POLITECNICO DI TORINO

Master degree in Computer Engineering

Master Degree Thesis

Library definition for an automotive ECU API layer (using Model-Based approach)

Advisor
Prof. Massimo Violante

Candidate
Luca Zannella

Internship tutor
Ing. Emilio Bertrand

November 2023

Sommario

ACRONYMS

- **API** Application Programming Interface
- **BOB** Break Out Box
- **BOM** Bill Of Material
- **BSW** Basic Software Layer
- **CAN** Controller Area Network
- **CNG** Compressed Natural Gas
- **CRF** Fiat Research Center
- **DBC** Database CAN
- **DLC** Data Length Code
- **DSL** Domain-Specific Language
- **ECM** Engine Control Module
- **ECU** Electronic Control Unit
- **EPR** Electronic Pressure Regulator
- **FI** Frequency Input
- **FLCU** Fuel Line Control Unit
- **FO** Frequency Output
- **GPL** Liquefied Petroleum Gas
- **HDS** Heavy Duty System
- **HEGO** Heated Exhaust Gas Oxygen sensor
- **HIL** Hardware-in-the-loop
- **HS** High Side
- **I/O** Input/Output
- **LIN** Local Interconnect Network
- **LNG** Liquefied Natural Gas
- **LS** Low Side
- **LUT** Lookup Table
- **MBD** Model Based Design
- **MCAL** Microcontroller Abstraction Layer
- **MCU** Micro Controller Unit
- **MIL** Model-in-the-loop
- **OBD** On-board diagnostic
- **OEM** Original Equipment Manufacturer
- **OTV** On Tank Valve
- **PIL** Processor-in-the-loop
- **PIM** Platform-Independent Model
- **PWM** Pulse-Width Modulation
- **SIL** Software-in-the-loop
- **SRD** System Requirement Document
- **UEGO** Universal Exhaust Gas Oxygen sensor

1. Introduction

In the last decades, the automotive industry has undergone a big transformation from different point of views, and it has reshaped the way vehicles are designed and manufactured.
Environmental sustainability is becoming a crucial factor in this change and thanks to technological advancement, the challenge of reducing gas emission and improving the energy efficiency of vehicles can be addressed.
For this purpose, more sustainable alternative fuels were sought and, natural gas and hydrogen have emerged as promising candidates.
This thesis project is done in collaboration with the Metatron S.p.A. company, which is specialized in the design and production of pressure regulator and Engine Control Units for alternative fuels.

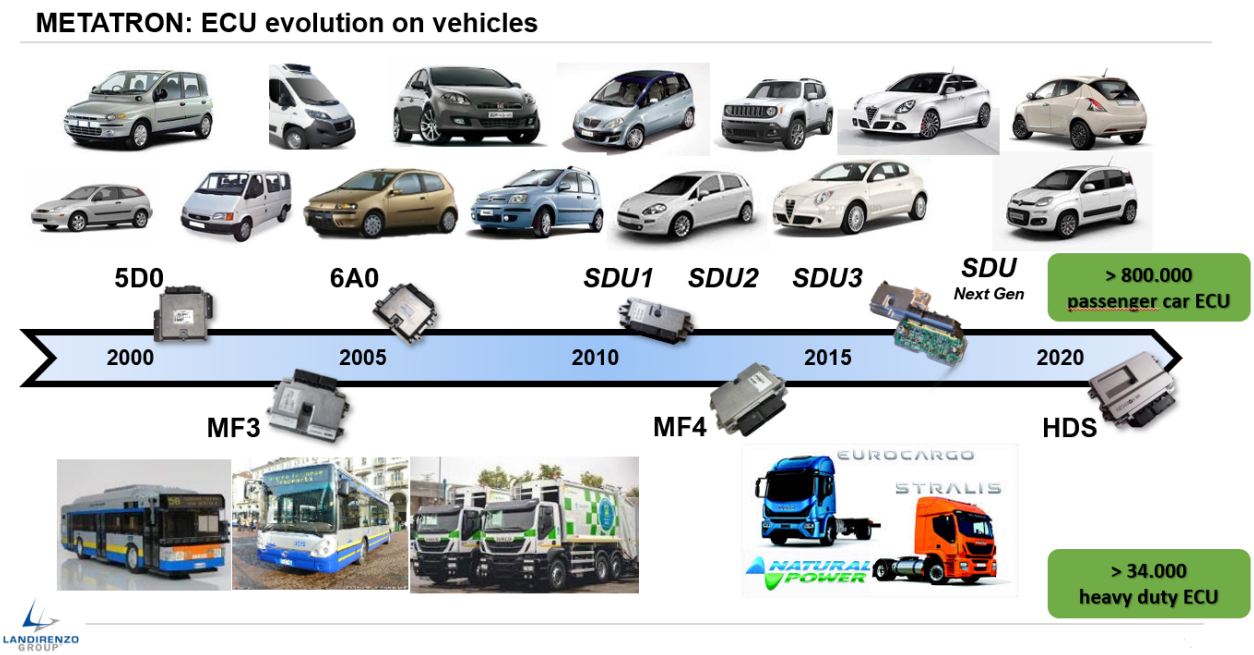
1.1 Metatron Overview

At the beginning of 90s, the Fiat Research Center (CRF) group identified in the natural gas fuel the best solution for reducing gas emission from the internal combustion engine both for passenger cars and heavy-duty systems.
To launch the industrial production of these natural gas systems, CRF found the Tartarini company in Bologna (specialized in 'aftermarket' systems for conversion of gasoline engines to methane) as a partner for the development and production. The first adopted solution was 'bifuel' (natural gas and gasoline supply) for light system and 'monofuel' (only natural gas) for heavy-duty systems such as commercial vehicles and public transport.

In 1998 some resources detached from Tartarini and created Metatron, still located at Bologna, with the goal to manufacture and sell CNG/LNG systems to OEM and not for the 'aftermarket' field.
Metatron became the exclusive supplier for Fiat Auto and IVECO of the main components for these type of systems, in particular electronic control unit and pressure regulator. After that, between 2008 and 2010 Metatron founded in Volvera (TO) a new division devoted to electronics technologies and their applications, acquiring from CRF resources that allowed to be independent from Fiat while maintaining different collaborations. This new division developed a secondary control unit for GPL systems of Fiat.
In few years, China became the major buyer of Metatron's pressure regulators with its producers of heavy-duty engines.

In 2014 Metatron acquired the control of Digigroup, a society specialized in development and supply of electronics components for Automotive Info telematic (ITS) and the following year was founded a new society called Metatronix, regarding all electronics applications.
After 4 years, in 2018, due to increasing differences between ITS and Powertrain field, Metatron decided to make Metatronix completely autonomous and to reinforce the group creating the Metatron Research Center.

In 2021, Landi Renzo Group signed a binding agreement for the acquisition of Metatron S.p.A. with the goal to reinforce and accelerate the position as leader in the supply of systems and components for Natural Gas and Hydrogen mobility in the Mid&Heavy-duty field which is going to grow quickly in the coming years.



1.2 Thesis goals

In the past years Metatron has designed a powerful solution for an Engine Control Module, called HDS. This ECU has been widely used to cover a great and different range of alternative fueled engines, with several architecture such as 4/6/8/12 cylinders engines mainly for heavy duty vehicles (both with 12V or 24V power supply systems) with different scopes (Trucks, Buses, Off-road vehicles, Agriculture tractors, Locomotives, Industrial/Civil Cogeneration applications, …) according to the latest pollutant emission standards and to the latest safety and cybersecurity standards (such as ISO26262 or ISO/SAE 21434).

The result of this development process is a very flexible hardware/software automotive general purpose platform that Metatron started to use as a rapid prototyping or proof-of-concept unit internally as basis for several spin-off project (such as Transmission Control Unit, Tank Control Unit, Pressure Regulator Control Unit, Injection Control Unit,…).

Consequently, Metatron is now capable to propose a cheap and robust automotive solution able to manage the typical automotive sensors/actuators, that allows to its customers to easily integrate their own application software (using the Model-Based approach) to be used to validate their concepts to control an automotive system with two main and big advantages:

1. Dramatically decreasing the investments typically needed for the concept phase, significantly reducing the time to market of the solution;

2. Validate and define the requirements for a tailored production control unit.

The ultimate goal of this thesis project is to explore and to identify a set of rules/guidelines to allow an easy integration of the customer application software on the HDS hardware platform.

To reach this target the idea is to make available to the customers a proprietary library already integrated in the development environment (MATLAB/Simulink) that make the modeling of the application software much easier and faster.

Since the software application is developed following a Model-Based approach and thanks to this library the customers will have at his disposal some Simulink blocks that help him with his workflow.

In particular, there has been created blocks to interface with the I/O channels, the communication layer (CAN and DBC integration) and the automatic generation of the code from the model.

Finally, the added value of this solution is the automation of some procedures that could inevitably lead to errors if hand made.

1.3 Hardware platform – HDS9

HDS stands for Heavy Duty System and it is an Engine Control Unit used for alternative fueled vehicles. This ECU has been created by Metatron using a high level of technology in terms of hardware components and it satisfies the requirements of the latest emission standards (EUVI/CHINAVI), on-board diagnostic (EOBD) and functional safety standards (ISO26262).

This ECU has been used as platform for the development of the thesis work. Thanks to its performance and its hardware specifications it is an optimal general purpose embedded platform where test and validate every type of automotive application software. The hardware characteristics of this platform will be explained in detail later. Here it can be seen the picture of the HDS9.

Figure 1 - HDS9

1.4 Prototype stages before industrialization

Before arriving at the industrialization phase of a final product, it is followed a certain development flow made by different types of prototypes that are necessary for a great work.

The starting point of this flow can be called 'Proto A' and it is basically laboratory instrumentation. An example of this first prototype can be the 'CompactRIO' made by National Instrument, it is a real-time embedded controller and its main characteristic is to have reconfigurable I/O modules and a FPGA module. Thanks to this characteristic the CompactRIO is extremely modular and reconfigurable according to your needs.
It is made by a chassis where can be attached the I/O modules and it also include a microprocessor for implementing control algorithms.
It is used as first prototype step because it can be adopted for the development of any product in case you don't have a hardware platform as HDS9.
The advantages of this prototype are:

- Flexibility: components can be easily swapped, upgraded or reconfigured enabling engineers to test various product. This flexibility accelerates the development process and facilitates the discovery of optimal solutions.
- Scalability: as the product's complexity and requirements evolve, additional modules can be integrated, accommodating changes in functionality or performance. Scalability minimizes the need to redesign the entire system for incremental improvements.
- Reusability: this type of prototype can be repurposed in other subsequent projects reducing the overall development time and cost and, in this way, it is covered the initial investment.

On the other hand, the main disadvantage could be the performance limitation because a dedicated platform with another type of microprocessor could have better performance and it would get closer to the final product.

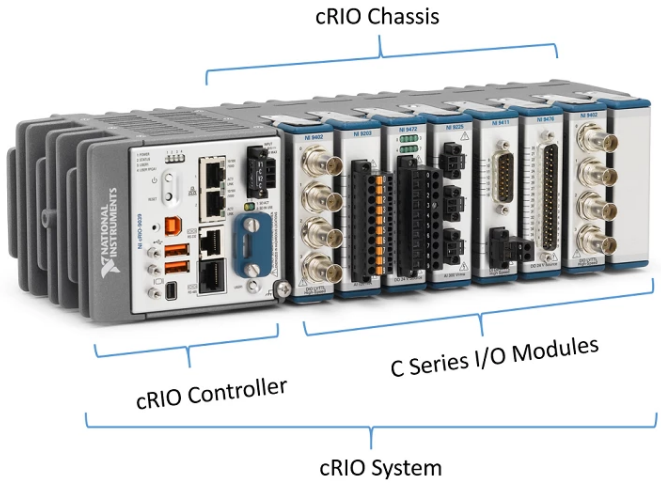


Figure 2 – CompactRIO

The next prototype step can be called 'Proto B' and it is essentially the role that plays in our case the HDS9. In this step there is a generic platform oversized in terms of power and hardware and in such

a way is possible to develop every type of application software that may require different type of I/O.
In this step having a good platform with all the basic functionality easily accessible is crucial for the success of the final product and the goal of this thesis goes in that direction. In this way the user will have access to the hardware layer in an easier way and it will be able to develop his application staying at a higher level of abstraction facilitating his workflow.

The last step is called 'Proto C' and it is obtained by cutting out all that is not necessary from 'Proto B' in order to reduce production costs. For example, if is not strictly necessary a microprocessor with the same performance of 'Proto B' it might be convenient install in the final product a less powerful one or if are not necessary all the I/O channel there could be deleted those that are not used. This prototype is the final product and then, after different test and validation phases, it will be industrialized.

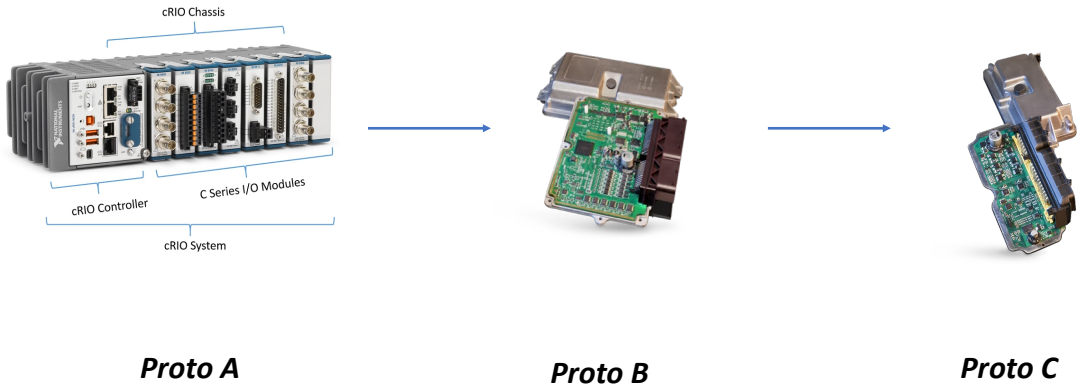


Figure 3 – Prototype stages

2. Embedded software architecture in the automotive field

The embedded software architecture, as shown in the *figure 4*, follows the key principles of AUTOSAR.
The basic idea behind AUTOSAR is the separation of the application software layer and the hardware layer. This leads to greater portability across different hardware platform.
Moreover, the software is divided into autonomous software components that can be developed, tested and updated independently. This modular approach simplifies complex software management and enhances components reusability.

The scheme of the embedded software architecture can be represented in this image.

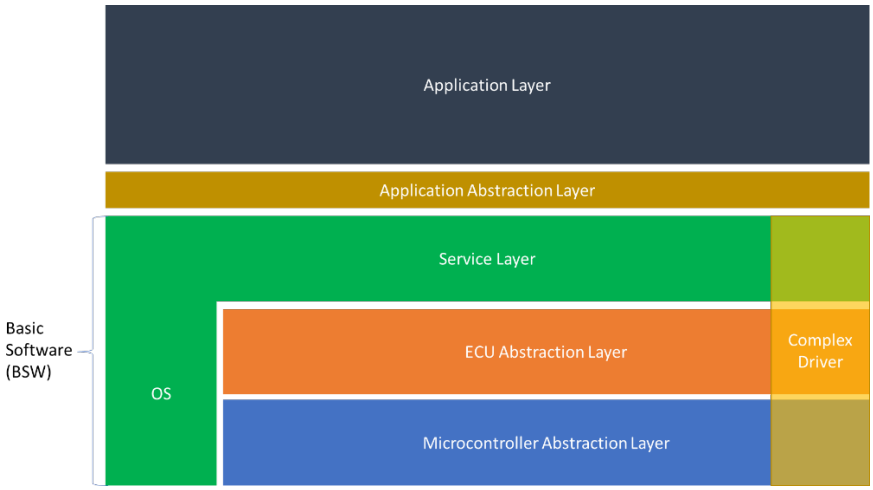


Figure 4 – Software Architecture

2.1 Architecture layers

The structure is divided in three different layers:

- **Basic Software Layer (BSW)**

  This layer provides a series of software modules that are essentially to use and communicate with different peripherals of MCU. It is composed by other different layers, each of them with a different purpose.
  The lowest layer of the BSW is the Microcontroller Abstraction Layer (also called MCAL) and it is strong dependent on the MCU in use, in fact it usually changes according to the type of microcontroller. It is very important because it contains drivers for accessing peripherals. The layer colored in orange is the ECU Abstraction Layer and his main purpose is to abstract the MCAL layer from upper layers and provides all the APIs for making available external and internal drivers. In this way upper layers of the ECU are independent from the hardware in use.
  The top layer colored in green is the Service Layer and it provides basic services for the

application. Some basic services are: Operating System functionality, communication services, memory services, ECU state management, etc.
The last layer of the BSW that is linked with all the other three is the Complex Driver and it is useful for writing function or drivers of external devices that are connected with our system.

- **Application Abstraction Layer**

The idea behind the project of this thesis is based on this layer, in fact the real strength of an architecture of this kind is to have this pillow layer which allows you to separate the Application Layer from the BSW Layer. In this way for example if you change the hardware platform you do not have to change the entire application software.
This meets the needs of customers who want to develop the application layer in-house, so the company can make available only the hardware platform with a dedicated Application Abstraction Layer.
This layer, that we internally also call 'API Layer', is the core of the architecture in fact it links the Application Layer with the BSW Layer and it has the goal of making these two layers independently.
Basically it implements the scheduling of the application software modules in different OS tasks and it is responsible for the I/O communication with the lower layer.
All the APIs available in this layer are written in C code and they are directly integrated in the development environment (Mathworks), but they will be analyzed in detail later.

- **Application Layer**

This is the highest layer of the architecture and it implements the specific automotive application. It is always distributed in different software modules with the scope to have more scalability, reusability and an easier implementation of the entire application.
The specific automotive application will be developed in MATLAB/Simulink using a model-based approach and in this way there will be many advantages. Following these guidelines the workflow will be more understandable and easier to manage.
Finally, different software modules can communicate each other exchanging input and output data that they can use for their control logics or in general their purpose.

2.2 Model-Based Design approach

Model-based design (MBD) is a fundamental approach in the automotive field for the development of real systems and it is used in many other areas.
It is basically the practice of doing simulation in a development environment to understand the behavior of a real physical system that will have to be built and controlled.
Each component of the physical system is represented through a model and can cover a wide range of disciplines such as mechanical, electrical, hydraulic, thermal, pneumatic, etc.
A physical system is usually defined as a set of components which interact each other exchanging information or data and perform a certain number of tasks.
In the MBD a model of a physical system tries to reflect the mechanism inside the real system using fundamental physical laws and engineering principles. Therefore, relying on the accuracy level of the system description, the entire model can be more or less similar to the real one.

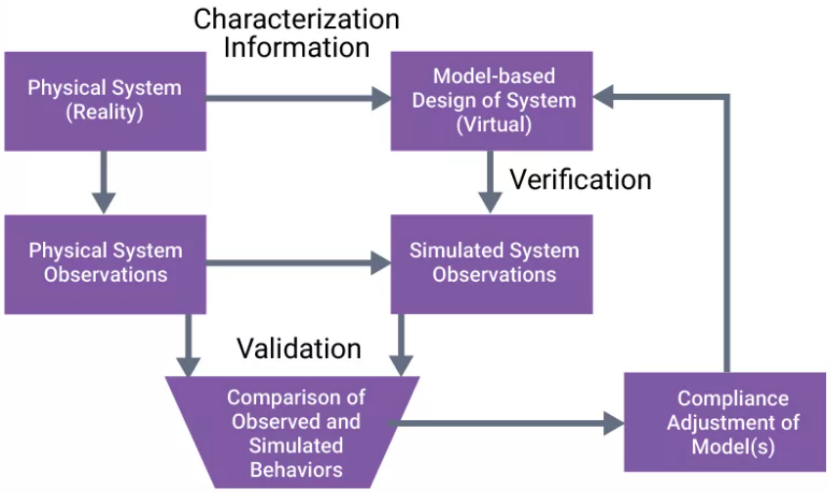


Figure 5 - Build a valid model

A key point of the MBD is the abstraction from specific realization technologies using a high-level language that have a visual approach, so roughly speaking through lines and block.
A graphical tool helps to develop high complex function with less effort, specially in real complex systems where split the entire model in more simply modules can make the work much easier to do and understand. One of the most famous tools used in the automotive field is Simulink, developed by MathWorks and directly integrated in MATLAB.
Using support tools, simulation and validation can be executed on the model (MIL) and once the model is ready and the expected behavior is correct, the Embedded Coder (a tool of MATLAB) will take care of generating the related software code, following a setup that the user can specify to obtain the desired code and files generated. This increases a lot the productivity and the efficiency because in this way is much easier generate the application software than write by hand the code of the entire model.
Testing the model before the integration in the real hardware target leads to reduce potentially expensive physical prototype iterations, in fact is possible to verify the design and the requirements of a system before its construction avoiding the waste of resources in terms of costs and times.

MBD in practice, is based on the separation of the application and the infrastructure to enhance the reusability of the model across different infrastructures. The basic idea of this concept is to model one time and build everywhere, for every type of hardware technology.
For all these reasons this type of approach has become very popular in the automotive field.
The complete workflow for the development of a system that follows the MBD will be explained later in the chapter 2.3 going on to detail the steps involved.

2.3 V-diagram of MBD flow

Model-Based Design follows a precise workflow divided in steps described by the V-diagram below.

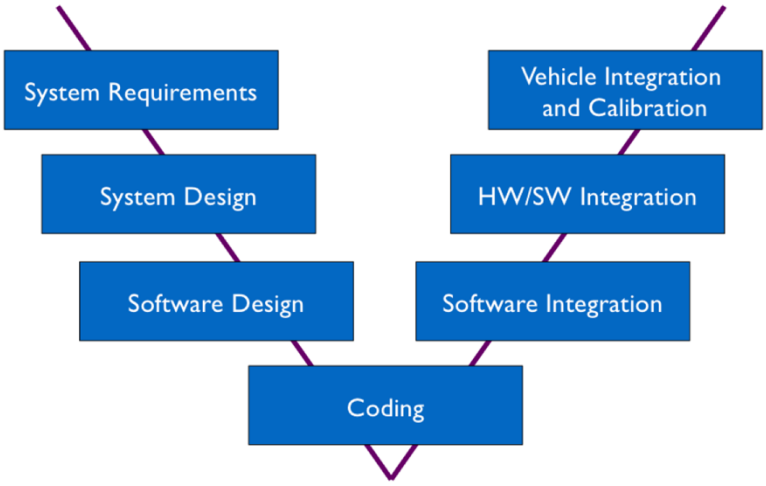


Figure 6 - V diagram

1. **System Requirements**

   The first important step to do is the analysis of System Requirements. It consists in a file, generally called System Requirement Document (SRD), that provides a detailed and clear description of the system in study and includes the declaration of all elements that are necessary for the correct implementation and operation of the system.
   The SRD is organized in a hierarchical way in order to be clear and understandable. At the higher level are described general system requirements while each 'child' of these higher requirements explain in detail what the individual component should do.
   Each requirement must be described in a detail way in order to be follow from the team of engineers that will develop that component in the future.
   System requirements describe the hardware components such as mechanical or electrical parts and functional requirements so functions that the system and all its sub-components should properly perform.
   In parallel to this document is also necessary a Software Requirements Specification document that has the goal to describe the function that software parts of each system component should execute.
   Each line of software requirement must have a reference to the system requirement to which is connected, a brief description of what it should do and an ID in order to draw up, later in the development of that component, some test case to verify that the software requirement is satisfied and to insert all the test result.
   In this way every system requirement is linked with some software requirements and the workflow is facilitated because the entire system is more modular. The single system requirement will be satisfied when all its software requirements will work properly.

2. System Design

The next step in the MBD flow is the System Design and it consist in describing all the modules, components and units that compose the system.
This process of design is at a high level of abstraction where an engineer can still evaluate and estimate some features about the system such as reliability and costs.
Iterations that usually are done in the design of a system will be made in this step so potentially problems can be solved before moving on the next phases of the flow.
Generally, to guarantee an optimal system design there are some practices to follow.
First of all, the communication between engineering teams must be done in the early stages of the development, so each of them should present many ideas as possible in order to have the best organization for the success of the project. The goal of all these preliminary stages is always the same that is to arrive at the development stage having the clearest possible ideas of what to do and so try to find the best solution for the problem before the implementation.
Another practice to follow is to make the design of the system as scalable as possible, because in this way it is ready to future improvements or additions.
A simple design is the key to success, it should be as clear as possible with the scope of be understandable by everyone.
Finally, through documentation is fundamental to ensure the validation and verification in the next step.

3. Software Design

In this step the system is modelled as a Platform-Independent Model (PIM) and in a suitable Domain-Specific Language (DSL) such as Simulink that is made of blocks that are very close to many domains like mechanical or electrical.
When the design of the entire system, that for example in a control system is made by the plant and the controller, is ready it is possible to simulate it several times in Simulink and in that way it helps to refine the model/controller and consider possible alternatives design.
This iterative phase which includes the first three stages of the V-diagram is called Model-in-the-loop testing (MIL).

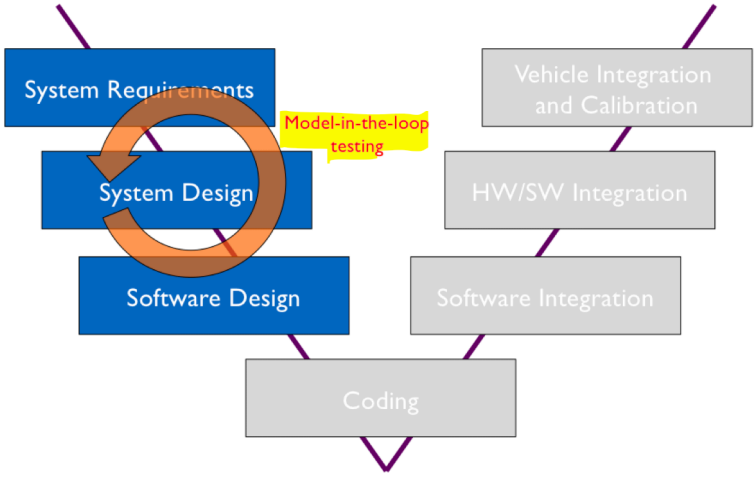


Figure 7 - Model-in-the-loop testing

Since the entire model exists in a simulation tool is very useful this type of testing in order to find possible bugs that in the future phases of the development would be much more serious in terms of time and costs. If this type of testing would not possible you should have the real system for doing test and this is not possible in many cases, furthermore it would cost a lot of money.

4. Coding

When you are sure that the system behaves as you expected, you can proceed with the code generation of your model. This is an important step because the model that is generated will run in the real system so you should try to optimize the implementation for the real target hardware.
There are various tools for the automatic code generation and each of them is designed to work with a specific program language.
Tools like Embedded Coder in Simulink have a configurator for the automatic code generator and permits you to define some guidelines and rules for reaching the desired code generation in terms of file created and programming style.
The main advantage of automatic code generation is that every time the model change, the code will update automatically so it has the goal to minimize the time required to write the code (minimizing also the costs) and to reduce the risk of manual coding errors.
Automatic code generation has become a widely used technique in recent years due to the increase of complexity of modern systems. As they become more complex, the effort for developers to manually write the code grow up and this technique offers a solution to this problem.

5. Software Integration

Once the code is generated it must be verified that it works and does what is expected and specially that the results are the same of the model-in-the-loop test phase.
This stage of verification is called Software-in-the-loop (SIL) and it essentially consists in running the generated code on a local computer and verify that the controller works properly.

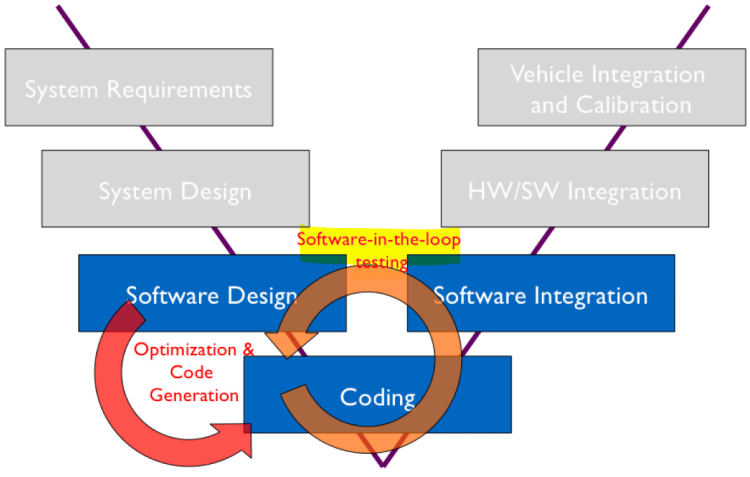


Figure 8 - Software-in-the-loop testing

During the simulation the plant remains in native simulation tool as Simulink while the controller running as executable code.
If it works no properly it means that there was an error in the generated code or in the model so they must be reviewed and corrected.

6. HW/SW Integration

After the correct integration of the software now is time to integrate the resulting code in the real embedded hardware like an ECU. In this step the software is deployed in the target hardware and it is co-simulated with the plant model to verify its correctness.
Also in this step the result must be the same of the previous MIL and SIL testing and if it is not the case some adjustment must be done.

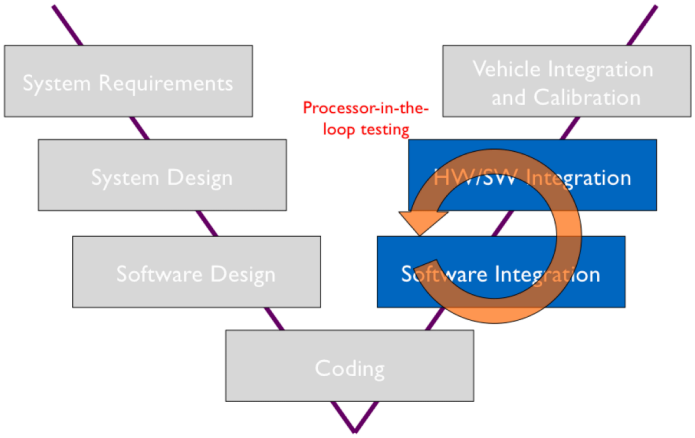


Figure 9 - Processor-in-the-loop testing

This iterative test phase is called Processor-in-the-loop (PIL). Here the controller run on the real embedded target hardware while the plant still remains simulated in the simulation tool and so it is missing the real time. The controller runs at a certain frequency and must communicates with the plant that is still simulated in the development PC.

7. Vehicle Integration and Calibration

In this final step the plant is simulated in a real-time simulator, so it performs simulations that are very close to the real word such as physical connections, I/O and communication protocols. Real time means that one second in the simulation are equivalent to one second in the real system.
The goal of all this step is to find issues related to interfaces and communications before going in the real system. The sooner errors are found, the lower is the cost to solve them and for this reason all these phases are made sequentially and iteratively with the scope of arriving in the real system without any problem.

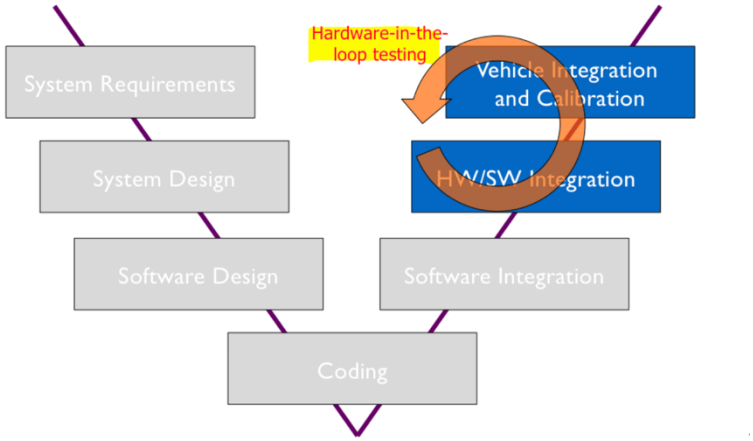


Figure 10 - Hardware-in-the-loop testing

After this final test phase, the product can be released end tested in the real word environment. Automotive customers typically adopt vehicle fleet tests to verify that the product respects their requirements. Once this last test session has been completed, the product can be considered as mature and producible. Design phase can be considered as completed and the production phase starts.

