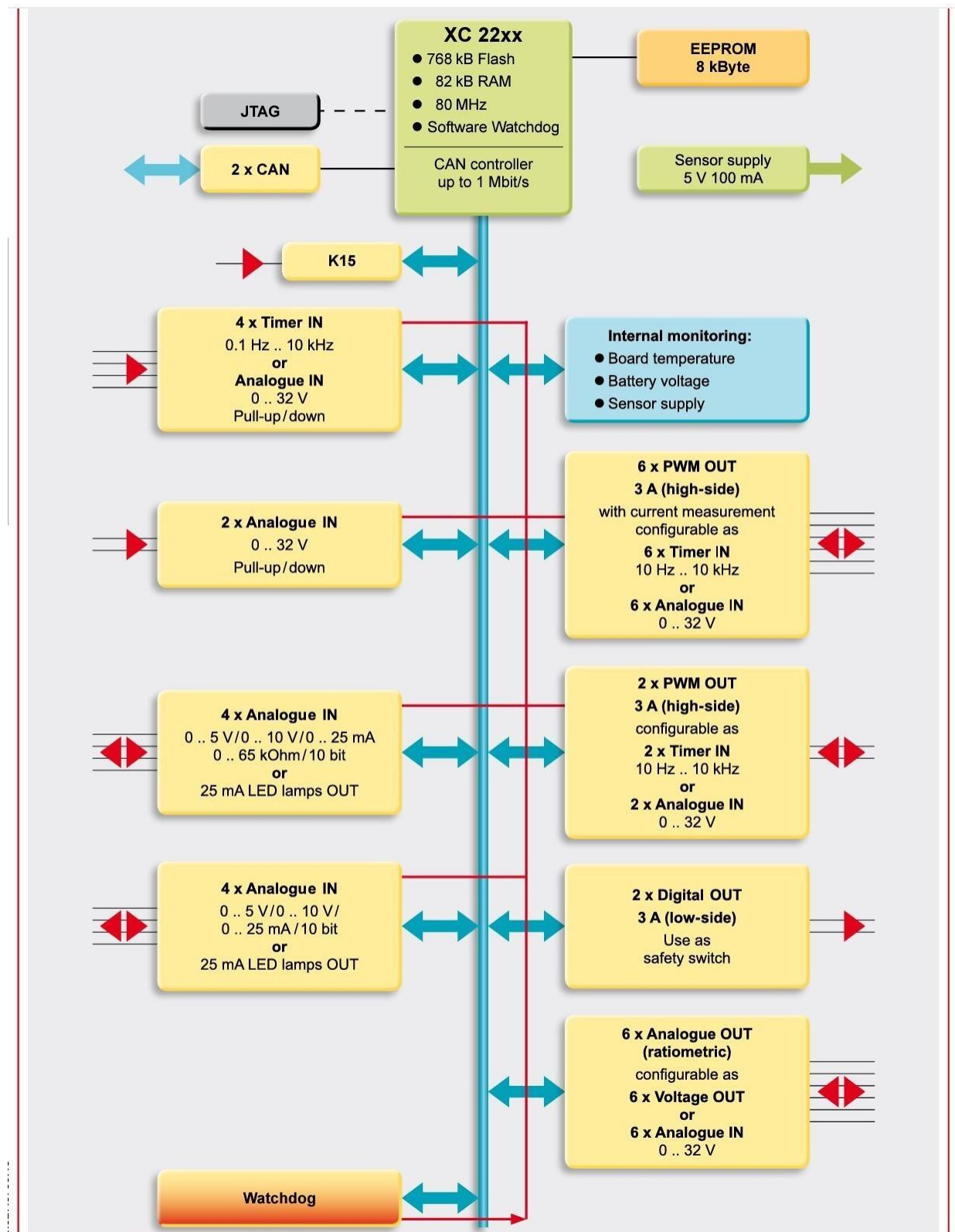


Figure 3. 9: 32S Block Diagram [22]

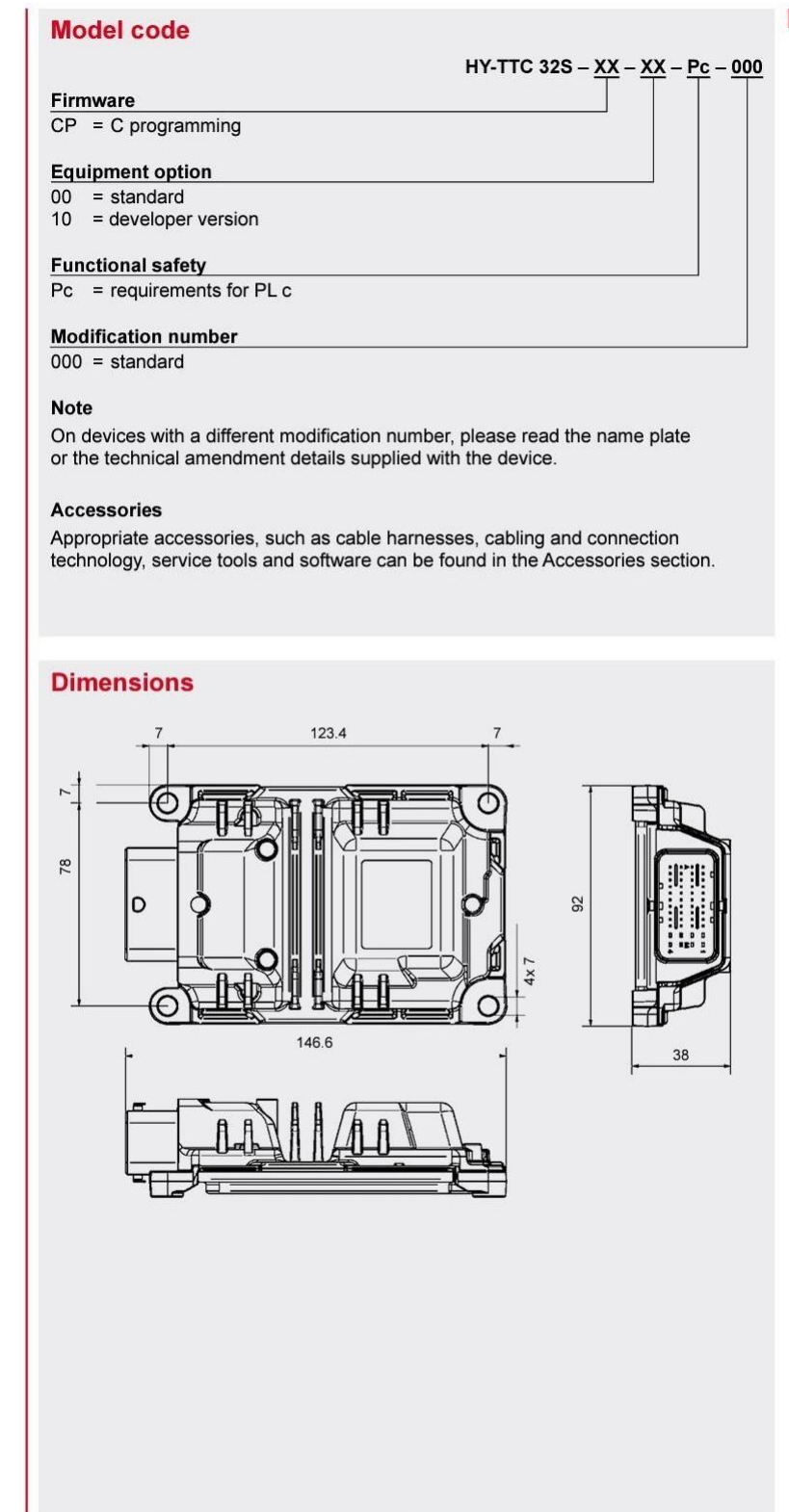


Figure 3. 10: 32S Model Code and Dimensions [22]

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. 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 infenion 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, especially 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 radiometric voltage outputs.
- Robust.
- Waterproof.
- 2 can buses.
- Pl c 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

The main.c file includes the initialization of the peripherals which are used in the tasks and initialize the real time operation system threads.

