Politecnico di Torino

Master’s Degree in Computer Engineering

AY 2023/2024

October 2024

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

Memory Management & OBD

Advisor:

Prof. Violante Massimo

Candidate:

Oliva Mattia

Internship Tutor:

Ing. Bertrand Emilio

Summary

1. Introduction

The never-ending strive of the automotive industry towards improvement has been driven, in recent years, by the idea of environmental sustainability, which has reshaped the way vehicles are designed and manufactured.
Improvements on pre-existing technologies' emissions and the adoption of new, more environmentally friendly fuels such as methane and hydrogen are just some of the changes introduced.
To enforce this search for sustainability, different standards have been defined, revised, and changed over the years.
This thesis has been made possible thanks to the collaboration of Metatron S.p.A., a world-renowned company specialized in the design and production of pressure regulators and Electronic Control Units.

1.1. Company Overview

Metatron's history began when, at the start of the 90s, the **Fiat Research Centre** (CRF), sited in Orbassano (Turin), determined that the best way to lower gas emissions for internal combustion engines was to use natural gas fuel with "three-way" catalysts.
To kick off the industrial production of natural gas systems, CRF partnered with **Tartarini**, a Bologna-based firm specialized in "aftermarket" systems for converting gasoline and diesel engines to methane. Tartarini managed the production of the system components, with "bifuel" for passenger cars and "monofuel" for heavy duty as the chosen technologies.

From Tartarini, in 1998, some resources detached to create **Metatron**, with the goal of moving from the "aftermarket" to manufacturing and selling CNG/LNG systems directly to OEMs. Metatron became the exclusive supplier of control units and pressure regulators for IVECO.

Then, in 2008~2010, Metatron founded a new division in Volvera (Turin), fully dedicated to the electronic technologies and applications. This division obtained the technical know-how in the gas supply field from CRF and went on to develop a secondary control unit for the LPG fuelled vehicles of FCA group (Fiat Chrysler Automobiles).

Moreover, since 2010, China has been the main market for Metatron's pressure regulator, for its production of heavy-duty engines, leading the company to open a new office in

Shanghai (MAP, Metatron Asia Pacific). In 2014, Metatron acquired Digigroup, a society specialized in both development and supply of electronic components for Automotive Telematics (ITS) and, the following year, Metatron relocated all the activities concerning electronics applications in the Volvera site, founding a new society named **Metatronix**. However, due to the increasing differences between ITS and Powertrain markets, in 2018 Metatronix was made completely autonomous and, to reinforce the Powertrain group, the **Metatron Research Centre** was created in Volvera.

In 2021, a binding agreement for the acquisition of Metatron S.p.A. was signed by the **Landi Renzo Group**. This would ultimately strengthen and accelerate the group's strategy aimed at reaching a leading position in the supply of systems and components for the **Natural Gas and Hydrogen Mobility in the Mid & Heavy-Duty** segment, which will keep on growing in the upcoming years.

Headquarter and plant • Castel Maggiore (Bologna - Italy)
Engineering & Electronics • Volvera (Turin - Italy)
Metatron Control Systems SHANGHAI Co., Ltd
METATRON
LANDIRENZO GROUP

Figure 1.1 - Metatron Offices

1.2. Thesis Goals

Over the years, Metatron has developed an impressive and layered code base,
ever adapting to the most recent standards and guidelines. However, due to the sheer number of these standards, the freedom of interpretation and implementation that they allow, and the different applications and customers' requests, this codebase has continued to grow in complexity while maintaining some obsolete functions' predispositions and oversized structures. This is particularly true for what concerns the management of the On-Board Diagnostic.

The end goal of this thesis work is the redefinition of the diagnostic managers to replace the existing ones, overstructured by years of evolving standards and requirements, in anticipation of future implementations and porting of the OBD system on different boards and applications (not exclusively focused on engine control systems).

Rather than simply refactoring the existing code and processes by removing the unnecessary procedures and artefacts dictated by now defunct or changed guidelines, for this thesis work we started anew by analysing the requirements of the state-of-the-art standards for the on-board diagnostic on heavy-duty systems (OBD2, WWH-OBD, J1939), while also taking into account the constraints dictated by Euro-VI and China-VI.
The information gathered by this analysis has then been used to define a set of requirements and implement a flexible strategy to handle fault detection collecting the information needed by the standards. Great focus has been placed on the abstraction of the adopted solutions to best suit the customers' needs and ease of use, and on the memorization and communication of the detected faults in accordance with the guidelines.

Before tackling the part concerning the OBD we defined an intermediate goal, propaedeutic to the work on the diagnostics: the management of non-volatile memory (NVRAM). The aim was to develop a way for the management of units to give the users the possibility to store and retrieve data from permanent memory with a safe approach. As per the other goal, the emphasis went on making the chosen solution as configurable as possible for the users.

For both the intermediate and final goals of this thesis, a set of APIs has been developed to allow customers to interact with the underlying system in a safe and reliable way.
As the applications are developed following a Model-Based design, blocks for the development environment (MATLAB/Simulink) of the APIs have also been created. In addition, blocks with graphical interfaces (*masks*) to facilitate the modification of parameters and the generation of code have been implemented.

1.3. Working Environment

The upcoming paragraphs present a concise summary of the key components comprising the development environment, hardware, and tools utilised throughout the course of this thesis.

1.3.1. HDS9

HDS9 is an Engine Control Unit created by Metatron for medium- and heavy-duty applications (HDS stands for Heavy Duty System) on methane-fueled engines. It has been developed based on the state-of-the-art of the available technology, and it is up-to-date with the most recent OEMs' global standards on emissions, on-board diagnostics, and safety, such as EU-VI and ISO 26262.

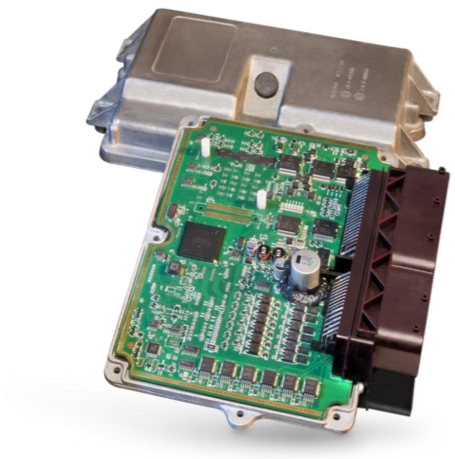


Figure 1.3.1 - HDS9

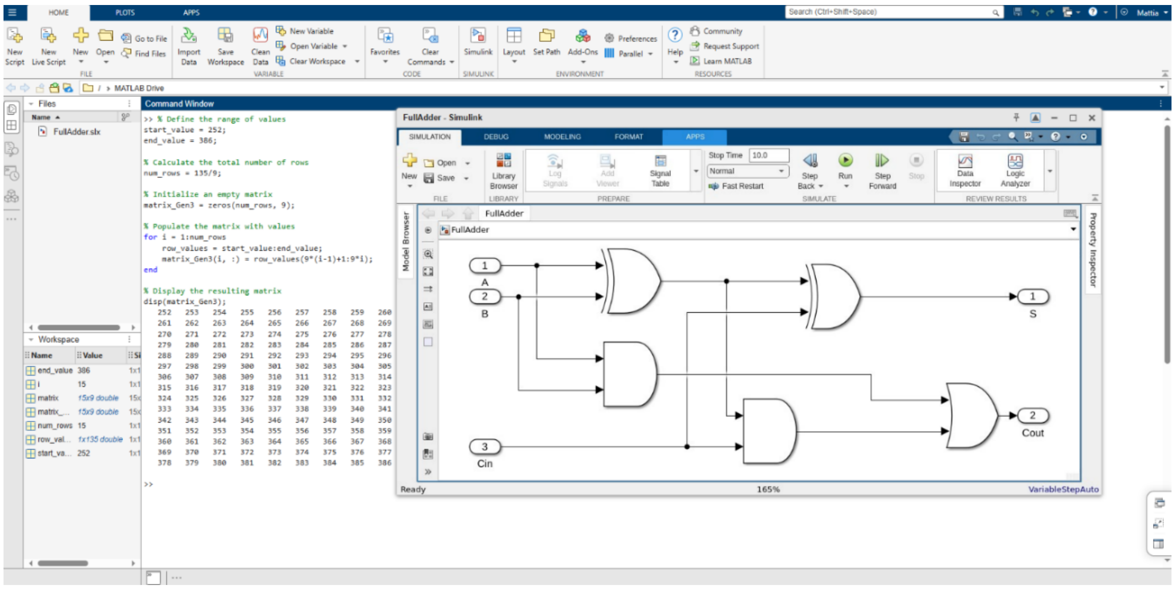
Thanks to its hardware specifications (described in more details in the "Hardware Architecture" chapter) and performances, this ECU has been an ideal platform to test and validate this thesis' work.

1.3.2. MATLAB & Simulink

MATLAB (a portmanteau of "Matrix Laboratory") is a numeric computing platform developed by MathWorks. It is specifically designed for engineers and scientists to analyse and design systems and products. The heart of this environment is the homonymous matrix-based programming language, which allows for the natural expression of computational mathematics. MATLAB can be used to analyse data, develop algorithms and applications, create and study models, and deploy the designed systems to embedded devices and enterprise applications. The versatility of use of

MATLAB is possible thanks to the possibility of combining the core environment with other products, such as Simulink.

Simulink is a widely used technology in the automotive industry, developed by MathWorks and incorporated into the MATLAB suite. It is a block diagram environment that supports both multidomain simulation and model-based design, enabling, among others, system-level design, simulation of both continuous and discrete time systems, and automatic code generation. Using this support tool, simulation and validation can be executed on the model (MIL) and once this is ready and the behaviour matches the expected one, the Embedded Coder will take care of generating the software code following the defined specifications for the target HW. The use of these tools helps increase productivity and efficiency, enhance modularity and portability introducing a separation between model and code, and reduce the chances of human errors.



```
>> % Define the range of values
start_value = 252;
end_value = 386;

% Calculate the total number of rows
num_rows = 135/9;

% Initialize an empty matrix
matrix_Gen3 = zeros(num_rows, 9);

% Populate the matrix with values
for i = 1:num_rows
    row_values = start_value:end_value;
    matrix_Gen3(i, :) = row_values(9*(i-1)+1:9*i);
end

% Display the resulting matrix
disp(matrix_Gen3);
   252   253   254   255   256   257   258   259   260
   261   262   263   264   265   266   267   268   269
   270   271   272   273   274   275   276   277   278
   279   280   281   282   283   284   285   286   287
   288   289   290   291   292   293   294   295   296
   297   298   299   300   301   302   303   304   305
   306   307   308   309   310   311   312   313   314
   315   316   317   318   319   320   321   322   323
   324   325   326   327   328   329   330   331   332
   333   334   335   336   337   338   339   340   341
   342   343   344   345   346   347   348   349   350
   351   352   353   354   355   356   357   358   359
   360   361   362   363   364   365   366   367   368
   369   370   371   372   373   374   375   376   377
   378   379   380   381   382   383   384   385   386

>>
```

Figure 1.3.2 - MATLAB and Simulink sample screen

1.3.3. LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a graphical programming environment developed by National Instruments widely used for data acquisition, instrument control, and industrial automation.