3. General purpose automotive HW ECU

An ECU (Electronic Control Unit) is an embedded system that has the purpose of control one or more electrical systems in the vehicle. Nowadays vehicles are equipped with many ECUs, each of which plays a specific role and thanks to the communication between them the correct functioning of the entire vehicle is guaranteed.
Some examples of the modules implemented in automotive field ECUs are the following:

- Engine Control Unit
  It controls multiple systems to guarantee the correct internal combustion engine. Main systems that are controlled include the Fuel Injection system, the Ignition system and the Variable Valve Timing system.
- Transmission Control Unit
  It manages the electronic automatic transmission using sensors from the car and data from the Engine Control Unit to calculate the best moment for the change gears in order to achieve the optimal performance in terms of fuel economy and shift quality.
- Door Control Unit
  It is responsible for managing the functions of a vehicle door such as locking and closing, windows movements and mirror adjustments.
- Break Control Module (ABS Control Module)
  It checks the braking system using data from wheel-speed sensor and from hydraulic break with the goal of release braking pressure at a wheel that is on the verge of lock up and start skidding.
- Battery Management System
  This module has the purpose to monitor the state of the vehicle battery in terms of voltage, temperature, current and state of battery cells.

As mentioned before, the hardware platform used for this thesis work is the HDS9 (*Figure 1*) and it is an Engine Control Unit for methane application currently in production by Metatron.
It is used in many fields such as medium and heavy-duty vehicles (buses and trucks), off-road vehicles (tractors and operating machines) and also stationary units mainly with natural gas to generate electricity.

3.1 Hardware architecture

The key elements of this ECU are essentially described in these macro areas:

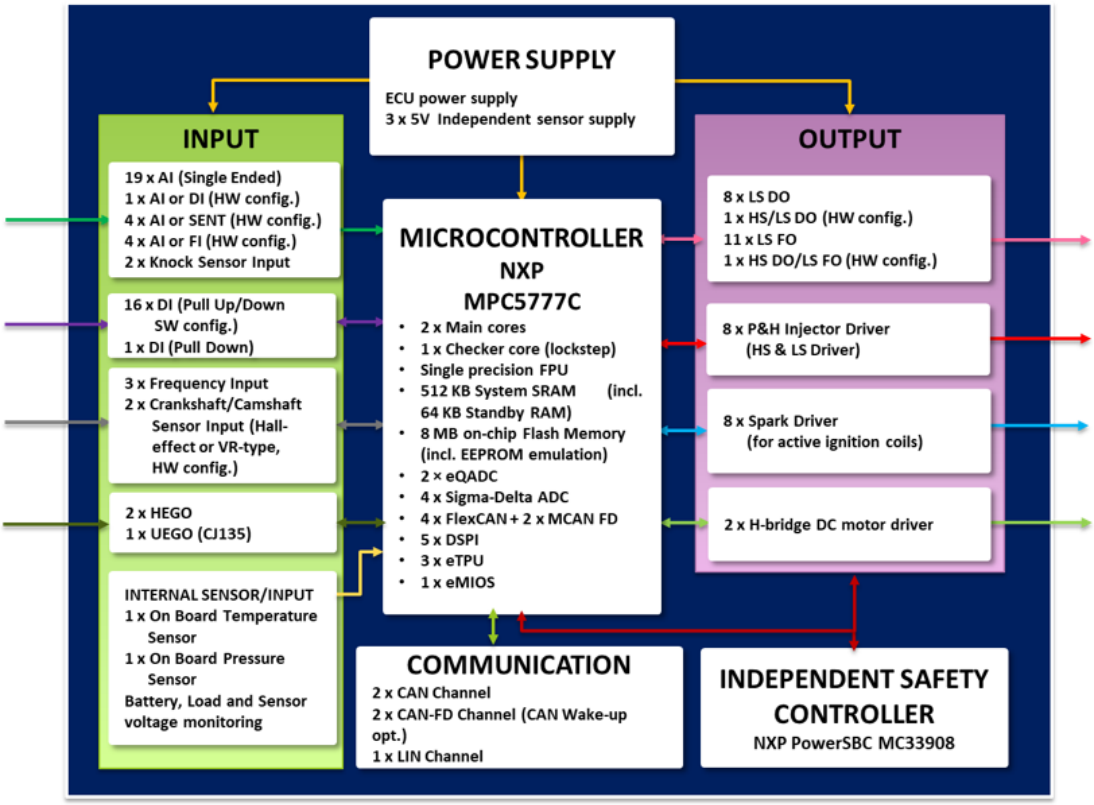


Figure 11 - HDS9 Hardware architecture

1. **Input**
   The available input channels in this platform are of Analog and Digital type. Analog channels are typically related to voltage sensors, such as temperature, pressure, actuator's position feedback, and level sensors. Digital channels are typically used for switches or binary level sensors.
   There are also some specific types of input such as SENT (Single Edge Nibble Transmission, that is a point-to-point protocol used from sensors to transmit data to the controller) and Frequency Input, typically used for the speed sensors.
   Input include HEGO (Heated Exhaust Gas Oxygen sensor) and UEGO (Universal Exhaust Gas Oxygen sensor), Crankshaft and Camshaft sensors, and Knock sensor.
   The board has also some internal sensor for monitoring the on-board temperature and pressure.

2. **Output**
   As output channels are available Digital Output and Frequency Output (PWM).
   These output channels are necessary to control for example actuators connected to the ECU and, based to the use case, it can be used a Low Side or a High Side channel to load (mainly resistive and inductive loads).

Usually, the Digital Input channels are used for ON/OFF actuators (such as electro valves) or for lamp indicators. Frequency Output are typically related to proportional actuators or gage indicators.
On the board are also present Peak&Hold Injector drivers and Spark drivers for active ignition coils able to manage up to 8 cylinders. Moreover, some H-bridge DC motor drivers are also present on the device.

3. Microcontroller

There are several types of microcontrollers for embedded systems from different companies like Freescale, Intel, Infineon etc. with different specifications.
The one chosen for this board is the NXP MPC577C of Freescale company. It is used for automotive and industrial engine application that require high performances and functional safety (ISO26262).
Here some general features:

- Two independent Power Architecture z7 cores (300 MHz)
- Single z7 core in lockstep that runs the same set of operations at the same time in parallel in order to detect and correct possible errors
- 8MB Flash memory
- 512kB SRAM (to have better performance than DRAM)
- Sigma-Delta and eQADC converters (analog to digital converters)
- eMIOS (enhanced Modular Input Output System) timer with 32 channels to generate or measure time events
- eTPU (Enhanced Time Processor Unit) timer with 96 channels to perform complex timing and I/O management regardless to the CPU

4. Communication

The hardware platform in use has 4 CAN channels that are fundamental for the communication in the automotive field. They permit the communication with other systems in the vehicle and in this way the ECU can work properly. It is also present a LIN channel that has the same purpose of the CAN, but it is based on a master-slave type of communication instead of a broadcast protocol.

3.2 CAN Communication

The Controller Area Network is the system which all the ECUs of a vehicle are interconnected. An ECU can exchange information through the CAN bus sending broadcast data, so the other nodes of the network, after receiving and checking these data, decide if accept or ignore them.

The physical communication happens via the CAN bus wiring harness consisting of two wires, CAN Low and CAN High, that have different voltage levels and are terminated with a 120Ω resistor. In particular, CAN High varies from 2.5V to 3.75V while CAN Low from 1.25V to 2.5V. When both CAN High and CAN Low voltage is 2.5V the signal is called 'Recessive' and it takes on the meaning of logical 1. Vice versa when CAN High is 3.75V and CAN Low is 1.25 the signal is called 'Dominant' and it is equivalent to the binary value of 0.
Using twisted pairs makes the CAN bus less sensitive to inductive spikes, electrical fields and other noise, so it is more robust.

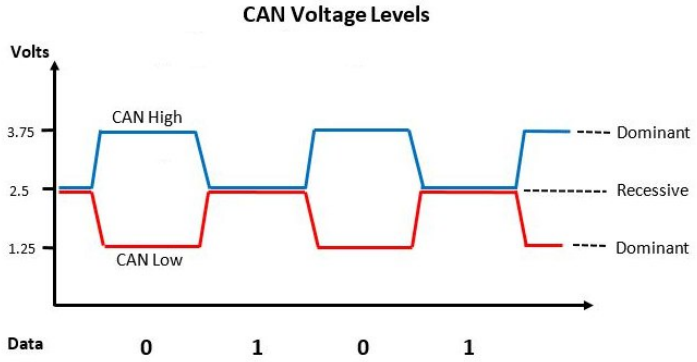


Figure 12 - CAN High & CAN Low

At the physical layer the baud rate of the classical CAN bus (High-Speed CAN) is up to 1Mbit/s but with the new protocol released in 2012 by Bosch called CAN FD (Flexible Data-Rate) it can go up to 5Mbit/s. This type of data-communication protocol is used in modern automotive ECUs which need a higher transfer rate to manage data with larger size in a faster way.

The main advantages of the CAN bus can be resumed as follow:

1. **Efficient and low cost**
   Thanks to a single CAN bus there is a sharp reduction of wires, weight end costs.

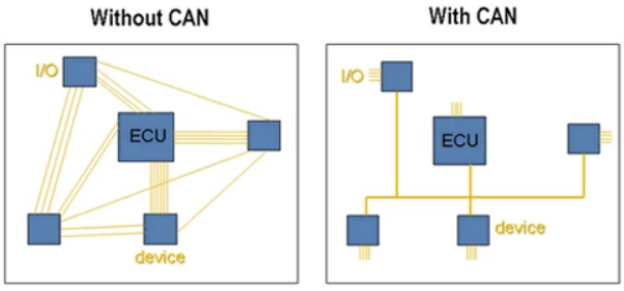


Figure 13 - Wiring comparison

2. **Easily accessible**
   The CAN bus is easily accessible because is possible to exchange data with all ECUs by a single access point making also easy diagnostic, data logging and configuration.

3. **Robust**
As mentioned before this bus is robust against disturbances and interferences, so it is very useful for application that require high levels of safety such as all the vehicles.

4. **Priority**
CAN messages are prioritized using the frame ID, in particular lower values have higher.

Communication is done through CAN frames which can be standard or extended.

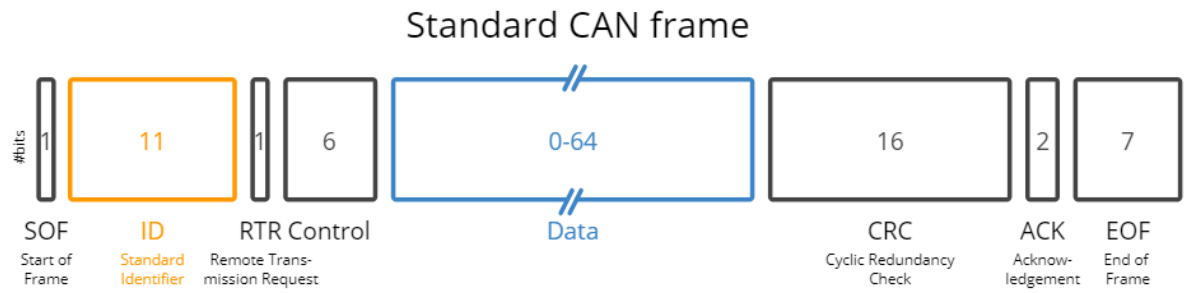


Figure 14 - Standard CAN frame

As it shown in the *figure 14* the CAN frame is composed in various field which are:

- **SOF** (start of frame): indicates the beginning of the frame and it is a 'dominant' 0.
- **ID**: is the identifier of the message, it uses 11 bit for standard frame and 29 bit for extended frame (used for heavy-duty vehicles in the J1939 protocol)
- **RTR** (remote transmission request): indicates if a node is requesting a certain frame from other nodes or is sending new data
- **Control**: includes the Identifier Extension Bit (IDE) that is '0' for the standard frame and the Data Length Code (DLC) that specifies the length of the data in the message (up to 8 bytes)
- **Data**: contains the payload to be transmitted of length indicated in the DLC
- **CRC** (cyclic redundancy check): used for error detection
- **ACK** (acknowledgment): indicates if the node has received data correctly
- **EOF** (end of frame): indicates the end of CAN frame

Raw CAN data frame without a decoding system are useless. For this reason, to interpret correctly all the frames that travel on the CAN bus, it is necessary a CAN database called DBC file. It contains decoding rules for the ID frame to understand signals from the payload.

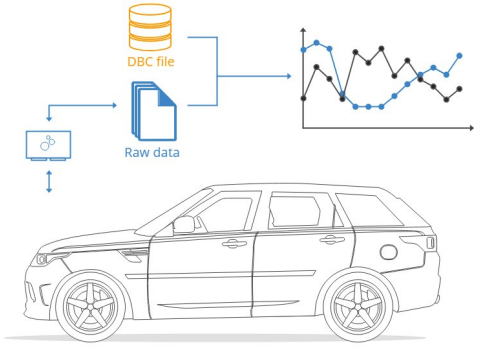


Figure 15 - Interpretation of raw signals

Generally, the messages that are sent or received, consist in one or more signals (in particular cases they have no signals) that describe data.

Here is an example of how messages and signals are described in the DBC file.

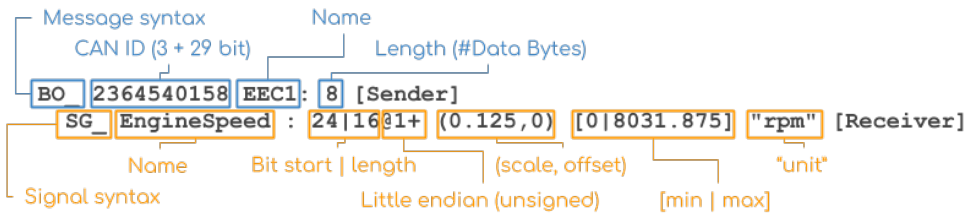


Figure 16 - CAN messages and signals in DBC file

Through this description signals are decoded in physical values and they can be effectively used.
As it can be seen (f*igure 16*) the message is identified by the CAN ID, preceded by 'BO_'. It must be unique (because it represents the address) and in decimal form. Then, in the same row, is described the name of the message, the DLC and the node of the network who send it.
Below and indented from the message, are present all its signals, each of them start with 'SG_'. Then all the parameters of the signals are described to give rules for decoding and read correctly physical values of the signals.
One of the goals of this thesis work is to give the possibility to integrate directly in MATLAB this type of file in order to send and receive message as fluently as possible, without additional complications related to implementation details. In this way, when you are modelling your automotive system, you have available some tools (library blocks) that permit you to send or receive message very easily. The strong point of this work is that you can directly send or use a signal without worrying about physical implementation.

As said before, the board has 4 CAN channels available, each of them is used for a specific purpose. The CAN 1 channel is set up for the communication between the ECU and the measurement/calibration system. Through this channel it is possible to read (measure) and modify (calibrate) signals and parameters of the ECU. This communication is made using XCP protocol, that is an interface to have access in r/w mode with the memory of the ECU. The memory access is address-oriented and the associations between symbols and address range is described in the A2L file. XCP works with a master-slave paradigm, in particular the measurement system assumes the master role while the ECU driver is the slave, so it responds to memory access requests.
This system can work with different type of transport layer, included CAN and CAN FD.
Some of tools that can be used for this purpose are Vector CANape and Etas Inca (for the thesis work it has been used the first tool).

The CAN 2 channel is used for the intravehicular communication, so with the other ECUs of the vehicle and in heavy-duty system it uses the J1939 protocol. Compared to light vehicles, in the heavy-duty systems there is a greater trend to make the communication as standard as possible. The J1939 protocol comes in handy defining an open standard for the communication in the commercial vehicle area. It comes from the SAE (Society of Automotive Engineers) and provides a Higher Layer Protocol based on CAN physical layer.

The CAN 3 channel is generally used for the vehicle diagnostic system. It is based on UDS (Unified Diagnostic Service) protocol and it is used to check errors and reprogram the ECU, so in case of a fault is possible to flash a new firmware in the Electronic Control Unit to solve the problem.

UDS works in a client-server modality, in particular the tester acts as client sending UDS requests while the ECU is the server that responds to the client.
To do this you have the possibility to connect a CAN bus interface with the OBD2 connector and start a diagnostic session to check the correctness of the system.
The OBD2 connector allows you to access information very easily, it is a connector made by 16 pins specified by the standard J1962. It is collocated next to the steering wheel usually behind dashboard panels.

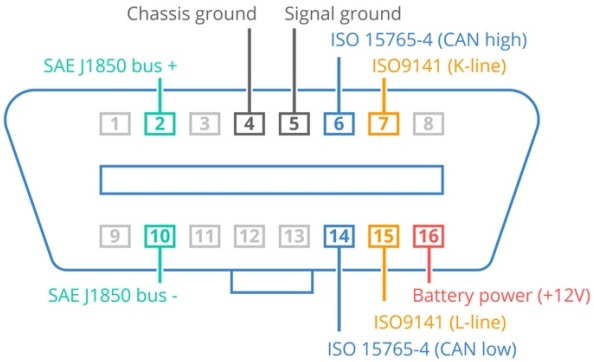


Figure 17 - OBD2 connector

As shown in *figure 17* the pin 16 is used to provide battery power and, since nowadays the CAN protocol is the most used, the pins 6 and 14 will be connected and will act as CAN High and CAN Low respectively.

The CAN 4 channel, also called private CAN, is used to implement a dedicated (and private) network among the ECM and other engine related smart devices.

Figure 18 - CAN Networks

4. API level design

This step is fundamental to reach the goal of creating a hardware platform ready to be used for developing the Application layer. The API level, as said before, puts in communication the Application layer with the Basic Software layer making available to the user a series of C functions.
All these functions are collected in a file called 'api.c' (and its relative header file 'api.h') and integrating them in Simulink through simple blocks, they can be ready for use in the modelling of the system.
Actually, this file already exists but if the user wants to use an API function in his model, it must be ensured that the function follows certain rules to be compatible with MATLAB and, after that, it should create the relative block in Simulink (usually by means of a S-Function). The major problem of this approach is the waste of time for the creation of all single blocks and the relative probability to introduce errors due to this iterative but manual task. Furthermore, after having created the block, if the user in the future applies some changes to the function, it must modify also the related block.
To solve these problems the API file has been changed with the correct rules and a block library does what the user did before.
The strong point of this work is that all these C functions are transformed in Simulink blocks automatically, so through a 'click' all the related blocks are created in the library.
This kind of approach brings a big advantage in the model development, in fact when the API file will be updated with new function or modified, Simulink blocks will be automatically adjusted accordingly. According to this approach, minimizing the number of functions in the api file would be effective and useful for having a clearer workflow.
Later there will be explained details regarding the implementation of this feature in the Mathworks tools ecosystem.

4.1 API file description

API functions covered by the activities for this thesis concern the I/O channels and the communication channels (CAN and DBC integration).

The API file contains all the enumerative types and data struct that are used by functions. They can be resumed as follow:

- `struct tTxPduInfo`
  It is used for transmitting a CAN message and it contains two fields that are the DLC and the payload (a uint8 vector of 8 elements since the maximum length of a payload is 8 bytes).

- `struct tRxPduInfo`
  Vice versa it is used for receiving a CAN message and for this reason it also has some fields related to the diagnostic that are the "DlcError" (if there is a mismatch in terms of DLC between the expected DLC and the received DLC), "TimeOut" (if the time limit for receiving the message has expired), "NewDataReceived" (report when new data is received), "ChannelIdErr" (if the channel is different from the expected one)

- `struct tPWMInInfo`
  It is used for receiving information about the PWM input signal. It contains the period and the duty of the signal.

- `enum tCANTxStatus`
  It enumerates the possible status of the CAN message in transmission. The status of the Tx message can be disabled, enabled or error.

- `enum tCANRxStatus`
  Idem for a received CAN message, the status can be OK, DLC error, Timeout error, New data error, Ch ID error and signal out of range (one of the message signals is out of range).

- `enum CANRxId`
  It enumerates all possible ID available for messages of all CAN networks. An example of ID for the CAN 2 network is 'CAN2_MSG_RX_000'.

- `enum CANTxId`
  It does the same thing for transmission messages.

- `enum DigInPinName`
  It enumerates all the available digital input pins of the board. An example of a DigInPinName is 'DIN_CH_ID_000' and it is mapped in a particular pin of the board (a dedicated file contains all the maps between name and pin number).

- `enum ANIN_Channel`
  It enumerates all analog input pins of the board.

- `enum PWMInPinName`
  It enumerates all PWM input pins of the board.

- `enum PWMOutPinName`
  It enumerates all PWM output pins of the board. They can be of two different type that are Low Side and High Side. An example of PWMOutPinName Low Side is 'PWMOUT_LS_CH_ID_000'.

- `enum DigOutPinName`
  It enumerates all digital output pins of the board in Low Side and High Side.

- `enum Std_eDiagStatusT`
  It enumerates all possible diagnostic error in the system.

Functions that deal with Input/Output channels by convention are preceded by 'API_' prefix followed by the type of I/O channel where:

- ANIN: Analog Input
- DIN: Digital Input
- DOUT: Digital Output
- PWMIN: PWM (Pulse Width Modulation) Input
- PWMOUT: PWM Output

They are described as follows:

- `uint16_T API_ANIN_getRawValue (ANIN_Channel u8ANIN_CH_ID)`
  This function returns the raw value (using 16 bit) of the specified analog input channel passed by parameter. Note that the maximum value of 2^{16}-1 is equivalent to 5 V.

- `real32_T API_ANIN_getADC (ANIN_Channel numPin)`
  It returns the value in [V] of the specified analog input channel. It is responsible for the conversion in Volt based on the channel where for example in the pin ANIN_CH_ID_000 the maximum value is 5 V while in the ANIN_CH_ID_023 (refers to the power supply) is higher.

- `uint8_T API_DIN_getDigIn (DigInPinName numPin)`
  It returns the value of the specified digital input channel and in details it can take the value of 1 or 0.

- `void API_DIN_setHwPullUp (DigInPinName u8DIN_PU_CH_ID, uint8_T status)`
  The function enables or disables (based on the status parameter) the hardware pull-up resistor for the channel specified as parameter.

- `uint8_T API_DOUT_setDigOut (DigOutPinName numPin, uint8_T value)`
  It sets the value of the specified digital output channel and return a certain value in case of error such as wrong pin number.

- `Std_eDiagStatusT API_DOUT_getDigOutErrorInfo (DigOutPinName numPin)`
  It returns the diagnostic status of the specified digital output pin.

- `tPWMInInfo API_PWMIN_getPeriodAndDuty (PWMInPinName u8PWMIN_CH_ID, uint8_T bNegative)`
  It returns the period and duty of the PWM input channel specified. The bNegative parameter allows to correctly compute the duty cycle according to the PWM polarity Low Side or High Side). The period is in microseconds while the duty in percentage.

- `void API_PWMOUT_setPeriodAndDuty (PWMOutPinName u8PWMOUT_CH_ID, uint16_T u16Period, uint16_T u16Duty)`
  It sets the period and the duty cycle of the specified PWM output channel. The period is expressed in microseconds and must be in the range [100, 62500] corresponding to a frequency range of [16, 10000] Hz. If the period is outside the range, it will be saturated. The duty cycle must be in the range [0, 1].

- `Std_eDiagStatusT API_PWMOUT_getPwmOutErrorInfo (PWMOutPinName numPin)`
  It provides information about the diagnostic errors of the specified PWM out channel.

Regarding functions that deal with communication, the most important and useful for the modeling phase are:

- `tRxPduInfo API_CAN_getRxPduInfo (CANRxId frameID)`
  It receives from the CAN Rx ID passed as parameter the PDU containing all information about the message. In particular, it is stored in the tRxPduInfo struct and in this way all the information, included the payload, are available to the user through its fields. As will be shown later, this approach is very useful in the model development because simply using the Simulink block associated to this function, the user will be able to use all data of the message in an easy way. Furthermore, since all the information about the message are available, including possible errors such as timeout error or DLC error, the user will have the possibility to manage the reception of the message based on these errors. Later it will be shown how this aspect will also automatically managed to allow the user the easiest model developing possible.

- `uint8_T API_CAN_setTxPduInfo (CANTxId frameID, tTxPduInfo data_SwSTXPdu)`
  It sends to the CAN Tx ID specified the message passed as parameters. In details, the message is a struct containing the DLC and the payload. Also in this case, the process of sending a message is automated and directly integrated with DBC files to allow the user a smoother workflow.

- `uint8_T API_CAN_setPduTxEnblDisbl (CANTxId frameID, uint8_T status)`
  It enables or disables the transmission of the message specified as parameter.

- `tCANTxStatus API_CAN_getPduTxEnblDisblStatus (CANTxId frameID)`
  It provides the enable status of the transmission message specified as parameter.

- `uint8_T API_CAN_setPduRxEnblDisbl (CANRxId frameID, uint8_T status)`
  It enables or disables the reception of the message specified as parameter.

- `tCANTxStatus API_CAN_getPduRxEnblDisblStatus (CANRxId frameID)`
  It provides the enable status of the receiving message specified as parameter.

- `uint8_T API_CAN_getPduTxStatus (CANTxId frameID)`
  It indicates if the last transmission request has been successful transmitted on the CAN bus.

In addition to these, are declared some callbacks that are necessary for each CAN network to specify certain parameters. These callbacks will not be transformed in blocks because they are not useful in the Simulink model phase, but they are fundamental later in the integration of the code.
For each CAN network are described these callbacks (here there are callbacks of CAN 2 network):

- `API_CAN2_setBaudRateCbk` (sets the baud rate for the specified CAN network)
- `API_CAN2_getDiagStatus` (returns the diagnostic status of the specified CAN network)
- `API_CAN2_setIdRxCbk` (specifies the ID, aka the address, of receiving messages according to the DBC file)

- `API_CAN2_setIdTxCbk` (idem for the transmitting messages)
- `API_CAN2_setIdeRxCbk` (specified if a receive message is standard or extended according to the DBC)
- `API_CAN2_setIdeTxCbk` (idem for transmission messages)
- `API_CAN2_getMsgStatusRx` (returns the status of a received message, so if it was properly received)
- `API_CAN2_initCbk` (sets parameters of messages according to the DBC file, in particular the init value for the transmitted messages, the period which a message has to be received or transmitted, the DLC, the timeout for receiving messages and the status of enable/disable of the single message so if it has to be sent/received)

All callbacks that are related to a DBC file were written by hand and could lead to mistakes. For this reason, all the process of populate the callbacks in the source file has been automated by a specific tool that does this job instead of user. This aspect will be explored in the next chapter.

API functions are translated into Simulink *'C Caller'* blocks[1]. This type of MATLAB blocks permits to call C functions declared in external source codes and libraries, so these files must be set in the configuration parameters of MATLAB.
When a C Caller block is created, all the values passed by parameter to the function are mapped into inputs of the block while the return value is the output of the block.

As mentioned before, to ensure that MATLAB is able to create all the blocks, some rules should be followed for a correct creation of the API functions:

- Pointers are not recommended because are difficult to integrate in MATLAB, so only parameters passed by value should be used.
- In case you need to return multiple values, you should create a struct containing all the values and return it, as in the case of '`API_CAN_getRxPduInfo`' where two different data return through a struct created ad hoc (these struct should be defined in the header file).
- To visualize input and output names of parameters in the C Caller block you have to put them also in the prototypes of the header file (api.h) just like in the api source file (api.c). In this way it will be clearer in the development phase use the C Caller block thanks to the presence of parameter names instead of a generic 'input 1' and 'input 2'.

In the MATLAB project used for the model development, it could be useful a copy of the original source file, with the simplification that the body function could be empty. This choice has been made because functions in the real source file call in turn other lower-level functions and MATLAB would not be able to handle them. In this way when the automatic code generator will translate C Caller blocks in source code, it will only write the name of the function and its parameters. This is enough for the company's goal because the integration with the real source file is done outside MATLAB.

Below is an example of C Caller block created from the `API_CAN_getRxPduInfo` function (*figure 19*), and as can be seen the received CAN message is ready to be used in your model simply by dragging a line from the block.
In particular, the block has as input the frame ID of the message to receive (in this example it is a message of the second available CAN network, CAN2), and as output the struct containing all the

[1] https://www.mathworks.com/help/simulink/ug/integrate-ccode-ccaller.html

message information. Using a 'bus selector' is possible to split the struct in its fields and use them in the user's model.

Figure 19 - C Caller block for receiving a CAN message

For those functions which have a struct datatype as input parameter (as for example the `API_CAN_setTxPduInfo`), the procedure is the reverse that is the use of a bus creator which "assembles" all single fields in the final struct.

This is another example of C Caller block created from the `API_DOUT_setDigOut` function where is set to 1 a certain Digital Output pin number and in this way for example, a LED connected to that pin can be turned on.

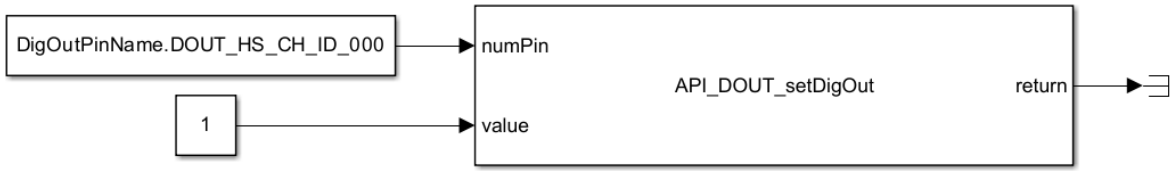


Figure 20 - C Caller block for Digital Output

These types of examples should allow easily to understand the potential of this approach and how simple is the communication between the model environment and the low level software.

Other functions present in the API file are those that are related to the management of the Operating System (OS). In particular, some specific functions are used to schedule the different tasks divided by execution time. When the code of a certain model is generated, it will be composed by a step function (and other file that will explain later) that must be inserted in the right task function based on his execution time. The different execution times available are 1ms, 2ms, 4ms, 10ms, 50ms, 100ms and 1s. The function in the API file that call these tasks is named with '`API_OS_Task`' plus the execution time. For example, the `API_OS_Task10ms` function has the purpose to call all functions that must be executed every 10 ms.
The remaining functions related to the OS are '`API_OS_LockOS`' and '`API_OS_UnlockOS`' that lock or unlock the operating system in order to prevent some possible task switching, '`API_OS_DriverEnable`' and '`API_OS_DriverDisable`' that enable or disable all external drivers.
In the future could also be managed functions in Simulink regarding the memory management (NVRAM) and the diagnostic modules (WWH-OBD) always with the idea of having available some blocks that allow the user to model more easily.

5. SW integration on the real target

When the MBD of your application is completed and the MIL testing iteration is done with the expected result, it is time to automatically generate the code with the scope to integrate it in the real target HW. To generate the code in the correct manner, it must be declared some configuration parameters in MATLAB. These parameters permit to create a software that is compatible with the ECU and the SW implementation strategies. After this step, trough other tools, is possible to generate the '.s19' and '.a2l' files that will be flashed in the ECU.

5.1 MATLAB configuration parameters for code generation

MATLAB makes available to the user many parameters that determine how the code generator produces code and builds an executable program.
The most important file generated from MATLAB are the following:

- *model*.c (contains the code for the model algorithm implementations and it is made by three main functions that are *model*_initialize, *model*_step, *model*_terminate)
- *model*.h (is the header file of *model*.c and contains the declarations of data structures, signals and calibrations used in the model. It also contains the prototypes of the three functions explained in the source file)
- *model*_private.h (contains local data that the model requires. It is automatically included along with *model*.h in *model*.c)
- *model*_types.h (provides user-defined types that the model requires)
- *model*_data.c (contains the declarations for the parameters data structure and the constant I/O blocks)
- rtwtypes.h (contains data types required by the generated code)

Concerning the functions generated in *model*.c, only *model*_initialize and *model*_step functions are necessary, without the terminate function. Furthermore, is not necessary the generation of a main function since '`API_OS_Task`' will take care to call the generated functions in '*model*.c'.