Main file is starting the real time operation system and initialize the all necessary peripherals with setting the starting values on them.

```
#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 dlmMulticastIP[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;
static IO_ErrorType adc_init_rc;
```

```
static IO_ErrorType pwm_init_00;
static IO_ErrorType pwm_init_01;
static IO_ErrorType pwm_init_02;
static IO_ErrorType pwm_init_03;
static IO_ErrorType pwm_init_04;
static IO_ErrorType pwm_init_05;
```

```

//task1
static ubyte2 pwm_duty_cycle_acce;
static ubyte4 pwm_duty_cycle_fb_acce;

static ubyte2 pwm_duty_cycle_freno;
static ubyte4 pwm_duty_cycle_fb_freno;

//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_fb_val_acce;
    ubyte4 pwm_duty_fb_val_freno;
    ubyte4 pwm_duty_fb_val_inv;

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

    driver_init_rc = IO_Driver_Init( IO_DRIVER_MODE_DEFAULT, NULL );
    /* set watchdog timeout */
    #ifndef DEBUG
        IO_WDTimer_Init(watchdog_timeout);

```

```
#endif
/* driver RTC initialization */
while (IO_E_OK != IO_RTC_Init())
{

}

/* driver EEPROM initialitation */
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
 */

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

// pwm output safety configuration

IO_PWM_SAFETY_CONF _io_pwm_safety_conf =
{
    IO_SAFETY_SWITCH_0,
    80,
    100,
    200
};
```

```
//pwm safety configuration
```

```
IO_PWM_SAFETY_CONF pwm_safety_cfg = {
    IO_SAFETY_SWITCH_1 // IO_PIN safety_switch
    , 50                // margin_lower_lim
    , 150               // margin_upper_lim
    , 200               // duty_cycle_tolerance
};
```

```
// initialize PWM output as safety-critical -TASK1
```

```
pwm_init_00 = IO_PWM_Init (
    IO_PWM_00 // 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 -TASK1
```

```
pwm_init_01 = IO_PWM_Init (
    IO_PWM_01 // 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 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_rc_acce = IO_PWM_SetDuty (
    IO_PWM_PIN_ACCE
    , pwm_duty_cycle_acce
    , &pwm_duty_fb_val_acce );

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

pwm_set_rc_freno = IO_PWM_SetDuty (
    IO_PWM_PIN_BREAK
    , pwm_duty_cycle_freno
    , &pwm_duty_fb_val_freno );

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

//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_DO_Init( IO_DO_00, 2500 ); //task1

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
```

```
#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 driver initialization
IO_UART_Init             // UART driver initialization
IO_ADC_ChannelInit       //ADC driver initialization
IO_PWM_Init              // PWM driver initialization

```

4.2 Acceleration and Brake Pedal Management Task

4.2.1 Task Architecture

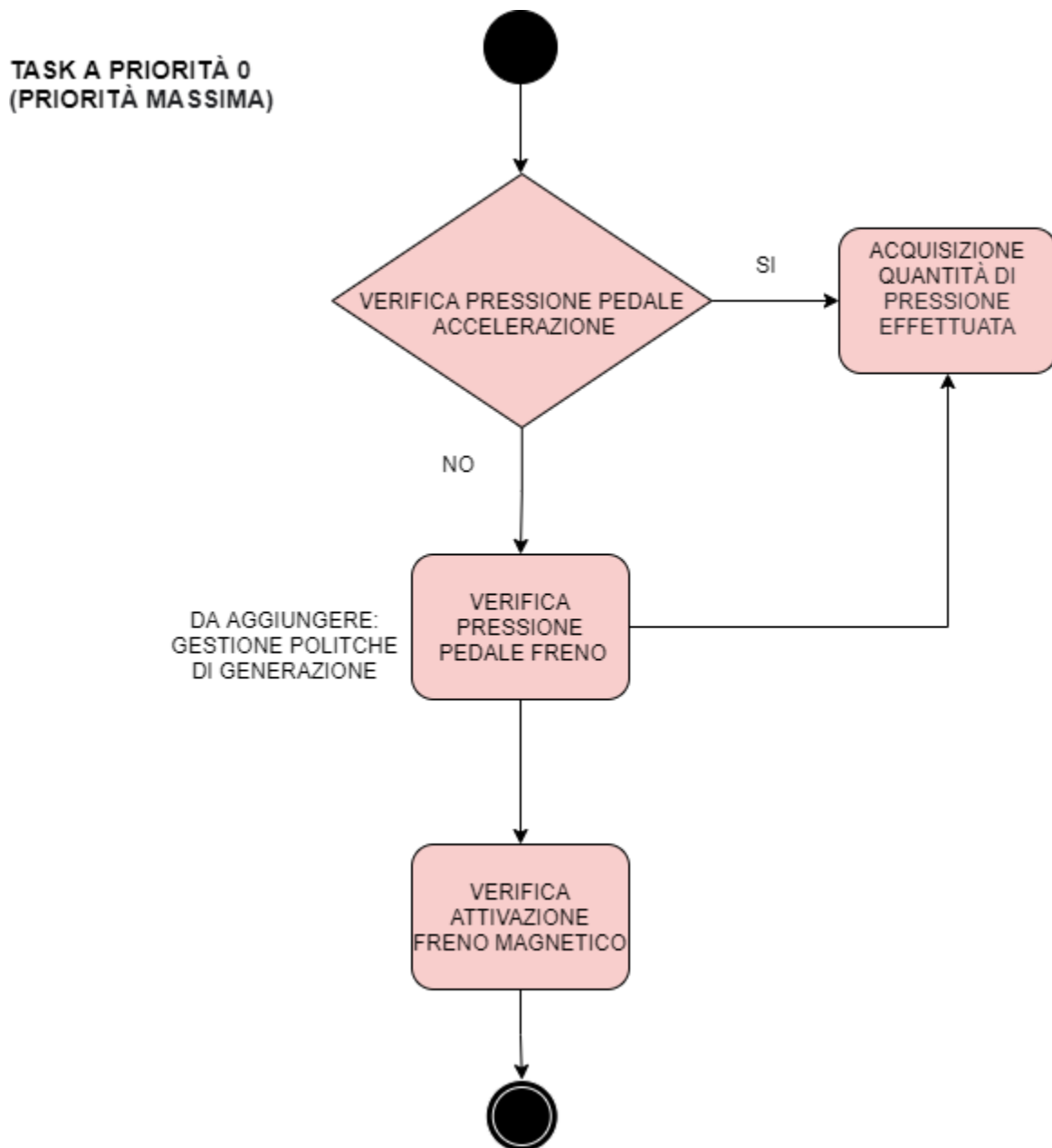


Figure 4. 1: Pedals management task architecture flow chart

The main goal of this task is creation of the software algorithm of the pedals which includes both accelerator and brake pedals.

This task has the maximum priority in the real time operation system which priority number is equal to 0. In our case the maximum priority is equal to minimum number and the minimum priority is equal to maximum value in a reverse logic.

The gas pedal is often referred to as the accelerator pedal. This pedal regulates the quantity of gas supplied into the engine and, as a result, controls speed of the car.

Electromagnetic brakes or electro-mechanical brakes use electromagnetic force to apply mechanical resistance, or friction, to slow or stop motion.

The concept of braking includes the conversion of mechanical energy to thermal energy. Whenever the brake pedal is pressed, a stopping force many times as strong as the force that sets the automobile or car in motion is triggered, and the resulting kinetic energy is absorbed as heat. Independent of the car's velocity, the brakes slow the vehicle down in a very short amount of time.

The software architecture and development algorithm of the task is as follows:

1. Verification of the acceleration pedal activation.
 - If a non-zero current value is received, this implies that there is some force on the gas pedal; thus, measure the pedal level.
 - If there is no current is returned that meaning there is no any pressure in the gas pedal and in that case we can move on with checking the brake pedal.
2. Verification of the break pedal activation.
 - If a non-zero current value is received, this implies that there is some force on the brake pedal; thus, activate the brake is under pressure.
 - If there is no current is returned that meaning there is no any pressure in the brake pedal.
3. Activation of the electromagnetic brake
 - If there is a pressure in the brake pedal then we need to activate the electromagnetic brake in the car as a last step of the algorithm.