LabVIEW is well known for its intuitive programming approach, based on the "G" graphical programming language, which enables users to efficiently realise complex test and measurement systems by building programs (here called Virtual Instruments, VI) by connecting functional nodes on a block diagram.

This environment boasts extensive support for connectivity to various instruments, and the added functionality of helping the users design an integrated interface for each program.
For the already mentioned qualities and many more, LabVIEW was utilised in this thesis to construct automated testing programmes to stress test the built solutions.

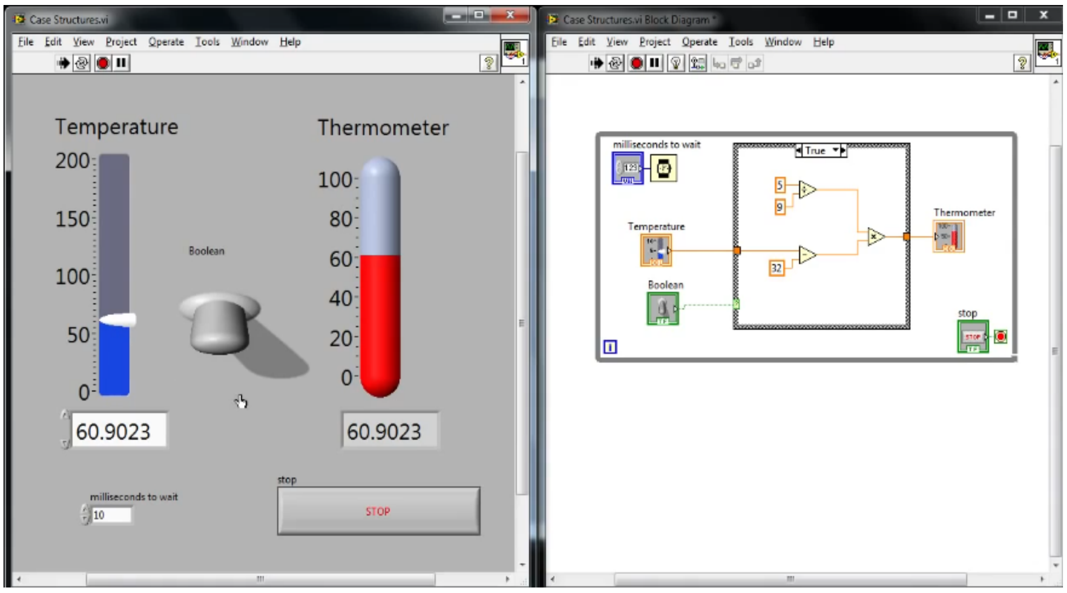
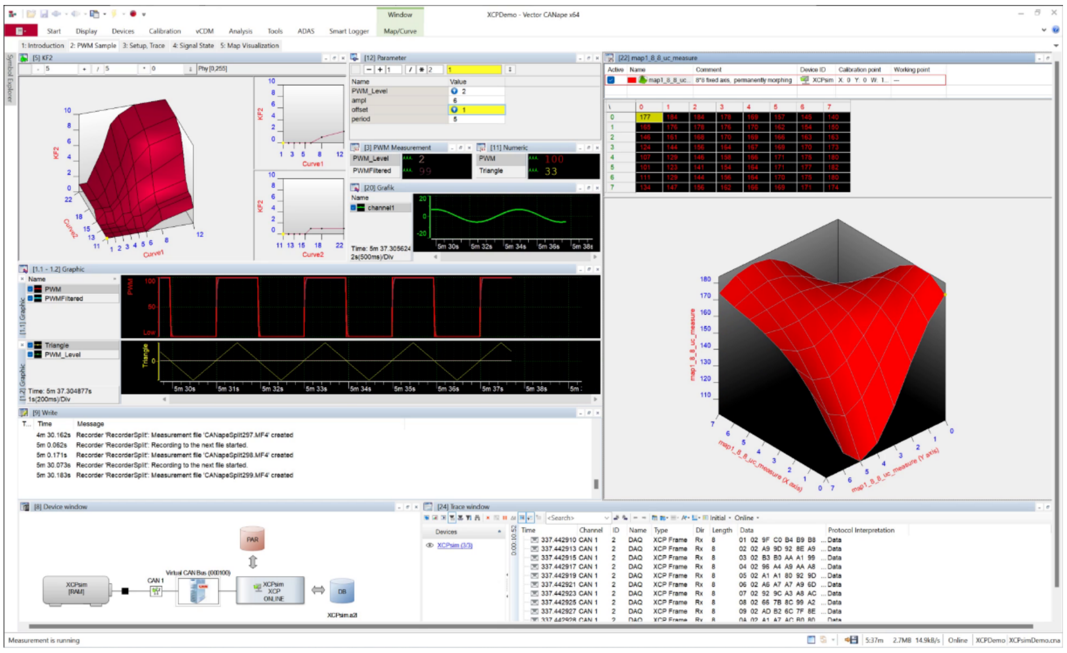


Figure 1.3.3 - Example of a LabVIEW VI with the generated interface

1.3.4. CANape

CANape is a tool designed by Vector Informatik for the measurement, runtime calibration, flashing, and logging of ECUs and ADAS sensors. It allows users to acquire various types of data and calibrate ECU parameters to adapt them to the vehicle.
CANape supports data analysis, logging, graphical visualisation, and automated report generation. It also enables symbolic access to data and functions via diagnostic protocol and supports calibration over XCP.
CANape uses its own scripting language, CASL, similar to the C programming language. The incredible versatility of this tool makes it a comprehensive solution for vehicle testing and development.



Name	Value
PWM_Level	2
ampl	6
offset	1
period	8

[22] map1_8_8_uc_measure

Active	Name	Comment	Device ID	Calibration point	Working point
✓	map1_8_8_uc...	8*8 fixed axis, permanently morphing	XCPsim	X: 0 Y: 0 W: 1...	---

\	0	1	2	3	4	5	6	7
0	177	184	184	178	169	157	145	140
1	165	176	178	176	170	162	154	150
2	146	161	168	170	169	166	163	163
3	124	144	156	164	167	169	170	173
4	107	129	146	158	166	171	176	180
5	101	123	141	154	164	171	177	182
6	111	129	144	156	164	170	175	180
7	134	147	156	162	166	169	171	174

[9] Write

```
T...  Time        Message
      4m 30.162s  Recorder 'RecorderSplit': Measurement file 'CANapeSplit297.MF4' created
      5m 0.062s   Recorder 'RecorderSplit': Recording to the next file started.
      5m 0.171s   Recorder 'RecorderSplit': Measurement file 'CANapeSplit298.MF4' created
      5m 30.073s  Recorder 'RecorderSplit': Recording to the next file started.
      5m 30.183s  Recorder 'RecorderSplit': Measurement file 'CANapeSplit299.MF4' created
```

[24] Trace window

```
Time              Channel  ID  Name  Type       Dir  Length  Data                        Protocol Interpretation
337.442910 CAN 1           2   DAQ   XCP Frame  Rx   8       01 02 9F C0 B4 B9 B8  ...Data
337.442913 CAN 1           2   DAQ   XCP Frame  Rx   8       02 02 A9 9D 92 8E A9  ...Data
337.442915 CAN 1           2   DAQ   XCP Frame  Rx   8       03 02 B3 B0 AA A1 99  ...Data
337.442917 CAN 1           2   DAQ   XCP Frame  Rx   8       04 02 96 A4 A9 AA A8  ...Data
337.442919 CAN 1           2   DAQ   XCP Frame  Rx   8       05 02 A1 A1 80 92 9D  ...Data
337.442921 CAN 1           2   DAQ   XCP Frame  Rx   8       06 02 A6 A7 A7 A9 6D  ...Data
337.442923 CAN 1           2   DAQ   XCP Frame  Rx   8       07 02 92 9C A3 A8 AC  ...Data
337.442925 CAN 1           2   DAQ   XCP Frame  Rx   8       08 02 66 7B 8C 99 A2  ...Data
337.442927 CAN 1           2   DAQ   XCP Frame  Rx   8       09 02 AD B2 6C 7F 8E  ...Data
```

Figure 1.3.4 - ECU calibration with CANape

1.4. Model Based Design

1.4.1. General Overview

Model-Based Design is a key development approach adopted in many engineering fields, including automotive, to shape and analyse complex systems.
It is based on performing simulations in a development environment to analyse the behaviour of the real physical system that will have to be built and controlled. The physical systems under examination are usually defined as a set of components, each of which can be represented by a model, interacting with each other, exchanging information, and performing certain tasks. Each component may span a wide range of disciplines, such as electrical, mechanical, thermal, hydraulic, pneumatic, optical, or any combination of these, ensuring the possibility to model very complex and differentiated systems. Depending on the accuracy level of the components' descriptions, the system can be more or less comparable to the original one. In the following picture, a schematic reconstruction of the realisation flow of a valid model is shown:

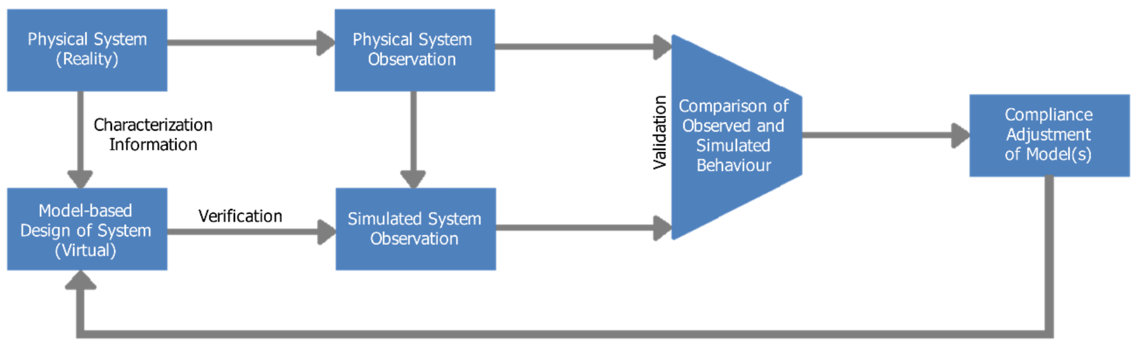

Physical System (Reality)
Physical System Observation
Characterization Information
Validation
Comparison of Observed and Simulated Behaviour
Compliance Adjustment of Model(s)
Model-based Design of System (Virtual)
Verification
Simulated System Observation

Figure 1.4.1 - Model Building Flow

The MBD focuses on abstracting from specific technologies through the use of high-level languages with a visual approach (e.g., through lines and blocks). Using a graphical tool can simplify the development of complex functions, especially in real-word systems, by breaking down the model into smaller modules that are easier to understand and implement.
These tools usually provide means of executing the model to perform testing; doing so before integration allows to reduce the risks of future issues that would result in greater costs and waste of time and resources.
Once the model behaves as intended, these tools may also generate the code with the defined settings (platform, language, and other specifications). This not only allows for higher productivity and portability, as only the coder settings need to be changed for

different platforms rather than the model itself, but also reduces the introduction of coding errors.
The before-mentioned Simulink is one of the most used tools of this kind.
To summarise, thanks to the advantages it introduces based on the separation of the application and the infrastructure (the concept of "model once, build everywhere"), Model-based Design has become more and more popular in the automotive fields.