Configuration parameters are divided in multiple fields, each of which concern a specific aspect of the code generation. The main ones are described below:

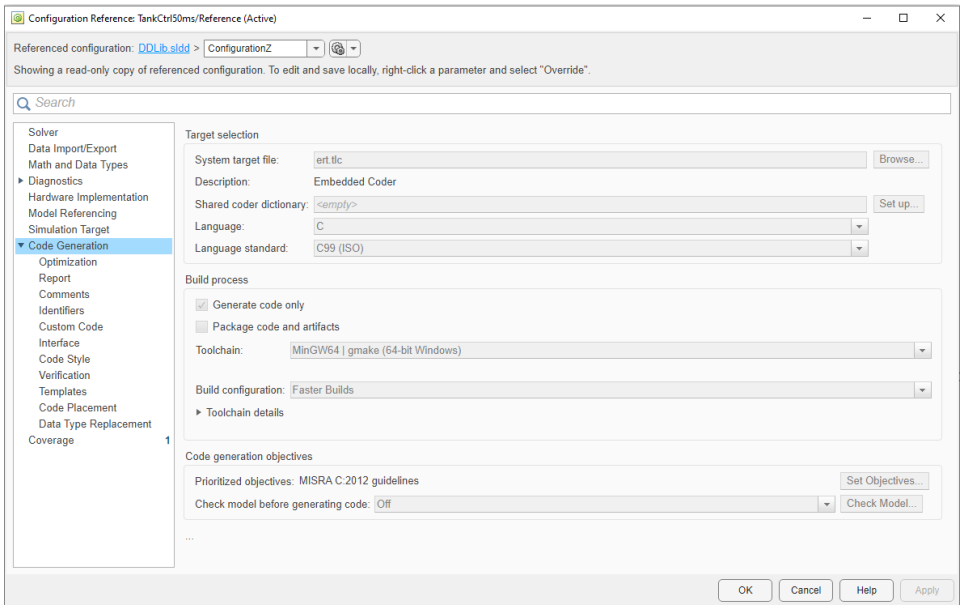


Figure 21 - General parameters

The System target file (*figure 21*) describes the file configuration used to control the code generation based on the final HW target and, in this case, it is set to 'ert.tcl' (embedded real-time target). In contrast with 'grt.tlc' (generic real-time target) that is used for a generic target (such as the host PC), the 'ert.tlc' has a better speed and memory optimization since the target is an Embedded system, so with lower power and space memory.

The language used for the code generation is the C. It follows the language standard C99 and the MISRA C 2012 guidelines. They have the scope to make easier some code characteristics in the embedded systems field such as security, safety, portability and reliability.

For what concern the build process, the MinGW64 is used to build the executable program and the build configuration option is set to 'Faster Builds' for optimizing the build time.

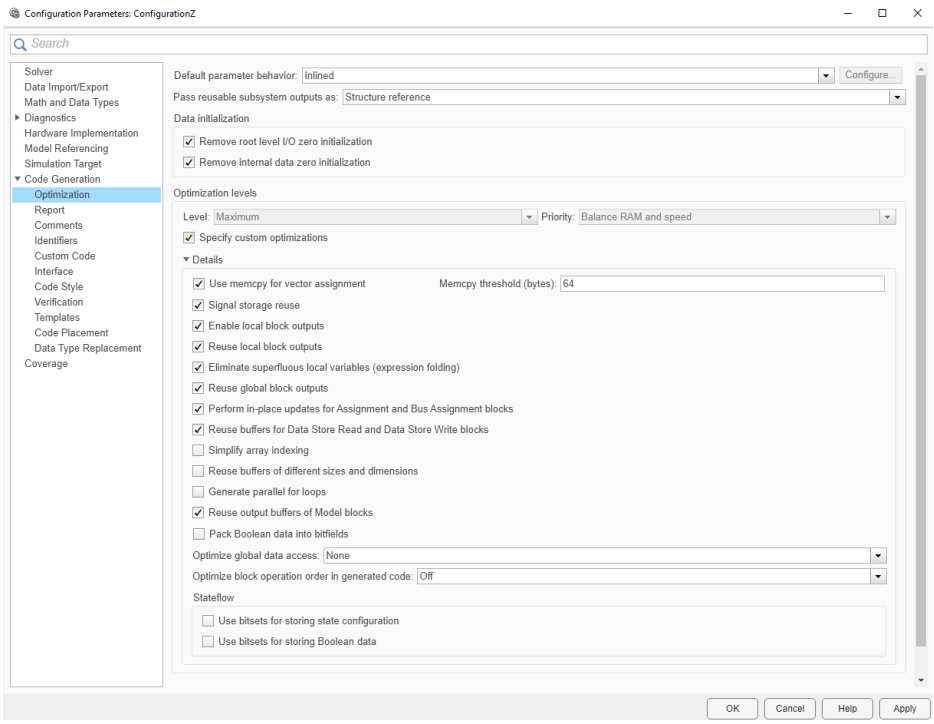


Figure 22 - Optimization parameters

As shown in the *figure 22*, MATLAB makes also available many optimization parameters for saving space memory and computational power. All these optimizations are related to the target in use, that in this case is an Embedded platform, in fact in these types of systems do not have all resources that can take for granted in a generic PC.
The first option 'Default parameter behavior' is set to 'Inlined' to not allocate memory for representing block parameters. In this way it reduces global RAM usage and increases efficiency of the generated code. The same scope has the second parameter that permit to pass reusable subsystem outputs as structure reference (pointer to it) to optimize the memory usage.
The following two flag disable the initialization of inports/outports and internal work structures, so the user will manage them. Then are available other parameters always with the scope to optimize the code generated, as for example the flag 'Use memcpy for vector assignment' that avoid for loops or 'Eliminate superfluous local variables' that increases the memory efficiency.

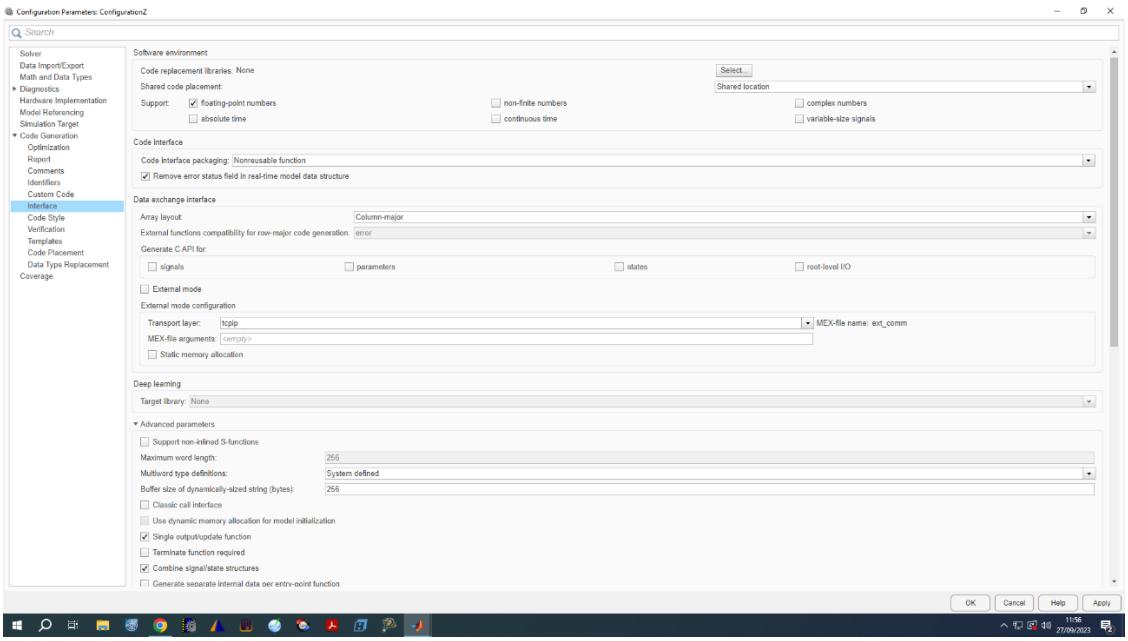


Figure 23 - Interface parameters

These parameters (*figure 23*) manage the interface of the generated code. The 'Shared code placement' is set to 'Shared location' to place the code for utility functions in a shared folder. In the support flags, only floating-point numbers are enabled for code generation while the others are disable, so for example complex numbers cannot be generated.
The 'Code interface packaging' set on 'Nonreusable function' permits to generates nonreusable code allocating model data structures in a static way.
The array layout is left to default as 'Column-major', so the matrix elements of the columns are contiguous in memory, and the transport layer used follows the TCP/IP mechanism.
As mentioned before the terminate function is not required, so the relative flag is disable. Furthermore, since it is necessary only the initialize and step functions, all API generations for signals, parameters and I/O are disable.

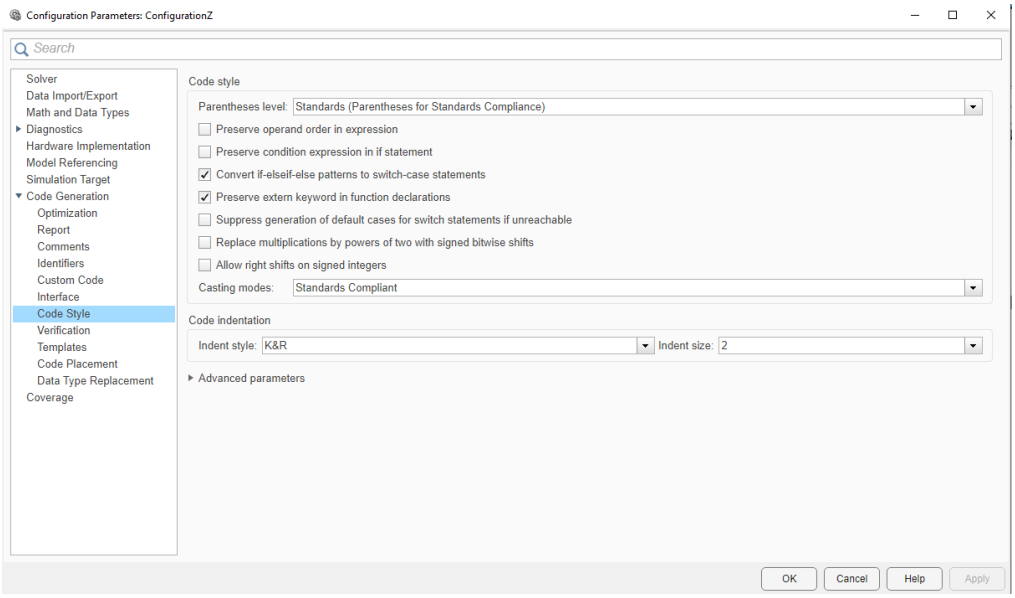


Figure 24 - Code style parameters

Code style parameters (*figure 24*) configures the appearance of the generated code. Through these parameters is possible to have a code that is conform to a specific standard, such in this case the MISRA C.
The first parameter 'Parenthesis level' is set to 'Standards' to have better code readability and to be conform to MISRA requirement. An important parameter is the 'Preserve extern keyword in function declarations' that permits to generate the model entry point functions, *model*_initialize() and *model*_step(), with the keyword 'extern' that explicitly indicates an external linkage.
The 'Casting modes' parameter is set to 'Standards compliant' to satisfy some MISRA rules, for example it can replace bitwise XOR operations with relational operations to satisfy the 10.1 MISRA rule.

In the Code Placement section of the configuration parameters windows, is possible to set the format of the file packaging. In particular, with the option 'Modular' all files described at the beginning of the chapter are generated, while using 'Compact' *model*_data.c, *model*_private.h, *model*_types.h are included in the source and header file.

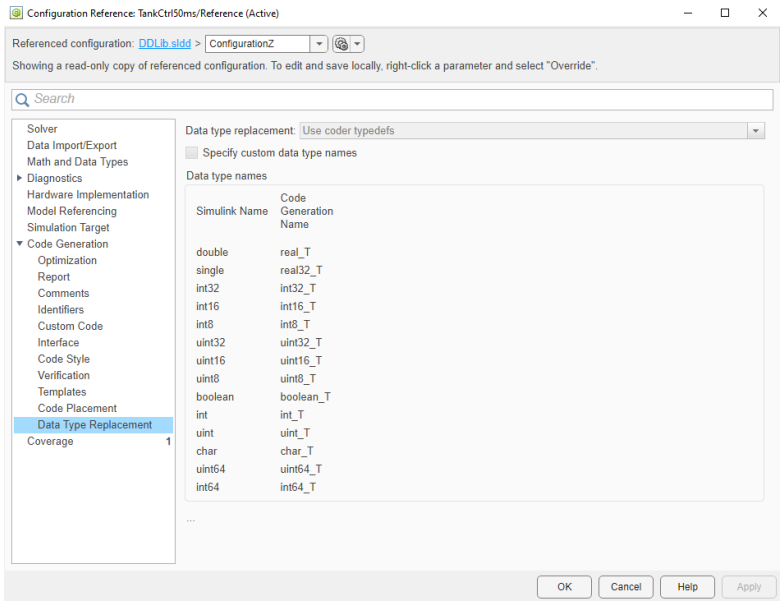


Figure 25 - Data type replacement parameters

The data type replacement (*figure 25*) section permits to replace built-in data type name with user-defined names in the code generation. In this case are used the default coder typedefs but in case of specific requests by a customer, is possible through the 'Specify custom data type names' flag to modify all names writing the desired ones.

5.2 Workflow from MATLAB to ECU

Once the MBD of a system is completed and the MIL testing is done with the expected results, as explained in the chapter before, the code of the model will be generated automatically. To run the software in the real hardware and test it, some steps have to be done.
In the PC used for development/test of the Model-Based application, should be created a project folder where insert all necessary files used for the project.
For a clearer understandability of the project architecture, the three layers of the SW architecture have been divided in different folders (BSWL, API, MBSL). The MBSL folder is dedicated for the application, in fact it shall contain all the source code files generated from the applicative models by MATLAB during the code generation.
The next step is to link these application files to the API layer, and this is done by including all the model header files in the API file.

The API file, as explained in previous chapters, contains some functions related to the OS that are executed at specific times. The '`API_OS_Init`' is in charge to do the task routine for the system initialization, so all the *model*_initialize() functions generated from MATLAB must be inserted here. In this way all models' initializations will be executed at the right time.

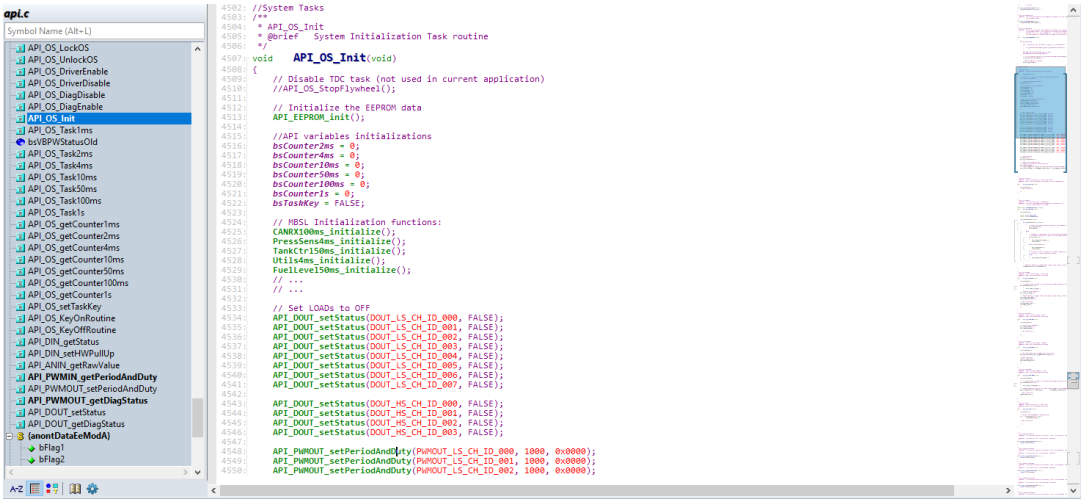


Figure 26 - API_OS_Init

As the *figure 26* shows, '`API_OS_Init`' function contains all initializations tasks such as the EEPROM initialization, the variables initializations or the I/O initializations. At line 4524 of the code in the figure, all the application initialization functions are inserted, and in this example, there are some functions related to the Demo Application that has been developed for testing the created library.

After that, *model*_step() functions must be inserted in the '`API_OS_Task`' function based on the frequency at which the task has to run. In the example below (*figure 27*) is shown the '`API_OS_Task50ms`' containing all tasks that must be run every 50ms. At the line 4700 are inserted two step functions derived from the related Simulink models. Obviously, the order in which functions are inserted depends on the application logic, so if is necessary to run model A first than model B, the order will be the same in the code.

```
4688: /**
4689:  * API_OS_Task50ms
4690:  * @brief   Task routine [50ms] - Medium
4691:  * @details MBSL can be executed from here.
4692:  */
4693: void    API_OS_Task50ms(void)
4694: {
4695:     bsCounter50ms++;
4696:
4697:     //Check Key status and trigger key on/off evet
4698:     API_OS_setTaskKey(API_DIN_getStatus(KeySw_ID));
4699:
4700:     // MBSL functions:
4701:     TankCtrl50ms_step();
4702:     FuelLevel50ms_step();
4703:     // ...
4704:     // ...
4705: }
```

Figure 27 - API_OS_Task50ms

Once this is done, is possible to proceed with the build of the project to transform the source code into an executable software for the embedded system. There are many software tools available for this task and in this case, it has been used 'HighTec Development Platform'.
After having selected the microcontroller used in the ECU, this tool manages the entire build process of the project so compiler, assembler and linker. The final result is the creation of two different files that are the '.elf' and the '.s19'.

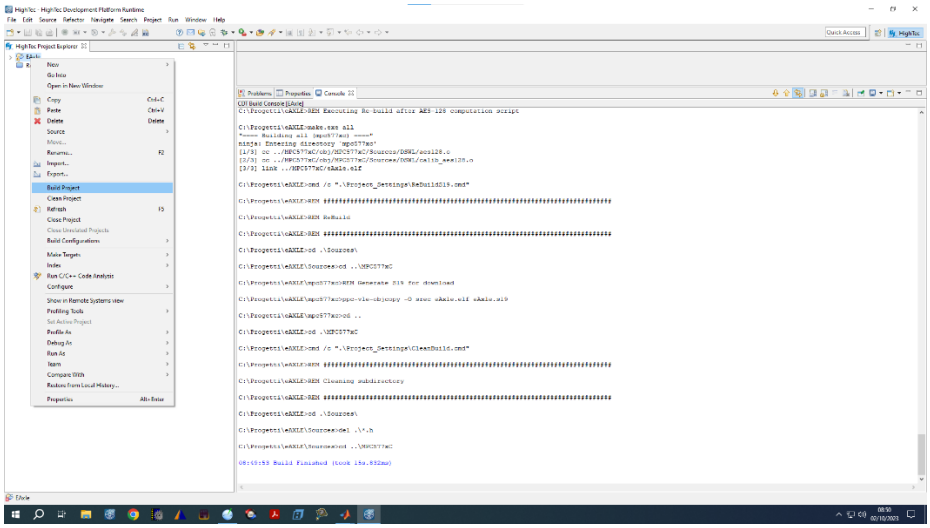


Figure 28 - Build project with HighTec

The ELF file (Executable and Linkable Format) is a standard file format for executable files, object code and shared libraries, it contains information for the execution of a program such as data memory addresses. The s19 file (S-Record file) contains the machine code compiled and it is used to program the EEPROM (non-volatile memory of the microcontroller).

The next step is the creation of the A2L file through some tools developed by Vector. The A2L file contains information about memory address and data type of all data objects (parameters, maps, signals, etc.) and, together with the s19 file, will be flash in the ECU.
It has been created a batch file to execute some instructions that allow the correct creation of the file:

- Through the Vector ASAP2 Merger tool, all A2L files belonging to the source codes of the project are merged into a single one, that will be called with the name of the project. It works with a master-slave paradigm, so is present a Master A2L file (Header A2L) that described rules for communicating with the ECU and it is merged with all the others that contains symbols.

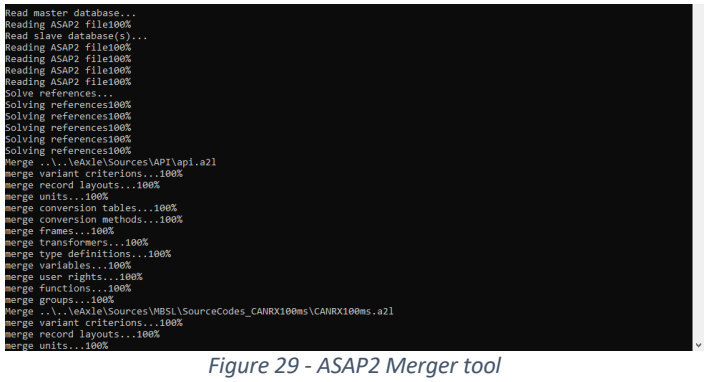


Figure 29 - ASAP2 Merger tool

- Through the Vector ASAP2 Updater, all the symbol addresses present in the A2L file are updated based on the ELF file generated from the last build process. This is done because object addresses may change from a build process to another and, if the A2L file is not update, the entire project may no longer work properly.

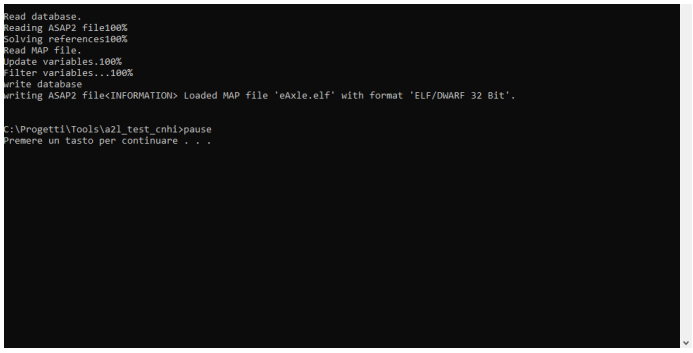


Figure 30 - ASAP2 Updater tool

When the A2L and S19 files are ready, they can be flashed in the ECU using a tool called CANape, developed by Vector. CANape is a platform able to connect with the ECU and perform many tasks including:

- Flashing (upload a new software in the ECU via XCP/CCP protocol, UDS or Ethernet)

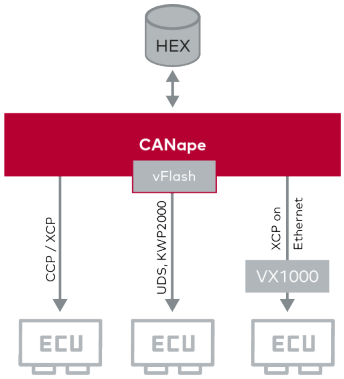


Figure 31 - Flash via CANape

- Data acquisition and analysis (measurement and processing of data and signals from ECU)

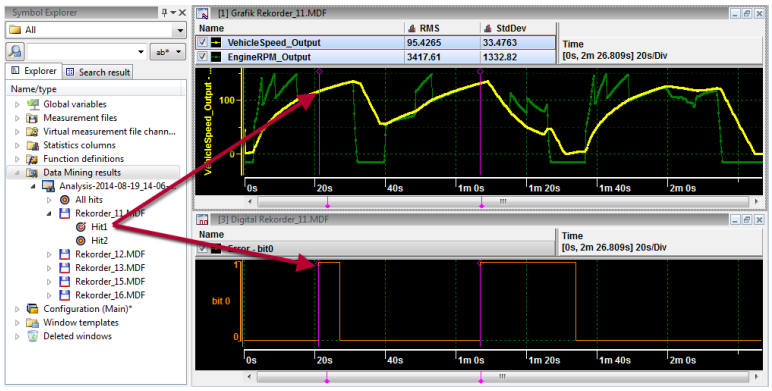


Figure 32 - Data acquisition

- Calibration (modify parameters and flash it to reach the expected behaviors of the system)

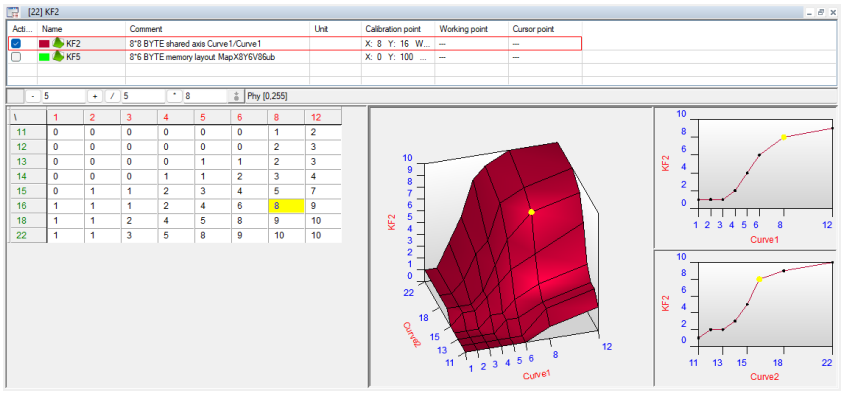


Figure 33 - Calibration

To start a session with CANape for flashing and testing your MBD application in the ECU, you should create a new CANape project (in a dedicated folder) defining all relative configurations and files needed to the project execution (such as S19 and A2L).

The figure below shows the physical links that must be done for the correct configuration of the environment work. The PC communicates with ECU through CANape using the XCP protocol.

Figure 34 - Physical links

From a practical point of view, the ECU is connected to a power supply and, as shown in the figure below, in this case it is set to 12V in DC.

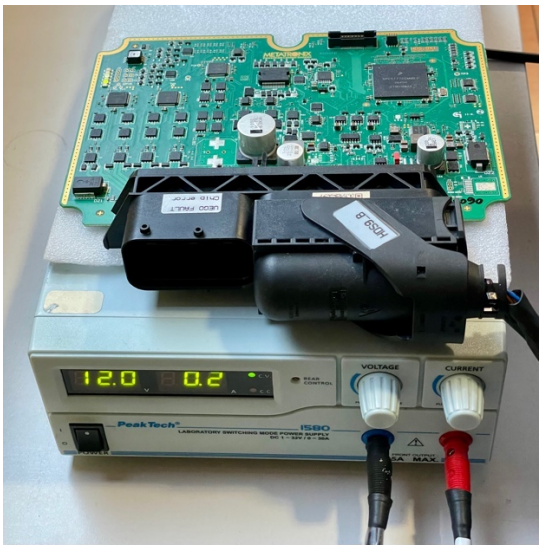


Figure 35 - ECU connected with power supply

The next step is to connect Vector CANape device to both the ECU and the PC. Concerning the connection with the PC is used an USB type B cable while a VGA port is used for the connection with the ECU.
As it can be seen (*Figure 36*), it is used the CAN 1 channel (as explained in the chapter 3.2) to exchange data with the PC and it is also necessary a termination resistor of 120 Ω (the little black box in the figure below between CANape and CAN 1 connector).

Figure 36 - Connection between PC and ECU

After correctly setting all the connections, is possible to launch the CNA file (CANape configuration file of the project). The tool will immediately notice that the software in the ECU is different from the one in the PC (basically looking the EPK, *Figure 37*) and is necessary to flash the new software.

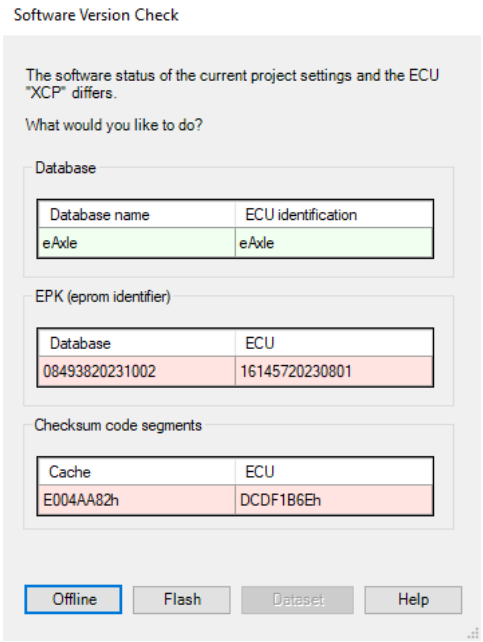


Figure 37 - Software version check

The tool makes available to the user three type of flash (*Figure 38*). 'CALIB' stands for calibrations and is used when the application software is the same but change only calibration parameters. 'APPLICATIVE' is used when the application part is different from the precedent one, for example if is added a gain block in the control module. The 'BOOT' flash is used in specific cases when is necessary to change the boot code in the memory (the code executed when the ECU is turned on).

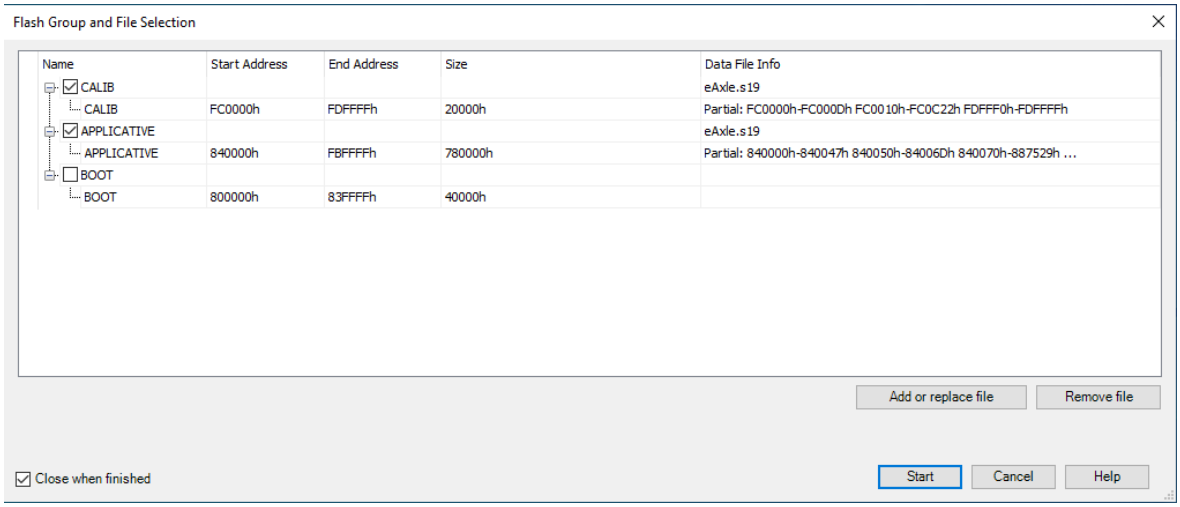


Figure 38 - Flash types

The first time that is flashed a new software version, calibration values in the software may differ from the displayed ones in CANape (*Figure 39*). The tool asks to the user if he wants to upload those in the software to the work screen of CANape or vice versa if he wants to download the old values present in CANape to the ECU.

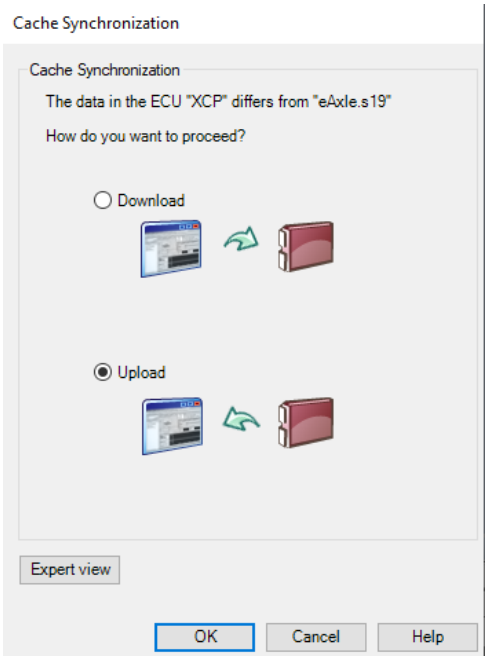


Figure 39 - Cache synchronization

After this step, if the user changes some calibration values during his test in CANape and flash them in the ECU memory, the two version of software will be aligned and the precedent popup will not appear in the next working sessions.

From a technical point of view, application and calibrations sections are divided in two different memory areas. This is done for a better management of the two sections and for enhance the security. Having the calibration section independent permits, in case of errors during the calibration writing, to preserve the applicative section. Furthermore, it can be subdivided in more sections based on their purpose and in this way is possible to give restricted accesses.

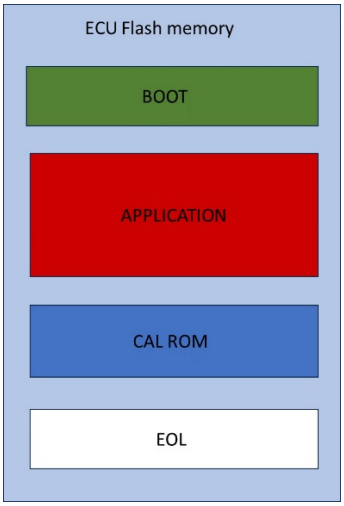


Figure 40 - Flash memory scheme

When the ECU is turned on, the MCU starts from the bootloader its routine tasks and, if is present an application software (checking the relative key presence), it updates the Stack Pointer with its first instruction. Then all calibration parameters are taken from the CAL ROM section of the flash memory and are copied in the CAL RAM section.
When calibrations are flashed from CANape to ECU, they are directly written to the CAL ROM and they will remain when the ECU will turn off.
If the application software should be flashed, the application key presence (located in a common storage area accessible by both the application and boot section) is canceled, and via CANape a request shall be sent for writing in the application section memory via XCP protocol. After that, will be possible to overwrite the application section with the new application software and to enable again the key presence.