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 Acceleration and Brake Pedal Management Code Implementation

4.2.2.1 Acceleration and Brake Pedals Management .c Code

```
/*  
 * task_1_gestione_pedali.c  
 *  
 * Created on 04/Jun/2021  
 * Author: Celal  
 */  
  
#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"  
#include "IO_PWM.h"  
#include "IO_DIO.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_set_rc_acce;  
static IO_ErrorType pwm_get_rc_acce;  
static IO_ErrorType do_set_rc;  
  
static IO_ErrorType pwm_set_rc_freno;  
static IO_ErrorType pwm_get_rc_freno;  
  
static ubyte2 pwm_current_fb_acce;  
  
static ubyte2 pwm_current_fb_freno;
```

```

static ubyte2 do_voltage_fb;

static ubyte1 diag_state;
static DIAG_ERRORCODE diag_error;

pwm_current_fb_acce = 0;
pwm_current_fb_freno = 0;

void task_1_gestione_pedali( )
{
    driver_task_begin_rc = IO_Driver_TaskBegin ();
    IO_ErrorType eeprom_error=512;
    ubyte4 task_1_timestamp;

    bool pwm_fresh_acce;
    ubyte2 pwm_current_fb_val_acce;

    bool pwm_fresh_freno;
    ubyte2 pwm_current_fb_val_freno;

    (void) IO_RTC_StartTime (&task_1_timestamp);
    (void) IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION,
ID_TASK_EXECUTION_LENHT, 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

    /*
    **      VERIFICA PRESSIONE PEDALE ACCELERAZIONE
    */

    // set the PWM output to the specified duty cycle

```

```
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 = pwm_current_fb_val_acce;
#ifdef DEBUG
    UART_Printf (IO_UART, "Acceleration Pedal pressure value: %u \n\r",
pwm_current_fb_acce);
#endif

}

}while(pwm_get_rc_acce == IO_E_OK);


//VERIFICA PRESSIONE PEDALE FRENO
pwm_current_fb_val_freno = TRUE;

while (pwm_current_fb_val_freno == TRUE){

// read the current feedback value
pwm_get_rc_freno = IO_PWM_GetCur (
    IO_PWM_PIN_BREAK
    , &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_current_fb_freno = pwm_current_fb_val_freno;

#ifdef DEBUG
    UART_Printf (IO_UART, "FRENO Pedal pressure value: %u \n\r", pwm_current_fb_freno);
```



```

    #endif

    //VERIFICA ATTIVAZIONE FRENO MAGNETICO

    // set digital output value for magnetic break

    do_set_rc = IO_DO_Set (IO_DO_PIN_MAGNETIC_BREAK
        , TRUE
        , &do_voltage_fb );

    if (do_set_rc != IO_E_OK)
    {

        #ifndef DEBUG
            UART_Printf (IO_UART, "Magnetic Break Error: %u mV\n\r", do_set_rc);
        #endif

    }

}

}

// retrieve diagnostic information
(void) DIAG_Status (
    &diag_state
    , &diag_error );

while (TRUE != Check_Task_End(task_1_timestamp, (ubyte4)TASK1_CYCLE_TIME))
{
    IO_RTC_PeriodicDeInit();
    #ifndef DEBUG
        UART_Printf (IO_UART, "\n\r wait for task 1 terminating %u !\n\r", task_1_timestamp);
    /*serve watchdog */
    #endif
    /*serve watchdog */
    #ifndef DEBUG
        while (IO_E_OK != IO_WDTimer_Service());
    #endif
}
/*serve watchdog */
#endif

```

```
while (IO_E_OK! =IO_WDTimer_Service());  
#endif  
driver_task_end_rc = IO_Driver_TaskEnd ();  
}
```

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

During the implementation different peripherals are used for the receiving information from the pedals and the electromagnetic brake.

For the acceleration and brake pedal we used PWM to communicate between pedal and the control unit. Electromagnetic brake is connected to the one of the digital input-output pin of the control unit.

Pulse Width Modulation is a method for controlling analog components with a digital signal. Alternately stated, an electronic product such as an MCU may emit a modulating signal to control an analog component.

It is one of the principal ways in which microcontrollers operate analog devices such as controllable motors, adjustable leds, controllers, and amplifiers. PWM signal is not analogue, though.

Digital Input/Output, or DIO, is a basic interface used in a broad variety of systems to successfully transmit digital signals from sensors, transducers, and mechanical components to other electrical electronic devices and circuits.

So for the PWM implementation and the communication we needed to use 3 different PWM API from HY-TTC30 family. These are `IO_PWM_Init`, `IO_PWM_SetDuty` and `IO_PWM_GetCur` functions.

1. **IO_PWM_Init** is basically setup the single PWM output pin of the board. We need 2 PWM pin initialized because one pin will be used for the acceleration pedal and the other one is used for the brake pedal. When we set the PWM pin API requires to set some parameters and returns the error codes.

The diagnostic margin should be configured to TRUE if a PWM channel is started as safety and a proper safety configuration is provided. Otherwise, startup will fail and the `IO_E_INVALID_PARAMETER` error code will be generated.

When any PWM channel of a frequency group is established, the other channels of this group may only be initialized with the same rate. The method returns `IO_E_GROUP_CONFLICT` otherwise.

The duty cycle cannot reach the diagnostic limit of 100us (lowest limit) and 250us (highest boundary) when diag margin is TRUE. This is an crucial phase for hydraulic circuits. If the diag margin argument is FALSE, no duty cycle range margin is used.

Parameters

- **pwm_channel** is used to choose which pwm channel will be used for the task. There are 12 pwm channel available to use in our board.
- **frequency** Pwm frequency should be set between 15Hz and 1000Hz, and only frequencies with a period of an integral multiple of 1ms are possible.
- **polarity** sets polarity of the output signal and we have TRUE or FALSE option for this parameter. True means that the margin will be on and False means that no margin will be applied in this time.
- **diag_margin** This parameter indicates if a margin should be applied or not and there are again 2 available inputs which are TRUE or FALSE.
- **overload_limit** it sets a current limitation in mA and whenever we have the current more than this limit `IO_E_PROT_USER_OVERLOAD` error flag will be accured. This value can be set to zero then no error will be report means this property is disabled. We can set to between 1mA-2999mA which means error will be accured in case of current value is higher than the set value. In case of the measured value exceeds 3000mA then other error code will be reported.
- **safety_conf** it is a safety configuration parameter.

Returns Values

IO_ErrorType will be returned after API is called by the task.

- `IO_E_OK` means that everything is fine and PWM pin is successfully set and ready to operate.
- `IO_E_CHANNEL_BUSY` means that Pulse width modulation channel or timer input channel in use by other task
- `IO_E_INVALID_CHANNEL_ID` means that the written pwm channel is does not exist
- `IO_E_INVALID_PARAMETER` means that given parameter is out of range
-

- **IO_E_CH_CAPABILITY** means that the capability of this pwm channel has not been activated.
 - **IO_E_DRIVER_NOT_INITIALIZED** means that the common driver init function has not been called before
 - **IO_E_SW_INTERNAL** means that there was an internal sw error when configuration tables are handled or index is out of the range.
 - **IO_E_GROUP_CONFLICT** means that there is a conflict when the configuration of the output is processed.
 - **IO_E_INVALID_SAFETY_CONFIG** means that one or more parameters of the safety configuration are not valid.
 - **IO_E_SAFETY_NOT_SUPPORTED** means that this channel does not support the safety properties
 - **IO_E_DRV_SAFETY_CONF_NOT_CONFIG** means that the driver is not initialized as a safety device and does not available in this channel
 - **IO_E_TASK_NO_FREE_SLOTS** indicates that there is no more available slots to setup task function.
2. **IO_PWM_SetDuty** is used to set the duty cycle of the used pwm channel. The duty cycle cannot reach the diagnostic limit of 100us (lowest limit) and 250us (highest boundary) when diag margin is TRUE. This is a crucial phase for hydraulic circuits. If the diag margin argument is FALSE, no duty ratio range tolerance is used.

We also need to consider as an important note that **IO_POWER_Set** should be activated for Output pins. If not, the ports will stay disabled.

Another important note is that PWM ports set with frequencies more than 250Hz can only provide 100mA of constant current.

Parameters

- **pwm_channel** It is one of the 12 channels.
- **duty_cycle** Should be arranged between the value of 0-65535
- **duty_cycle_fb** feedback value for the arranged channel, it is optional

Returns Values

IO_ErrorType will be returned after API is called by the task.

- **IO_E_OK** Indicates that everything is okay.
- **IO_E_CHANNEL_NOT_CONFIGURED** An IO controller task function will return this error if the channel has not been established. Calling the corresponding Init method initializes the channel.
- **IO_E_INVALID_CHANNEL_ID** Returned if a port ID that does not exist was supplied to the function.
- **IO_E_CH_CAPABILITY** This error can be occurred when specified pin is does not provide

the requested duty.