1.4.2. MBD Flow V-Diagram

Mode-based Design follows a rigorous workflow composed by different steps, disposed in the so-called V-diagram. In the following picture, a generalization of the V-diagram applied to the automotive field:

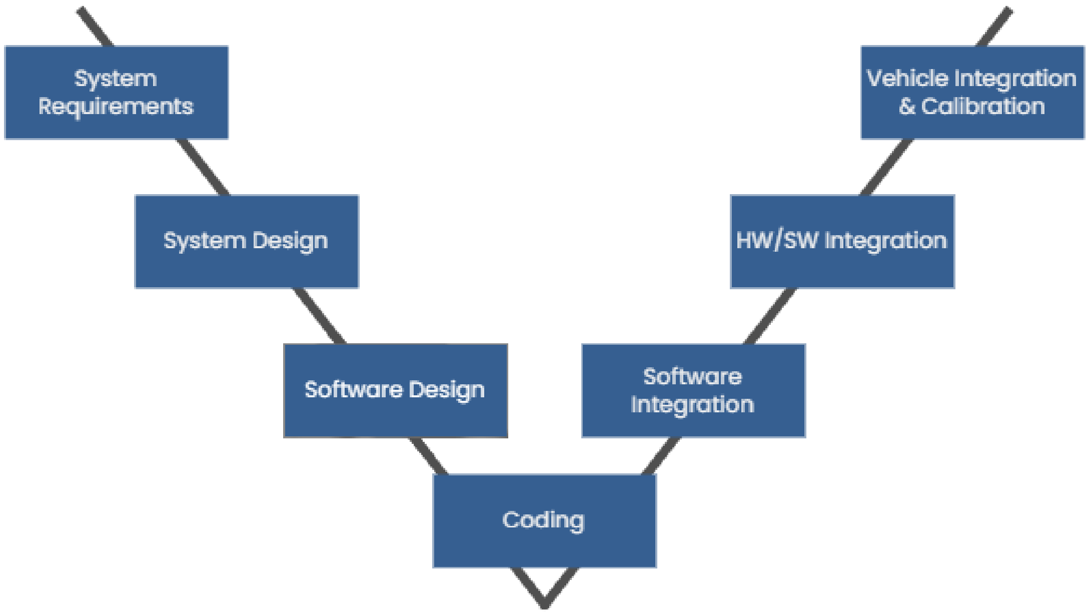


Figure 1.4.2 - V Diagram

1. System Requirements

The first step consists in analysing the system's requirements and using the results of this analysis to redact the System Requirements Document (SRD). This report will not only contain a comprehensive description of the analysed system but also a definition of all the necessary elements for the correct implementation and operation of the target system.
The document should present a well-defined hierarchical structure to favour understandability, starting from the general system requirements at a higher level and proceeding towards more detailed and restrictive 'child' requirements, each explaining in detail the expected behaviour and implementation of a module.

As SRD describes the hardware components, such as mechanical or electrical parts, and the functions they should perform, a Software Requirements Specifications document should be redacted in parallel. Every line of this document should include an identifier, a reference to the related system requirement, and a brief description. This structure makes it so that each system requirement is linked with one or more software requirements, facilitating the workflow. A system requirement will be considered satisfied once all its software requirements work properly. To ensure this, different test cases must be written and run.

2. System Design

The second phase of the V diagram goes deeper into the description of all the modules, components, and units that make up the system. Starting from the requirements document, the engineers analyse the feasibility of the requests, trying to find possible solutions and implementation strategies while performing additional estimations such as reliability and costs. During this step, it's still possible to introduce changes to the SRD before moving on to the next phases.

To ensure an optimal system design, one should follow some practices:

- Communication between the working teams must be present from the beginning, even in the preliminary phases. This will consent to arrive at the development stages with as clear as possible ideas.
- The system's design should be as scalable and modular as possible to reduce costs for future improvements, additions, and changes.
- A simple design is the key to success.
- Comprehensive and clear documentation is fundamental.

3. Software Design

This step involves modelling the system as a Platform-Independent Model (PIM) by means of an appropriate Domain-Specific Language (DSL), like Simulink, composed of blocks close to many domains, such as mechanical and electrical. When the design of the whole system is ready, it is possible to simulate it in order to refine it or find alternative designs. The possibility to conduct tests on the model, existing entirely inside the simulation tool, helps find bugs and issues in the earlier stages of development, thus reducing the costs that their correction and identification would require in later stages.

The iterative phase that includes this and the previous two steps of the V diagram is called Model-in-the-Loop testing (MIL).

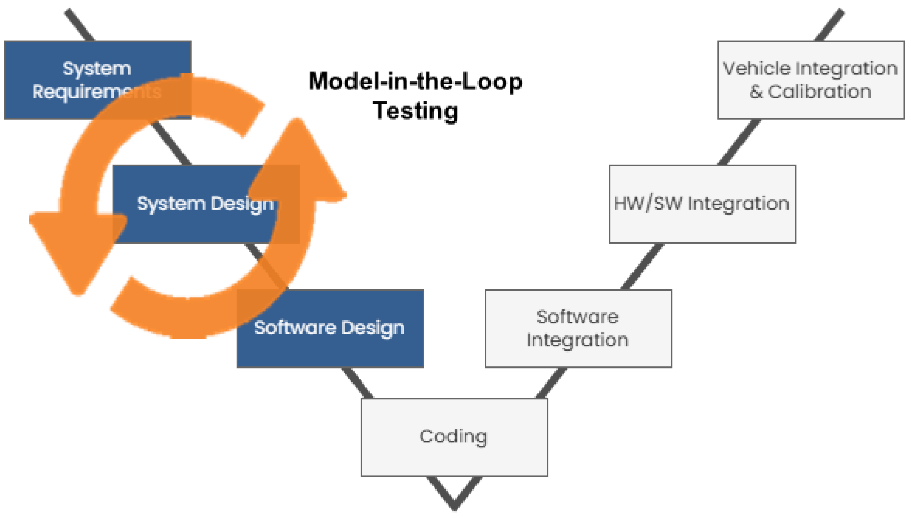


Figure 1.4.3 - MIL Testing

4. **Coding**

Once we have made sure that the system behaviour is the expected one, it's time for the generation of the code. This step will produce what will actually run on the target system; as such, one should try to optimise the generation parameters for the implementation on the desired HW.
There's a multitude of tools for automatic code generation, each with its own set of languages and customisable parameters, such as the Embedded Coder in Simulink.
Automatic code generation has the main advantage of erasing the need to update the code when the model changes, reducing not only costs and times but also the risk of manual coding errors.
The increasing complexity of modern systems has led to the widespread adoption of this type of coding approach, thanks to the before-mentioned advantages.

5. **Software Integration**

After the code has been generated, we need to confirm that it works as intended and that its behaviour and outcomes match those of the model-in-the-loop phase.
This verification process, consisting in running the generated code locally to confirm whether it is operating as intended, is called Software-in-the-Loop (SIL) and also covers the two previous phases.
If an erroneous behaviour emerges, it means that there was a mistake in either the model or the code generation, and they need to be checked and appropriately fixed.

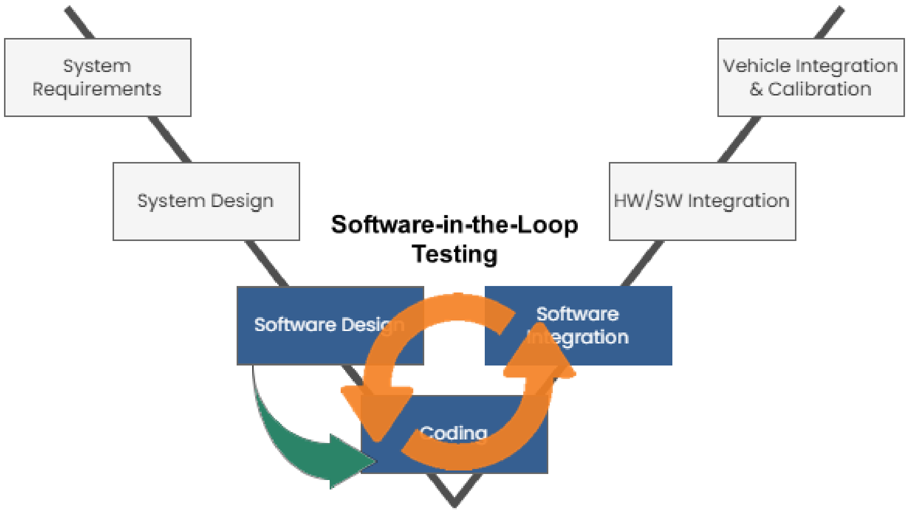


Figure 1.4.4 - SIL Testing

6. HW/SW Integration

After the software has been adequately verified, it is time to integrate the generated code into real embedded hardware, e.g., an ECU. The software is then deployed on the target hardware and co-simulated with the system model to verify its correctness.
Additionally, the outcome of this phase must match those of the MIL and SIL testing steps; if not, some adjustments must be made.
This step, with the "Software Integration" one, composes an iterative test phase called Processor-in-the-Loop (PIL). Whilst the PIL does not present a real-time testing situation, as only the controller is running on the real, embedded target hardware while the rest of the plant is being simulated, it is still of fundamental importance as it can help identify underlying mistakes before the costs of correcting them grow higher.

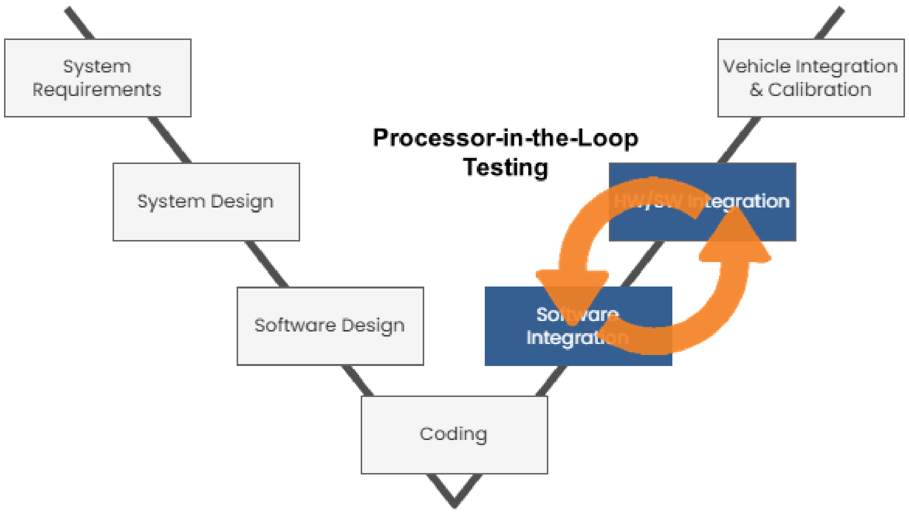


Figure 1.4.5 - PIL Testing

7. Vehicle Integration & Calibration

In this final step, the plant is simulated via a real-time simulator, to produce a behaviour as close to the real-world one as possible, e.g., in physical connectivity, I/O, and communication protocols. This is the last step before moving to the real system; after this test phase, the product can be released and tested in a real-world environment, which in the automotive world typically translates to performing vehicle fleet tests to ensure that the product meets the requirements.

Once this last verification session, often referred to as Hardware-in-the-Loop (HIL), has been completed, the design phase can be considered done and the production cycle can finally begin.