The overall workflow from MATLAB to the ECU can be resumed in the following schema.

Figure 41 - Workflow from MATLAB to ECU

6. Modeling environment and Library design

As previously explained, the model environment used for this thesis work is MATLAB and it is one of the most famous tools for the Model-Based Design of systems.

MATLAB makes available a large number of libraries and, integrated with Simulink, is possible to develop complex systems with a MBD approach in an easy and fast way.
Furthermore, the community of MATLAB is very large and so in case of problems is easier to find online resources and receive support.

For these types of works is very useful to use a MATLAB 'Project', that is an environment where is easier to manage files of different types including: MATLAB files, DBC files, source code files, requirements file, reports, generated files etc.

For the description of the modeling environment, it has been considered the demo application developed to test and validate the created library. It tries to follow the basic rules and hints that a good project development should have.

6.1 Custom Storage Classes

To meet some specific implementation requirement, there has been created some custom storage classes that add some features to default MATLAB classes.

The custom storage class 'Calibration' has been created for all parameters that must be tunable (calibratable) during the following phases of testing.
To do this, it has been defined in MATLAB a new memory section called 'CalRam' that is a section of the RAM dedicated to calibration parameters and it is characterized by a pragma section.
A pragma is a directive that gives the possibility to assign additional information to the compiler and in this way decide some compilation details which are generally not modifiable.
Using this directive, MATLAB will add the pragma section to the code when it declares all parameters saved as 'Calibration'.
These figures show how the memory section is created and then how MATLAB declares all calibration parameters in the generated code with the correctly use of the pragma statement.

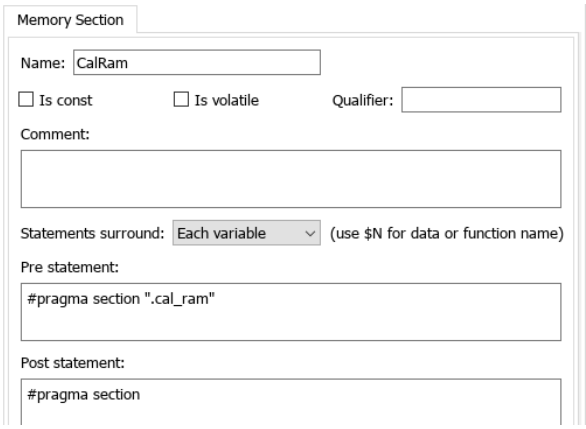


Figure 42 - CAL RAM creation

```
/* Definition for custom storage class: Calibration */
#pragma section ".cal_ram"

real32_T xsEMPTY_TANK_THR = 5.0F;      /* Referenced by:
                                        * '<S1>/Switch'
                                        * '<S1>/Switch1'
                                        */

#pragma section
```

Figure 43 - Pragma section use

The same procedure is done for the 'Map' custom storage class that is an extended class of the Lookup Table (LUT). In this way all the parameters of the LUT declared as 'Map' could be modified since is stored in the CalRam memory section. This is important when you have to find the right parameters of the table through various simulations.

The last two custom classes created are named 'Signal' (used for input/output) and 'TestPoint' (used for intermediate signals).

I/O signals are used with the custom class 'Signal', in particular input signals shall be imported from other models while output signals are declared as extern.
TestPoint signals are those that are used inside the model and only in that model. However, from a "code generation" point of view they are treated as output signals so they are declared as extern and they can be visualized in the measurement tool later.

The *figure 44* shown a use case example of these created storage class. On the left is present the LUT called 'zvRailPConv1' and it is related to a 'Map' object in the data dictionary with the same name. In the line coming out of the LUT is connected a 'TestPoint' called 'zsRailPressureTmp', as explained before it is a signal that remains in the current model and is not used from other models. After that is present a switch that permits to set a fixed value instead the real signal if a enable parameter is turned on. This parameter (zfRAILPRESSURE_VALUE) and the fixed value (zfRAILPRESSURE_EN) are saved in the data dictionary as 'Calibration' and so they are modifiable in CANape during the HIL. Outgoing from the switch is present a 'Signal' that can be used in other models.

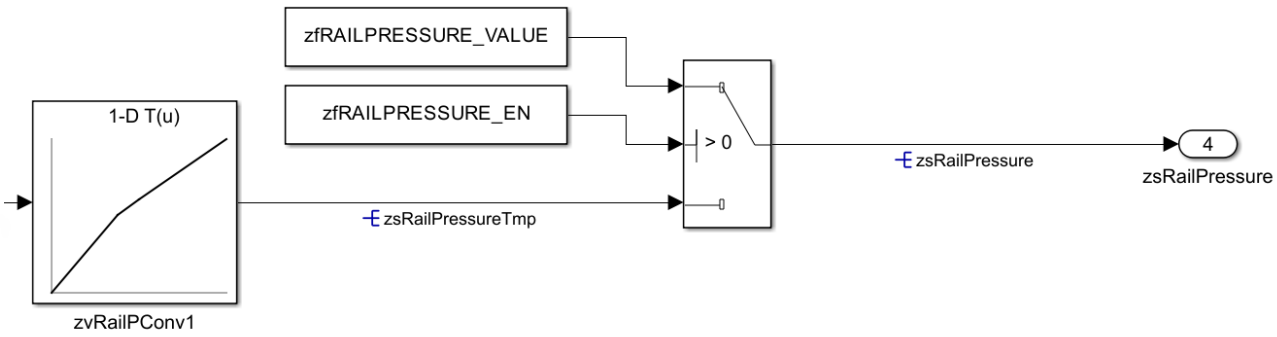


Figure 44 - Custom storage class example

6.2 MATLAB Project architecture

A project should follow a modular architecture with the goal of making the work environment well-structured for improving the comprehensibility and facilitating the system development.
After having created a new MATLAB project, should be created folders for different working areas, so for example a library folder where to put all files inherent to the library blocks, and a model folder containing all files of system models.
Regarding models of the system, they are divided by different task (e.g. ignition module, injection module, turbo module, etc.) each of which is in turn composed of multiple simulink models divided by execution time. Each model has an associated data dictionary that contains all its data such as parameters, signals or lookup tables.

Upstream of all data dictionaries is present a data dictionary that is linked with all the others and so it contains data of all models. This is done because each model will be linked with this 'father' data dictionary to visualize and use data of all the other models.
Downstream of all models is present a data dictionary containing all the enumerative data types and data struct of the project. It will be linked and so available to all models allowing its use.

In the figure below is shown the folder management of the Demo Application (chapter 7) in MATLAB. As it can be seen, in the 'DemoApp' folder are present all different models, each of which has the Simulink model, the data dictionary associated and a test harness model to verify the correct functionality of the model (MIL).
In the 'Librerie' folder are present the library and all files associated to it like the API file, custom class files and DBC files. Then is present the data dictionary 'father' (DDLib.sldd) and the data dictionary of all enumeratives and data type (eAxle_enum.sldd)

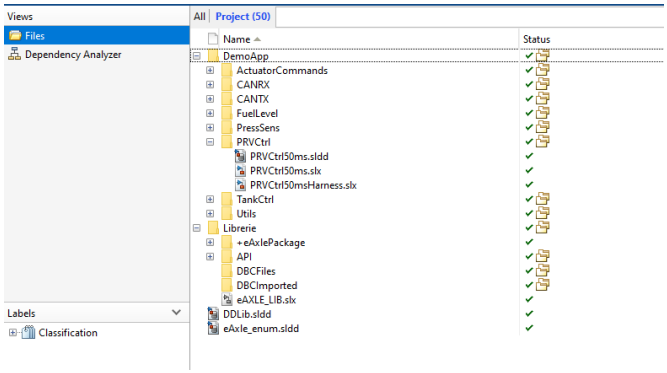


Figure 45 - Project folder management in MATLAB

MATLAB provides a very useful tool for visualizing all dependencies between project files that is called 'Dependency Analyzer'. It helps the user to understand the links present between the various models and data dictionaries.

As it is shown in the *figure 46*, all models (characterized by the red label and the suffix ".slx") are connected with the 'father' data dictionary (called in this example 'DDLib.sldd' and with yellow label) and, as said before, each model can use signals of other models. Each model data dictionary is connected to 'eAxle_enum.sldd' for having access to all enumerative data types.
The created library for this thesis work ('eAXLE_LIB.slx') could have access to the data dictionary of enumerative data types but it cannot be linked with the other data dictionaries.

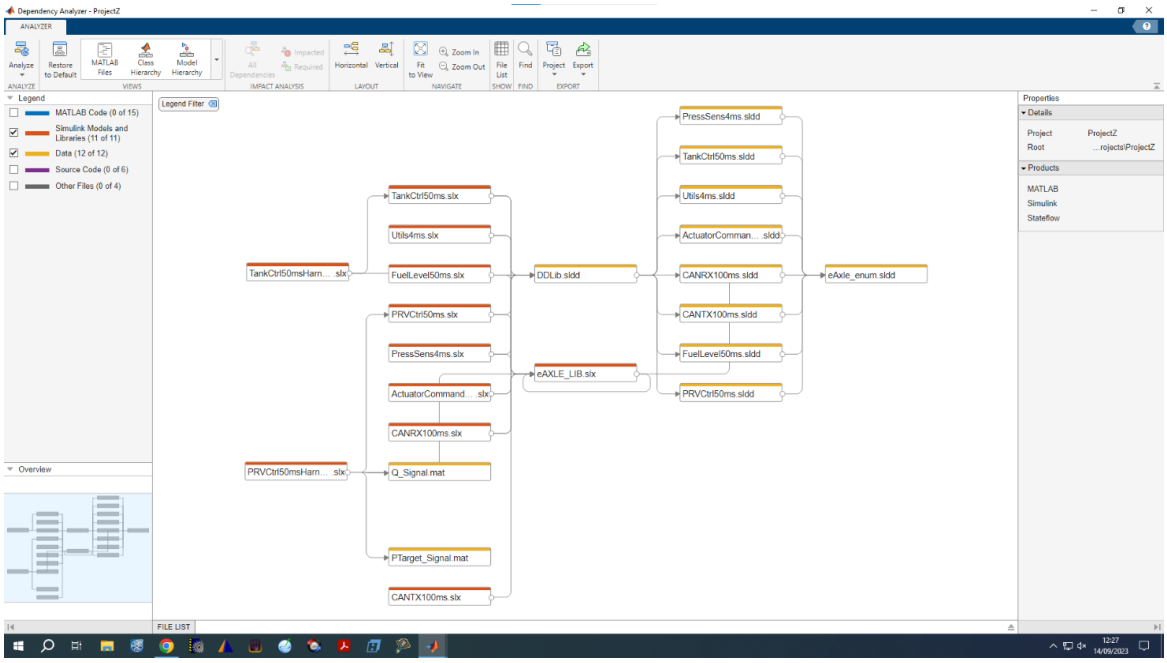


Figure 46 - Dependency Analyzer

6.3 Blocks library design

To create a new library is necessary to start Simulink and click on 'Blank Library'. Using your own library in addition to the existing ones, allows the user to have some functional blocks available that help the model development. These blocks library performs actions with the aim of reducing development time and avoid errors due to human mistakes.

All blocks that require an input from the user such as click a button, are created simply by adding a new empty subsystem to the library model and modifying its mask. Through the 'Mask Editor' of the block is possible to create a block library with which to dialog using buttons, check box, editable parameters etc. All buttons are related to a specific script file (saved in the same folder of the project with the suffix ".m") that is executed at the time the button is pressed. These script files are written in MATLAB programming language that is similar to the C language.

To have available the created library directly in the library browser some actions have been performed following guidelines in the MathWorks website[2]. This leads to the advantage of having all blocks available directly in the model by simply dragging and dropping them.

The mainly blocks that have been created are resumed in the following table.

Block name	Description	Reference
Update	Update enumerative types and API function blocks	6.3.1
Import DBC	Import in MATLAB DBC files and generate related callbacks in API file	6.3.2
CANRX_MESSAGE	Receive message belonging to DBC files imported	6.3.3

[2] https://www.mathworks.com/help/simulink/ug/adding-libraries-to-the-library-browser.html

CANTX_MESSAGE	Send message belonging to DBC files imported	6.3.4
GenCode	Build model wrapper to manage I/O signals and generate code	6.3.5

The *figure 47* shows the created library and its blocks.

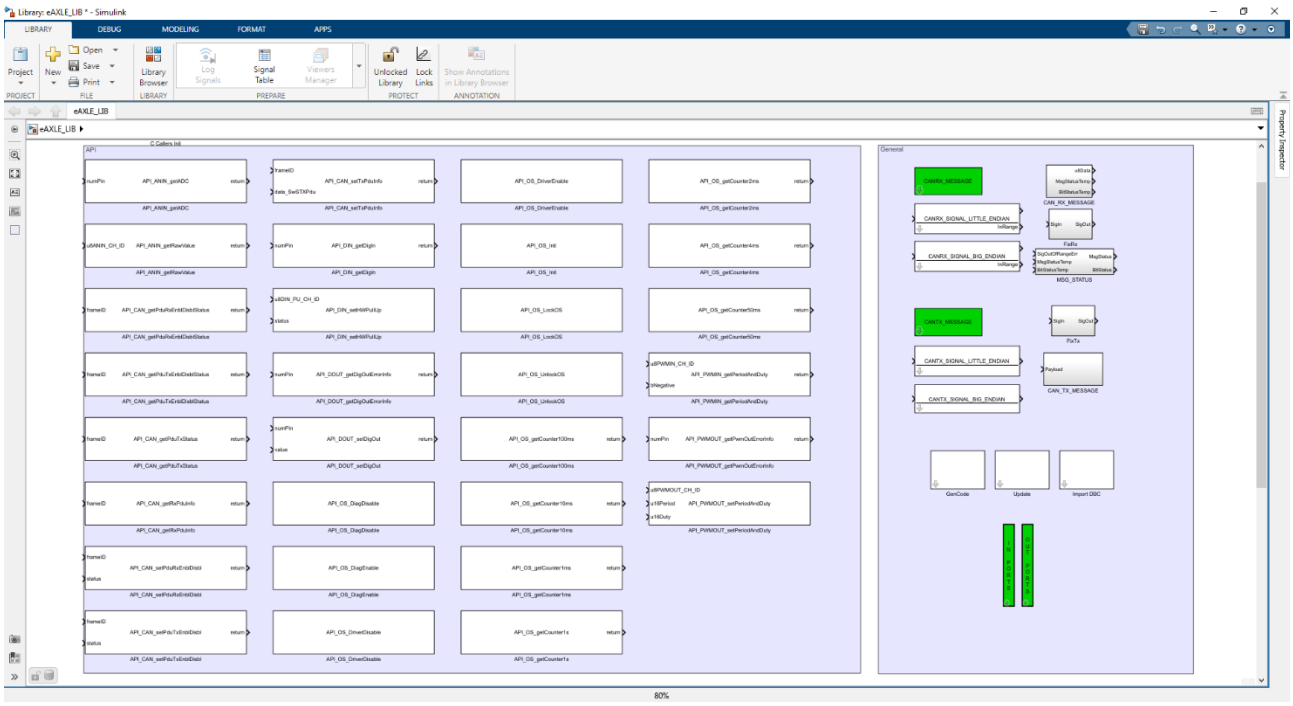


Figure 47 - Created library

6.3.1 “Update” block library

In the MATLAB environment, when you start a new project, you have to create one by one in the appropriate data dictionary all enumerative data types that are present in the API file. Once you have done, if are necessary some updates or if there are new data types in the API file that must be added, you must modify the data dictionary by hand acting one data type at a time. The same problem concerns the creation of C Caller blocks for using API functions and this can take a lot of time.

This block library has been created whenever is required to create or update all the enumerative data types and C Caller blocks from the API file.
When you have just created your new MATLAB project for the model-based design of your application, you should add the API source and header files in your project folder and declare them in the library configuration parameters under the heading ‘Code information’.

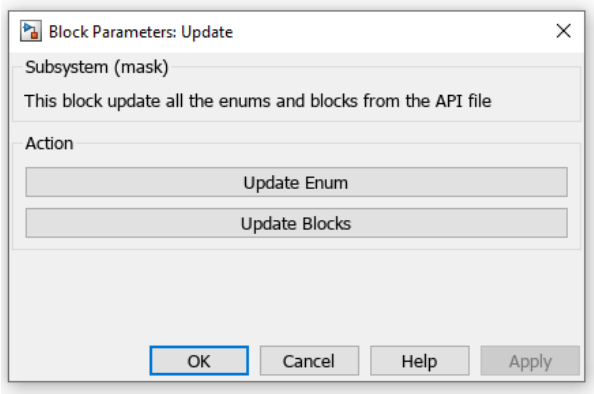


Figure 48 - "Update" block library

The block shown in *figure 48* is composed of two buttons, the “Update Enum” button is in charge to add all the enumerative data types and data struct from the API header file to the ‘eAxle_enum.sldd’ (the data dictionary used for this purpose). Before do this, it deletes all entries of the data dictionary in order to start with the original state (where the data dictionary is empty), so in case the button is pressed for an update it correctly regenerates all enumerative data types.
After that, through the “`Simulink.importExternalCTypes`” function used in the script, it effectively adds entries to the data dictionary and then it converts all storage types of ‘Native Integer’ to ‘uint8’, to be aligned with the code generation implementation. This is the data dictionary of the enumerative types after having pressed the “Update Enum” button and, as it can be seen from the *figure 49,* all data types are created (enumerative data types denoted by the yellow grid and struct data types by the three black lines) and the storage type of each data is ‘uint8’.

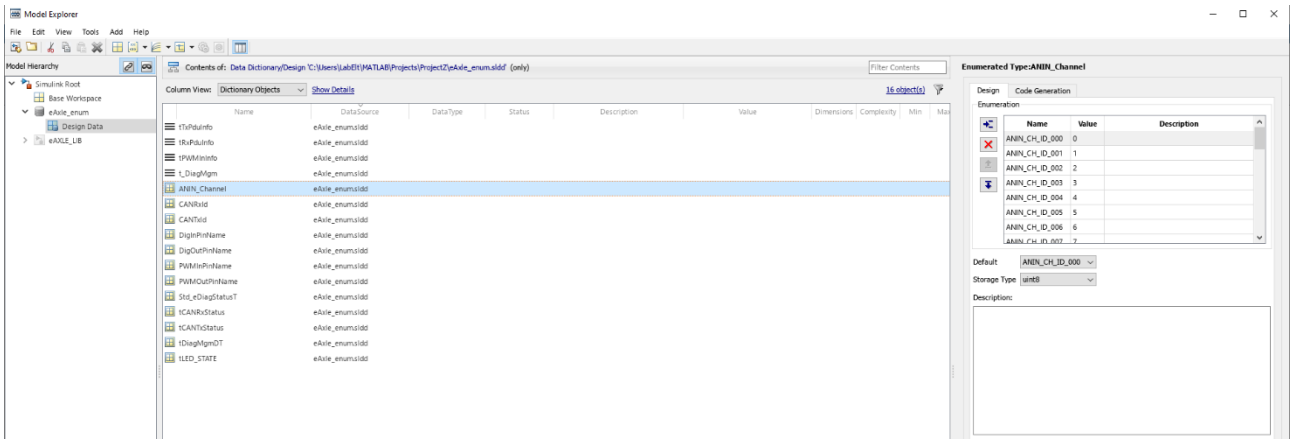


Figure 49 - "Update enum" result

The second button ”Update Blocks” performs the same actions with the API functions, so it creates all C Caller blocks with the scope of calling the API function associated. When the button is pressed, present C Caller blocks are eliminated for the same reason of the previous case, then is created a ‘C Caller block Init’ that has the scope of generate all the other blocks. The script updates the list of the available functions in the ‘C Caller Init’ (*figure 50*) and then it creates them one by one.
As the *figure 51* shows, after having pressed the “Update Blocks” button in the library will be present in an orderly manner all the C Caller blocks, ready to be used in the model.

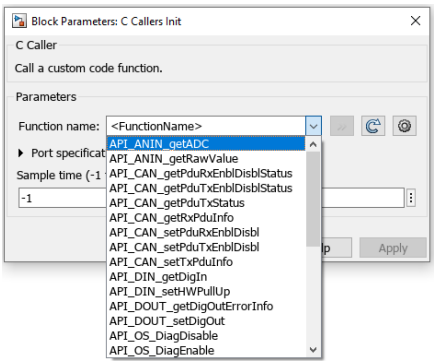


Figure 50 - C Caller Init block

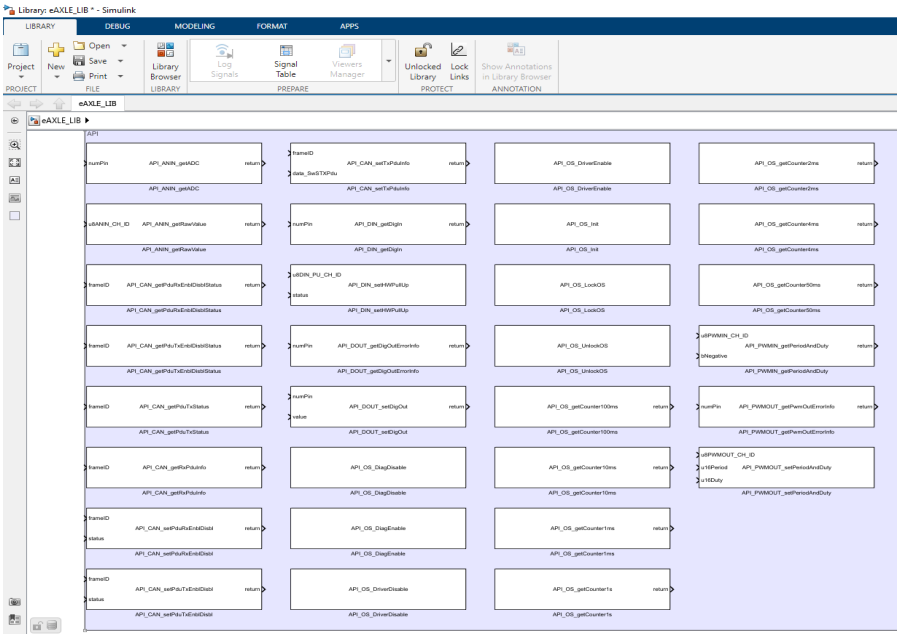


Figure 51 - "Update Blocks" result

This block library is very useful because it permits to configure the work environment quickly and easily. Its strong point is the scalability because large numbers of enumerative data types or different functions are not a problem since all the work is automatically done.

6.3.2 “Import DBC” block library

When it is necessary to receive or transmit messages via CAN, the user should implement in Simulink the composition of signals for every message that must be used. This process can take a lot of time and the risk of making mistakes is present since many things must be implemented such as encapsulation of payload, data conversions, signal resolution and units.
To have a better management of CAN messages and consequently use them in the model development, the integration of DBC files in MATLAB is fundamental.
For this scope, it has been created a specific block. It gives the possibility to the user to add or delete a DBC file in the project and integrates in MATLAB all its messages and signals. Furthermore, is possible to automatically generate all callbacks (specified in chapter 4.1) that are related to DBC files in the API file (and programmatically populate the file during the DBC import procedure).

To manage messages and signals, two different classes have been defined. The ‘Message’ class, used for managing all messages has the following properties:

- Name (name of the CAN message)
- ID (represent the address of the message expressed in hexadecimal form)
- NetworkType (indicates if it belongs to CAN1, CAN2, CAN3 or CAN4 network)
- UniqueID (flag that indicates if the message address is unique or not)
- DLC (Data Length Code of the message, the length of the payload in byte)
- Period (period in [ms] which a message must be received/transmitted)
- Timeout (time limit within which wait a message)
- Enable (flag that indicates if the message can be transmitted/received)

- Type (Extended or Standard message)
- MboxId (the frame ID associated to the message)
- Role (if is a transmitted or received message)
- Signals (list of all its signals)

The 'SignalOfMsg' class used for all message signals is composed by:

- Name (name of the signal)
- BitStart (start bit of the signal in the message payload)
- Length (bit length of the signal)
- ByteOrder (if expressed in Little Endian (Intel) or Big Endian (Motorola))
- ValueType (if the signal is signed or unsigned)
- Factor (precision factor of the signal)
- Offset (possible offset of the signal)
- Min (min value it can take)
- Max (max value it can take)
- Unit (unit of measurement)
- Comment (signal comment string from DBC file)

For what concerns the mapping between MboxId available (frame ID) and messages in DBC files it has been created a class called 'canMap' that manages this relationship. This class contains the list of all MboxId available, a flag that indicates if each frame ID exist in the API file (so if it has been managed at low level), the name of the possible message associated (empty if no message is associated yet) and the number of total messages associated. For a clearer management of network and messages, it is created an object of this class for each CAN network and for each 'Role'. For example, the CAN2 network will have a 'canMap' object for messages in reception (called CAN2RX) and another for those in transmission (called CAN2TX).

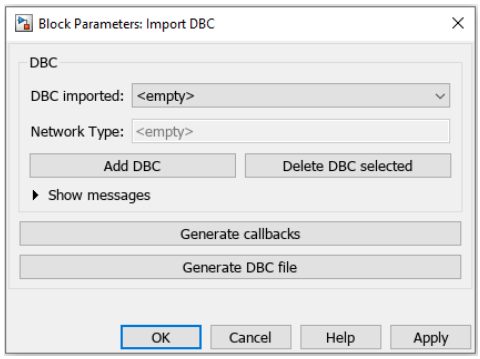


Figure 52 - "Import DBC" block library

After opening the block library, the first action to do is to add a new DBC file in MATLAB pressing the 'Add DBC' button (*figure 52*). At that moment, an open file dialog appears to choose the DBC file from the file system and then the tool will ask the user to select the CAN Network and the transmission node (*figure 53*).

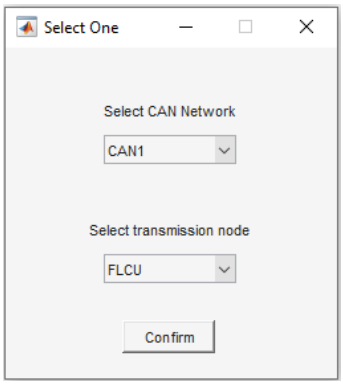


Figure 53 - Select CAN network and transmission node

This is done because based to transmission node, the tool can understand which messages are transmitted and which are received (since in the DBC file each message has among its properties the transmission node). Then it will update the correct canMap related to the selected CAN Network.
The script will analyze the DBC file and based to its syntax (*figure 16,* chapter 3.2), it will create message and signal objects saving them in a specific file called as the DBC with the suffix '.mat' (type of MATLAB file used to store data). After saving the objects, the 'Import DBC' block will show all the message properties of the selected one in the popup menu (in the example of the *figure 52* it is shown the 'FLCU_TO_PCM_003' message, including a popup that shows the signals associated to the message). The script is predisposed for changing all message properties and consequently update the MATLAB object but for the moment, the user can only modify the flag for enabling or disabling the message.
If the user adds more DBC files, they will integrate with the existing ones, so the block library will assign new MboxId from those not yet occupied (checking from the canMap) and it will check the address univocity for the network of belonging.
All DBC files imported are visible through the first popup called 'DBC imported'.

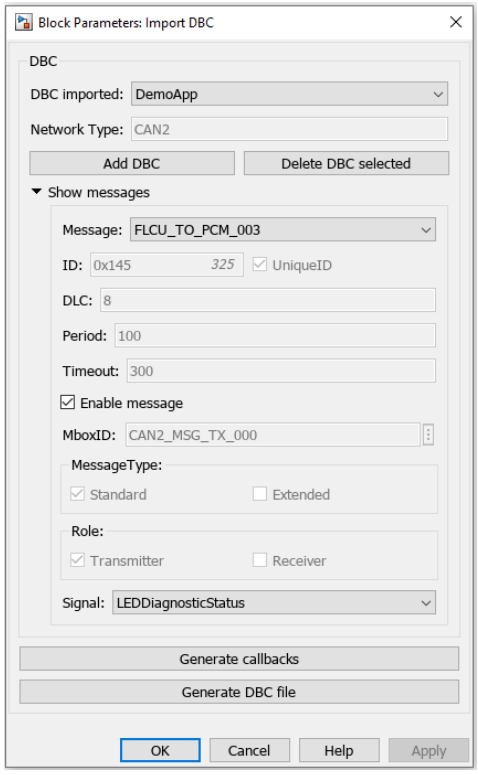


Figure 54 - Message properties

The next step after importing all DBC files into MATLAB, is to generate automatically all callbacks related to messages. Pressing the 'Generate callbacks' button, the script will write in the correctly code portion of the API file all the properties of the messages. As shown in the *figure 55,* in the API file are present some markers (line 3111 and 3143 highlighted of the example code) that have the purpose to indicate the starting and ending point of the callback in such a way the script knows where to write the code.
Every time the callbacks are generated, the script deletes the old ones and rewrite the new ones. In this way is easier to manage update of existing messages or add more.
The big advantage of this feature is to eliminate all possible human errors due to incorrect transcription of message callbacks.
The 'Generate DBC file' button is inactive and in the future can be used for regenerate the DBC file according to all modifies done by the user.

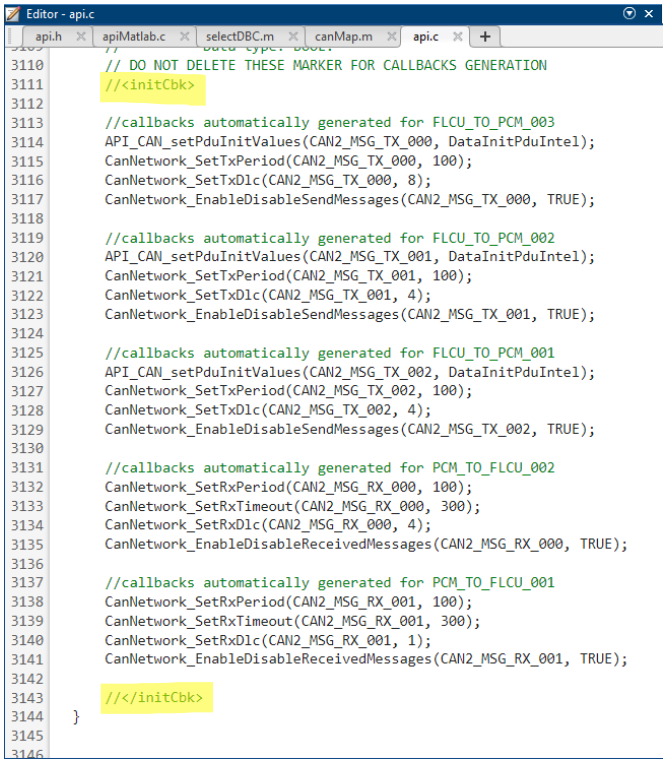


Figure 55 - Callbacks generated automatically

The last thing that the user can do with this block is to delete an imported DBC file using the 'Delete DBC selected' button. In this way, after having chosen the DBC file to delete through the first popup and pressed the button, the script will automatically update the callbacks to be aligned with the DBC files imported at that time.

6.3.3 "CANRX_MESSAGE" block library

After integrating DBC files, for using them is necessary to send and receive their messages. This block library (and its dual) concludes the management and integration of CAN communication in the MATLAB environment. It makes available to the user a simple way to use signals of received

messages in the model development and for doing this, it is integrated with all DBC files imported thanks to the previous block library explained.

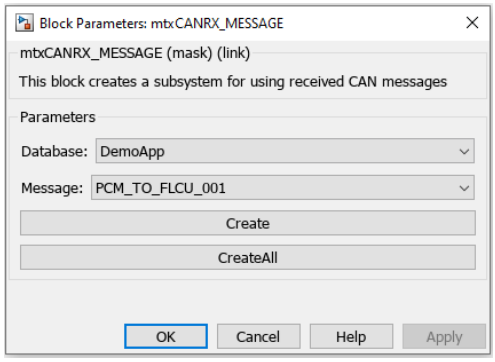


Figure 56 – CANRX_MESSAGE block

As it is shown in the *figure 56* the user has available all messages of DBC files imported in MATLAB and they are viewable through the two popups. After selecting the message and pressing the 'Create' button, the tool will create a subsystem with all message signals as output and the user will be able to use them directly in the model. The created subsystem also has two diagnostic outputs called with the name of the message plus '_Status' and '_BitStatus' and they are used to check the diagnostic status of the message.