- **IO_E_PWM_OPEN_LOAD_OR_SHORT_BATTERY** If the fault persists after 50ms, based on the output pin's operating voltage
- **IO_E_PWM_OPEN_LOAD** returned if the voltage is in open load range
- **IO_E_PWM_SHORT_CIRCUIT** This exception is generated if the output signal cannot be monitored via the timer return and the value on the response channel is low.
- **IO_E_PWM_SHORT_BATTERY** Short Circuit is detected in the battery.
- **IO_E_PWM_OUTPUT_DISABLED** when the pwm outputs are disabled then this error will be occurred.
- **IO_E_PWM_CHANNEL_STARTUP** means that PWM output is not in the startup phase
- **IO_E_PWM_OUTPUT_STARTUP_ERROR** there is an error on the channel mode
- **IO_E_ADC_INVALID** There is an error in the ADC
- **IO_E_SW_INTERNAL** Configuration table gives an error in the internal software
- **IO_E_PWM_CAPTURE_ERROR** This failure may arise if two edges of the feedback signal are too close together and the inner timer is no longer able to monitor the time change.
- **IO_E_PROT_USER_OVERLOAD** The value of output current is more than the initialization-configured threshold.
- **IO_E_PROT_TEMP_OVERLOAD** Meaning that output current is higher than 3A.

3. **IO_PWM_GetCur** returns the measured current from the arranged channel pin.

Parameters

- **pwm_channel** It is the arrangement of channel.
- **Current measured current value**
- **Fresh Provide** is the value is new or the same as before and indicates TRUE or FALSE

Returns Values

IO_ErrorType will be returned after API is called by the task. Current measurements for all PWM channels are equally spaced, indicating that sampling occurs synchronously with the PWM period. Every 1 millisecond, the current value is collected. The collected current values will be averaged across one cycle of the PWM signal before being sent to the software. Returns are the same as explained in details before.

- **IO_E_OK** Everything fine
- **IO_E_CHANNEL_NOT_CONFIGURED**
- **IO_E_INVALID_CHANNEL_ID**
- **IO_E_ADC_INVALID** An ADC
- **IO_E_PWM_CURRENT_INACCURATE**
- **IO_E_CH_CAPABILITY**
- **IO_E_NULL_POINTER**

Acceleration pedal important variables will be explained in the next. **pwm_get_rc_acce** is the return value

of the `IO_PWM_GetCur` function for the gas pedal and only if it returns the code of `IO_E_OK` then we are updating the feedback value of the acceleration which the variable of `pwm_current_fb_acce`. This process will be working while returned code is always indication of everything is fine with the system. Same algorithm is used for the brake pedal, `pwm_get_rc_freno` variable is used for the return value. Lastly digital output is set for the activation of the electromagnetic brake and the return variable is `do_set_rc`.

4.3 Battery Key management Task

4.3.1 Task Architecture

TASK A PRIORITÀ 1

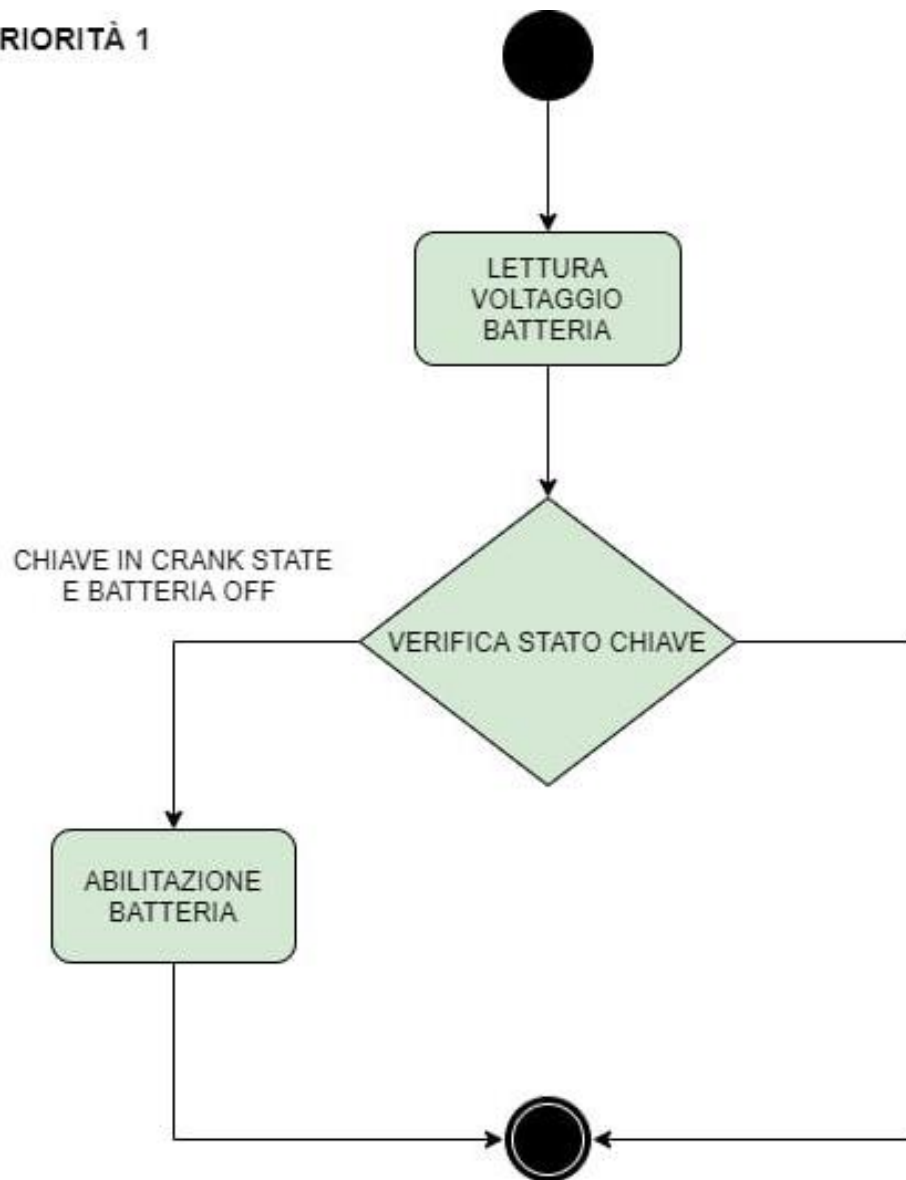


Figure 4. 2: Battery Key Management task architecture flow chart

The main goal of this task is creation of the software algorithm of the battery which is being enable with the status of the key.

This task has the high priority in the real time operation system which priority number is equal to 1. In our case the maximum priority is equal to 0.

The battery pack is composed of battery management system and the battery modules. The main condition to enable the battery in the car is that to key should be crank state in that moment. Whenever the conditions are set then control unit should send the activation signal to the battery management system of the car.

This software architecture aims to use the battery in a most efficient way without wasting any energy from the car and also be sure that safety conditions with receiving the car and key status. Newer cells have BMS (Battery Management Systems) with designed safety measures to regulate excessive internal currents and overheating. It is essential that safety-critical software and hardware function as expected and are adequately resilient.

The software architecture and development algorithm of the task is as follows:

1. Verification of the battery voltage and log the value.
 - If a expected return value is received, this implies that we successfully received the value of the battery to a variable which is specified before.
 - Then we use this variable to log the valur of the battery.
2. Verification of the crank state of the key.
 - If a expected return value is received from the key status function, this implies that key is in the crank status.
 - We also receive the state of the battery and compare them with the key status before enable the battery.
3. Activation of the battery enable.
 - If key is in the crank state and battery state is false at the same time then we enable the battery successfully.