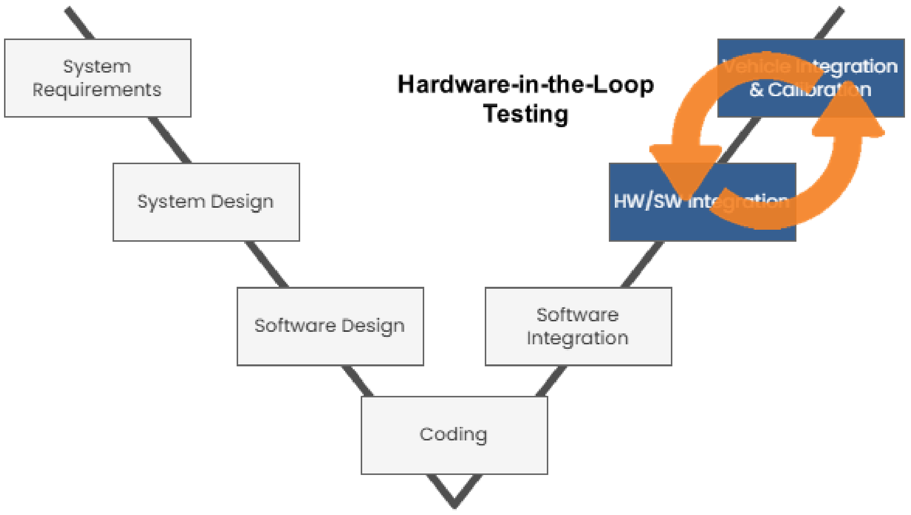


Figure 1.4.6 - HIL

2. Hardware Architecture

The ECU Heavy Duty System (HDS) is an engine control unit dedicated mainly to CNG/LNG-fueled engines with a maximum of eight cylinders to be used for commercial/industrial vehicles (Light-Duty and Heavy-Duty vehicles), and for stationary units using natural gas to generate electricity. The ECU has the capability to control the whole engine.

As an engine control unit, HDS9 is able to control multiple systems, such as the Fuel Injection system, the Ignition system, and the Variable Valve Timing system, in order to ensure the correct functioning of the internal combustion engine.
To help perform its duty, the ECU is equipped with different sensors.

The following schematic illustrates the key components of the ECU:

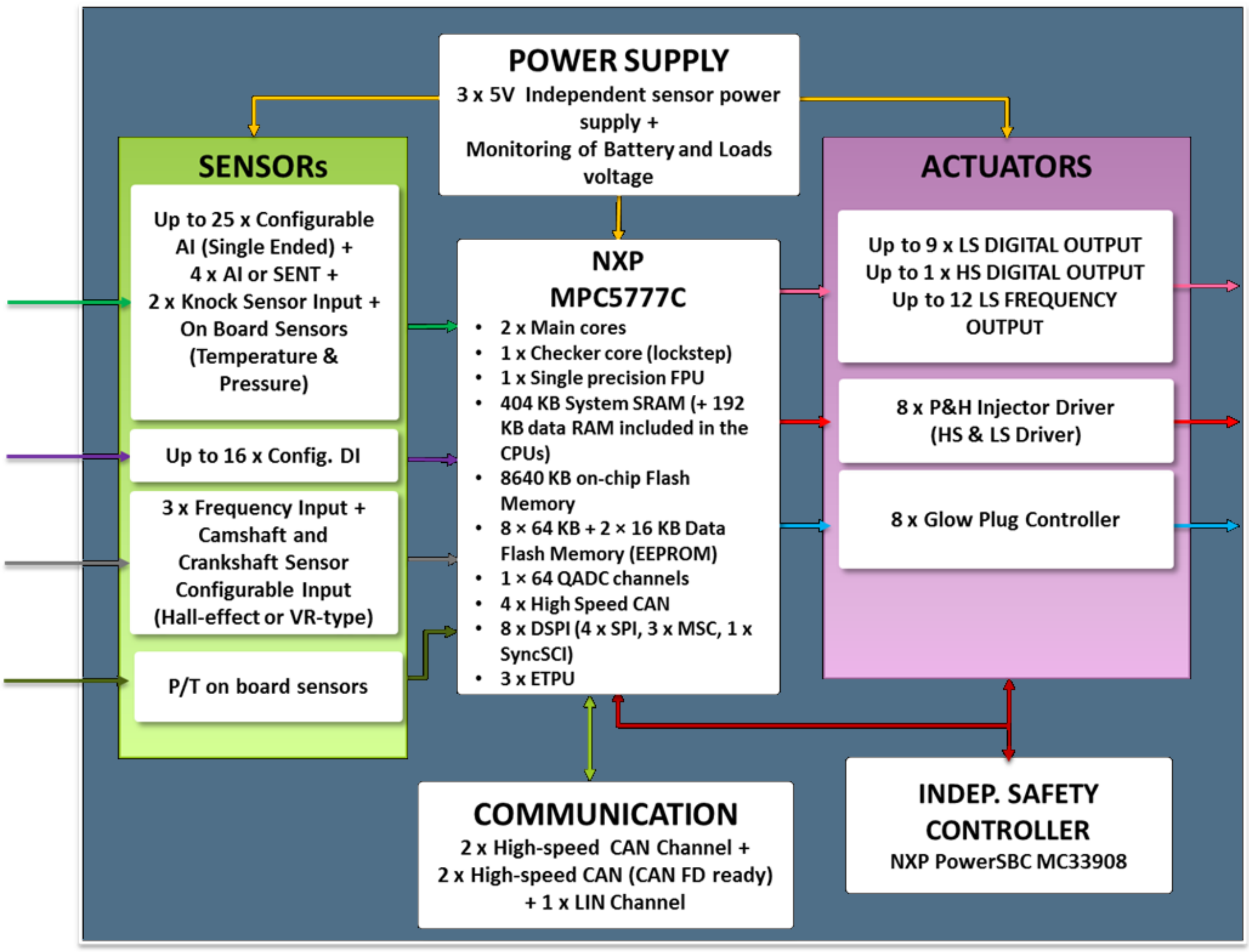

POWER SUPPLY
3 x 5V Independent sensor power supply +
Monitoring of Battery and Loads voltage
SENSORs
Up to 25 x Configurable AI (Single Ended) + 4 x AI or SENT + 2 x Knock Sensor Input + On Board Sensors (Temperature & Pressure)
Up to 16 x Config. DI
3 x Frequency Input + Camshaft and Crankshaft Sensor Configurable Input (Hall-effect or VR-type)
P/T on board sensors
NXP MPC5777C
2 x Main cores
1 x Checker core (lockstep)
1 x Single precision FPU
404 KB System SRAM (+ 192 KB data RAM included in the CPUs)
8640 KB on-chip Flash Memory
8 × 64 KB + 2 × 16 KB Data Flash Memory (EEPROM)
1 × 64 QADC channels
4 x High Speed CAN
8 x DSPI (4 x SPI, 3 x MSC, 1 x SyncSCI)
3 x ETPU
ACTUATORS
Up to 9 x LS DIGITAL OUTPUT
Up to 1 x HS DIGITAL OUTPUT
Up to 12 LS FREQUENCY OUTPUT
8 x P&H Injector Driver (HS & LS Driver)
8 x Glow Plug Controller
COMMUNICATION
2 x High-speed CAN Channel + 2 x High-speed CAN (CAN FD ready) + 1 x LIN Channel
INDEP. SAFETY CONTROLLER
NXP PowerSBC MC33908

Figure 2.1 - ECU block diagram

2.1. Inputs

The ECU presents both Analog and Digital input channels. There are 28 analog input conditioning circuits for external sensors, used for temperature, pressure, position, and HEGO/UEGO lambda sensors.

The HDS9 also mounts 15 digital input conditioning circuits for external switches. Each input channel can be configured, via software, with a pullup or pulldown resistor according to the switch connection to ground or to battery voltage. In addition, two specific inputs are dedicated to turning on the ECU when active: the key switch and the auxiliary key switch.

There are also frequency (PWM) inputs, including 5 Hall effect sensors.

The board also presents some internal sensors for monitoring the on-board temperature and pressure.

2.2. Outputs

The ECU presents both Digital and PWM/Frequency output channels, used to control the actuators connected to the ECU. To better adapt to the mounted actuators, both output types come in Low Side and High Side channels.

The digital channels are normally used as ON/OFF outputs and present two reserved outputs, specific to the starter command and the pump command.

Pulse-Width Modulation outputs are generally associated with proportional actuators or gauge indicators.

The board also includes Peak & Hold Injector drivers and Spark drivers for active ignition coils, capable of managing up to 8 cylinders.

In addition, the HDS9 includes two channels for H-bridge actuators.

2.3. Microcontroller

The HDS9 is equipped with an NXP microprocessor COBRA55 (MPC5777C).

The MPC5777C Power Architecture MCU is dedicated to industrial and automotive control applications requiring advanced performance, timing systems, security, and functional safety capabilities.

This microcontroller offers a high-performance multicore design and an industry standard eTPU-based timer system. It features a Flash solution allowing for code expansion, a security module, and packaging options, as well as the highest level of functional safety (ASIL-D) support.

Below, a short list of the microcontroller's main features:

- 2 x Main Power Architecture z7 cores + 1 x Checker core (lockstep) running the same set of operations in parallel to detect and correct possible errors.
- 1 x Single precision FPU
- 404 KB System SRAM (+ 192 KB data RAM included in the CPUs)
- 8 MB on-chip Flash Memory
- 8 × 64 KB + 2 × 16 KB Data Flash Memory (EEPROM) that can be used to implement memory recovery strategies
- 1 × 64 QADC channels
- 4 x High Speed CAN for communication
- 8 x DSPI (4 x SPI, 3 x MSC, 1 x SyncSCI)
- 3 x eTPU top perform complex timing and I/O operation management independently from the CPU