The '_Status' signal describes through the enumerative data type 'tCANRxStatus' (page 27) the diagnostic status of the message but it can represent only one state of error. For this reason, error state is displayed following a priority order. The most important error is the 'Channel ID' error because in presence of this error the message is not received. In case there were other errors besides this one, they will not be displayed due to priority management. Continuing with the priorities order, there is the TimeOut error, it means that the time limit imposed to receive the message has expired, so that message will be lost. The next error is the DLC error, it means that the DLC of the message received is not equal to the expected one. After that, the NewDataReceived state indicates when a new message is received raising a bit to 1. For transforming it to an error it has been inserted a 'NOT' block in such a way the error status rises when the new message is not arrived (before the Timeout error). The last error in order of priority is the SignalOutOfRange error and it means that one of the message signals assumes an out-of-range value.

To investigate deeper and visualize all possible status error, it is used the '_BitStatus' signal. It is a bit word made by 8 bit, each of which represent a specific status error. To compose it, every status has been weighted by a multiple of 2 and summing all of them you get the bit word.
In the *figure 57* it can be seen the subsystem created after having pressed the 'Create' button of the library block and it is composed, as mentioned before, from two diagnostic signals and the message signals (in this case the message contains only the signal 'csPCM_TO_FLCU_001_EnableFLCU').

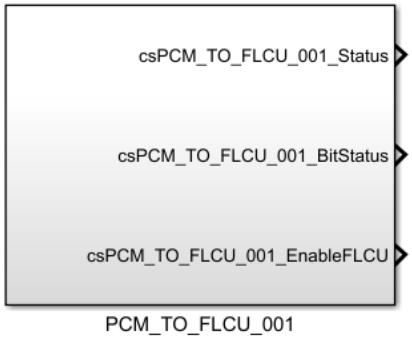


Figure 57 - CAN RX subsystem

If the user presses 'CreateAll' a window like the one in the *figure 58* will appear to ask the rate of messages that he wants to receive. Selecting the rate in [ms] the tool will create all subsystems as those in *figure 57*, one for each message.

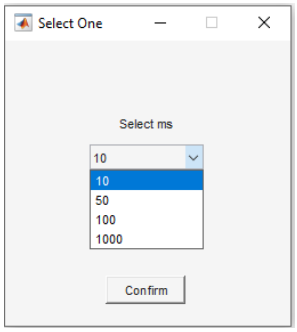


Figure 58 - CreateAll messages

The CANRX subsystem is in turn composed from other subsystems that permit its purpose, so it is created from a script using other blocks library, specially made for this scope.

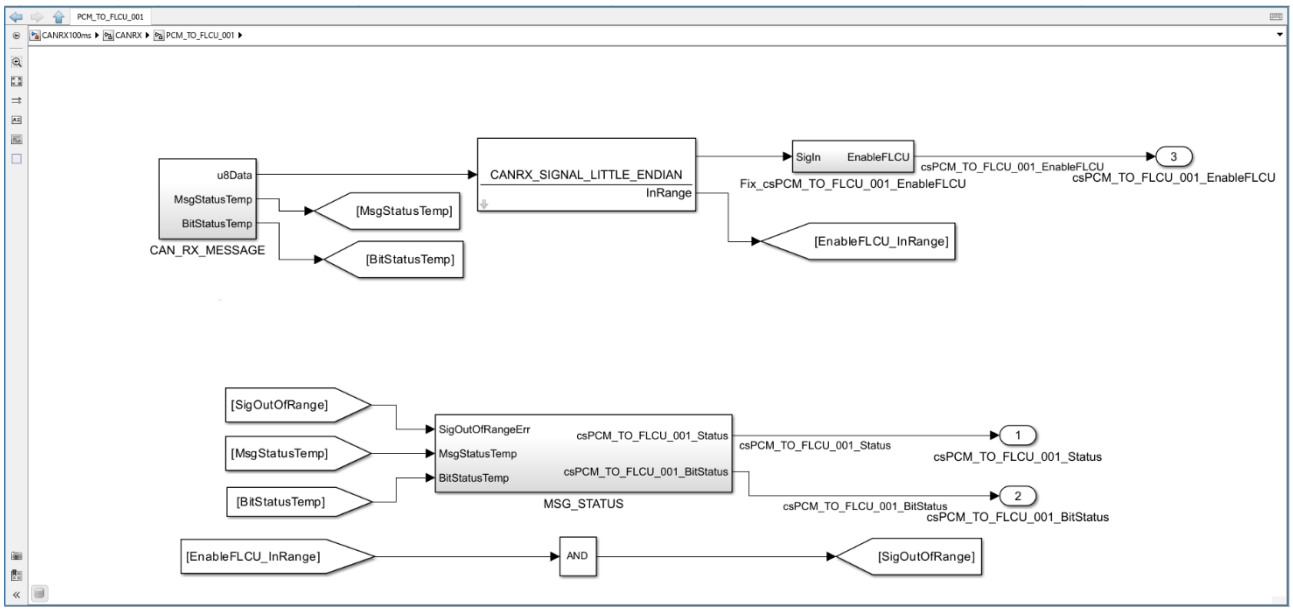


Figure 59 - CAN RX subsystem composition

As shown in the *figure 59* there are other blocks that do different task. The upper part is in charge of giving in output the message signals while the bottom part takes care about diagnostic status errors.

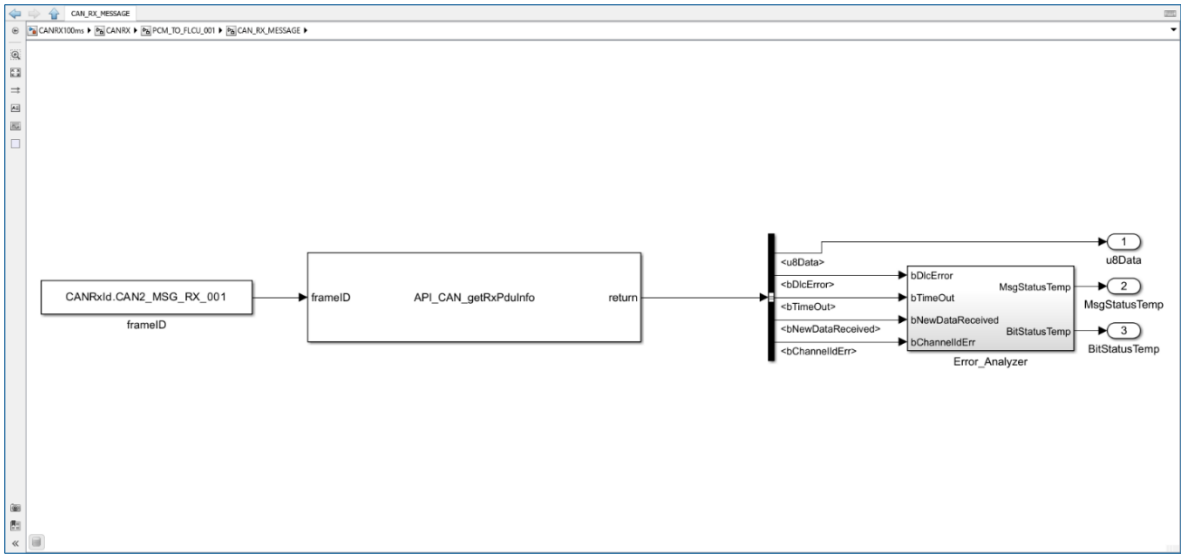


Figure 60 - CAN_RX_MESSAGE

The first subsystem in the upper left corner of *figure 59*, called 'CAN_RX_MESSAGE', is described in the *figure 60.* It takes the frameID associated to the selected message and call the '`API_CAN_getRxPduInfo`' function to receive the message (through the C Caller block created using the library, chapter 4.1 *figure 19*). The C Caller block gives as output the CAN frame (containing the payload and the error fields) and it is spun off using a bus selector.

The 'Error_Analyzer' subsystem (*figure 61*) takes in input all the errors of the message and, based to priority, gives in output the two error signals (they are denoted as 'Temp' because is still missing the SignalOutOfRange error that is managed in another part of the subsystem). As it is shown below (*figure 61*) the priority concept is modeled in MATLAB using a cascade of switch blocks (that can be interpreted as IF-THEN-ELSE) so if the first switch is verified (it passes the true condition) the others are ignored. Each switch corresponds to a possible error status described by a constant value, for example the 'ChannelIdErr' is represent by the number 4 because it is its enumerative (accordingly to the enum data dictionary).
For creating the bit word status, each error signal is multiplied by a power of two and they are summed.

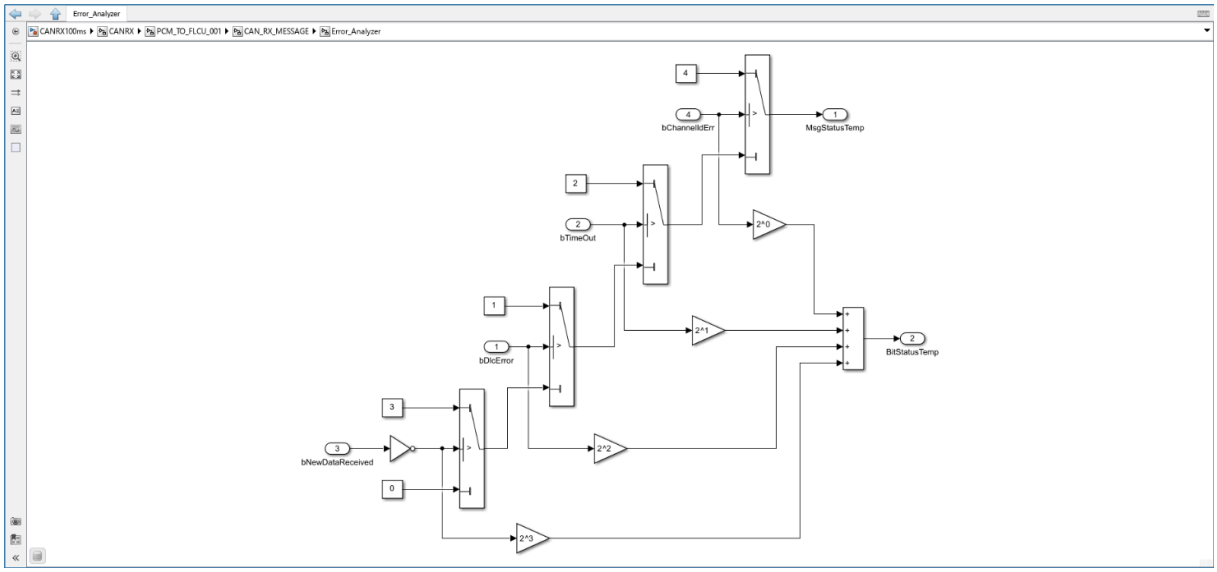


Figure 61 - Error Analyzer

To complete the two diagnostic signals is also necessary the 'SignalOutOfRange' error but it is managed separately because it is a property of the single message signal and not of the entire message. For doing this all 'SignalOutOfRange' errors of each message signal (that comes from the 'CANRX_SIGNAL_LITTLE_ENDIAN' block as will be shown later) are inserted in an AND block that gives as output the 'SignalOutOfRange' error of the message (in the example in *figure 59* there is only one input in the AND block because there is only one signal but in case of multiple signals the AND block would adapt with the same number of input). In this way, if just one signal is out of range (denoted by a 0), the signal 'SigOutOfRange' (output of AND block) will be zero (the NOT block raises the error flag) and the two diagnostic signals can be completed in the 'MSG_STATUS' subsystem (*figure 62*).

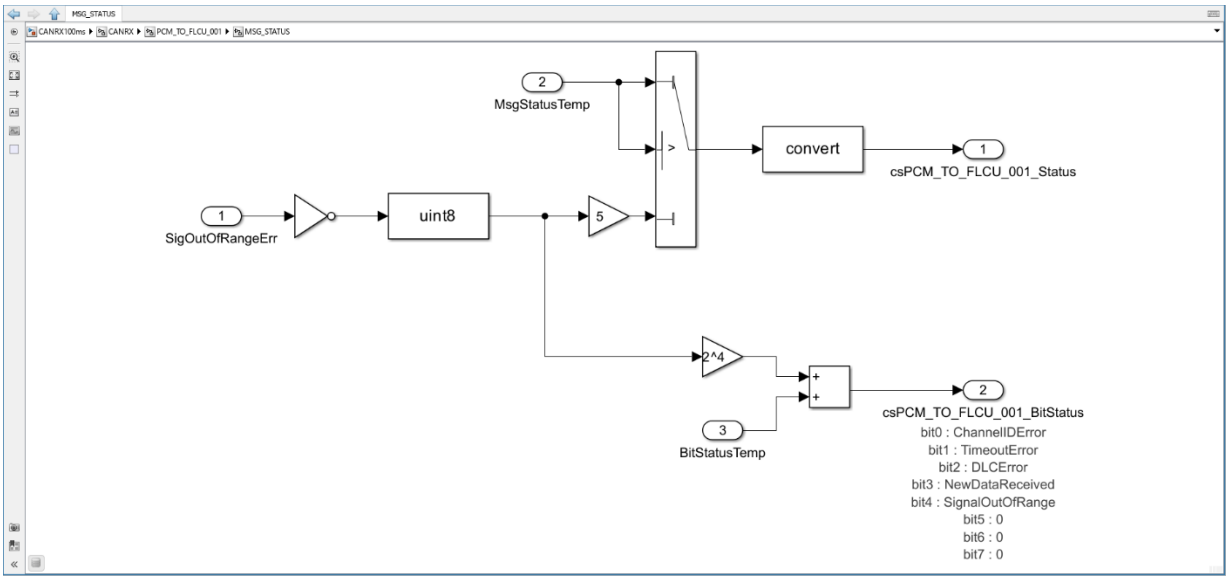


Figure 62 - MSG_STATUS subsystem

In the figure above in the lower right corner it also can be seen the composition of the bit word status where each bit means a specific diagnostic error.

For what concern the extraction of single signals from the payload data, the block 'CANRX_SIGNAL_LITTLE_ENDIAN' (*figure 59*) has this purpose. Through the information about the signal, this block is able to extract the correct bits from the entire payload and create the correct signal. In case of a message with more than one signal, it will be present a block of this type for each one.

Inside this block is also done the check of the 'SignalOutOfRange', in particular the signal is saturated to the minimum and maximum value before going out from the subsystem. Then it is compared the signal before and after the saturation block and, if they are different, it means that the signal value is out of range.

After this step, the user will have the opportunity through the 'Fix_' block (in the *figure 59* is called 'Fix_csPCM_TO_FLCU_001_EnableFLCU' because it assumes the name of the message signal) to enable a fixed value for that signal instead the real one.

As the *figure 63* shown, each signal has two calibrations. The first calibration, cfSIGNALNAME_EN, has the purpose to enable the fixed value represented by the second calibration, cfSIGNALNAME_VALUE. Through a switch block is checking if the enable calibration is equal to one, and in that case, the signal output will take the value of the value calibration. Otherwise, if it is zero the signal remains the original one. This type of approach is used very often during the test phases

because thanks to these calibrations the user can test some values inside the operating range of the signal.

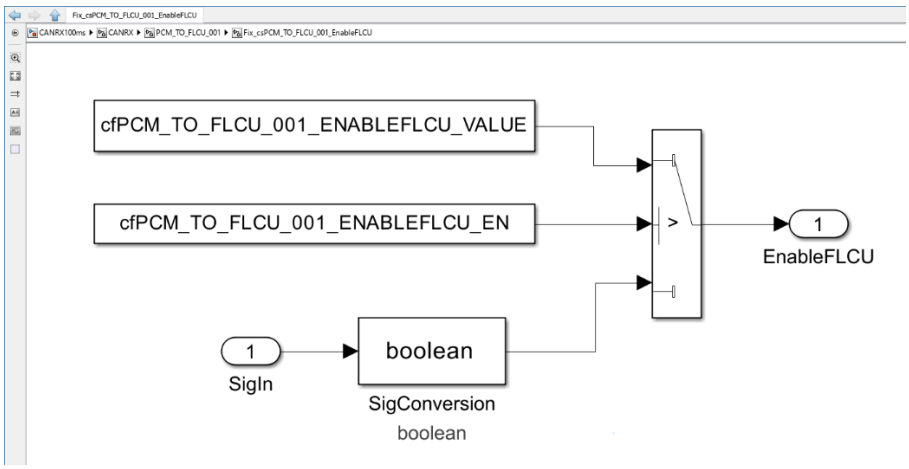


Figure 63 - Fix subsystem

6.3.4 "CANTX_MESSAGE" block library

From the dual side, this block has the purpose to send CAN messages in the network. The window that appears is the same as in the 'CANRX_MESSAGE' block (*figure 56*) but the messages proposed to the user are the transmitted messages instead received messages. The aspect of the created subsystem for each message is similar to the receive ones, in fact it has as inputs all message signals that are going to be send (*figure 64,* in this example the message FLCU_TO_PCM_001 is composed by two signals that are 'FuelLevel' and 'FuelLevelLow').

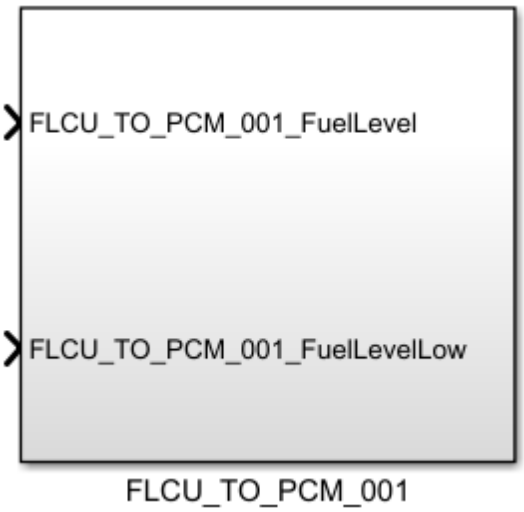


Figure 64 - CAN TX subsystem

Also in this case, the subsystem is composed by other subsystems that have different role. The 'CANTX_SIGNAL_LITTLE_ENDIAN' (*figure 65*) blocks have the purpose to compose the payload of the message. Based to the signal information such as start bit, number of bits, factor etc., each signal data is routed to a bitwise OR to create the entire payload data.

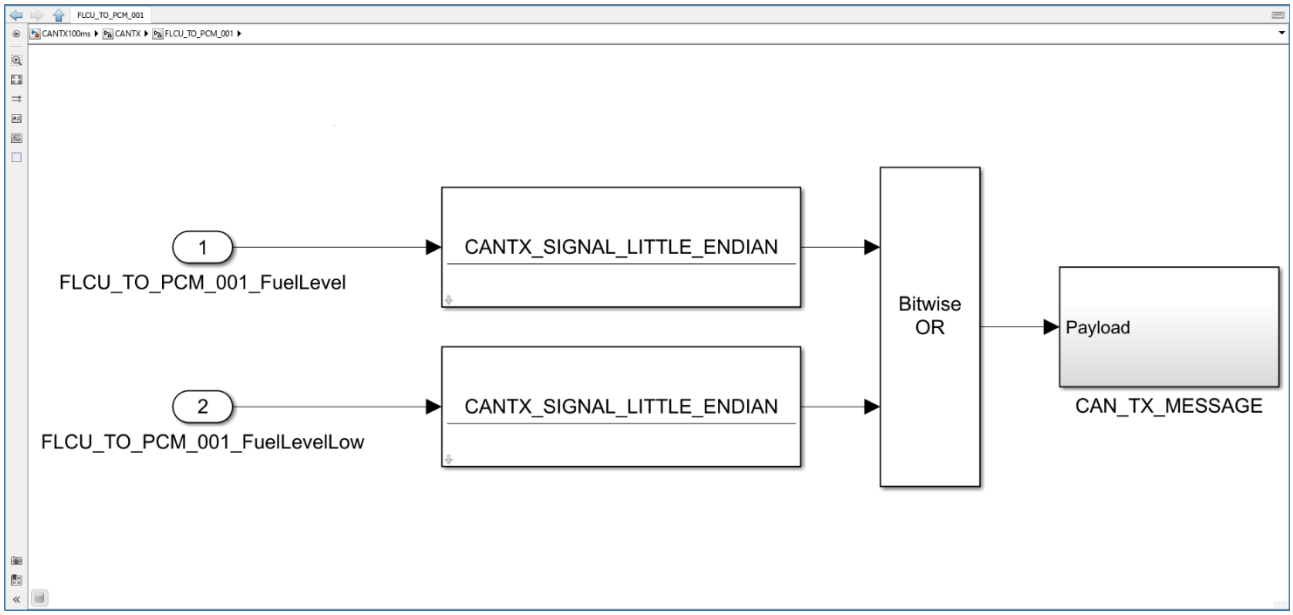


Figure 65 - CAN TX subsystem composition

Inside the 'CANTX_SIGNAL_LITTLE_ENDIAN' block (*figure 66*) is present the 'Fix_' subsystem, since also in this case is possible to send specific values using the calibrations (they are called with the same syntax in received messages). The block is inserted before the creation of the signal data in order to put the right values in the payload, original signal or fixed signal.

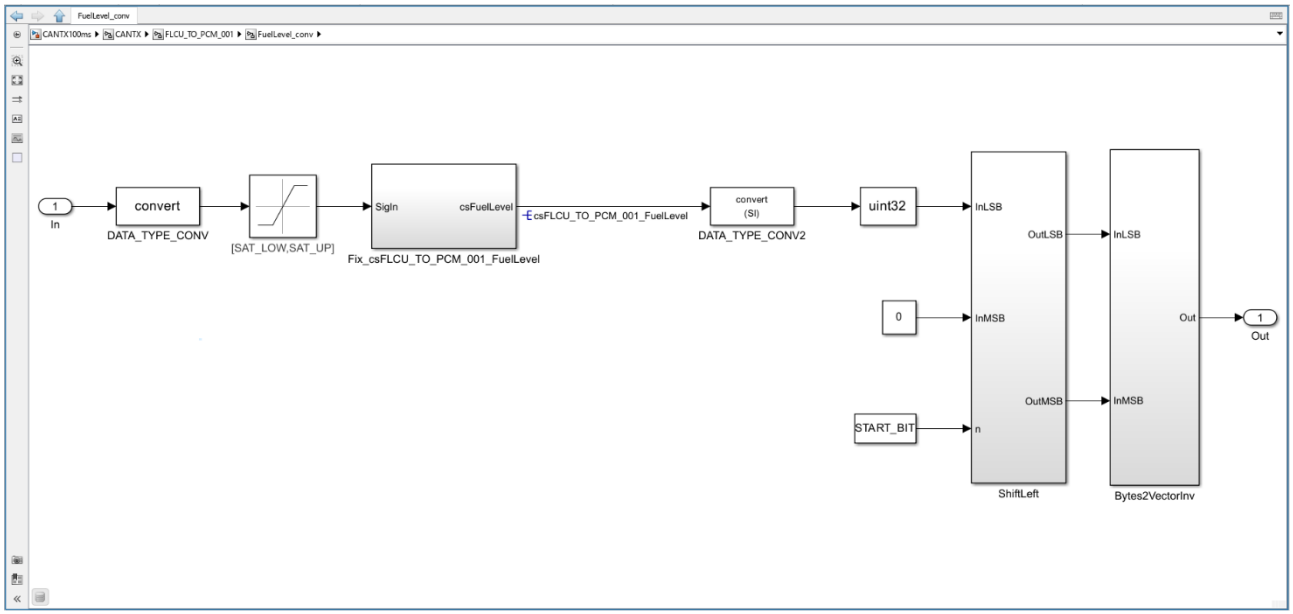


Figure 66 - CANTX_SIGNAL_LITTLE_ENDIAN

The Bitwise OR block in the *figure 65*, that is adapted based to the number of signals, sends to the 'CAN_TX_MESSAGE' block the payload data. This subsystem (*figure 67*) is in charge of sending the CAN frame to the associated frameID (MboxId) and, as shown in the figure below, it is done automatically using the message information. The tool uses the '`API_CAN_setTxPduInfo`' block creating, through a bus creator, the CAN frame composed by the DLC and the payload.

It also makes available to the user a calibration, called ‘csMessageName_En’ that enable or disable the state of message transmission. The calibration is converted in ‘uint8’ according to the function parameter.

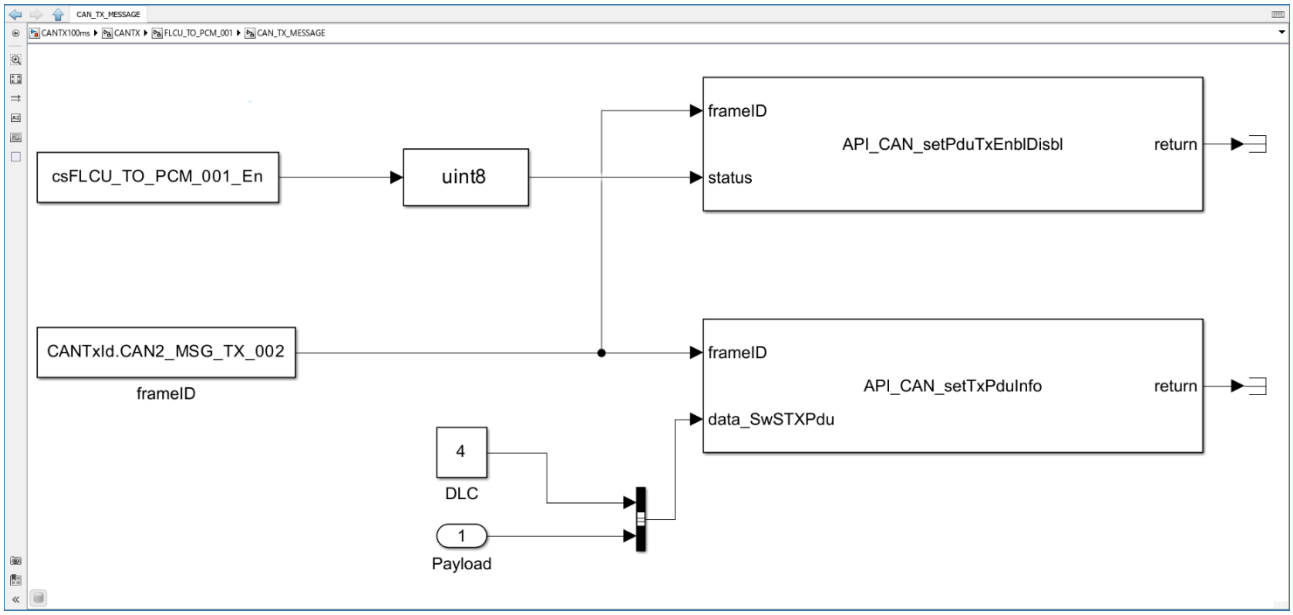


Figure 67 - CAN_TX_MESSAGE subsystem

6.3.5 “GenCode” block

When the model development is finished, MATLAB takes care of generating the related code. To properly handle signals between different modules, certain operations need to be carried out. This block has the scope to do these actions, in addition to the code generation, that are necessary for the correct workflow of the project and that were done by the user, having the possibility to make mistakes or forgetfulness.

For what concern Lookup tables, they are used creating a ‘Map’ object in the data dictionary of the model and specifying its table and breakpoint vectors. This block has been created to have a clearer representation of the LUTs in the code generated, in fact it updates all the struct type definition names before the code generation in such a way each Lookup table will have the struct type name equal to the LUT object name plus ‘_str’ at the end (to indicate that it is a struct). Furthermore, it assigns the name of each LUT object to the related LUT block in the model.

```
#ifndef DEFINED_TYPEDEF_FOR_zvRailPConv_str_
#define DEFINED_TYPEDEF_FOR_zvRailPConv_str_

typedef struct {
  uint32_T N1;
  real32_T BP1[4];
  real32_T Table[4];
} zvRailPConv_str;

#endif
```

Figure 68 - Typedef struct definition

```
/* Definition for custom storage class: Map */
#pragma section ".cal_ram"

zvRailPConv_str zvRailPConv = {
  4U,

  { 0.5F, 2.0F, 2.5F, 4.5F },

  { 0.0F, 25.0F, 30.0F, 50.0F }
} ;                                  /* Referenced by: '<S2>/zvRailPConv1'

#pragma section
```

Figure 68 – LUT object definition

As can be seen from the figures above, the LUT called ‘zvRailPConv’ is declared as a Map in a pragma section (*figure 69*), so due to the fact that is stored in the CAL RAM, it is calibratable in the test phases. Furthermore, its data type has been set with the same name of the LUT (*figure 68*). In this way the code generated is clearer and more understandable.

As mentioned before, this block takes care about the management of signals. Since complex systems are divided in several models, signals are often shared between them. In particular a signal produced by a certain model can be used from another model (for this reason is used a main root data dictionary that includes all data dictionaries). The tool created does operations through a script before and after the code generation following the flow below.
Before to proceed with the code generation the script does the following tasks:

- Check if Input signals are present in the data dictionary of the project and set ‘ImportFromFile’ as Storage Class. If the header file from where to import the signal is not present it gives error because is not possible use the signal.
- Set the parameters Min, Max, Data type and Unit of the inport blocks equal to the signal associated. It is necessary for the report created at the end of the model.
- Check if Output signals are present in the data dictionary and set the storage class as ‘ExportedGlobal’ in such a way they are declared as *‘extern’* in the code generated.
- Set the parameters Min, Max, Data type and Unit of the outport blocks equal to the signal associated.

Then the code generation started:

- Update and generate code of the model.

- Save the most important generated files in a directory created ad hoc for the current model ('.c', '.h', '_private.h', '_types.h', '.a2l').
- Save shared files in a directory common to all project models.

After the code generation phase the script:

- Sets for each Output signal 'ImportFromFile' as storage class and assign the name of the current model (adding '.h' at the end) as header file in such a way the next model that uses that signal will have already set up the correct header file.
- Saves all changes of the data dictionary father.

The script that does all these actions is inserted inside a block library in a dedicated button called 'Generate code' (*figure 70*). The following block is meant to be used at the end of the design of each model.

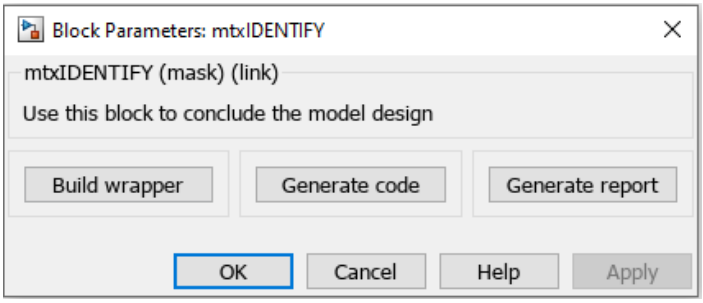


Figure 70 - "GenCode" block library

When the user has finished the model design it should wrap the model in a subsystem obtaining a new block with inputs and outputs. The first button in the block called 'Build wrapper' has the scope to join all inputs in a single block, link the inports to the related signal and lock/unlock the OS to avoid conflicting with other OS task during the acquisition of that signals.

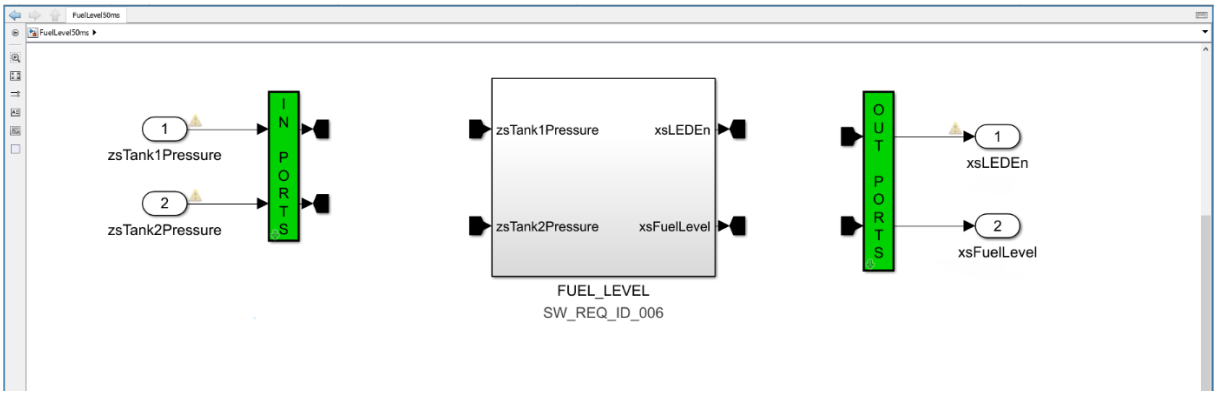


Figure 71 - Build wrapper result

As is shown in the *figure 71*, all the input ports enter in the green block and with 'Goto' and 'From' blocks the signals are sent in the subsystem of the model. Inside the 'IN PORTS' green block it can be seen (*figure 72*) from the blue trident shaped symbol that the inport is connected to the signal object denominated in the same way. Furthermore, are present the C Caller blocks '`API_OS_LockOS`' and '`API_OS_UnlockOS`' to do the actions mentioned before. To ensure the correct order of execution, that is Lock the OS, assign the signal object to the inport and then Unlock the OS, block priorities were changed through their properties. The '`API_OS_LockOS`' has 1 as priority, the square block interposed between inports and outports (it has no function, only to pass

the signal from one side to the other) has priority 2 and finally the ‘API_OS_UnlockOS’ has priority 3.