4.3.2 Battery Key Management Code Implementation

4.3.2.1 Battery Key Management.c Code

```
/*
 * task_2_gestione_chiave_batteria.h
 * Created on 05/Jun/2021
 * Author: Celal
 */

#include <string.h>
#include <stdarg.h>

#include "task_2_gestione_chiave_batteria.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
#include "IO_ADC.h"
#include "IO_DIO.h"
#include "IO_UART.h"

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const ubyte1 TASK2_ID [1] = {2};

void task_2_gestione_chiave_batteria ()
{
    ubyte2 ubat;
    bool ubat_fresh;
    bool key_crank_state;
    bool battery_state;
    IO_ErrorType adc_error;
    IO_ErrorType crank_state_error;
    IO_ErrorType battery_enable_error;
    IO_ErrorType battery_state_error;

    ubyte4 task_2_timestamp;
    driver_task_begin_rc = IO_Driver_TaskBegin ();
    (void) IO_RTC_StartTime (&task_2_timestamp);
    (void) IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION,
    ID_TASK_EXECUTION_LENGTH, FALSE, TASK2_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

// retrieve and log the battery voltage
While (IO_E_OK == IO_ADC_Get(Battery_voltage_pin, &ubat, &ubat_fresh) )
{
    #ifdef DEBUG
        UART_Printf (IO_UART, "Battery voltage: %u mV\n\r", ubat);
    #endif

    crank_state_error = IO_DI_Get( Crank_state_pin , &key_crank_state);
    battery_state_error = IO_DI_Get( Battery_state_pin , &battery_state);

    //key is in crank state and battery is off
    if (key_crank_state == TRUE && battery_state == FALSE)
    {
        //battery enable
        battery_enable_error = IO_DO_Set (Battery_enable_pin , TRUE, &battery_enable);

        #ifdef DEBUG
            UART_Printf (IO_UART, "Battery voltage: %u mV\n\r", battery_enable_error);
        #endif
    }
}
while (TRUE != Check_Task_End(task_2_timestamp, (ubyte4) TASK2_CYCLE_TIME))
{
    #ifdef DEBUG
        UART_Printf (IO_UART, "\n\r wait for task 2 terminating %u !\n\r", task_2_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 ();
}

```

4.3.2.2 Battery Key Management.h Code

```
/*
 * task_2_gestione_chiave_batteria.c
 * Created on 05/Jun/2021
 * Author: Celal
 */

#ifndef TASK_2_GESTIONE_CHIAVE_BATTERIA
#define TASK_2_GESTIONE_CHIAVE_BATTERIA
#include "ptypes_xe167.h"
#define DEBUG
#define TASK2_CYCLE_TIME 100000u

void task_2_gestione_chiave_batteria(void );
/*
 * ADC and DIO pins could be modified later
 */
#define Battery_voltage_pin IO_ADC_UBAT
#define Crank_state_pin IO_DI_00
#define Battery_state_pin IO_DI_01
#define Battery_enable_pin IO_DO_01
#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_ADC_Get

IO_DO_Init

IO_DI_Get

IO_DO_Set

EEPROM driver

Real Time Clock (RTC)

Digital inputs and outputs (DIO) driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.3.4.2 Development and Implementation Documentation

During the implementation different peripherals are used for the receiving information from the battery and the key status.

For the battery level we used ADC to receive the value from the battery to the control unit. Key and battery state is connected to the one of the digital input-output pin of the control unit.

The ADC implemented as a microprocessor peripheral has the ability to measure the analog input signal. So for the ADC implementation and the communication we needed to use 1 different ADC API from HY-TTC30 family. This is *IO_ADC_Get* function and to take the battery and key status we simple used *IO_DI_Get* function and *IO_DO_Set* function for the enable the battery in the car.

1. ***IO_ADC_Get*** is a function that we used to receive the value of the battery from the battery management system in the car. We have 3 input parameters for this function and a return.

Parameters:

adc_channel is a first parameter and which presents the input or output channel which connection is used between battery management system and the control unit in the car.

Channels could be the one of the 41 IO_ADC channels and additionally could me more specific possibilities available in the battery.

Type of this parameter should be IO_PIN from the driver.

adc_value is a second parameter which a pointer to receive the battery value and store that address. There is a range which is depend on the input group and there is also specific configuration changing with the channel.

Type of this parameter should be ubyte2 *const from the control unit side.

fresh is the last parameter of this function and it indicates if the received battery value is new value or it is the same received value as before.

It is the type of bool pointer and stores true or false.

Return Values

IO_ErrorType is a return type of this function and this type includes:

IO_E_OK means everything went fine.

IO_E_ADC_CHANNEL_STARTUP Channel is still in the init step.

IO_E_INVALID_CHANNEL_ID The ADC value provided is inaccurate or unavailable.

This problem has been detected twice. First, if such an ADC quantity will be taken out soon after initializing the ADC channels and the ADC transformation of the associated channel has not yet begun, the ADC conversion of that channel will not be performed (after startup).

Second, this issue is displayed at runtime if the IO-Driver has identified a converting error, which indicates that the ADC failed to convert the channel during the most recent conversion cycle.

IO_E_CH_CAPABILITY The required functionality is not supported by the IO channel (IO pin).

Two circumstances can result in this error code:

- When trying to start or utilize an ADC pin as an Output pin, for instance.
- When attempting to configure an IO for a pin function not supported by the ECU variation. (For instance, while attempting to configure a PWD input that is not physically mounted on the ECU variation in use)

IO_E_NULL_POINTER The function has just been supplied a NULL pointer.

This issue is generated if a non-optional function pointer argument has been changed to NULL.

IO_E_FET_PROTECTION An inner switching (FET) has been deactivated to prevent harm to the hardware.

If the current on an inner FET is too large, program will disable the FET to prevent its damage. After a one-second timeout, the driver attempts to re-enable the FET.

Whenever a FET has been turned off by the means of protection, the relevant task function will produce this error messages. Therefore, the values obtained are incorrect and should not be utilized in future calculations.

IO_E_CHANNEL_NOT_CONFIGURED Neither the IO channel nor the IO pin have been activated. An IO driver task function will return this exception if the connection has not been established. Calling the corresponding Init method initializes the channel.

IO_E_ADC_INVALID The ADC value provided is invalid or unavailable.

This problem has been reported twice. Firstly, if an ADC result will be read out instantly after instantiating the ADC channels and the ADC transformation of the associated channel has not yet begun,

the ADC conversion of that channel will not be performed (after startup).

Second, this error is displayed at runtime if the IO-Driver has identified a conversion error, which indicates that the ADC did not convert the channel during the most recent conversion cycle.

- 2. IO_DI_Get** It is a simple digital input status receiver in the current channel pin. Returns the digital input state.

We have 2 inputs and a return value for this function.

Parameters

di_channel it indicates one of the 4 digital input channels available in the control unit and the type of the parameter is IO_PIN.

di_value It is a pointer variable and indicates true or false depending on the received digital signal in the channel. Type of this parameter is bool *const.

Return Values

IO_E_OK
 IO_E_INVALID_CHANNEL_ID
 IO_E_CH_CAPABILITY
 IO_E_NULL_POINTER
 IO_E_CHANNEL_NOT_CONFIGURED
 IO_E_ADC_INVALID
 IO_E_DI_SHORT_CIRCUIT
 IO_E_DI_SHORT_BATTERY
 IO_E_DI_OPEN_LOAD
 IO_E_DI_INVALID_VOLTAGE
 IO_E_DI_OPEN_LOAD_OR_SHORT_CIRCUIT

The digital input is governed by the principles and in the correct sequence:

- 1.If the voltage level falls within low thresh1 and low thresh2 specified with IO DI Init, a digital value of FALSE is provided. If the voltage falls between high thresh1 and high thresh2, the digital value TRUE is displayed. In both circumstances, the IO E OK error code is returned.
- 2.If a pull-up is setup on the intake and the voltage falls within 4.75V and 5.5V, di value is invalid and the operation gives IO E DI OPEN LOAD.
- 3.If a pull-down resistance or no pull-resistor is specified on the intake and the voltage is somewhere between 0V and 1.25V, the value in di value is invalid and the operation gives IO E DI OPEN LOAD OR SHORT-CIRCUIT.
- 4.If the voltage level falls within UBat and UBat - 1.25V, di value is set to TRUE and IO E DI SHORT BATTERY is returned.
- 5.In all the other circumstances, the result in di value is incorrect, and the function returns IO E DI INVALID VOLTAGE, since the calculated value does not fall inside the user-defined range nor

any diagnostic range.

The voltage levels that the user specifies through the IO DI Init variable limitations generally take precedence over diagnostic functionality.

If an intake is equipped with a pull-up resistor and the voltage range 0-6V is declared as valid limited, this operation will not give an open-load failure.

3. IO_DO_Set It controls the simple digital output value in the current channel pin.

We have 3 inputs and a return value for this function.

Parameters

do_channel it indicates one of the 35 digital output channels available in the control unit and the type of the parameter is IO_PIN.

do_value It is a bool variable and indicates true or false depending on the arranged digital signal in the channel.

voltage_fb It is an ADC value takes the feedback signal from the outside. The range is between 0-32780mv.

Return Values

IO_E_OK
 IO_E_INVALID_CHANNEL_ID
 IO_E_INVALID_DIAG_STATE
 IO_E_CHANNEL_NOT_CONFIGURED
 IO_E_DO_SHORT_CIRCUIT
 IO_E_DO_OPEN_LOAD
 IO_E_DO_SHORT_BATTERY
 IO_E_DO_OPEN_LOAD_OR_SHORT_BATTERY
 IO_E_DO_DIAG_TRANSIENT_OSC
 IO_E_DO_CHANNEL_STARTUP
 IO_E_DO_OUTPUT_DISABLED
 IO_E_DO_OUTPUT_STARTUP_ERROR
 IO_E_PROT_USER_OVERLOAD
 IO_E_PROT_TEMP_OVERLOAD
 IO_E_PROT_ACTIVE
 IO_E_PROT_FATAL
 IO_E_PROT_REENABLE
 IO_E_SW_OUTPROT_SM
 IO_E_ADC_INVALID
 IO_E_CH_CAPABILITY

- For the low-side output signals IO DO 10 through IO DO 11, the voltage return adc value reading is only possible in the OFF condition. The value obtained will remain 0 during ON phase.

- If the device is started as safety, the low-side ports IO DO 10.. IO DO 11 may only be enabled and deactivated during the driver's basic phase.

- For open-load/short-circuit monitoring on all digital output channels, the corresponding ADC channel will be utilized.
- To acquire a realistic result for voltage fb, the IO DO Set() method must be used a second time at least 10 milliseconds later.
- With IO POWER Set, the high-side digital outputs IO DO 00.. IO DO 05 and IO DO 20.. IO DO 25 must be activated. If not, the outputs will stay disabled.
- IO POWER Set must also be used to activate the push-pull digital outputs IO DO 30.. IO DO 35; else, those outputs will stay disabled.

4.4 Inverter Management Task

4.4.1 Task Architecture

TASK A PRIORITÀ 2



Figure 4. 3: Inverter management task architecture flow chart

The main goal of this task is creation of the software algorithm of the inverter management which is being enable with control unit.

This task has the high priority in the real time operation system which priority number is equal to 2. In our case the maximum priority is equal to 0.

A car voltage inverter is a component that transforms your vehicle's DC electricity to Ac voltage. Then, it may be used to power smaller electrical gadgets. DC power and AC power cannot be interchanged; thus, you should convert it before inserting an AC-powered equipment into a DC socket and conversely.

This software architecture aims to use the inverter in a most efficient way without wasting any energy from the car and also be sure that safety conditions with receiving the car and key status.

The software architecture and development algorithm of the task is as follows:

1. Verification of the inverter.
 - Reading the digital channel pin to get the state of the inverter.
2. Receiving the effective pressure of the pedal.
 - Read the pressure with using PWM current value.
3. Inverter management
 - Using pwm manage the interter from the control unit channel.

4.4.2 Inverter Management Code Implementation

4.4.2.1 Inverter Management .c Code