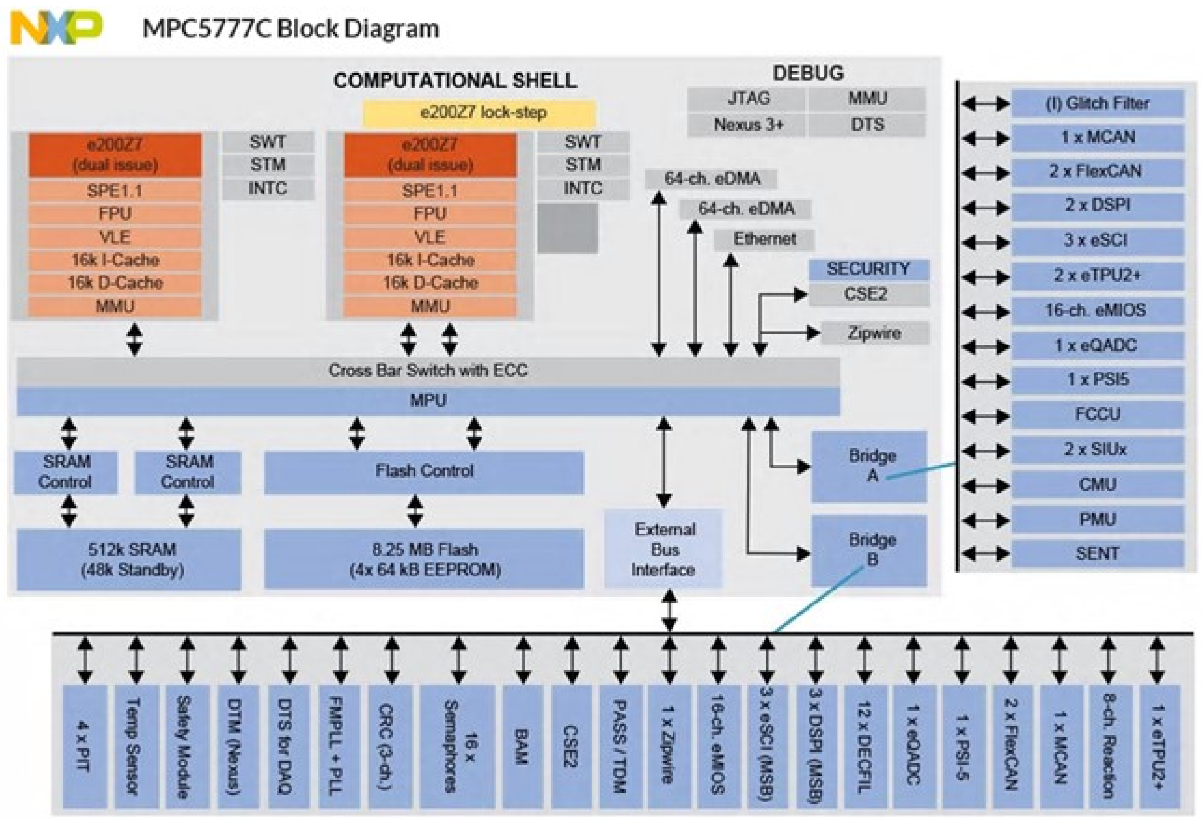


Figure 2.3.1 - Microcontroller schematics

2.4. Communication

The board uses four Controller Area Network (CAN) modules, one of which also supports CAN FD extension, for communication with other systems in the vehicles and external tools. Each of the available channels serves a different purpose.

The CAN 1 channel is used for communication between the ECU and the measurement/calibration system, allowing for the reading (measurement) and modification (calibration) of ECU signals and parameters. This communication is carried out using the XCP protocol to interface with the system's memory in R/W mode, following a master-slave paradigm, where the measurement system (e.g., CANape) assumes the master role and the ECU is the one responding to the address-oriented memory access requests. This access is, and the correspondences between symbols and addresses are defined in an A2L file. This channel can also operate with CAN FD.

The CAN 2 channel is set up to allow intravehicular communication, using the J1939 protocol. The J1939, designed by the Society of Automotive Engineers (SAE), is an open standard for the communication commonly used in heavy-duty vehicles to define the information exchange between Electronic Control Units. It operates on the CAN and provides standardisation, robustness, and scalability.

The CAN 3 channel is designed to be used for the vehicle diagnostic system. It uses the Unified Diagnostic Service (UDS) protocol to detect problems and reprogram the ECU. When a malfunction occurs, a new firmware can be flashed to resolve the issue.
The UDS operates with a client-server paradigm, with the tester issuing requests and the ECU responding as the server. By connecting a CAN bus to the OBD2 port, one can start a diagnostic session to ensure that the system is working as intended.

The CAN 4 channel, also called "private CAN", allows for the implementation of a private network between the Engine Control Module (ECM) and other engine-related devices.

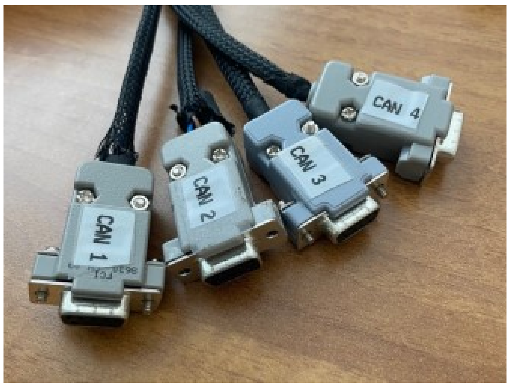


Figure 2.4.1 - HDS9 CAN Connectors

In addition, the HDS9 presents a LIN transceiver, based on a master-slave communication protocol rather than CAN's broadcast.

3. Software Architecture

HDS9's embedded software architecture follows the separation principles proposed by **AUTOSAR**; the structure is layered in order to standardise functional interfaces to the HW platform and at the same time define an architectural reference that could be extended to the various operating areas of the software while remaining easily accessible to the various technical figures operating on its implementation.
The modularity of this solution leads to greater portability across different HW platforms, as well as the possibility of independent development, testing, and update of every single module.

3.1. AUTOSAR Principles

AUTOSAR is a global partnership of leading firms in the automotive and software industry with the aim of developing and establishing *the* standardised software framework and open E/E system architecture for intelligent mobility.

The idea at the base of the AUTOSAR software framework is to improve complexity management for integrated E/E architectures by enabling the reuse and interchangeability of software modules between OEMs (Original Equipment Manufacturers) and suppliers.

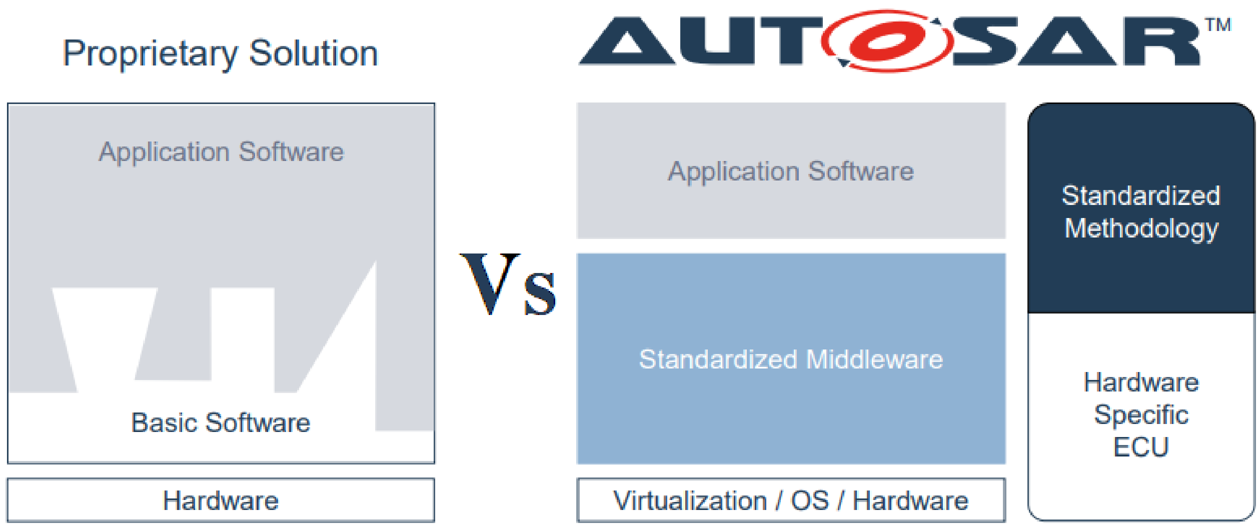


Figure 3.1.1 - Proprietary vs. AUTOSAR Middleware Approach

AUTOSAR offers a comprehensive environment for innovative electronic systems with a high focus on performance, safety, and security standards. It does so by adhering to a fundamental set of principles:

- Hardware and software should be widely independent from each other.
- The development should be distributed and done in parallel, thanks to the abstraction between horizontal levels, reducing development time and costs.
- The reuse of software is the basis for enhancing both quality and efficiency.

The layered architecture that allows for great results following these principles is generally designed to support hardware abstraction, scheduling of runnable tasks via the OS, communication between applications on the same hardware and over the network, and safety and security services in conjunction with diagnosis and diagnostic services.

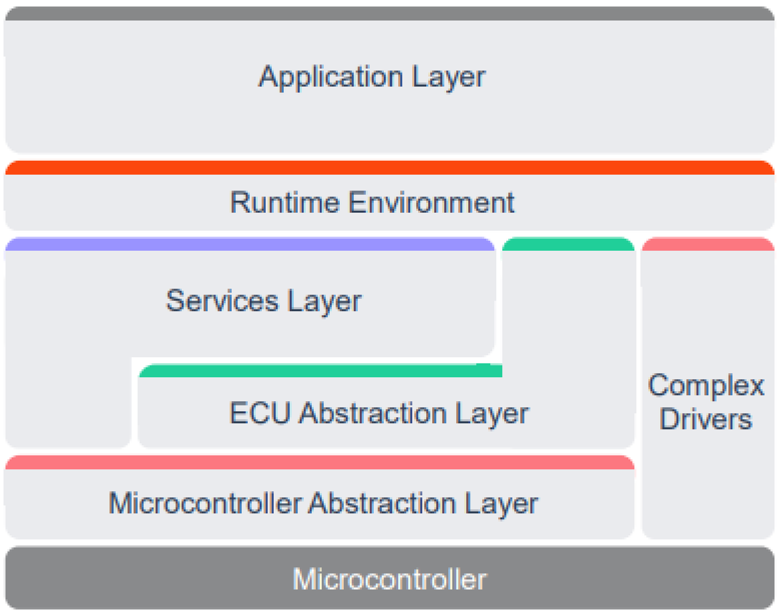


Figure 3.1.2 - AUTOSAR layered architecture

The image above depicts an example of a layered software architecture that serves as the basis for Metatron's actual structure, as described in the following paragraphs.

Even with all the advantages that AUTOSAR comes with, it still isn't free of defects:

- It introduces costs for the license royalties, as only "big players" provide MCAL and ECUAL, and for the required configuration tools (which need to be certified).
- It makes debugging harder and requires constant updates to keep up with the new releases, introducing additional maintenance costs.

- Reduces the competitive advantage: if everyone uses the same approach, cost is the same for all, so those who produce many ECUs and have more spending capacity are at an advantage over small to medium players.
- Low configurability, especially for what concerns the heavy-duty market; AUTOSAR drivers and services are generally designed for passenger systems, as the production volumes are far higher rather than for heavy-duty systems.
- Need to define custom and complex drivers to manage chips and application-specific sensors and actuators.
- The redundancy of functions and symbols in RTE/MCAL requires high-performance processors without a real need for them.

For all these reasons, Metatron decided not to directly integrate third-party AUTOSAR but rather implement its own structure, following the same principles and philosophy of abstraction levels.

3.2. Architecture Levels

The following image reports the main separation between the Basic Software, also known as Firmware, and the Application software. This distinction allows us to abstract the control strategy from the real HW solution and the actual implementation.