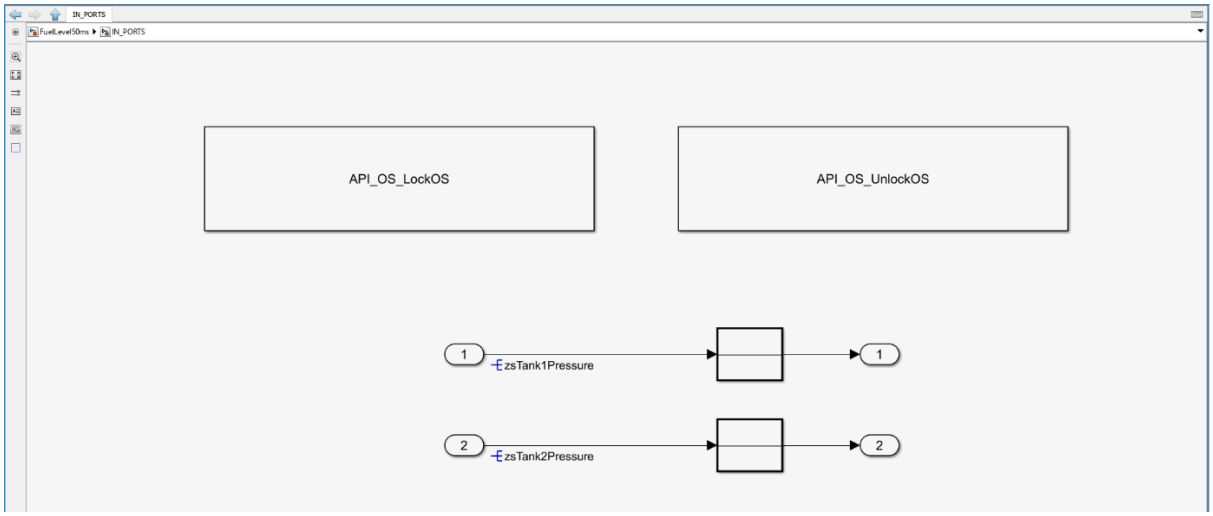


Figure 72 - IN PORTS block

The result of the priority orders in the generated code is the following:

```
50  API_OS_LockOS();
51
52  /* Outputs for Atomic SubSystem: '<S2>/BYPASS1' */
53  /* SignalConversion generated from: '<S6>/In' incorporates:
54   *  Inport: '<Root>/zsTank1Pressure'
55   */
56  rtb_In_o = zsTank1Pressure;
57
58  /* End of Outputs for SubSystem: '<S2>/BYPASS1' */
59
60  /* Outputs for Atomic SubSystem: '<S2>/BYPASS2' */
61  /* SignalConversion generated from: '<S7>/In' incorporates:
62   *  Inport: '<Root>/zsTank2Pressure'
63   */
64  rtb_In = zsTank2Pressure;
65
66  /* End of Outputs for SubSystem: '<S2>/BYPASS2' */
67
68  /* CCaller: '<S2>/API_OS_UnlockOS' */
69  API_OS_UnlockOS();
```

Figure 73 - Signals assignment avoiding conflicts

The last button ‘Generate report’ has the scope to generate a specific report for the current model to describe it. This button was already present and was created by previous colleagues.

6.4 Blocks validation

After having created the new library, all blocks must be validated to be ensure that they work as expected.
To validate I/O blocks is necessary to interface with the pins on the board and know the mapping with the related channel IDs.
The Break-Out Box (BOB) (*figure 74*) is used in these situations for having available all physical pins of the board and doing test. It replicates all pins and each of them has a specific ID number (e.g. B50).
The test equipment changes according to the project and to the development phase of the product. For the rapid prototyping it is possible to connect some electronic components such as resistors or LEDs and measure voltages through tester or oscilloscope for testing the model functionalities directly in the hardware.
Dedicated HIL benches with real or emulated loads can be used in more advanced phases of the project.

The mapping between pin numbers and MATLAB channel ID is made at a lower layer and a file that describe it is necessary for connecting electronic devices to the correct pin number. For example, the pin B53 is mapped to the channel ID 'DOUT_LS_CH_ID_000' so in the model phase if the user set a Digital Output LS to a certain value, he should check the result through that pin of the Break-Out Box.

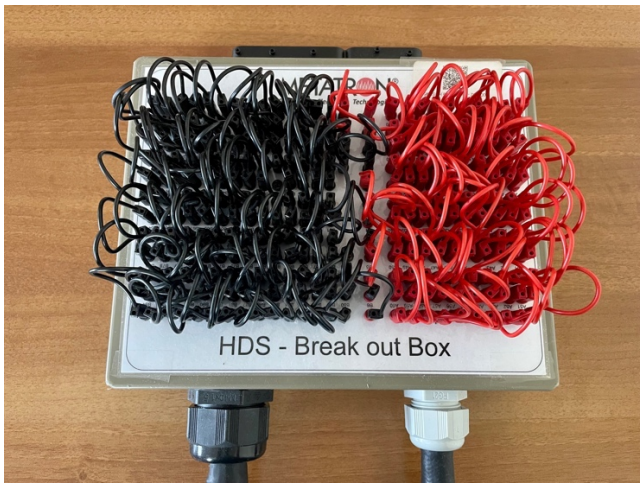


Figure 74 – HDS9 Break Out Box

With the BOB is possible to test and validate blocks that use Input and Output channels such as the example in the *figure 20* which is present a Digital Output set to one. Connecting a LED, in this case it occurs that effectively when the channel is set to one, it turns on.

Figure 75 - LED on

Regarding communication blocks, they have been validated through a real use case for a test implementation of the third CAN channel as possible intravehicular channel communication. To do this, it has been created a DBC file containing the largest possible number of messages and every possible task frequency. The goal is to check if all these messages are sent and transmitted correctly using CAN 3.

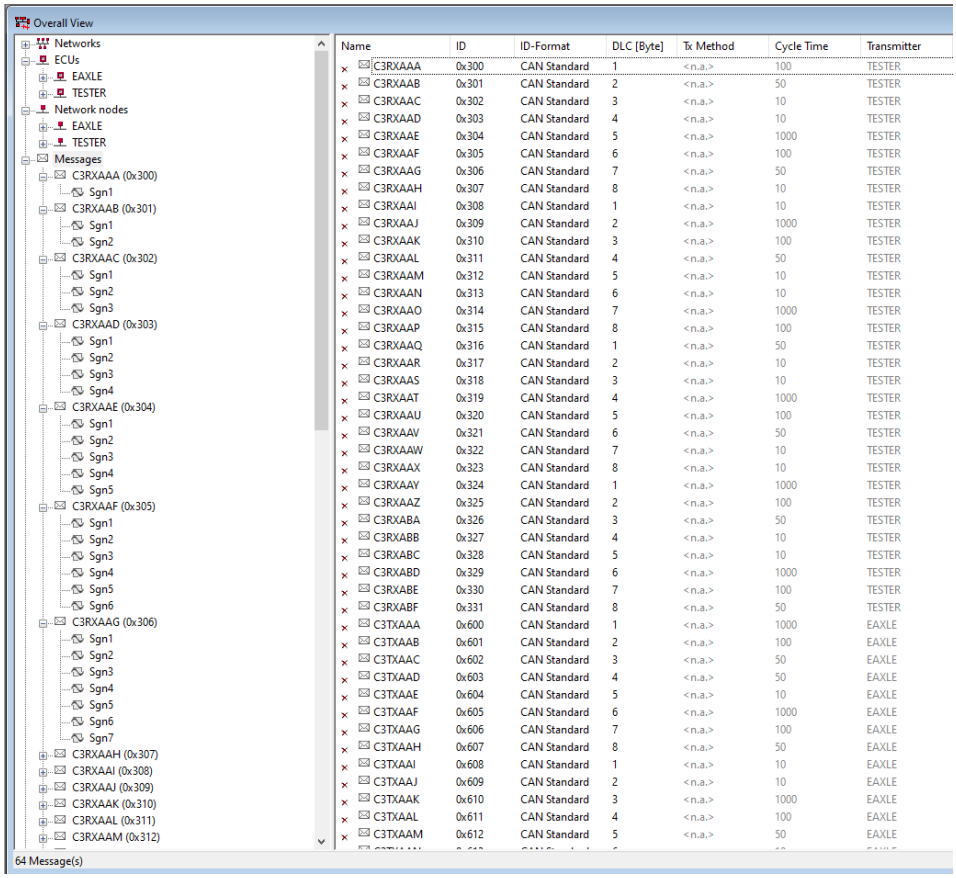


Figure 76 - DBC file

As shown in the *figure 76,* each message has from one to eight byte as DLC and, for simplicity, one signal for each byte. The ID starts from 0x300 (since all messages are standard and not extended) and increases by one for each message, the cycle time to send or receive the message is 10, 50, 100

or 1000 ms. The network has two nodes where 'EAXLE' is the transmitter (the ECU) and 'TESTER' is the receiver (the PC). All message names are fictitious and start with C3 (stands for CAN 3), then are followed by RX or TX to indicate their role in the network and at the bottom there is an incremental string that start from 'AAA'.

After this first step, the DBC file has been imported in MATLAB through the "Import DBC" block library and the result is visible in the *figure 77*. In this case is shown the 'C3RXAAA' message and its properties. By default, the timeout is set as three times the period.
The tool assigns a different MboxId for each message and checks if the address is unique. In case it is not unique the flag 'UniqueID' would be disable.

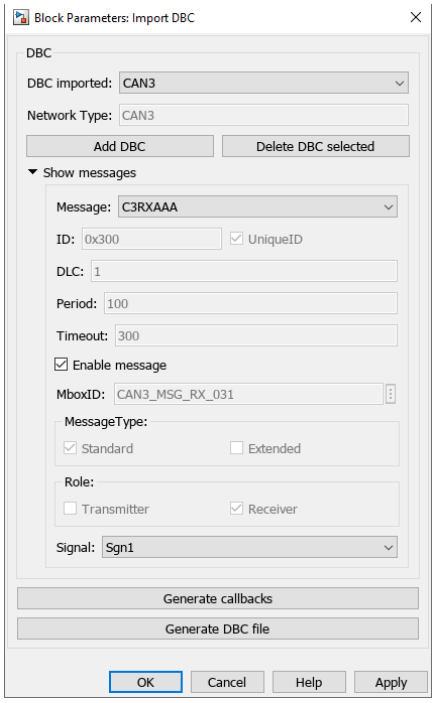


Figure 77 - DBC imported in MATLAB

For what concern the design of this validation application, there has been created one model for each period and for each role. This is done because each model will run in different OS task based on its frequency. Each model also has a data dictionary where all signals are saved.

Models for received messages have the only scope to receive messages of a certain period and visualize, through CANape, their signals. From the dual side, models for transmitted messages have to send some fictitious signals.
Using the 'CANRX_MESSAGE' and 'CANTX_MESSAGE' blocks library, all subsystems of messages are created automatically in the model in a few seconds. In this use case, it can be appreciated the time saving that these blocks have provided since before, the creation and integration of the DBC file in MATLAB could take three weeks.

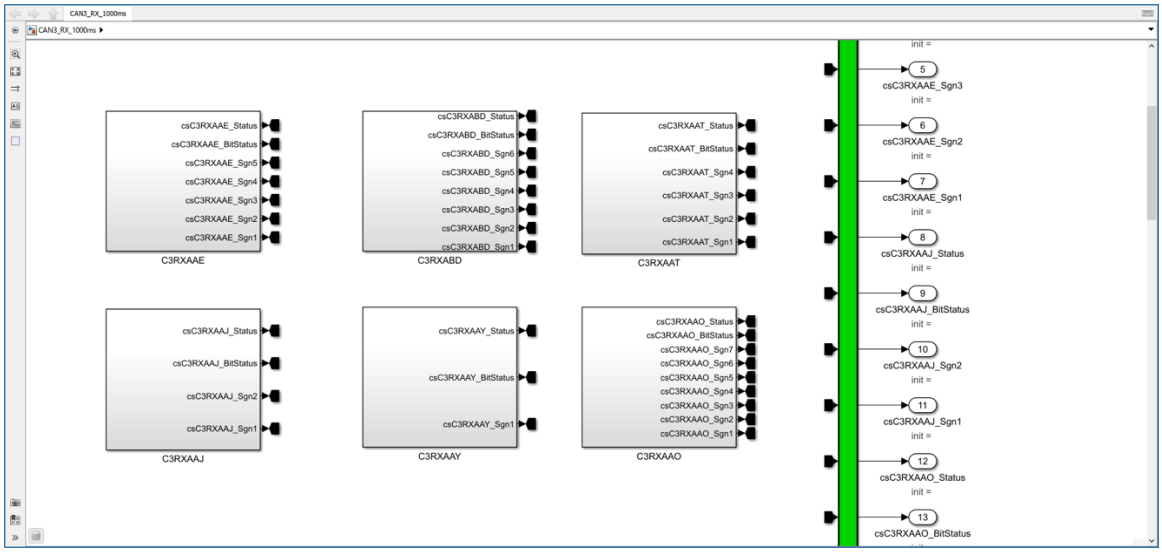


Figure 78 - CAN3_RX_1000ms

The *figure 78* shows the model created for received messages with period 1000ms. All the outports connected to the green block are created automatically using the 'Build wrapper' button of the block library and, consequently, all signals are connected to each related outport.

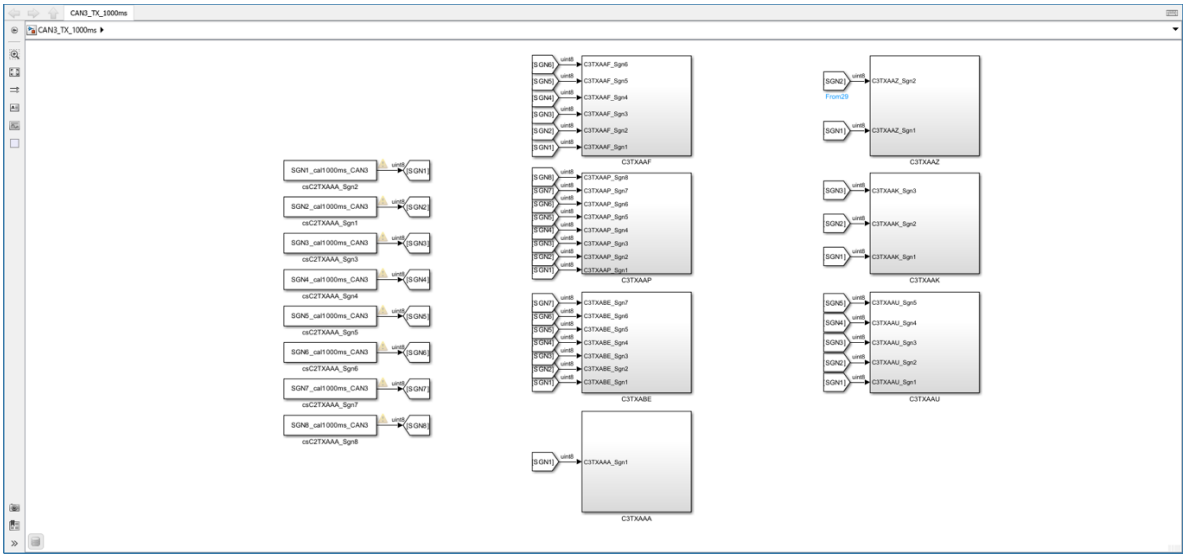


Figure 79 - CAN3_TX_1000ms

On the other hand, the *Figure 79* shows the model for transmitted messages with period 1000ms. To send signals, some calibrations have been created and saved in the data dictionary appropriate. In this way, in the next validation step with CANape, is possible to modify the transmitted signals and verify that the communication channel works properly.
After having done all models for all types of messages, it is possible to automatically generate the code and, through the 'Import DBC' block, generate all the callbacks related to the DBC messages.
The final step is to flash the software in the ECU following the flow explained in the chapter 5.2 and though CANape visualize all signals messages.
For the thesis, the 'TESTER' node in the network is done by the PC using a Peak dongle (PCAN-USB, visible in the *figure 80*) connected to the CAN 3. Through a tool called 'PCAN-view', a network sniffer, is possible to visualize network messages as well as send them. After having opened the tool is necessary to set the type of messages that are Standard or Extended. In this case it is set to Standard since all messages created in the DBC file have the ID of that type.

Figure 80 - Peak dongle

To validate the functionality of the program developed for this test, it has been tested both received and transmitted messages. The *figure 81* shows a simple check to ensure the functionality of CAN 3. On the left side of the screen is present the PCAN-View tool that as said before plays the 'TESTER' role. Consequently, the received messages in PCAN-View are those transmitted by EAXLE, vice versa the transmitted messages in PCAN-View are received by EAXLE.
On the right side of the screen is shown Vector CANape tool that represents EAXLE messages.

For what concern transmitted messages (EAXLE -> TESTER), it can be seen from PCAN-View that are received all messages from ID 0x600 to 0x631. In addition to that it has been done another type of test, changing the value of the signal 'SGN1_cal100ms_CAN3' from 0x01 to 0x08 (since it is a calibration it is modifiable from CANape), is received the updated message value in fact, the blue circle shows that. This signal is used from all messages that have cycle time 100ms and it represents the first byte of the payload.
In addition to the message with ID 0x616, it is correctly updated in messages with ID 0x601, 0x606, 0x611, 0x621, 0x626, 0x631.

On the other hand (TESTER -> EAXLE), for testing received messages it has been created a new message to transmit from the TESTER. To correctly create the message all setups must be done according to the DBC file, so in this case since is sent the message with ID 0x300, the DLC is set to 1 and the cycle time is set to 100ms. The payload data sent is 0x05 and from CANape is visible the correct payload circled in red. In this case the message received is the 'csC3RXAAA_Sgn1' and its status is CAN_RX_OK since the reception has no problem.

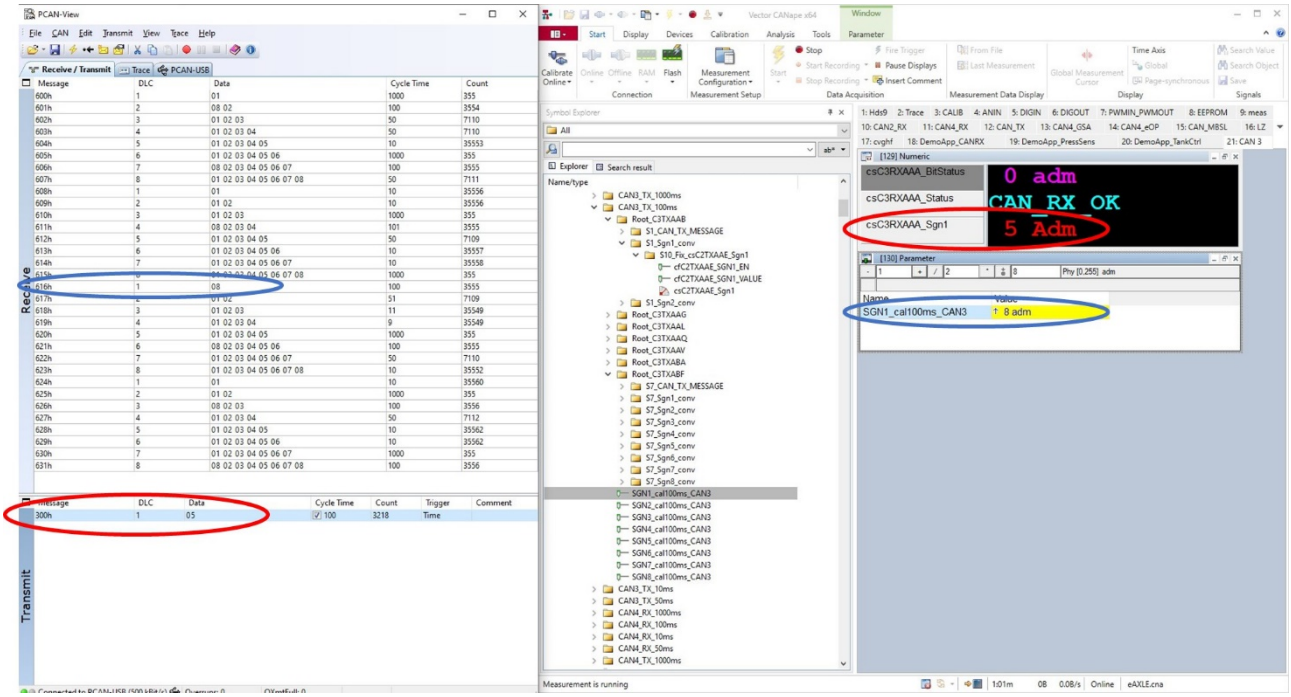


Figure 81 - CAN 3 functionality

7. Demo Application- Rail pressure regulator and Fuel tank management

To have a complete validation of the created library, a demo application has been developed. The scope is to follow the entire workflow from requirements to SW integration in order to emulate (in a simplified way) the creation of a real work project: a Fuel Line Control Unit.

The Fuel Line Control Unit (FLCU) is an important component of the fuel delivery system in internal combustion engines. Its primary function is to regulate and monitor the fuel flow, ensuring optimal performance and efficiency.

7.1 System requirements

The FLCU system must meet specific requirements to ensure proper fuel management. They describe the purpose of each component of the system and how they are interconnected with the other components.
The following subchapters represent the key requirements for the FLCU.

7.1.1 Fuel Rail Pressure Control

The Fuel Rail Pressure control is one of the most important requirements for the development of the FLCU as it directly impacts engine performance and fuel efficiency. By regulating the pressure of the Fuel Rail in a correct manner, many advantages can be achieved such as optimized combustion, precise fuel delivery and reduced emissions.
To do this, a proportional-integral-derivative (PID) controller can be employed in order to continuously monitors the actual fuel rail pressure, obtained from pressure sensors of the system, and compares it to the target pressure provided by the Powertrain Control Module (PCM).
The PCM is an external module that is in charge of managing the desired engine torque and communicates with the FLCU via CAN.
Based on the comparison between the actual and the desired rail pressure, the PID controller calculates the appropriate adjustment signal to control the Pressure Regulator Valve (PRV).
The control mechanism of the PRV is based on the Pulse Width Modulation (PWM) technique. Modifying the duty cycle of a fixed-frequency square wave signal, the PWM signal controls the actuator connected to the PRV. In particular, if the duty cycle increases, the PRV opens more allowing more fuel to enter in the fuel rail and increasing its pressure. On the other hand, decreasing the duty cycle restricts the fuel flow, reducing the pressure in the fuel rail.
The PID controller permits to the FLCU to correctly regulate the fuel rail pressure remaining within the desired range specified by the PCM.

7.1.2 Fuel Level Monitoring

The fuel level monitoring requirement in the FLCU is necessary for maintaining a correct fuel supply and preventing fuel depletion.
For this reason, the FLCU interfaces with fuel pressure sensors installed in the fuel tanks also called On Tank Valve (OTV). These sensors provide continuous measurements of the fuel tank pressure,

allowing the FLCU to detect a low fuel level condition when the pressure in both tanks falls below a tuneable threshold. After detecting a low fuel level condition, the FLCU should activate a visual warning to alert the driver (fuel reserve LED). Additionally, it should transmit this information to the PCM, which can further optimize engine operation based on the actual fuel level.

7.1.3 Tank Valve Management

Tank valve management is required for maintaining balanced fuel flow and pressure between the fuel tanks. The FLCU employs a dedicated control system for managing the opening and closing of the tank valves. This control system continuously monitors the pressure difference between the tanks using an additional pressure sensor at the tank manifold (after both the OTVs).
For implementing the tank valve management are used solenoid valves as actuators that are controlled by the FLCU to maintain the pressure difference between the tanks below a configurable threshold.
By implementing precise tank valve management, the FLCU ensures optimal fuel distribution, trying to keep consistent fuel pressure throughout the system. This in turn, promotes efficient engine operation and minimizes the risk of fuel starvation.

7.1.4 Communication with PCM

Establishing effective communication between the FLCU and the PCM is necessary for coordinated operations and it is done via CAN.

The communication enables seamless exchange of information from the PCM to the FLCU related to:

- The target fuel rail pressure to ensure the desired engine torque
- The system enabling conditions (binary condition to activate the FLCU control)

Vice versa the FLCU provides to the PCM information about:

- The diagnostic status of the sensors
- The diagnostic status of the actuators (valves and LED)
- The actual rail pressure
- The fuel level information

The CAN database that describes the communication is the following.
Received messages:

- PCM_TO_FLCU_001 (ID: 0x110)
  EnableFLCU (Signal): Indicates the system enable/disable signal
- PCM_TO_FLCU_002 (ID: 0x220)
  TargetRailPressure (Signal): Represents the desired fuel rail pressure

Transmitted messages:

- FLCU_TO_PCM_001 (ID: 0x167)
  FuelLevelLow (Signal): Indicates a low fuel level condition
  FuelLevel (Signal): Indicates the actual fuel level
- FLCU_TO_PCM_002 (ID: 0x123)

RailPressure (Signal): Indicates the actual rail pressure

- FLCU_TO_PCM_003 (ID: 0x145)
  Tank1PSDiagStatus (Signal): Diagnostic status of Tank1 Pressure Sensor
  Tank2PSDiagStatus (Signal): Diagnostic status of Tank2 Pressure Sensor
  ManifoldPSDiagStatus (Signal): Diagnostic status of Manifold Pressure Sensor
  RailPSDiagnosticStatus (Signal): Diagnostic status of Rail Pressure Sensor
  OTV1DiagStatus (Signal): Diagnostic status of OTV1
  OTV2DiagStatus (Signal): Diagnostic status of OTV2
  PRVDiagStatus (Signal): Diagnostic status of PRV
  LedDiagStatus (Signal): Diagnostic status of LED

7.1.5 Sensor/Actuator Management

Effective management of the pressure sensors within the Fuel Line Control Unit (FLCU) is essential for accurate monitoring and control of the fuel system. The sensor management requirement includes the following aspects:

- Data Acquisition: The FLCU must acquire pressure readings from multiple sensors, including those located before each OTV, before and after the PRV (tank manifold pressure sensor and rail pressure sensor). The FLCU should establish reliable and efficient data acquisition mechanisms to capture sensor data accurately and in a timely manner.
- Error Handling: The FLCU should implement robust error handling mechanisms to detect and handle sensor failures or abnormal readings. This includes monitoring sensor output for inconsistencies, identifying sensor malfunctions, and generating appropriate error codes. The FLCU should also have the capability to switch to backup sensors if primary sensors fail, ensuring continuous monitoring and control of the fuel system.
- Diagnostic Capabilities: The FLCU should provide diagnostic functionality to identify potential sensor faults or actuator anomalies. This result in running diagnostic algorithms to analyze sensor data and detect discrepancies, generating diagnostic signals in case of problems.

With a correct managing of the pressure sensors, the system ensures accurate and reliable measurement of fuel pressures at different points in the fuel delivery system and additionally has an efficient fault diagnosis procedure. In this way the reliability of the entire system is enhanced.

7.2 System Architectural Design

For a better understanding of the system, it is necessary to have a system layout that represent each component of the system and how is connected with the other elements.
As it can be seen from the *figure 82,* the FLCU exchange data with other components of the system that are described in detail afterwards. The figure shows that it receives data from sensors (represented by yellow rectangles) and exchange messages via CAN with the PCM module. Green lines represent actuator controls while blue lines represent the fuel flow.

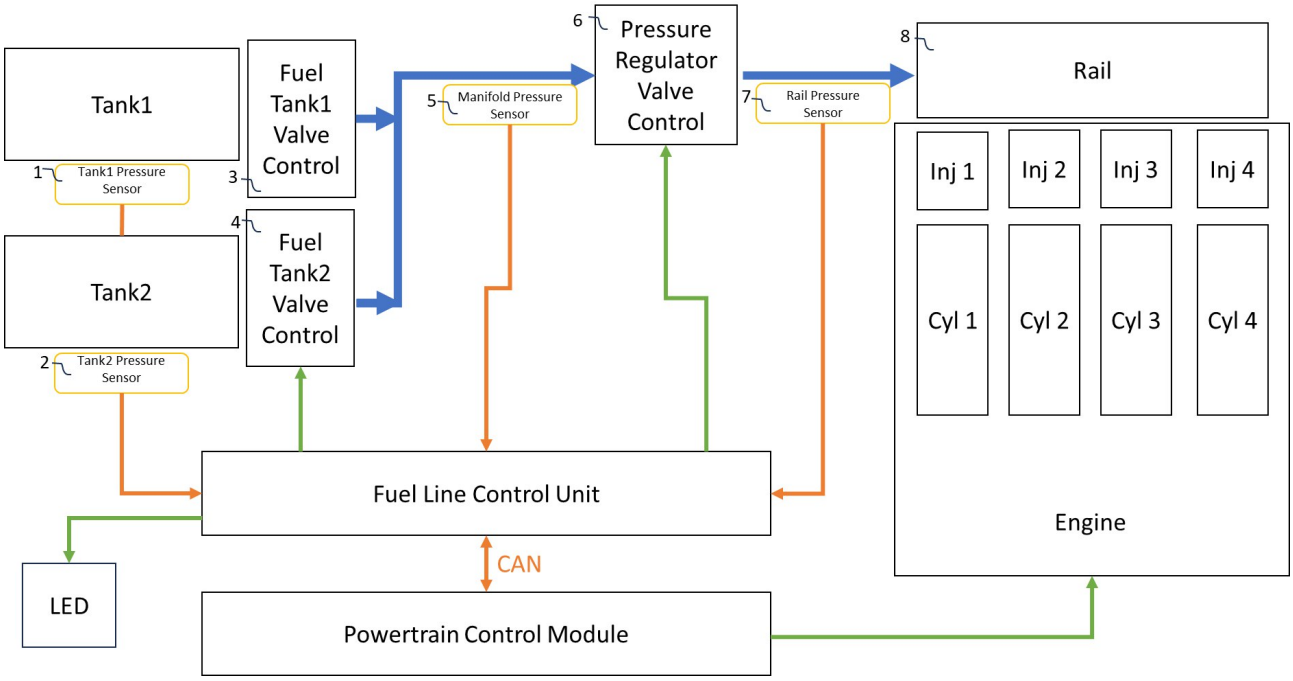


Figure 82 - System Layout

The following list outlines the main components interfaced with the FLCU:

1. **Pressure Sensors**: The FLCU interfaces with pressure sensors installed in the fuel tanks to monitor the tank pressure and with pressure sensor installed before and after the PRV. These sensors provide accurate measurements of the fuel pressure, enabling the FLCU to detect low fuel level conditions.
2. **PRV Actuator**: The FLCU interfaces with the actuator responsible for controlling the pressure regulator valve (PRV). The FLCU adjusts the actuator based on the output from the PID controller to regulate the fuel rail pressure.
3. **Tank Valves**: The FLCU interfaces with the valves connected to the fuel tanks. These valves are responsible for controlling the fuel flow between the tanks and the main fuel line. The FLCU manages the opening and closing of these valves to maintain the pressure difference within the desired range.
4. **PCM Interface**: The FLCU establishes a communication interface with the Powertrain Control Module (PCM). It receives the target fuel rail pressure from the PCM for maintaining the desired pressure level and an enabling signal for the entire system.
5. **LED**: The FLCU interfaces also with a LED that turn on in case of low fuel level.

7.2.1 BOM description

The Bill Of Material (BOM) represent the list of material necessary to develop the system in study and some other information about them.