```
/*
 * task_3_gestione_inverter.c
 * Created on 07/Jun/2021
 * Author: Celal
 */

#include "task_3_gestione_inverter.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
#include "IO_PWM.h"
#include "IO_DIO.h"

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const uint8_t TASK3_ID [1] = {3};
static IO_ErrorType pwm_set_rc_inv;
static IO_ErrorType pwm_get_rc_inv;
```

```

static ubyte2 pwm_current_fb_inv;
static ubyte2 current;

static DIAG_ERRORCODE diag_error;
static ubyte1 diag_state;
static IO_ErrorType pwm_set_rc_inv_man;
static IO_ErrorType pwm_get_rc_inv_man;
static ubyte1 diag_state;
static DIAG_ERRORCODE diag_error;

void task_3_gestione_inverter( )
{
    bool inverter_state;
    bool pwm_fresh_inv;
    ubyte2 pwm_current_fb_val_inv;
    ubyte4 task_3_timestamp;

    driver_task_begin_rc = IO_Driver_TaskBegin ();
    (void) IO_RTC_StartTime (&task_3_timestamp);
    (void) IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION, ID_TASK_EXECUTION_LENHT,

FALSE, TASK3_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

// GET THE STATE OF THE INVERTER

while(IO_E_OK == IO_DI_Get(inverter_state_pin,&inverter_state))
{

    // read the current feedback value
    pwm_get_rc_inv = IO_PWM_GetCur (
        IO_PWM_PIN_INV
        , &pwm_current_fb_val_inv
        , &pwm_fresh_inv);

    // only update the current feedback value if fresh
    if ((pwm_get_rc_inv == IO_E_OK) && (pwm_fresh_inv))
    {

```

```

    pwm_current_fb_inv = pwm_current_fb_val_inv;

    #ifdef DEBUG
        UART_Printf (IO_UART, "Inverter Pedal pressure value: %u \n\r", pwm_current_fb_inv);
    #endif
}

//INVERTER MANAGEMENT PWM SET
pwm_get_rc_inv_man=IO_PWM_SetCur (IO_PWM_PIN_INV_Man, current, NULL);


(void) DIAG_Status (
    &diag_state
    , &diag_error );
}
while (TRUE != Check_Task_End(task_3_timestamp, (ubyte4)TASK3_CYCLE_TIME))
{
    #ifdef DEBUG
        UART_Printf (IO_UART, "\n\r wait for task 3 terminating %u !\n\r", task_3_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 ();
}

```

4.4.2.2 Inverter Management .h Code

```

/*
 * task_3_gestione_inverter.h
 * Created on 07/Jun/2021
 * Author: Celal
 */

#ifndef TASK_3_GESTIONE_INVERTER
#define TASK_3_GESTIONE_INVERTER
#define DEBUG
#define TASK3_CYCLE_TIME 100000u

void task_3_gestione_inverter(void );

#define inverter_state_pin      IO_DI_02
#define IO_PWM_PIN_INV         IO_PWM_02
#define IO_PWM_PIN_INV_Man     IO_PWM_03