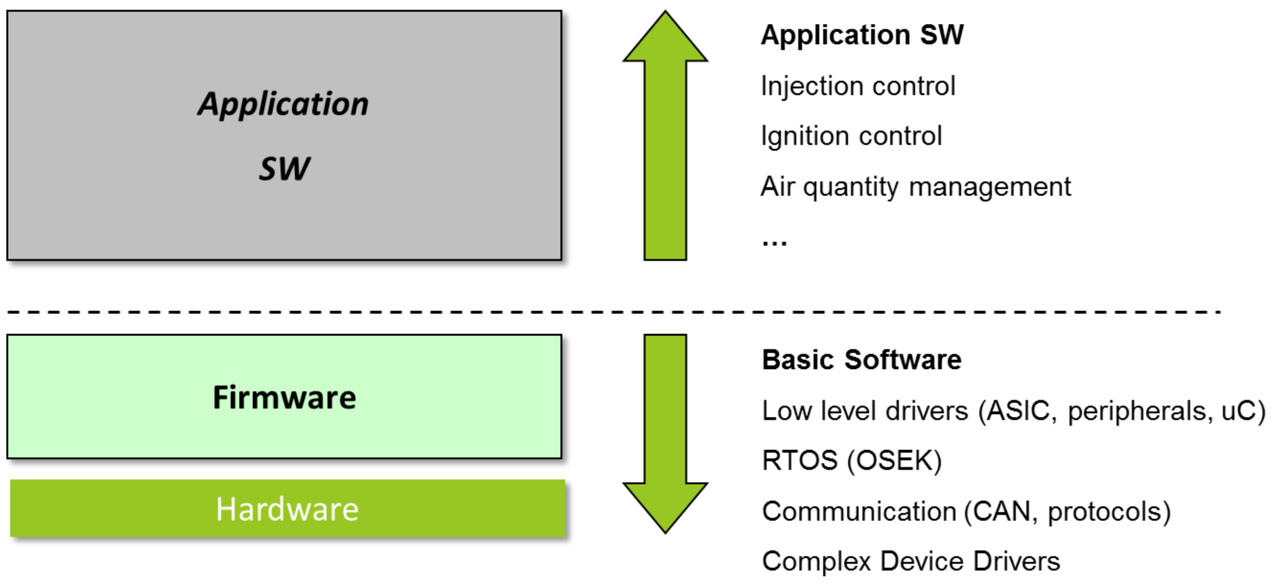


Figure 3.2.1 - HDS9 SW architecture main separation

To preserve the distinction between these two levels, an intermediary layer is added, resulting in the three-layer design shown in the following picture:

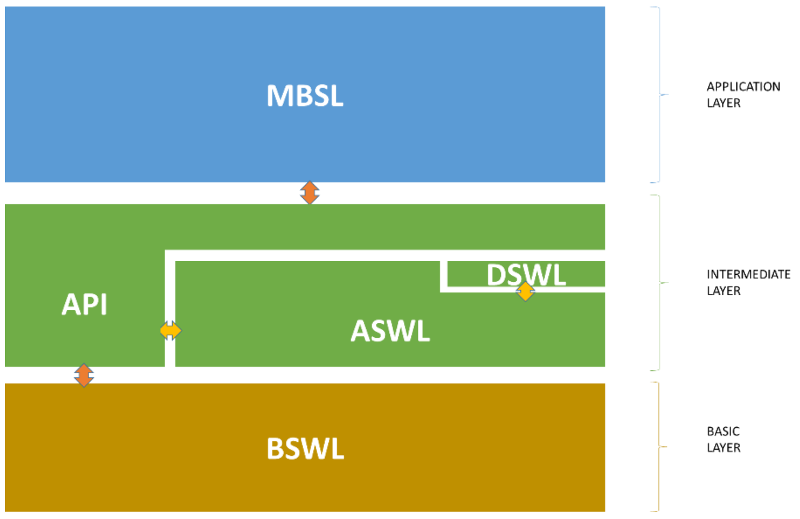


Figure 3.2.2 - HDS9 SW architecture layers

3.2.1. Application Layer (MBSL)

The highest-level layer implements the code specific to the automotive application. It is composed of a set of different software modules, called wrappers, programmed in a model-based design fashion, that can communicate with each other by exchanging data through means of global variables (called signals). These variables are accessible on entry to the module by including the producers' interfaces, i.e., the file handle.

If the module needs to access resources made available directly from the low level, it does so by calling appropriate functions provided by the underlying layer (API Interface). While the original paradigm was more akin to a classic get/set approach, in recent years Metatron has moved towards more generalised functions that allow for a more flexible management of variables, and specialised methods like the one used, in this thesis context, for memory access or diagnostic management.

In any case, the API interface is the only means of accessing the underlying functionalities. This way, it is sufficient for the application module to include the API interface file; moreover, having this single access point compels a complete abstraction of the application software with respect to the basic software. Finally, this strategy allows for the reuse of the same application software while adjusting the implementation of API methods without changing the prototype.

3.2.2. Intermediate Layer (API, ASWL, DSWL)

As previously stated, the goal of this layer is to introduce an abstraction level to better separate the platform-independent application software from the hardware-dependent base-level software.

It does so thanks to this layer's main component, the API Interface, which is the only access point for the application layer and whose methods can directly access the base-level software modules. In addition, this component also defines routines that the operating system calls for the various tasks required by the application, e.g., scheduling and I/O operations. This enables a single entry-point for the application software for each functionality, avoiding the need to interact with operating system modules.

Another important component of this layer is the Hand Coded Support Functions (ASWL), a group of modules that support several applicative functions, communicating directly with the basic software or with API Interface software, such as the "xcpmgr" module that manages the XCP services or the "J1939" module that implements that specific protocol functionalities interacting with the board communication drivers.

This layer is the core of the HDS9 software's modularity and abstraction, with all the advantages that this approach comes with.

3.2.3. Basic Software (BSWL)

The basic software level aims to provide interfaces to the HW, hiding the details related to the physical location of each signal and its implementation while still allowing an easy association between interfaces and the relative electrical signals, as expected at the connector of the control unit itself. Moreover, it allows for high configurability of the devices used to implement such features.

This layer, as the intermediate one, is composed of different sub-layers, each of them with a different purpose:

- The Microcontroller Abstraction Layer (MCAL) is the lowest-level one and the most dependent on the MCU in use. It's a key component, as it contains the actual drivers needed to access the peripherals.
- The ECU Abstraction Layer, just above the MCAL, implements the abstraction of the MCAL for the upper layers, providing all the required APIs for the external and

internal drivers, so that the upper layers of the ECU are independent of the effective HW.

- The Service Layer, "mounted" on top of the ECU abstraction layer, provides basic services for the applications, such as ECU state management, memory and communication services, and Operating System functionality. The Operating System provides a task switching mechanism that is fully preemptive based on the priority scheme, including an idle mechanism (background task), which is active when no other system or application functionality is active. No extended tasks are managed. Services for critical region protection or resources are available.

All three sub-layers interface with the Complex Driver sub-layer, usually dedicated to the implementation of peculiar functionalities involving both the microcontroller and external devices on the board.

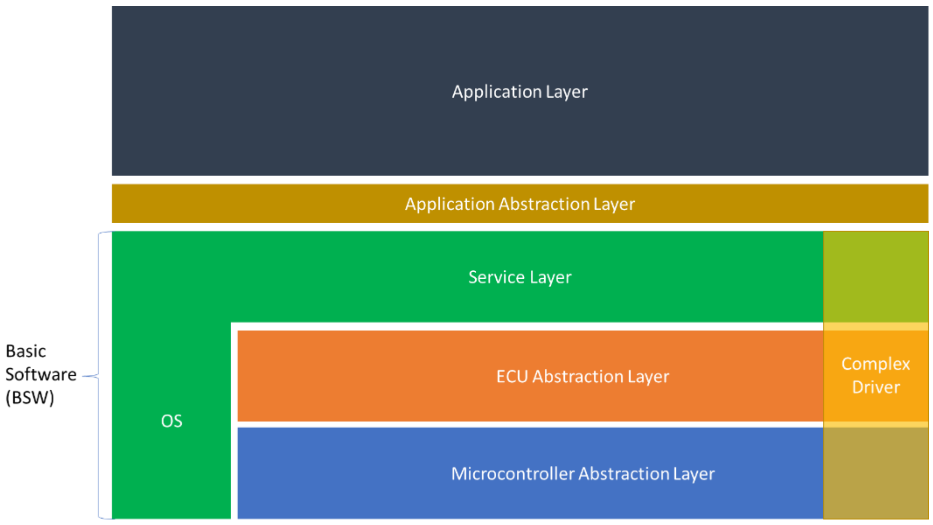


Figure 3.2.3 - HDS9 BSWL internal modules and layers subdivision

3.3. API Design

As previously stated, the API level makes communication possible between the Application layer and the BSWL, providing users with access to the defined C functions. These functions are defined in the C file "api.c" and the related header file "api.h" and are then imported into Metatron's Simulink library in the form of function blocks that can be used for the model-based design of the application.

API methods are classified according to their functional group in either:

- Get/set functions, operating on a single variable and following a getter/setter paradigm.

- Specialised functions, operating on several variables.

In recent years, Metatron moved from the getter/setter approach with functions specific to a single variable towards a strategy focused on more generalised methods that act as getter/setter for a certain category of variables and take the target variable as a parameter.

This has been done to further improve abstraction and increase code maintainability and readability. Moreover, this generalisation helps minimise the number of API blocks imported in Simulink, which can be incredibly useful for creating a clearer workflow for the customers that will interface with the library.

In this regard, it is important to cite the work done by L. Zannella, documented in the thesis titled "Library definition for an automotive ECU API layer (using Model-Based approach)", which introduced a set of changes and improvements for what concerns the design and development process of the APIs and whose work served as the foundation for this thesis.

4. Part 1 - Memory Management (NVRAM)

The first step in this thesis work, after studying the system documentation, was the definition and implementation of a memory management strategy for the HDS9 non-volatile memory, here called EEPROM (*Electrically Erasable Programmable Read Only Memory*), focused on increasing the reliability of the system while also defining a more general-purpose structure to better suit any customer's request.

This operation served to form a better understanding of the system and the implementation flow, as well as improved memory management, which would be needed for the second part of the thesis.

4.1. Strategy

The first thing to do was to identify a strategy that covered all the given requirements and, starting from it, define a flow of actions that would then have to be translated into code and function calls.

The starting requirements were:

- Manage the memory at startup and shutdown, also taking into account sudden shutdowns.
- Identify possible errors and restore the most recent backup when needed.
- Implement the stored data structure to be as versatile as possible, to fit any data a customer could store.
- Enable the customers to access the memory for R/W operations.
- Avoid overcomplicated strategies (a.k.a., follow the KISS principle).

The chosen strategy is based on employing two of the memory modules available on the device; the basic idea is to alternatively select one module or the other at startup and then store the values on the other module at shutdown. On the next activation of the system, the last written module will be selected to read the stored values and load them into the volatile memory.

With this approach, we ensure that a module will always contain the previous backup of the memory state, and we will be able to retrieve that data in case of issues with the latest loaded module. In addition, we had to cover the possibility, however remote, of both memory modules malfunctioning at the same time.

To detect whether a module's backup was corrupted or not, we needed a consistency check function. This would have to be computed on the data stored in the memory module at the shutdown and then stored with them inside the module.