Name	Description	Type	Range
Tank1 pressure sensor	Point 1 of chapter 7.2	Analog	0-5 V
Tank2 pressure sensor	Point 1 of chapter 7.2	Analog	0-5 V
Manifold pressure sensor	Point 1 of chapter 7.2	Analog	0-5 V
Rail pressure sensor	Point 1 of chapter 7.2	Analog	0-5 V
Fuel Tank1 Valve Control	Point 3 of chapter 7.2	Digital	0-5 V
Fuel Tank2 Valve Control	Point 3 of chapter 7.2	Digital	0-5 V
Pressure Regulator Valve Control	Point 2 of chapter 7.2	PWM	0-5 V
LED	Point 5 of chapter 7.2	Digital	0-5 V
Fuel Line Control Unit	ECU of the system	Analog	8-32 V
Powertrain Control Module	Extern ECU interfaces with the FLCU	Analog	8-32 V

7.3 Software Architectural Design

To have a clearer idea of how organize the model development is necessary a scheme of the software architecture that represent all system modules. Each block will be modeled in MATLAB and will satisfy a precise software requirements.

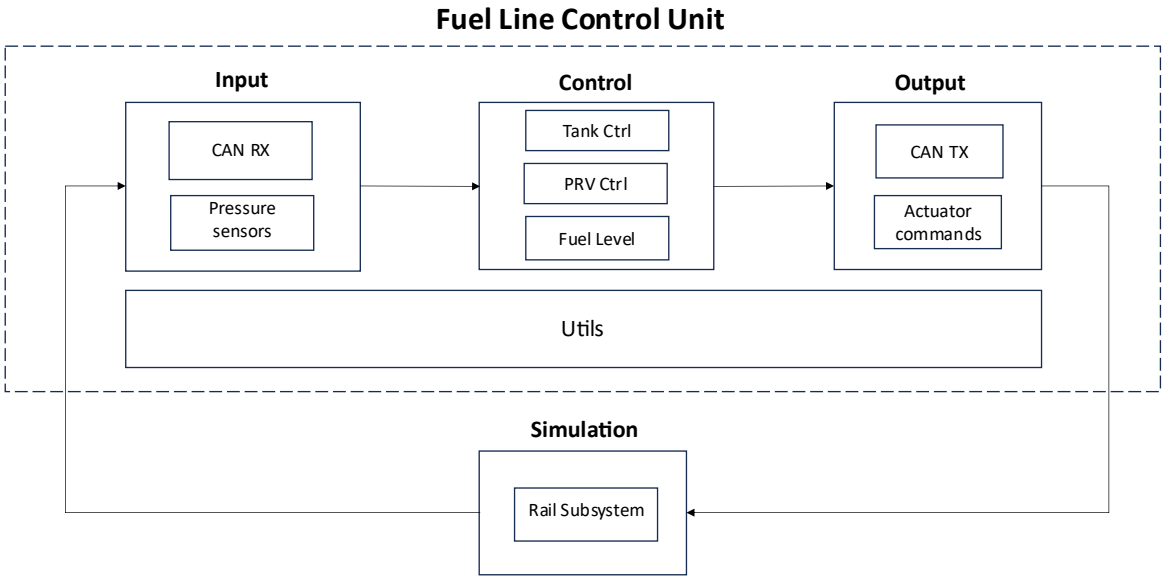


Figure 83 - Software architecture

Software requirements:

ID	Name	Description	Notes	System req. chapter reference
SW_REQ_ID_001	CANRX	Read the message according to the frame ID every 100 ms		7.1.4
SW_REQ_ID_001.1	CANRX_ENABLE_SYS	Read the message containing the enable status of the system in the frame ID 0x110	Frame ID according to DBC file	7.1.4
SW_REQ_ID_001.2	CANRX_PRAIL_TARGET	Read the message containing the target rail pressure in the frame ID 0x220	Frame ID according to DBC file	7.1.4
SW_REQ_ID_001.3	CANRX_FIX_VALUES	Receive the fix values of the enable signal and target rail pressure signal instead the original ones	Frame ID according to DBC file	7.1.4
SW_REQ_ID_001.4	CANRX_SAT_PRESS	Receive the saturated value of the pressure in case of signal out of range and warns the user setting the correct message diag. status		7.1.4
SW_REQ_ID_002	TANK_CTRL	Manage the fuel tank valves every 50 ms		7.1.3
SW_REQ_ID_002.1	TANK_CTRL_DSBL_SYS	Close all fuel tank valves in case of enable status as 0		7.1.3
SW_REQ_ID_002.2	TANK_CTRL_DIFF_PRESS	Keep the difference pressure between the two tanks below a delta calibration set to 20 bar initially		7.1.3
SW_REQ_ID_002.3	TANK_CTRL_EMPTY_TANK	Close both valves in case of low fuel level		7.1.3
SW_REQ_ID_003	PRV_CTRL	Manage the PRV based on target pressure rail and current pressure rail every 50 ms		7.1.1
SW_REQ_ID_003.1	PRV_CTRL_SET_DC	Set the Duty Cycle of the square wave to		7.1.1

		reach the desired rail pressure		
SW_REQ_ID_004	CANTX	Send message according to the frame ID every 100 ms		7.1.4
SW_REQ_ID_004.1	CANTX_ACTUAL_PRAIL	Send a message to frame 0x123 containing a signal with the actual pressure rail	Frame ID according to DBC file	7.1.4
SW_REQ_ID_004.2	CANTX_FUEL_LEVEL	Send a message to frame 0x167 containing a signal with the actual fuel level and another signal containing a boolean value that indicates if the fuel level is low.	Frame ID according to DBC file	7.1.4
SW_REQ_ID_004.3	CANTX_DIAGNOSTIC	Send a message to frame 0x145 containing sensors and actuators diagnostic	Frame ID according to DBC file	7.1.4
SW_REQ_ID_005	PRESS_SENS	Read the pressure values from the sensors every 4ms	Analog PIN according to the BOB	7.1.5
SW_REQ_ID_005.1	PRESS_SENS_TANK1	Read and convert using a LUT the Tank1 pressure		7.1.5
SW_REQ_ID_005.2	PRESS_SENS_TANK2	Read and convert using a LUT the Tank2 pressure		7.1.5
SW_REQ_ID_005.3	PRESS_SENS_TANK_MANIFOLD	Read and convert using a LUT the manifold pressure		7.1.5
SW_REQ_ID_005.4	PRESS_SENS _RAIL	Read and convert using a LUT the rail pressure		7.1.5
SW_REQ_ID_005.5	PRESS_SENS_RAIL_SAT	Read and convert the saturate level of rail pressure		7.1.5
SW_REQ_ID_006	FUEL_LEVEL	Manage the fuel level		7.1.2
SW_REQ_ID_006.1	FUEL_LEVEL_COMPUTATION	Calculate the fuel level based on tank1 and tank2 pressure		7.1.2
SW_REQ_ID_006.2	FUEL_LEVEL_LOW	Check if the fuel level falls below the threshold		7.1.2

SW_REQ_ID_007	ACTUATOR_COMMANDS	Give commands to the valves based on the computations of control blocks		
SW_REQ_ID_007.1	ACTUATOR_COMMANDS_TANK1_VALVE	Open/Close tank1 valve according to diagnostic status (close in case of diagnostic error)		7.1.3
SW_REQ_ID_007.2	ACTUATOR_COMMANDS_TANK2_VALVE	Open/Close tank2 valve according to diagnostic status (close in case of diagnostic error)		7.1.3
SW_REQ_ID_007.3	ACTUATOR_COMMANDS_PRV	Give the command to PRV based on computation of control block according to diagnostic status (DC=0 in case of diagnostic error)		7.1.1
SW_REQ_ID_008	UTILS	Manage diagnostic of sensors and actuators		7.1.5 Diagnostic capabilities
SW_REQ_ID_008.1	UTILS_TANK1_PRESSURE_SENSOR	Check if the tank1 pressure sensor works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.2	UTILS_TANK2_PRESSURE_SENSOR	Check if the tank2 pressure sensor works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.3	UTILS_MANIFOLD_PRESSURE_SENSOR	Check if the manifold pressure sensor works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.4	UTILS_RAIL_PRESSURE_SENSOR	Check if the rail pressure sensor works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.5	UTILS_LED	Check if the LED works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.6	UTILS_PRV	Check if the PRV works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.7	UTILS_TANK1_VALVE	Check if the Tank1 Valve works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities
SW_REQ_ID_008.8	UTILS_TANK2_VALVE	Check if the Tank2 Valve works properly	Correct Range: 0-5 V	7.1.5 Diagnostic capabilities

7.4 System development

The system in study has been developed using a MBD approach. Firstly, it has been created a new MATLAB project where to put all folders and files related to the system. Next, using the created blocks library, it has been updated blocks and enumerative data types (using the block library in *figure 48*) and has been imported the DBC file containing all messages (as shown in the *figure 54*).
As mentioned before, every module in the software architecture has a corresponding Simulink model and in turn, a data dictionary.

7.4.1 CAN RX module

Concerning Input modules, the CANRX model uses the created blocks library to receive messages declared in the DBC files and makes available to the other models the signals received.
As it is shown in the figure below, the model is inside the subsystem and contains the description of the software requirement referred to (in this case the CANRX model is referred to the SW_REQ_ID_001).

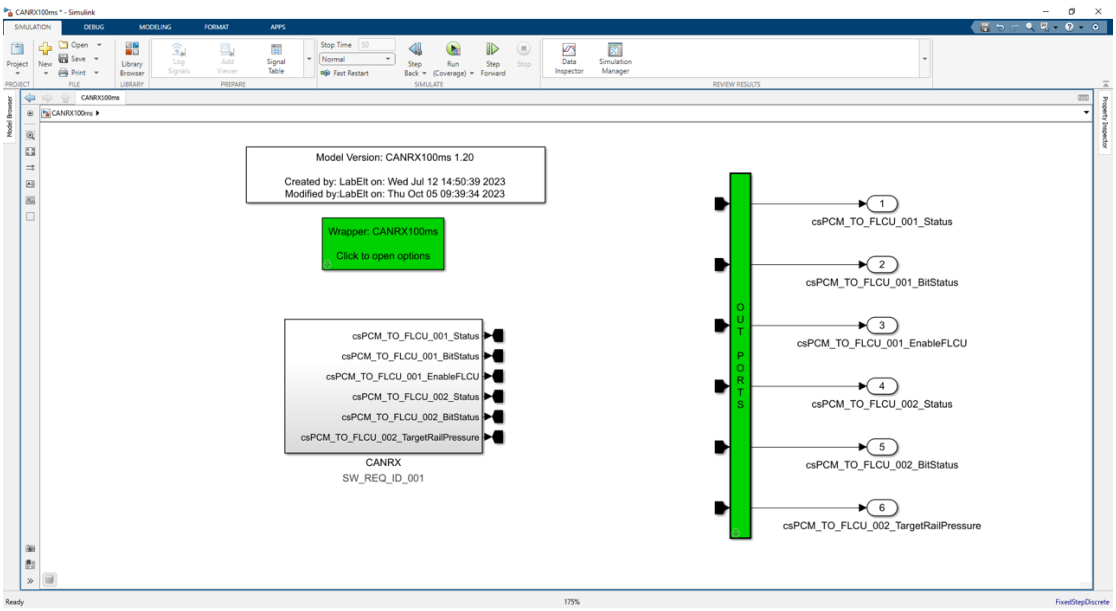


Figure 86 - CANRX module

7.4.2 Pressure Sensor module

The Pressure Sensor module (*figure 85*) uses the Analog Input blocks for reading voltages from pins related to all pressure sensors. After having received the voltage values, they pass in a EMWA (Exponentially-Weighted Moving Average) filter to smooth out short-term fluctuations and then they are saturated in the range [0.5 – 4.5] V. Then through LUTs, voltage values are converted to the unit of measurement of pressure (Bar) in order to be used from the other modules. Before output, all pressure signals pass in a fix block (switch block) to permit to the user to change their values in the testing phase through calibrations.

This module, as the name remember, has to run every 4ms so after generating the code it will be inserted in the right OS task.

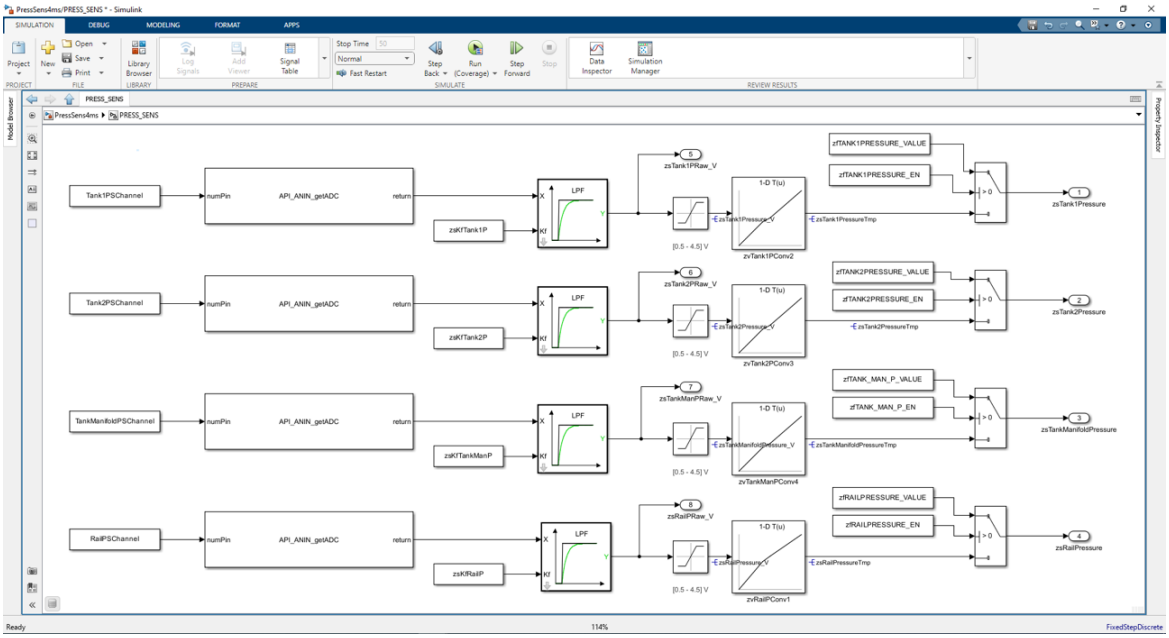


Figure 85 – Pressure Sensor module

7.4.3 Fuel Level module

Control modules are the core of the system. They are divided in three modules each of which have a specific control objective. They must be run every 50ms to have an optimal control of the fuel delivery system.

The Fuel Level control logic is very simple. Taking in input the two tank pressures from the precedent module, it checks if both are below a certain calibratable threshold (in this case is equal to 5 bar) and in that case gives in output the signal (xsLEDEn) to turn on the reserve fuel LED. Furthermore, it calculates the total fuel level (xsFuelLevel) summing the two tank pressures and reproportion it in percentage.

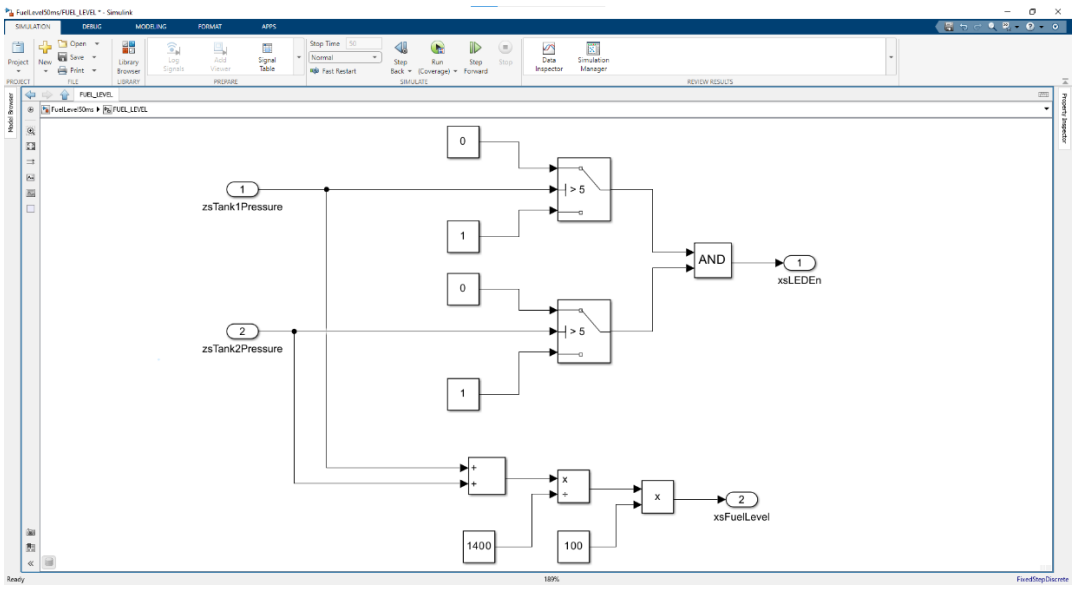


Figure 86 - Fuel Level control module

7.4.4 Tank Control module

The Tank control logic (*figure 87*) is developed using Stateflow for representing state machines. The main two state are represented by 'ON' / 'OFF' and indicate the enabling state of the tanks. The default entry point is in the 'OFF' state where is initialized the signal of the state valve (xsValveState) to 0. The system remains in this state while it is in fuel reserve (both tank pressure below the reserve threshold) for preserving and not damage the vehicle.
If the FLCU system is enabled (xsSystemAbilitation==1) the system passes to the 'ON' state, where one of the two tank valve is opened and the natural gas can flow. As explained in the requirements, the tanks are opened in an alternating manner to avoid big pressure changes, so when the pressure difference between the tanks is above a calibratable threshold (xsSWITCH_TANK_THR) the control logic change valve passing from TANK1_ON to TANK2_ON and vice versa. In those states, is set the valve state signal to the proper value based on the tank to open.
The system returns to the 'OFF' state when it receives the signal to disable the FCLU system (xsSystemAbilitation=0) or if the fuel is on the verge of running out.

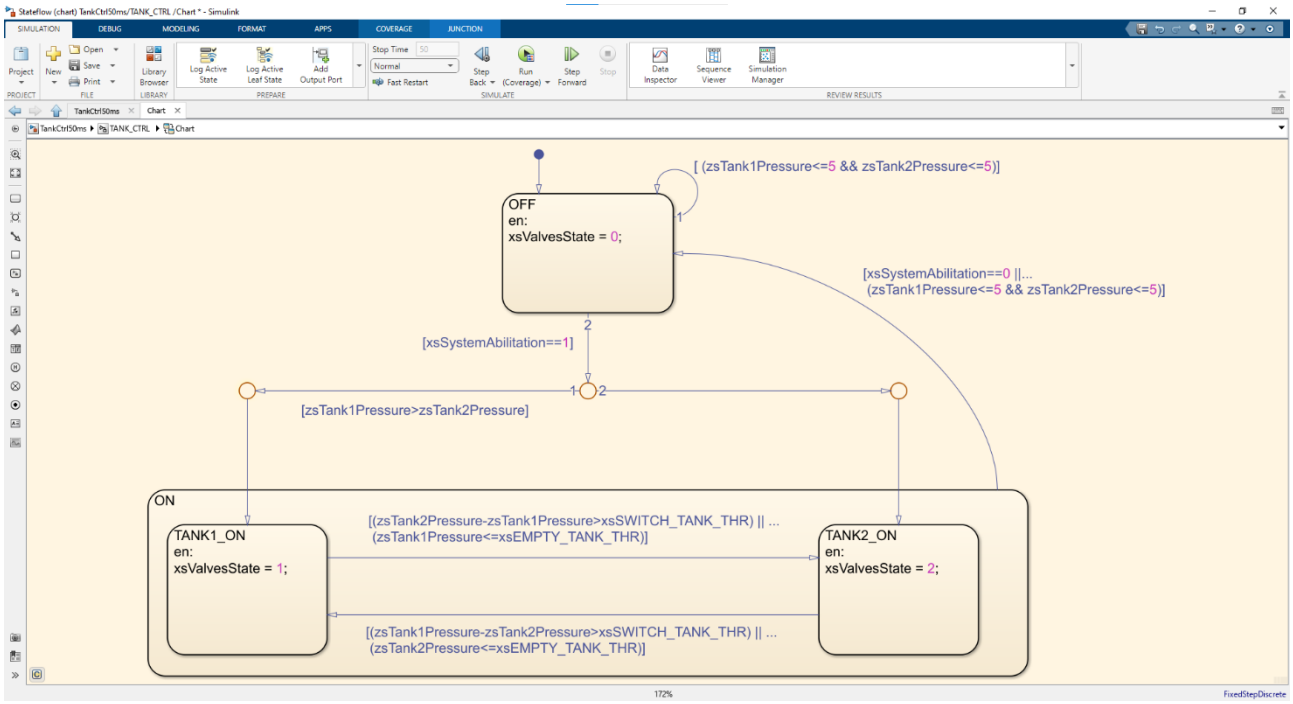


Figure 87 - Tank control module

7.4.5 PRV control module

The PRV control uses a PID controller to control the pressure flow of the natural gas. The controller is of discrete type since it must run on a real hardware every 50ms.
This module receives the target rail pressure via CAN and the actual rail pressure from sensors. It calculates the error value subtracting measured pressure from the desired pressure. The error will be used from the three parts of the PID controller to calculate its proper contribute.

The proportional part (*figure 88*) follows the formula $P = K_p * \varepsilon[n]$ where K_p is the proportional gain while $\varepsilon[n]$ represent the tracking error in that moment. Its contribute is directly proportional to the error and in Simulink is modeled as in the *figure 86.*

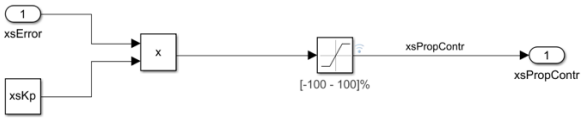


Figure 88 - Proportional part of PID controller

Obviously, the parameter K_p is a calibration to permit to be modifiable during various test.

The integral part (*figure 89*) is proportional to the sum over time of the error and follows the formula $I = \left(K_I * \frac{\varepsilon[n]}{T_s}\right) + I[n-1]$ where K_I is the integral gain calibratable, T_s is the sampling time (in this case is equal to 0.05 since the controller has to run every 50ms) and $I[n-1]$ is the integral contribution in the previous instant. The delay block (characterized by z^{-1} block) has the scope to produce in output the signal of the previous instant and besides the input signal has two other inputs, the initial condition (represented by x_0) and the external reset (represented by the up arrow) that impose the initial condition when is triggered.

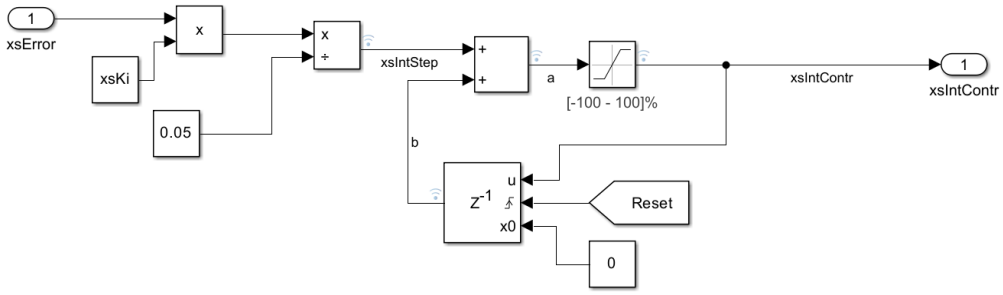


Figure 89 - Integral part of the PID controller

The derivative part (*figure 90*) is proportional to the speed of the error signal changing (derivative over time) and is characterized by the following formula: $\frac{(\frac{d\varepsilon}{dt}[n]+\frac{d\varepsilon}{dt}[n-1])}{2} * K_D$ where $d\varepsilon = \varepsilon[n] - \varepsilon[n-1]$.
In this case it has been used the average of the last two sample for a better integration.

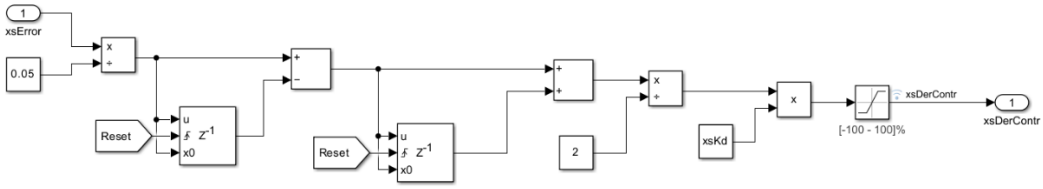


Figure 90 - Derivative part of the PID controller

The overall implantation of the PID controller (*figure 91*) for the PRV management, also takes into account the system enable signal (xsSystemAbilitation) to set the DC to 0 in case the system must be disable. Furthermore, when the system is disabled, all delay blocks are triggered to be reset.
From the figure, it can be seen that the model after summing all three contributes of the PID, gives in output the duty cycle (saturated with minimum and maximum value calibratable) of the PWM signal that control the valve actuator.
A duty cycle saturated to 100% means that the valve is completely opened and, vice versa, when the system is disabled the duty cycle is 0 in order to close the valve and deny the passage of natural gas.

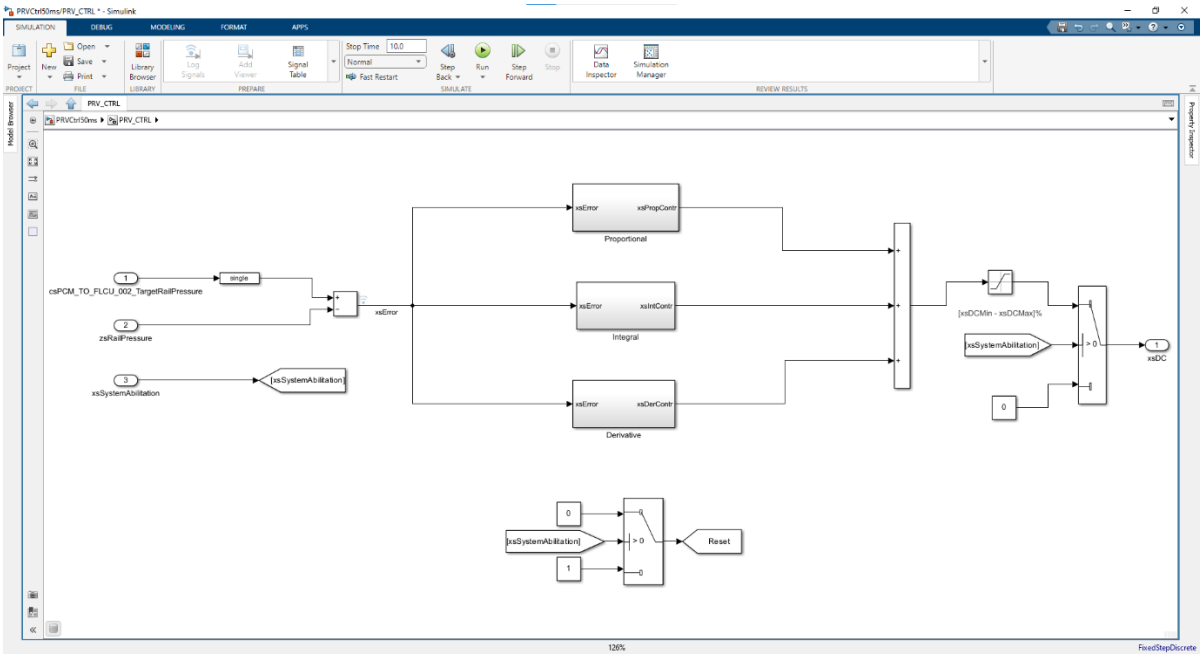


Figure 91 - PRV control module

The MIL phase of this module has been developed in Simulink using a Test Harness. It is another Simulink model that isolates the module under test and, through various stimuli, verifies the output. To do this, a closed loop control system is created emulating the plant, so at each step time the actual Rail Pressure and the target Rail Pressure to give as input to the PID controller are known.
The Powertrain Control Module is in charge of managing the desired engine torque, in particular it manages injection times and gives the command to open injectors. In this way is generated a fuel quantity variation that is described by a signal Q_{inj} ($\frac{dm}{dt}$) where 'm' is the mass of the fuel and for this reason is necessary to keep a certain rail pressure acting on the PRV. The signal Q_{inj} and the duty cycle of the square wave used to control the PRV are combined in a LUT to compute the delta of the actual Rail pressure to send in input to the PID controller. The fuel quantity variation (Q_{inj}) is created using the signal builder of Simulink.

The test harness is represented in the *figure 92* and as it can be seen the controller is represented by the grey subsystem which is connected to the real model of the PRV control. As mentioned before the duty cycle and the Q_{inj} are combined to produce the delta pressure that is summed to the previous pressure to give the new actual pressure. Then this pressure is saturated between [0:50] bar (the range of the rail pressure) and filtered in the EMWA filter to produce a clearer pressure to send as input to the controller. Both the Q_{inj} and the Target Rail Pressure signals are created using the signal editor and as shown there have been tested static and dynamic case of these two signals.

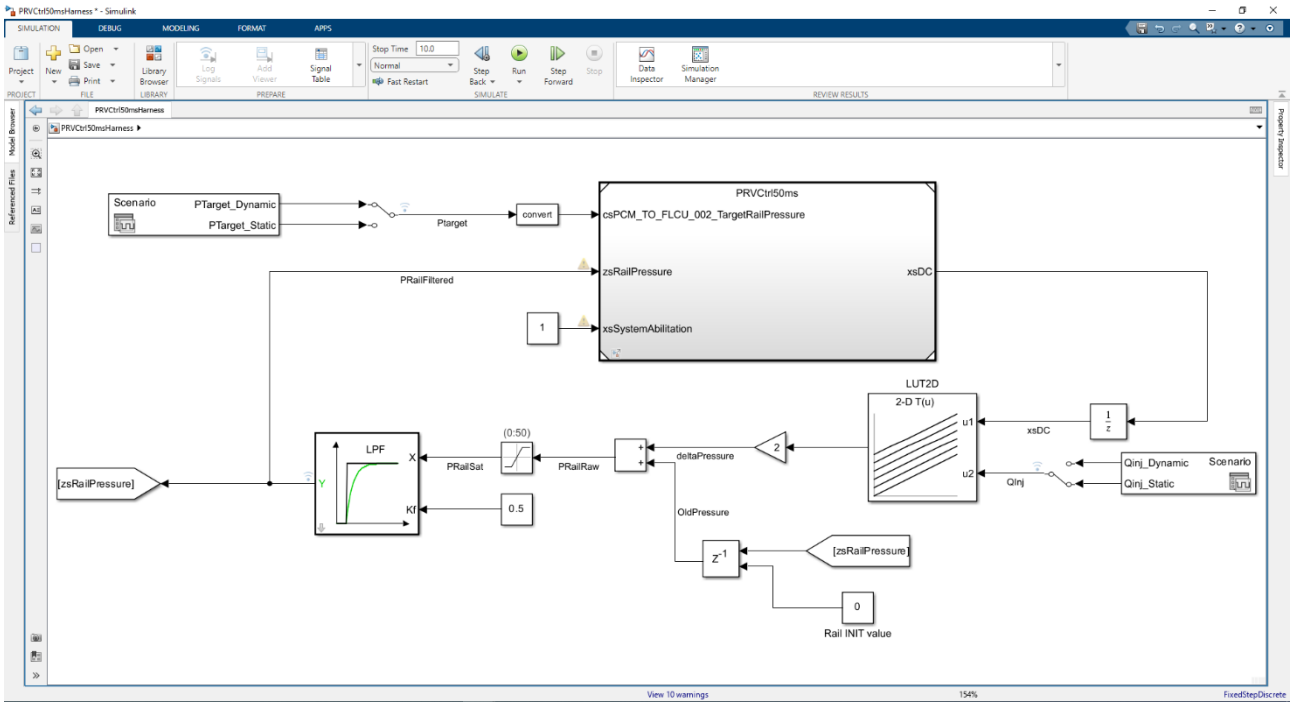


Figure 92 - Test Harness PID

The *figure 93* shows the pressure changes of the LUT. In the X axis is present the gas flow rate [kg/h] while in the Y axis the DC range [%]. If for example the flow rate is empty and the valve is completely opened, the rail pressure will have a positive pressure variation of 5 bar in the unitary step.