#endif

```

4.4.3 Development and Implementation

4.4.3.1 Functions and Drivers Utilized

IO_Driver_TaskBegin
 IO_Driver_TaskEnd
 IO_RTC_StartTime
 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
 Digital inputs and outputs (DIO)
 driverPulse Width Modulation
 (PWM) driverReal Time Clock

(RTC) driver

Universal Asynchronous Receiver Transmitter (UART)

driverEEPROM driver

EEPROM preload functions

4.4.3.2 Development and implementation documentation

During the implementation different peripherals are used for the receiving and sending information from the inverter of the vehicle.

For the communication between control unit and the inverter we used PWM to receive the information and send commands to the inverter from the control unit.

The PWM implemented as a microprocessor peripheral has the ability to communicate with other components.

So for the PWM implementation and the communication we needed to use 3 different API from HY-TTC30 family. These are IO_PWM_SetCur and IO_PWM_GetCur and lastly to get the status from the inverter we used IO_DI_Get function. As explained before PWM_GetCur and IO_DI_Get functions here will be explained only PWM_SetCur function.

1. **PWM_SetCur** function sets the current for current-controlled specified pwm channel. We have 3 input parameters and a return variable for this function.

Parameters

pwm_channel it should be one of the 6 available pwm channels.

current This parameter should be current output which is indicated with mA.

duty_cycle_fb It is a duty cycle feedback for the channels.

Return Values

IO_E_OK
 IO_E_CHANNEL_NOT_CONFIGURED
 IO_E_INVALID_CHANNEL_ID
 IO_E_CH_CAPABILITY
 IO_E_PWM_OPEN_LOAD
 IO_E_PWM_SHORT_CIRCUIT
 IO_E_PWM_SHORT_BATTERY
 IO_E_PWM_OUTPUT_DISABLED
 IO_E_PWM_CHANNEL_STARTUP
 IO_E_PWM_OUTPUT_STARTUP_ERROR
 IO_E_ADC_INVALID_A_ADC
 IO_E_SW_INTERNAL
 IO_E_PWM_CAPTURE_ERROR
 IO_E_PROT_USER_OVERLOAD
 IO_E_PROT_TEMP_OVERLOAD
 IO_E_PROT_ACTIVE
 IO_E_PROT_FATAL
 IO_E_PROT_REENABLE

4.5 Pump Management Task

4.5.1 Task architecture

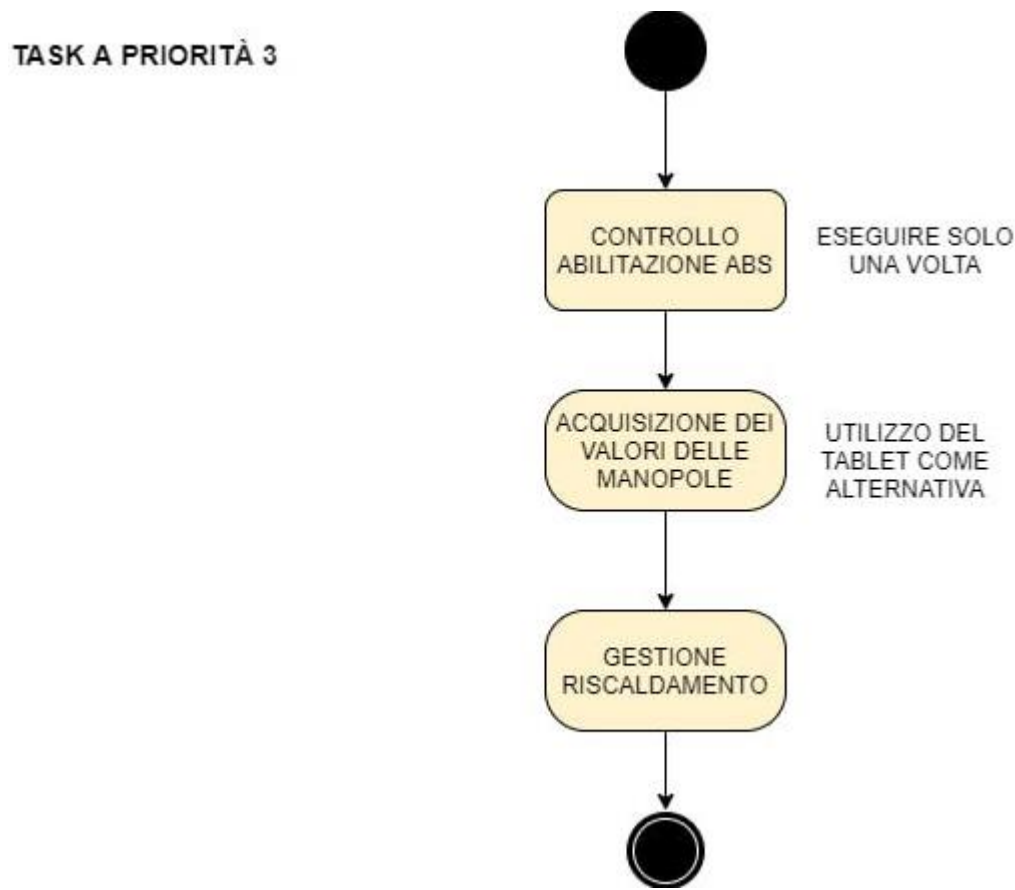


Figure 4. 4: Pump management task architecture flow chart

The main goal of this task is creation of the software algorithm of the pump management which will be controlling the heating of the car. Doing that task is basically verify firstly the ABS and taking the data from a know in the car.

This task has the maximum priority in the real time operation system which priority number is equal to 3. In our case the maximum priority is equal to 0.

Many HVAC systems rely on pumps to provide the energy necessary to transport water and other fluids through pipes, fittings, and machinery. Several aspects must be considered when choosing a pump for an HVAC application, including operational expenditures, energy costs, system dependability, and lifespan. To use a systems method that focuses on the whole performance of the system as opposed to individual parts will assist in the choice of a durable pump that operates effectively. This article examines the most prevalent pump types utilized in HVAC applications, as well as selection factors and system selection.

The software architecture and development algorithm of the task is as follows:

1. Enable the ABS.
 - We used a digital pin to enable the ABS as a starting of the management of the pump and the heating system together.
2. Reading the value of the specific know.
 - Using a one of the available pwm channel in the control unit we are able to read the expected heating value and with that data we can manage the heating system of the car to arrange the expected heating.
3. Heating management.
 - Heating management is arranged with taking data from the know and communication managed with using PWM in the system.

4.5.2 Pump Management Code Implementation

4.5.2.3 Pump Management.c Code

```
/*
 * task_4_gestione_pompe.c
 * Created on 07/Jun/2021
 * Author: Celal
 */

#include "task_4_gestione_pompe.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
#include "IO_PWM.h"
#include "IO_DIO.h"

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const uint8_t TASK4_ID [1] = {4};

static IO_ErrorType do_set_rc_abs;
static uint16_t do_voltage_fb;

static IO_ErrorType pwm_set_rc_knob;
static IO_ErrorType pwm_get_rc_knob;

static uint16_t pwm_duty_cycle_knob;
static uint32_t pwm_duty_cycle_fb_knob;
static uint16_t pwm_current_fb_knob;

static uint16_t duty_cycle_set, current;
static uint32_t duty_cycle_rb;
static uint32_t duty_cycle_fb;

static DIAG_ERRORCODE diag_error;
static uint8_t diag_state;

static IO_ErrorType pwm_set_rc_inv_heat;
static IO_ErrorType pwm_get_rc_inv_heat;

duty_cycle_set = duty_cycle_heat_management;
```

```

pwm_duty_cycle_knob = 0;

pwm_current_fb_knob = 0;

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

void task_4_gestione_pompe( )
{
    bool pwm_fresh_knob;
    ubyte2 pwm_current_fb_val_knob;
    ubyte4 pwm_duty_fb_val_knob;
    ubyte4 task_4_timestamp;

    driver_task_begin_rc = IO_Driver_TaskBegin ();
    (void) IO_RTC_StartTime (&task_4_timestamp);
    (void) IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION, ID_TASK_EXECUTION_LENHT,

    FALSE, TASK4_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

    // ENABLE ABS WITH DIGITAL OUTPUT PIN
    do_set_rc_abs = IO_DO_Set (ABS_ENABLE_PIN
                                , TRUE
                                , &do_voltage_fb);

    if (do_set_rc_abs != IO_E_OK)
    {
        #ifdef DEBUG
            #endif

        // UART_Printf (IO_UART, "ABS Enable ERROR: %u mV\n\r", do_set_rc);

        while (do_set_rc_abs == IO_E_OK){

            // USE PWM TO READ KNOB
            pwm_set_rc_knob = IO_PWM_SetDuty (
                                IO_PWM_PIN_KNOB
                                , pwm_duty_cycle_knob
                                , &pwm_duty_fb_val_knob);

```

```

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

// read the current feedback value
pwm_get_rc_inv = IO_PWM_GetCur (
    IO_PWM_PIN_KNOB
    , &pwm_current_fb_val_knob
    , &pwm_fresh_knob);

// only update the current feedback value if fresh
if ((pwm_get_rc_knob == IO_E_OK) && (pwm_fresh_knob))
{
    pwm_current_fb_knob = pwm_current_fb_val_knob;

#ifdef DEBUG
    UART_Printf (IO_UART, "Knob value: %u \n\r", pwm_current_fb_knob);
    UART_Printf (IO_UART, "Knob Value duty cycle: %u mA\n\r", pwm_duty_cycle_fb_knob);
#endif
}

//HEATING MANAGEMENT PWM SET
pwm_set_rc_inv_heat=IO_PWM_SetDuty (IO_PWM_PIN_HEAT    , duty_cycle_set,
&duty_cycle_rb);
pwm_get_rc_inv_heat=IO_PWM_SetCur (IO_PWM_PIN_HEAT    , current, NULL); //set current
and duty cycle

(void) DIAG_Status (
    &diag_state
    , &diag_error);
}
while (TRUE != Check_Task_End(task_4_timestamp, (ubyte4)TASK4_CYCLE_TIME))
{
    #ifdef DEBUG
        UART_Printf (IO_UART, "\n\r wait for task 4 terminating %u !\n\r", task_4_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 ();  
}
```

4.5.2.4 Pump Management.h code

```
/*  
 * task_4_gestione_pompe.h  
 * Created on 07/Jun/2021  
 * Author: Celal  
 */  
  