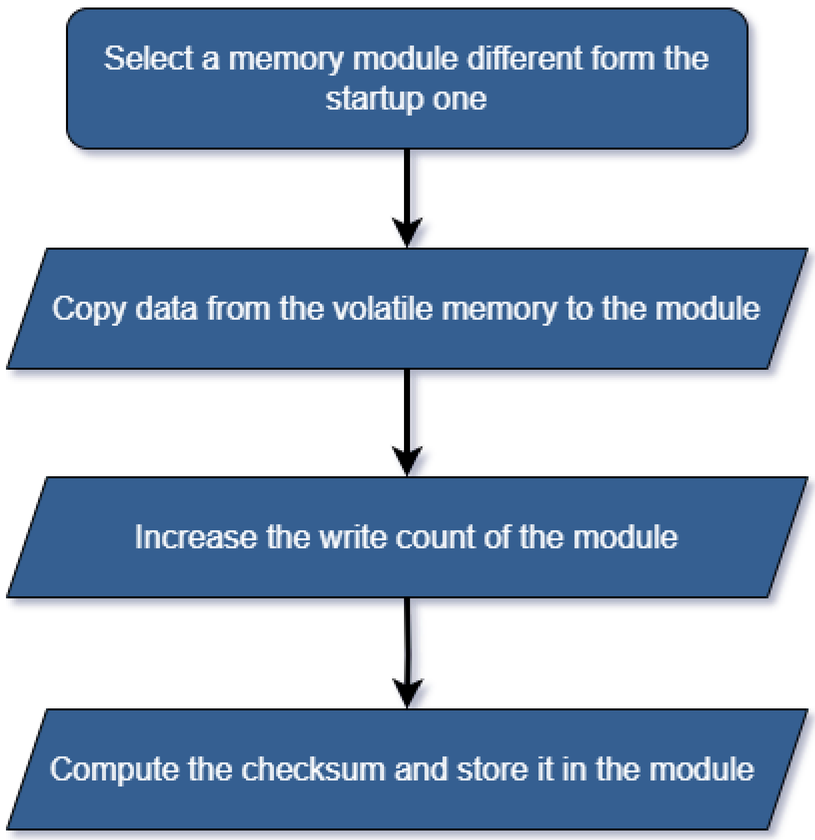

Select a memory module different form the startup one
Copy data from the volatile memory to the module
Increase the write count of the module
Compute the checksum and store it in the module

Figure 4.1.1 – NVRAM shutdown strategy

At startup, the system would recompute the consistency check on the data stored in the selected module. In addition to the user-inserted data, the module would also contain the number of times the memory has been rewritten. This value can be used to determine the module to read from.

In case of a mismatch between the newly computed value and the one stored at the previous shutdown, the system would try to select the other module containing the last backup, again performing this check. In the unfortunate case in which the second module was also corrupted, a default set of values would replace the memory module contents.

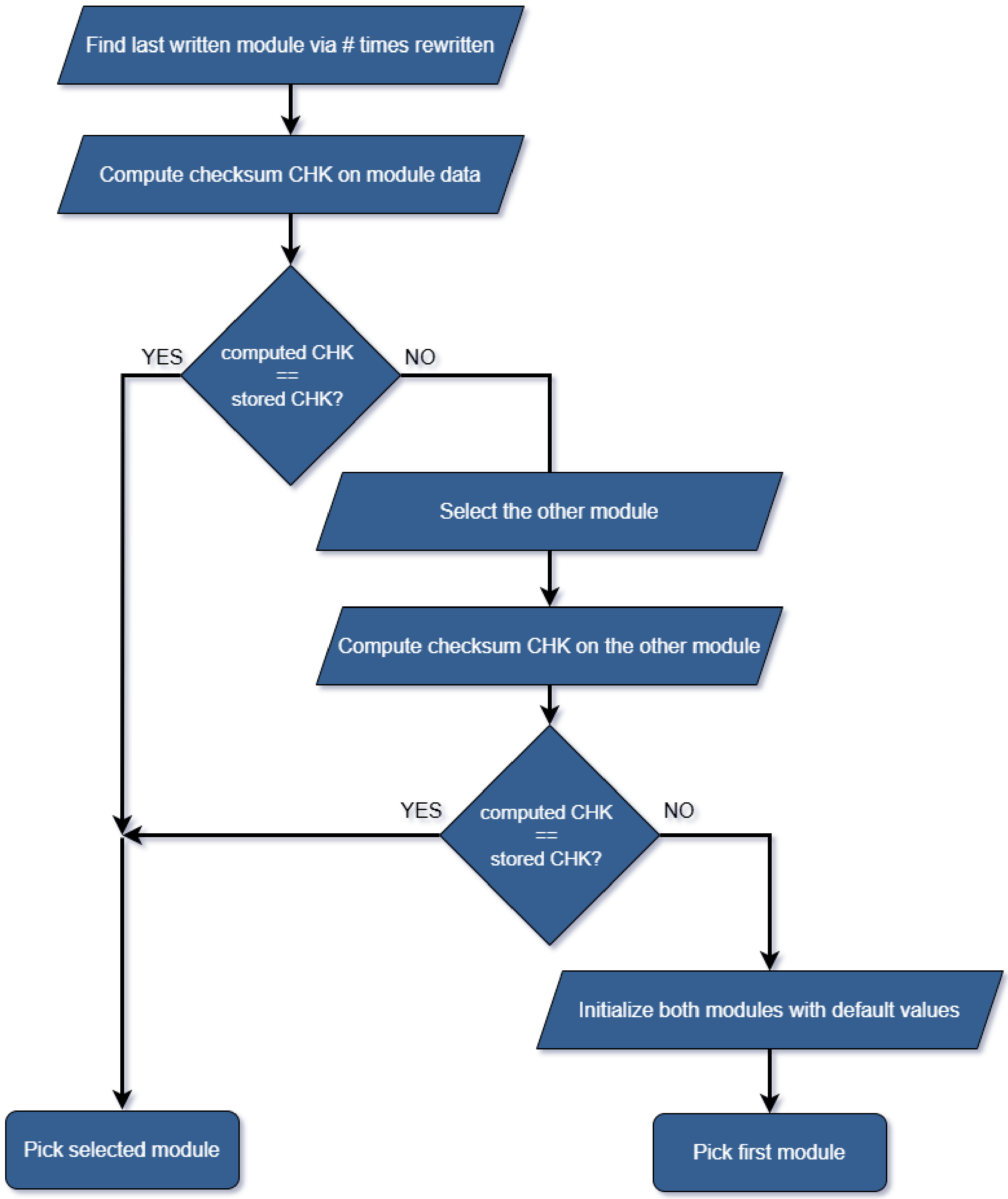

Find last written module via # times rewritten
Compute checksum CHK on module data
computed CHK == stored CHK?
YES
NO
Select the other module
Compute checksum CHK on the other module
computed CHK == stored CHK?
YES
NO
Initialize both modules with default values
Pick selected module
Pick first module

Figure 4.1.2 – NVRAM startup strategy

In terms of the format of the data that the users would be able to store into the NVRAM, our focus went again to looking for the simplest and most flexible strategy. In the end, we decided to opt for a simple array-like structure, whose fields would be of the most general type that the board could support (that is, the largest memory-wise) and whose length could be regulated by the user with ease.

While this solution allows for more adaptable and efficient storage management, it does have some drawbacks; in particular, adopting this approach would require users to convert the types they want to store in order for them to fit correctly and be retrieved later.

At the same time, as the users would need to correctly manage the form of the inserted data, they could perform optimisations (such as storing multiple Boolean values as an array of bits in a single entry) that could greatly improve the usage of the memory, thus making it less of an issue and more of an occasion.

The final point that our strategy tried to cover was the optimisation of the access operations; as per the previous requirements, we looked for a solution that could be both efficient and generic while keeping it as simple as possible. Starting from the designed memory structure, we decided to go with direct access R/W operations. This solution introduced a compromise between efficiency and user-demanded tasks, as they would have to know the exact position (index) of the value to change/retrieve in the memory structure. As you will see in the implementation paragraphs, however, it is possible to introduce some workarounds to reduce the burden on the user.

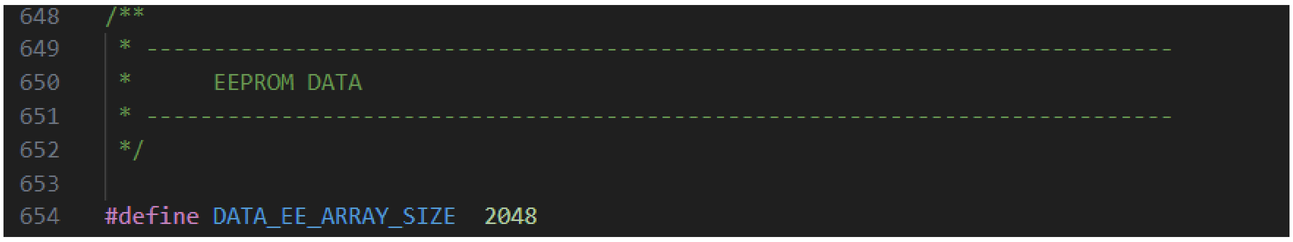
Basically, as we decided to follow an approach focused on simplicity and flexibility, we had to move part of the management into the hands of the users, with the aim of improving the platform flexibility.

4.2. Implementation

In this section, we'll describe how the strategies illustrated in the previous paragraphs have been implemented in the system via C coding. All the structures, variables, and functions present in the following paragraphs are located in the "api.h" and "api.c" files previously mentioned in the "Software Architecture" chapter of this thesis.

4.2.1. Memory Structure

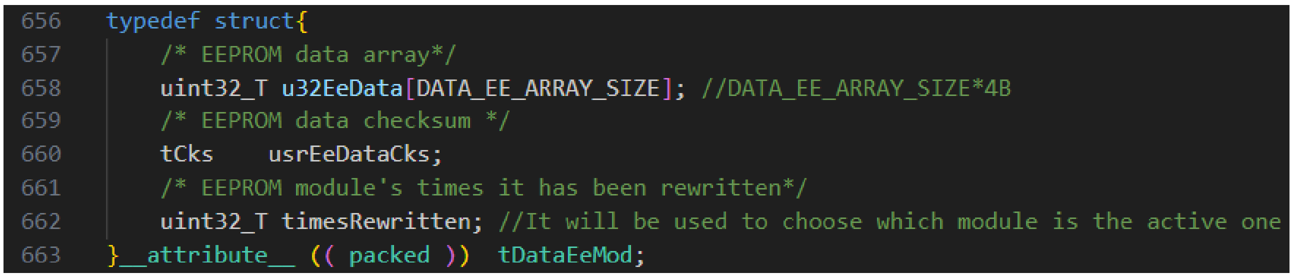
As previously stated, we decided to store the user-defined data inside a simple array-like structure. The size of this array can be defined by the user by changing the value of the #define **DATA_EE_ARRAY_SIZE** (here set by default to 2048). The operations involving this array use the defined value, thus allowing for a single-point change to adapt the whole code.



```
/**
 * ----------------------------------------------------------------------
 *      EEPROM DATA
 * ----------------------------------------------------------------------
 */

#define DATA_EE_ARRAY_SIZE  2048
```

Figure 4.2.1 - NVRAM array size #define

Simply using an array wouldn't provide us with the information required by the strategies underlined in the previous paragraphs. As such, a wrapper structure, **tDataEeMod**, has been defined in order to neatly pack the user's data array together with both the checksum value and the counter of the number of times the module has been rewritten. Note that this wrapper is completely transparent to the users.



```
typedef struct{
    /* EEPROM data array*/
    uint32_T u32EeData[DATA_EE_ARRAY_SIZE]; //DATA_EE_ARRAY_SIZE*4B
    /* EEPROM data checksum */
    tCks    usrEeDataCks;
    /* EEPROM module's times it has been rewritten*/
    uint32_T timesRewritten; //It will be used to choose which module is the active one
}__attribute__ (( packed ))  tDataEeMod;
```

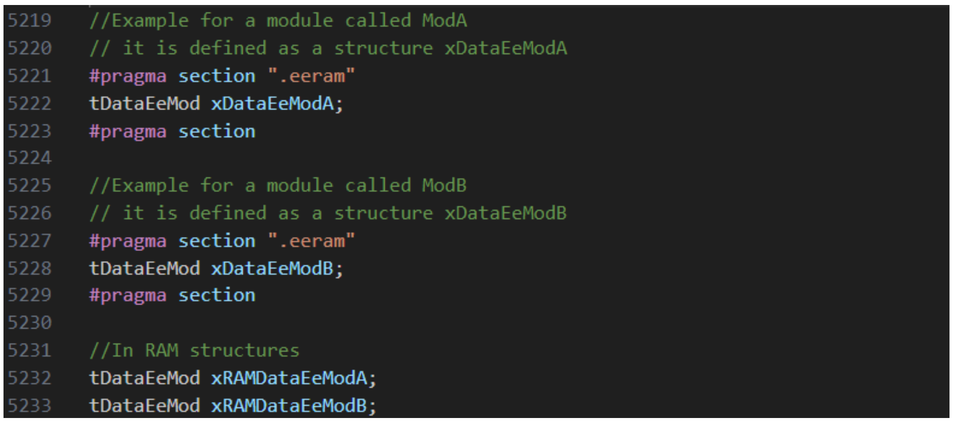
Figure 4.2.2 - NVRAM wrapper structure

As seen in the picture above, the selected type for the user-defined data array, the variable **u32EeData**, is a 32-bit unsigned integer. As explained in the "strategy" section, this allows us to store the highest variety of values, gifting us the outmost flexibility without impact on system performance, as the microcontroller registers are native 32-bit.