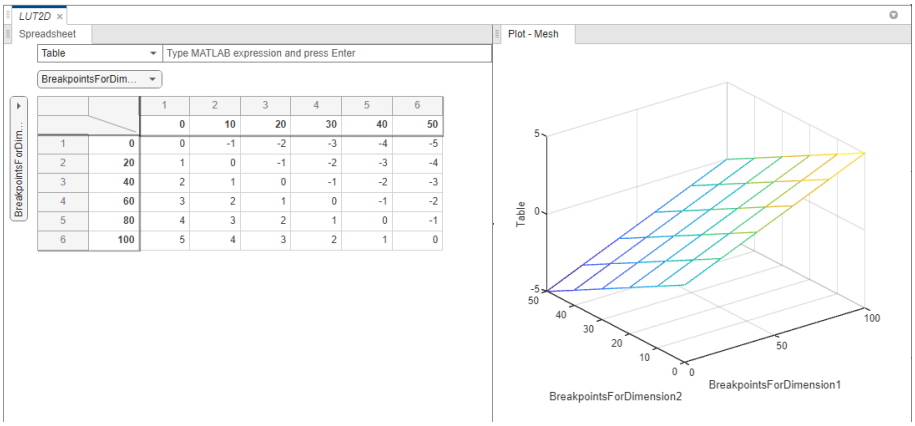


Figure 93 - LUT of delta pressure

To find the right values of the PID parameters (Kp, Ki, Kd) various test has been made. Each parameter affects the contribution of the proportional, integral and derivative part. To start the calibration of parameters both pressure target and flow rate are set to static, in particular the Rail Pressure Target to reach is 25 bar and the flow rate is 20 kg/h.
To better understand the behavior of the controller, are logged many parameters such as all the P-I-D contributes, the duty cycle acted to the valve, the flow rate and, the most important, the comparison between target and real rail pressure.

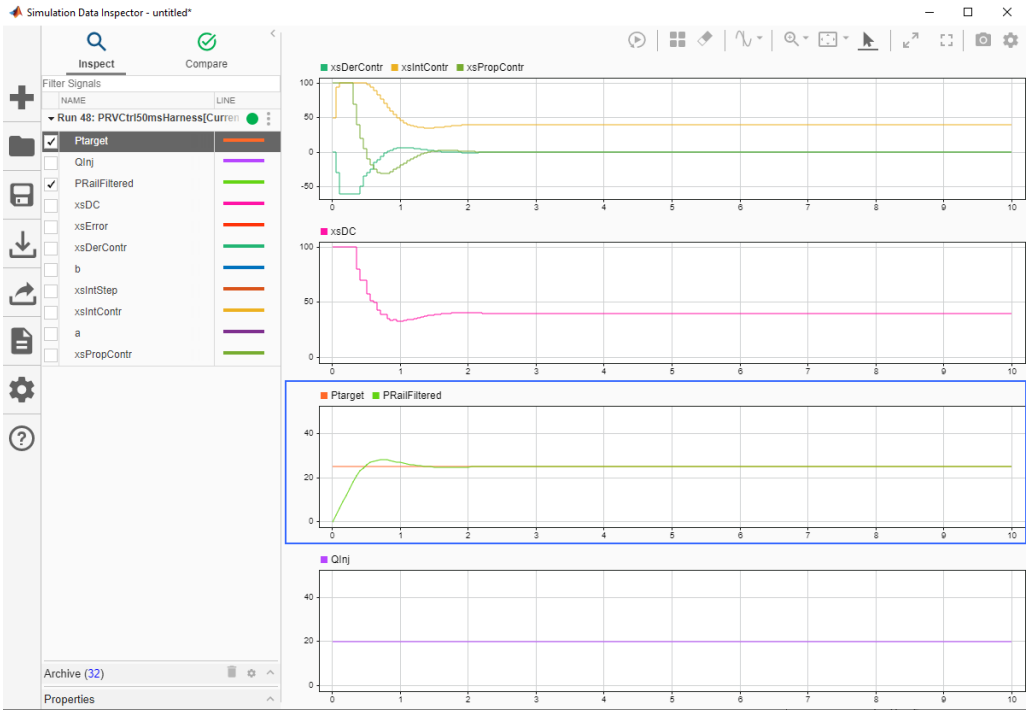


Figure 94 - Simulation data inspector, 1

From the simulation data inspector (*figure 94*) is highlighted in blue the comparison between measured and desired rail pressure and it can be seen that in about 1 second, starting from 0 bar, the Rail pressure reach the pressure goal of 25 bar with a slight overshoot.
Starting from the PID parameters obtained from this first simulation, the next step is to create a simulation more similar to reality, so target pressure and flow rate varying over time.

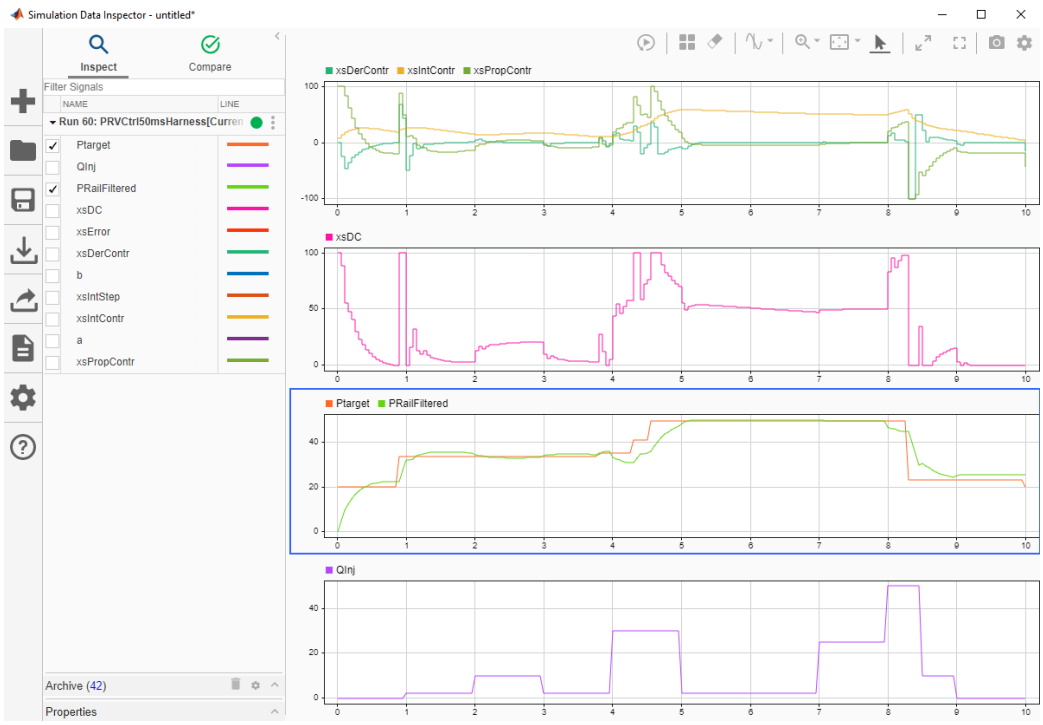


Figure 95 - Simulation data inspector, 2

The results (*figure 95*) show how the PID controller works to follow the desired rail pressure at each step time acting on the duty cycle (xsDC) to keep the error as minimum as possible.

In the future months will be conducted various test (HIL) in the fluid dynamics laboratory to verify the correct functionality of the PID controller implemented in an Electronic Pressure Regulator (EPR).

7.4.6 CAN TX module

The module is responsible to send messages to PCM module via CAN. It uses the block library to automatically create subsystems related to each message and the final result is visible in the *figure 96.*

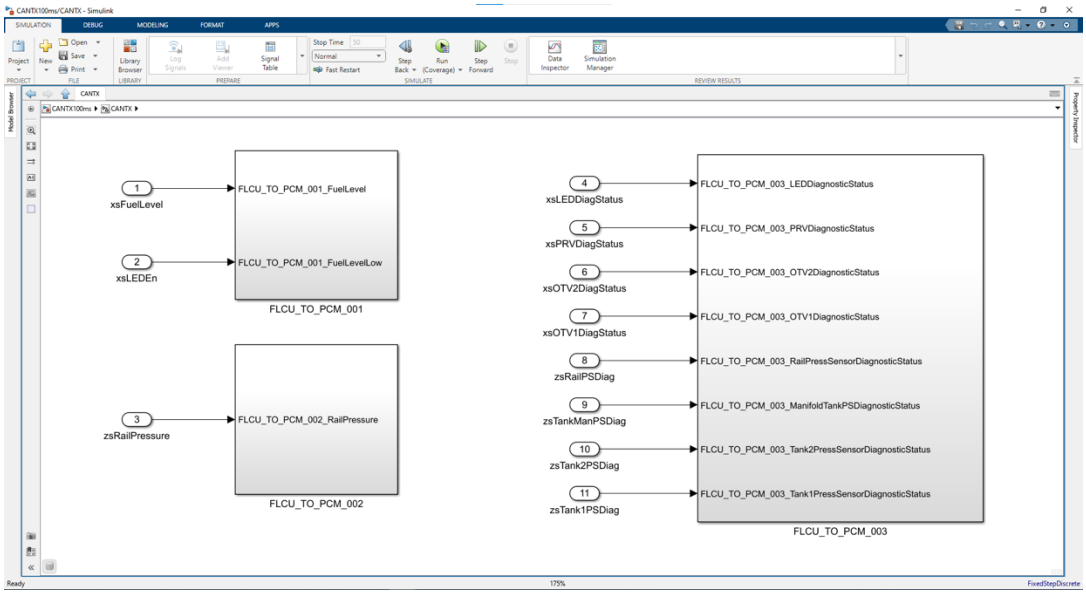


Figure 96 - CANTX module

The messages created are those present in the DBC file imported, 'FLCU_TO_PCM_001' contains the fuel level and the LED enable signal. 'FLCU_TO_PCM_002' contains the rail pressure and the 'FLCU_TO_PCM_003' all the diagnostic status of all sensors/actuators.

7.4.7 Actuator commands module

This module (*figure 97*) is in charge of managing the actuators present in the FLCU. To do this, are used API blocks library automatically created that concern with I/O. In particular, OTVs and LED are managed with digital output block library, so it is only necessary to pass to the C Caller block the DOUT pin channel related to the actuator and the binary enable state (converted to 'uint8' to be aligned with the API function) as input.
The PRV is managed via PWM, so it is used the PWMOUT library block with the duty cycle produced by the PRV control module. The PWM channel and the period of the square wave are also passed as input to correctly call the API function.

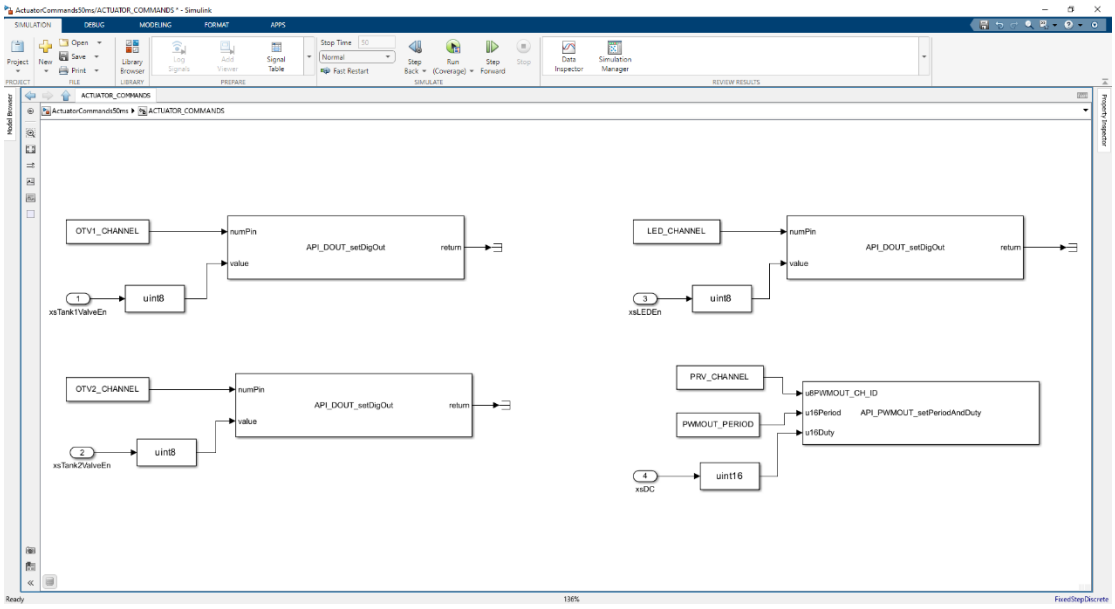


Figure 97 - Actuator commands module

7.4.8 Utils module

This last module manages (*figure 98*) all diagnostic signals of sensors and actuators and produce in output the system enable system based on the overall system status. If one of all sensors or actuators has some problem, the system will be disable.

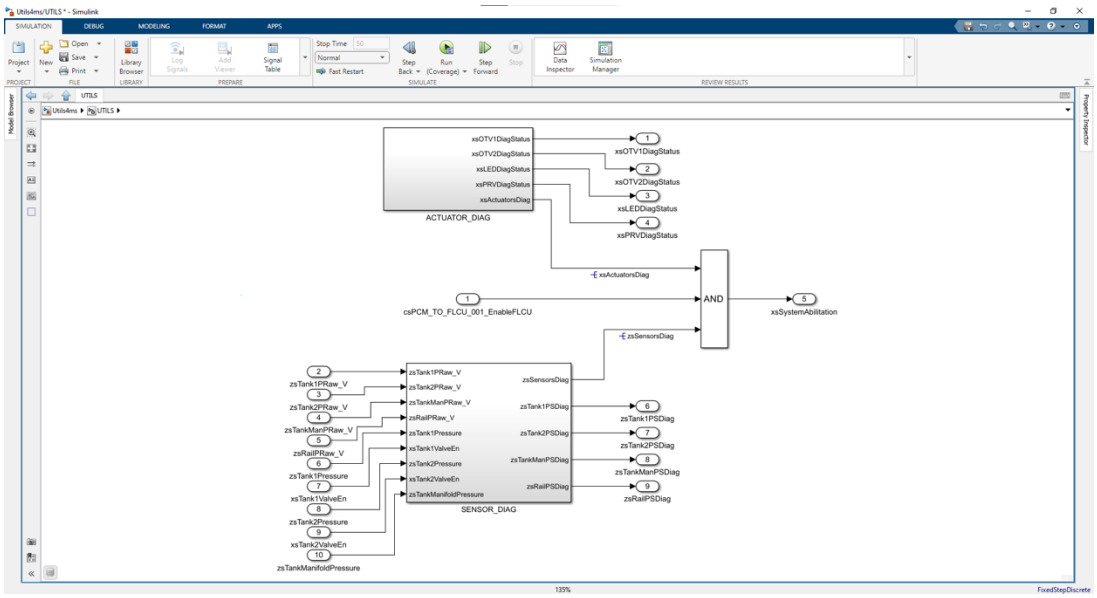


Figure 98 - Utils module

The module combines in an AND block the CAN message received for the FLCU system enabling, the actuator diagnostic signal (representative of all actuators) and the sensor diagnostic signal (representative of all signals) with the scope of arrest immediately the FLCU system in case of failure.

The actuators diagnostic (*figure 99*) is made by API 'ErrorInfo' blocks that gives information about the status of a specific I/O channel.

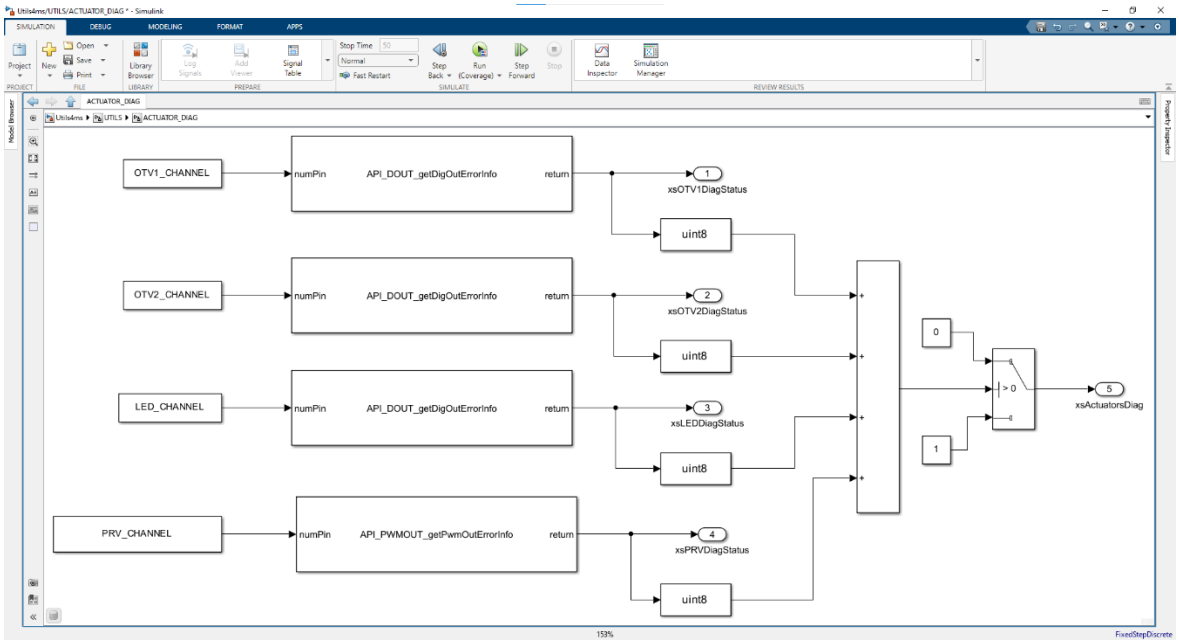


Figure 99 - Actuator diagnostic

All these blocks give as output '0' if the actuator works properly and a value higher than 0 if there is some error. The correct state is reached when the sum of all these values is 0, so in this case the actuators diagnostic signal (xsActuatorDiag) is set to 1 (to be aligned with the FLCU system enable CAN message).

The sensors diagnostic (*figure 100*) check if all raw value measured in [V] are in the correct range [0.5-4.5] V and if the tank manifold pressure sensor is equal to the tank pressure opened at that moment.

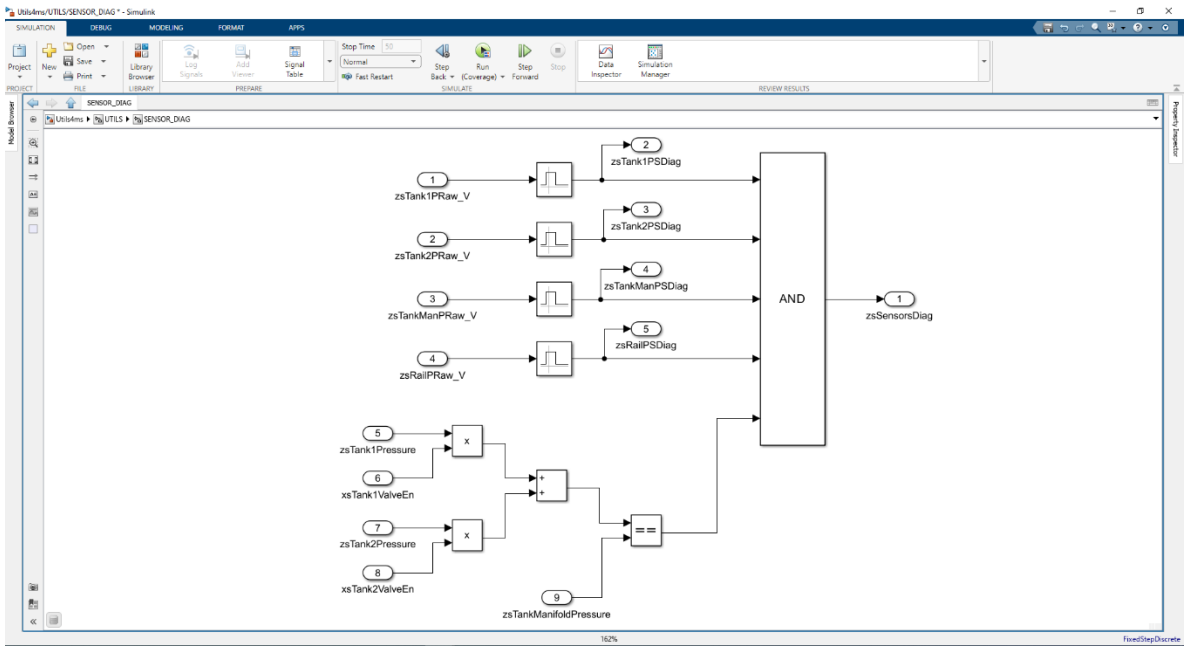


Figure 100 - Sensor diagnostic

If one of the two checks fails, the AND block gives '0' as result and the model disable all the FCLU system.

7.5 Software integration

After finishing the model design and the MIL phase, is time to integrate the automatic code generated in the real target hardware. When each single module has been developed and tested, the code is generated inside the model through the appropriate block library and the final result is a folder containing all source codes. As explained in the chapter 5.2, each module function bust be called in the right `API_OS_task` to be executed at the correct frequency and, when the API file is ready, all step explained in that chapter can be done. After flashing in the ECU the 's19' and 'a2l' file is possible to start the HIL phase.

The Tank Control is taken as example of this phase and in the *figure 101* is shown the initial test configuration. The biggest graph represents the value of pressures over time and the two tank pressure are initially full at 700 bar. With the parameter windows is possible to view and modify all fixed values that represent calibrations. The bottom right graph represent the status of the valve and they are also replicated above in a numeric window.

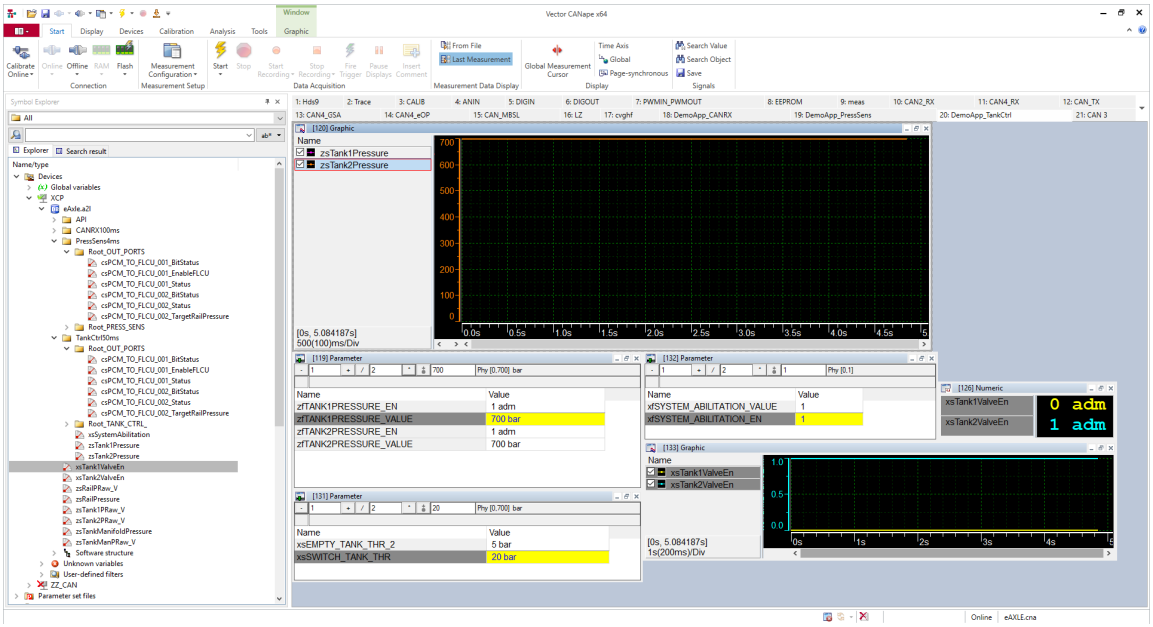


Figure 101 - Initial condition test in CANape

To verify the proper functioning of the system, the two tank pressures are modified manually to obtain the switch tank (close a valve and open the other) from the controller.
As is shown in the *figure 102,* when the active tank (Tank n.2) reaches the switch tank threshold (difference pressure higher than 20 bar) at 679 bar, the controller closes the OTV2 and open the OTV1. This action is visible in the bottom right graph at about 6.5 sec when the two lines interchange.

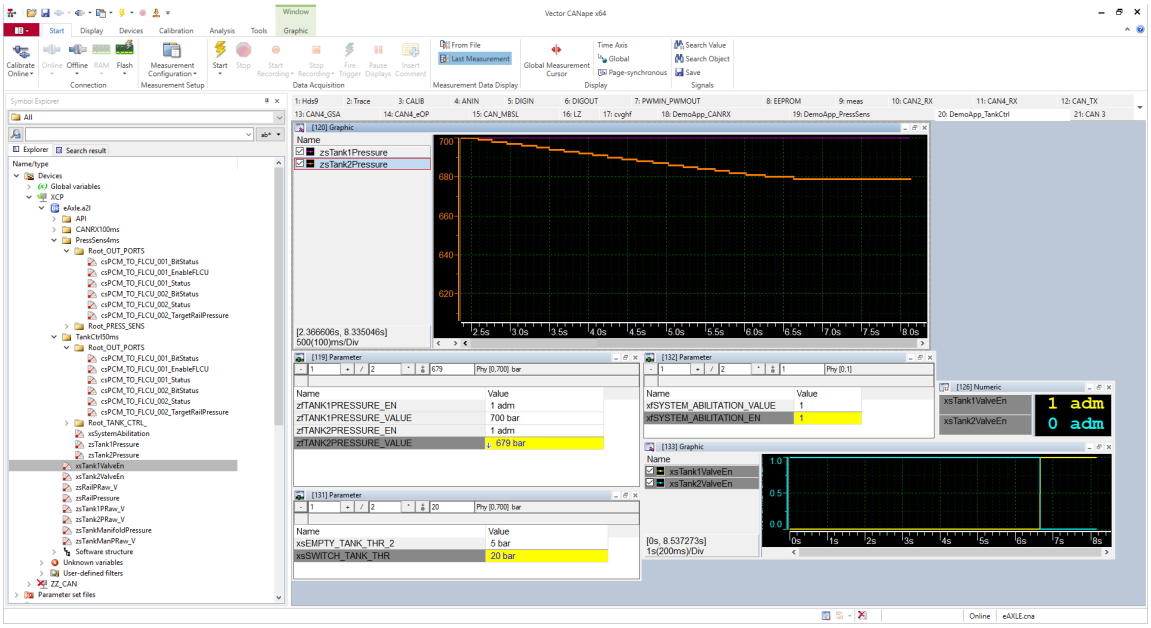


Figure 102 - Switch tank

Another feature that should be test in this module is the valve closure in case of empty tanks. In this example (*figure 103*) the empty tank threshold is set to 5 [bar] and when both tanks reach 4 bar the graph (and the numeric window) show that valve states are set to 0 [bar].

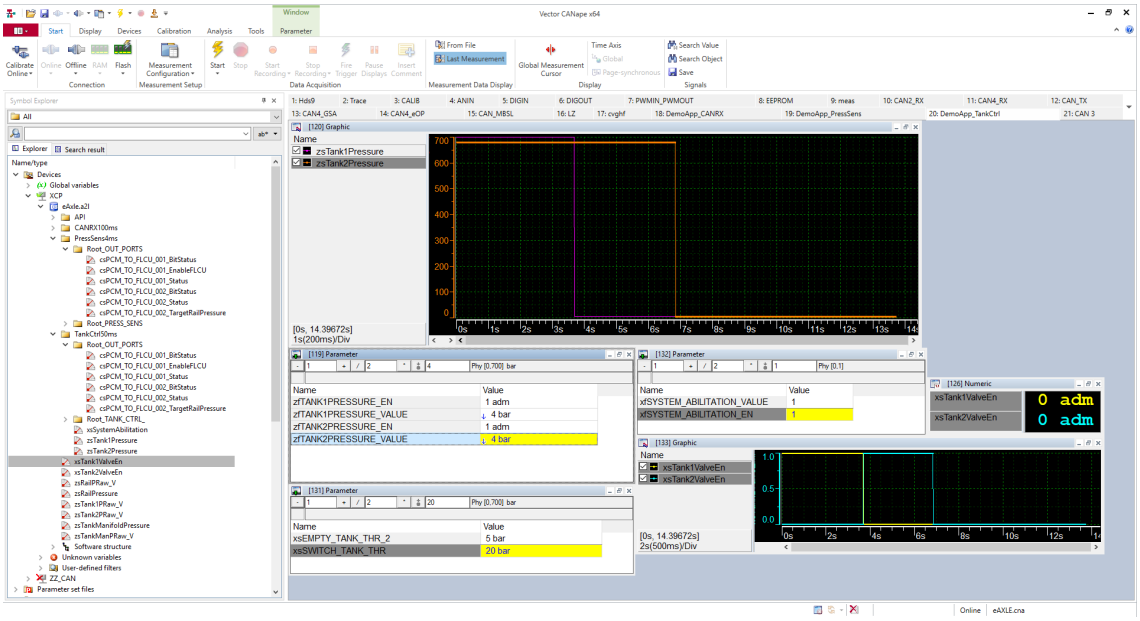


Figure 103 - Valve closure

8. Conclusion

Have a general purpose HW platform able to manage the main automotive sensors and actuators and in parallel a dedicated library for the development phase, allows the customer to integrate in a faster way every type of application software. Reduce the initial cost investment and the development time, implies a shorter time to market of the product with a higher profit.

The overall work of this thesis can be considered satisfactory. The Demo Application developed with the HW platform and the dedicated library, demonstrates the achievement of goals. Thanks to the added features in MATLAB, it has been possible to model the system more easily and quickly.
As said in the chapters before, all actions done by these library blocks were done by the user taking a lot of time and with the risk of making mistakes.

Avoid typing errors means reducing model design loops and so development time. Reduce development time means saving money and a lower time-to-market.

At the moment, the main benefits encountered in the modeling of the FLCU system brought by the library, concern the integration with the I/O channels, which is possible to easily manage some actuators, and the implementation of the DBC files in MATLAB thanks to which is possible to send and receive CAN messages directly in the model.

In the future, possible additions and enhancements will make the library increasingly complete and functional for modeling a general application in the automotive field.

Some possible addition to be implemented regard the management of:

- Diagnostic (OBD2)
- Memory
- Injectors
- Ignition coils
- Lambda sensors (HEGO and UEGO)
- Functional safety (ISO26262)
- Knock sensors
- H-bridges

Acknowledgments

I would like to thank the people who helped me through this long university journey, first and foremost my girlfriend and my family who always supported me in the most difficult times and celebrated in the happiest moments to give me motivation in achieving my goals.

A special thank goes to Emilio, without which this project would not have been possible, for all the help and dedication he has given me in these months.

To all Metatron team who from the beginning treated me as a member of their reality and helped me in times of need.

To professor Violante for his disposition and professionalism in carrying out his work.

Writing these words marks the end of one journey and the beginning of another one much longer and more challenging that will be undertaken with the same perseverance and motivation.

Bibliography

- https://www.metatron.it/it/prodotti/ch4-natural-gas-vehicle/ecus/hds
- https://autosartutorials.com/autosar-basic-software-bsw/
- https://www.synopsys.com/glossary/what-is-model-based-design.html
- https://www.collimator.ai/post/model-based-development
- Introduction-to-model-based-software-designed M.Violante
- https://www.collimator.ai/reference-guides/what-is-automatic-code-generation/
- https://en.wikipedia.org/wiki/Electronic_control_unit
- https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/mpc5xxx-microcontrollers/ultra-reliable-mpc57xx-mcus/ultra-reliable-mpc5777c-mcu-for-automotive-and-industrial-engine-management:MPC5777C
- https://forum.digikey.com/t/overview-of-the-can-bus-protocol/21170
- https://support.squarell.com/index.php?/knowledgebase/article/view/94/0/can-high--can-low
- https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial
- https://www.vector.com/int/en/know-how/protocols/xcp-measurement-and-calibration-protocol?etcc_cmp=Calibration_EN&etcc_grp=XCP&etcc_med=SEA&etcc_par=Google&etcc_bky=xcp%20protocol&etcc_mty=e&etcc_plc=&etcc_ctv=449102319289&etcc_bde=c&etcc_var=CjwKCAjwjOunBhB4EiwA94JWsHS56WeodomCtCJXGrM7TJgswBTLOQgY__LhglC_LToNmSv0XOPnGBoCGokQAvD_BwE&gclid=CjwKCAjwjOunBhB4EiwA94JWsHS56WeodomCtCJXGrM7TJgswBTLOQgY__LhglC_LToNmSv0XOPnGBoCGokQAvD_BwE
- https://it.mathworks.com/help/ecoder/ug/generate-code-modules.html
- https://cdn.vector.com/cms/content/products/asap2/Docs/ASAP2Tool-Set_Manual_EN.pdf
- https://www.vector.com/int/en/products/products-a-z/software/canape/canape-application-areas/#c314093