#ifndef TASK_4_GESTIONE_POMPE  
#define TASK_4_GESTIONE_POMPE  
#define DEBUG  
#define TASK4_CYCLE_TIME 100000u  
  
void task_4_gestione_pompe(void );  
  
#define ABS_ENABLE_PIN          IO_D0_02  
#define IO_PWM_PIN_KNOB         IO_PWM_04  
#define IO_PWM_PIN_HEAT         IO_PWM_05  
#define duty_cycle_heat_management 0x8000  
  
#endif
```

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_DO_Init
IO_DO_Set
IO_POWER_Set
IO_PWM_GetCur

Pulse Width Modulation (PWM)

driverEEPROM driver

Real Time Clock (RTC)

driverWindow Watchdog

driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.5.3.2 Development and implementation documentation

During the implementation different peripherals are used to manage the heating and the pump in the car. For the communication between heating system and the knob we used PWM to receive the information and send commands to the HVAC from the control unit.

So for the PWM implementation and the communication we needed to use 3 different API from HY-TTC30 family. These are IO_PWM_SetCur and IO_PWM_GetCur and lastly to enable the ABS of the car we used IO_DO_Set function as we used before the other tasks.

As a summary of this task we simply enable the ABS firstly and set a knob to read the desired value from the user and data communication provided by the pwm and lastly we transmit this information to the heating system again with using pwm peripheral of the control unit.

4.6 Debouncer Task

4.6.1 Task architecture

The main goal of this task is creation of the generic software algorithm of a debouncer which will be used in the buttons of the control unit.

Debouncing is the process of reducing undesirable input noise from knobs, switches, or other user inputs. Debouncing avoids frequent occurrences of excess activations or delayed processes. Debouncing is implemented in hardware switches, software, and websites.

In coding, debouncing occurs when a process filters input from the user prior to initiating an action. Incorrect debouncing of user input may result in poor performance, duplicate activation, or user annoyance. Often, a generic debouncing procedure is employed instead of developing fresh code for each input action. The debouncing function follows the user input and then calls the button's intended action. Numerous programming libraries offer a debounce function by default.

A result of mechanical and physical difficulties, pushbuttons often create erroneous open/close transitions when pushed; these transitions may be interpreted as several pushes in a very little period of time, tricking the computer. This example explains how to debounce an event, which involves doing two checks in a brief amount of time to confirm that a button has been touched. Without debouncing, a single button hit may result in unforeseen outcomes.

The software architecture and development algorithm of the task is as follows:

1. Read the input value.
 - We are reading the input value with using the `IO_PWM_GetCur` function and assign the received value to the specified variable.
2. Filtering algorithm of the reading value.
 - With using a `RTC_StartTime` and `RTC_GetTimeUS` functions, we are executing the algorithm in a while loop and filtering the received data in a correct way.
3. Set the output value.
 - We set the output value with using `PWM_SetCur` function after filtering process is finished.

4.6.2 Debouncer Code Implementation

4.6.2.1 Debouncer.c Code

```
/*
 * debouncer.c
 * Created on Jul 8, 2021
 * Author: Celal
 */

#include "debouncer.h"
#include "utility.h"
#include "eeprom_address.h"
#include "IO_RTC.h"
#include "IO_WDTimer.h"
#include "IO_PWM.h"
#include "IO_DIO.h"
#include "DIAG_Functions.h"
#include "DIAG_Constants.h"

static IO_ErrorType driver_task_begin_rc;
static IO_ErrorType driver_task_end_rc;
const uint8_t Debouncer_ID [1] = {8};

static uint16_t pwm_duty_cycle;
static uint32_t pwm_duty_cycle_fb;

static IO_ErrorType pwm_set_rc_output;
static IO_ErrorType pwm_get_rc_input;

static uint32_t value;
static DIAG_ERRORCODE diag_error;
static uint8_t diag_state;

void debouncer ()
{
    pwm_duty_cycle = 0;
    pwm_current_fb = 0;

    uint32_t value;
    uint16_t reading_out = 0
    uint32_t lastValue = 0;
    uint16_t pwm_current_fb_val_input;
```



```

bool pwm_fresh_input;
ubyte4 debounce_time = 40;

ubyte4 debouncer_timestamp;
ubyte4 time_stamp;
driver_task_begin_rc = IO_Driver_TaskBegin ();
(void) IO_RTC_StartTime (&debouncer_timestamp);
(void) IO_EEPROM_PreloadWrite (ID_TASK_EXECUTION, ID_TASK_EXECUTION_LENHT,
FALSE, Debouncer_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

// read input value
pwm_get_rc_input = IO_PWM_GetCur (
    IO_PWM_PIN_INPUT_debouncer
    , &pwm_current_fb_val_input
    , &pwm_fresh_input);

if (pwm_get_rc_input == IO_E_OK)
{
    value = (ubyte4)pwm_current_fb_val_input;
}
do
{
    IO_RTC_StartTime (&time_stamp);
    lastValue = value;
    if (IO_RTC_GetTimeUS (time_stamp) > debounce_time)
    {
        pwm_get_rc_input = IO_PWM_GetCur (
            IO_PWM_PIN_INPUT_debouncer
            , &pwm_current_fb_val_input
            , &pwm_fresh_input);
        if (pwm_get_rc_input == IO_E_OK)
        {
            value = (ubyte4)pwm_current_fb_val_input;
        }
    }
} while (lastValue != value);
reading_out = value;

```

```

pwm_set_rc_output = IO_PWM_SetCur (
    IO_PWM_PIN_OUTPUT_debouncer
    , reading_out
    , &pwm_duty_cycle_fb);

(void) DIAG_Status (
    &diag_state
    , &diag_error );

while (TRUE != Check_Task_End (debouncer_timestamp, (ubyte4) DEBOUNCER_CYCLE_TIME))
{
    #ifdef DEBUG
    UART_Printf (IO_UART, "\n\r wait for debouncer terminating %u !\n\r", task_3_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 ();
}
}

```

4.6.2.2 Debouncer.h code

```

/*
 * debouncer.h
 * Created on Jul 8, 2021
 * Author: Celal
 */

#ifndef SRC_DEBOUNCER_H_
#define SRC_DEBOUNCER_H_
#define DEBUG
#include "ptypes_xe167.h"
#define DEBOUNCER_CYCLE_TIME 100000u

void debouncer(void);

#define IO_PWM_PIN_INPUT_debouncer 1 // input pwm pin
#define IO_PWM_PIN_OUTPUT_debouncer 2 //output pwm pin

```

```
#endif      /* SRC_DEBOUNCER_H_ */
```

4.6.3 Development and implementation

4.6.3.1 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
 RTC_StartTime
 RTC_GetTimeUS
 IO_POWER_Set
 IO_PWM_GetCur

Pulse Width Modulation (PWM)

driverEEPROM driver

Real Time Clock (RTC)

driverWindow Watchdog

driver

Universal Asynchronous Receiver Transmitter (UART) driver

4.6.3.2 Development and implementation documentation

During the implementation different peripherals are used to manage the debouncing of the data received by the input of the control unit.

For the communication between control unit and the button we used PWM to receive the information and send back the data to specified peripherals from the control unit.

So for the PWM implementation and the communication we needed to use 3 different API from HY-TTC30 family. These are IO_PWM_SetCur and IO_PWM_GetCur and from the timer point of view we used RTC_StartTime and RTC_GetTimeUS function.

1. **RTC_StartTime** function returns a timestamp so which can be used for another timer functions

available in RTC.

Parameters

period The time interval that the event handler shall be invoked.

event_handler It is a function pointer to the event handler

Return Values

IO_E_OK

IO_E_NULL_POINTER

IO_E_INVALID_PARAMETER

IO_E_BUSY

2. **RTC_GetTimeUS** Returns the time elapsed. The method returns the time in us since the specified timestamp was captured (through the procedure IO RTC StartTime()).

Parameters

timestamp Timestamp obtained from an IO RTC StartTime call ()

The method will return 0 if the RTC unit has not been initialized.

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 Skonhøft. 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://efaidnbmnnnibpcajpcgclefindmkaj/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://efaidnbmnnnibpcajpcgclefindmkaj/viewer.html?pdfurl=https%3A%2F%2Fwww.ttccontrol.com%2Fwp-content%2Fuploads%2FTTControl-HY-TTC-32S-Datasheet.pdf&cflen=262721&chunk=true