The variable used to store the computed checksum, **usrEeDataCks**, is of the type *tCks*, defined in the base-level software precisely for this kind of value.

Lastly, as we estimated that a value of 16 bits might not suffice to maintain the counter of the writings performed on the modules for all the possible applications of the system, we chose to define the **timesRewritten** field of the struct as a 32-bit unsigned integer.

Each memory module presents an associated tDataEeMod structure, as shown in the following picture:



```
//Example for a module called ModA
// it is defined as a structure xDataEeModA
#pragma section ".eeram"
tDataEeMod xDataEeModA;
#pragma section

//Example for a module called ModB
// it is defined as a structure xDataEeModB
#pragma section ".eeram"
tDataEeMod xDataEeModB;
#pragma section

//In RAM structures
tDataEeMod xRAMDataEeModA;
tDataEeMod xRAMDataEeModB;
```

Figure 4.2.3 - Modules' structures declaration

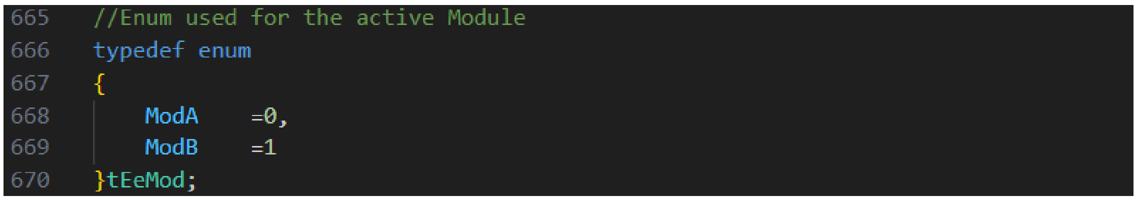
The two modules' structures are declared inside of a '#pragma section ".eeram"' directive. This compiler-specific C construct is used to instruct the compiler to place certain code or data into a specific memory section. In this case, the data contained inside the "**.eeram**" sections of the memory will be the one to be physically stored inside the NVRAM modules at shutdown via some base-level software methods.

The actual read/write operations are performed on temporary structures stored in the system's RAM rather than on those in the "*.eeram*" sections, for multiple reasons, from speed to the need to reduce the number of writings on those sectors. The declaration of those in-RAM structures can be seen in the previous image.

4.2.2. Read & Write Operations

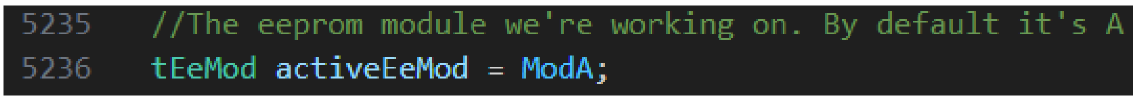
Although multiple modules are present, there can only be a single active module at any time. The operations of retrieving and storing data done by the user can only be performed on this active module.

The active module identifier is stored inside the **activeEeMod** variable, of type **tEeMod**. The tEeMod is an enumerative type containing the identifiers of the available modules (here **ModA** and **ModB**). By default, the active module is ModA.



```
//Enum used for the active Module
typedef enum
{
    ModA    =0,
    ModB    =1
}tEeMod;
```

Figure 4.2.4 - NVRAM modules enumerative tEeMod



```
//The eeprom module we're working on. By default it's A
tEeMod activeEeMod = ModA;
```

Figure 4.2.5 - Declaration of activeEeMod with default value

The two operations available to users via API are the get and set of data in a given position. Both APIs require the index in the memory array of the value to read or write, passed as the 16-bit unsigned integer parameter named **u16slotID**. We opted to leave the association between value and index to the users in order to provide higher efficiency via direct indexed access. The index parameter is stored on 16 bits, as it has been found to be the best trade-off between the amount of data usually required by the users and the number of available memory entries in the modules.

Both the APIs return an 8-bit unsigned integer value to report a possible error code, although at the moment only two values (success and generic error, respectively, 0 and 1) are available.

The read API is **API_EEPROM_getData,** which also takes as a parameter the pointer to a uint32_T variable, **u32eData**, where to store the value read at the given index, as the return value of this function is used to indicate the presence of errors.

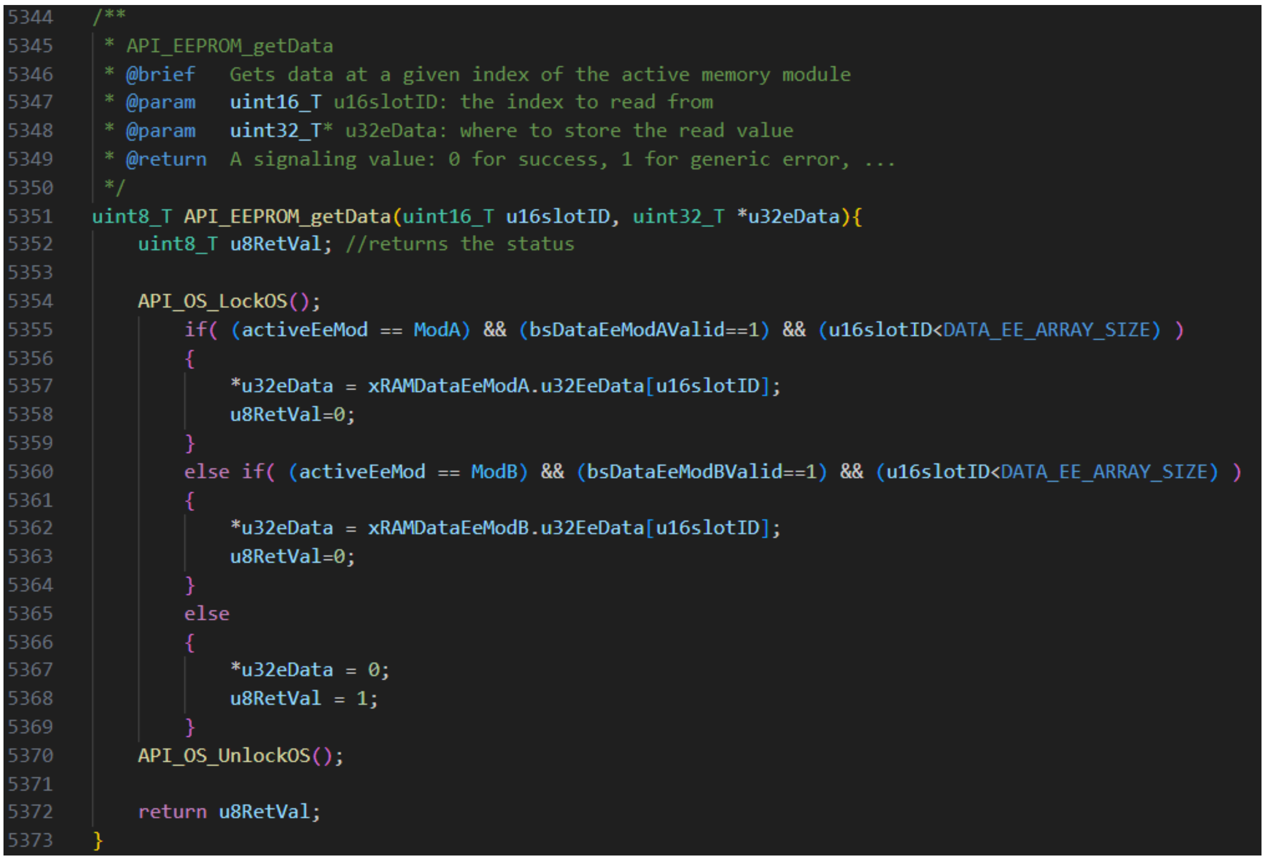
The function starts by calling the Operating System API **API_OS_LockOS** before entering the critical section, in order to avoid race conditions. This BSWL ensures that only one task will be operating inside our critical region.

The function then proceeds by checking three conditions: whether the active module is ModA, that the data inside the module is not corrupted (bsDataEeModAValid data validity flag), and that the passed index does not exceed the memory array size defined as DATA_EE_ARRAY_SIZE. In the event that all the conditions are satisfied, the value of the ModA struct's u32EeData field at the given index is stored via the pointer passed as a parameter, and the return value (variable u8RetVal) is set to 0.

In the event that one or more of those conditions fails, the function proceeds to check the ModB using the corresponding data validity flag and the same check on the index. In case of success, the value at the chosen index in the ModB structure's array is written to the given pointer.

In the eventuality that the checks on both ModA and ModB fail, the return value is set to 1 to signal a generic error, and the value pointed by u32eData is set to zero.

Before returning the value to the caller, the function releases the lock on the critical section, calling the **API_OS_UnlockOS** method provided by the operating system.



```
/**
 * API_EEPROM_getData
 * @brief   Gets data at a given index of the active memory module
 * @param   uint16_T u16slotID: the index to read from
 * @param   uint32_T* u32eData: where to store the read value
 * @return  A signaling value: 0 for success, 1 for generic error, ...
 */
uint8_T API_EEPROM_getData(uint16_T u16slotID, uint32_T *u32eData){
    uint8_T u8RetVal; //returns the status

    API_OS_LockOS();
        if( (activeEeMod == ModA) && (bsDataEeModAValid==1) && (u16slotID<DATA_EE_ARRAY_SIZE) )
        {
            *u32eData = xRAMDataEeModA.u32EeData[u16slotID];
            u8RetVal=0;
        }
        else if( (activeEeMod == ModB) && (bsDataEeModBValid==1) && (u16slotID<DATA_EE_ARRAY_SIZE) )
        {
            *u32eData = xRAMDataEeModB.u32EeData[u16slotID];
            u8RetVal=0;
        }
        else
        {
            *u32eData = 0;
            u8RetVal = 1;
        }
    API_OS_UnlockOS();

    return u8RetVal;
}
```

Figure 4.2.6 - API_EEPROM_getData

The write API, **API_EEPROM_setData**, takes as parameters both the index and the 32-bit unsigned integer value, **u32eData**, to be written at the given position.

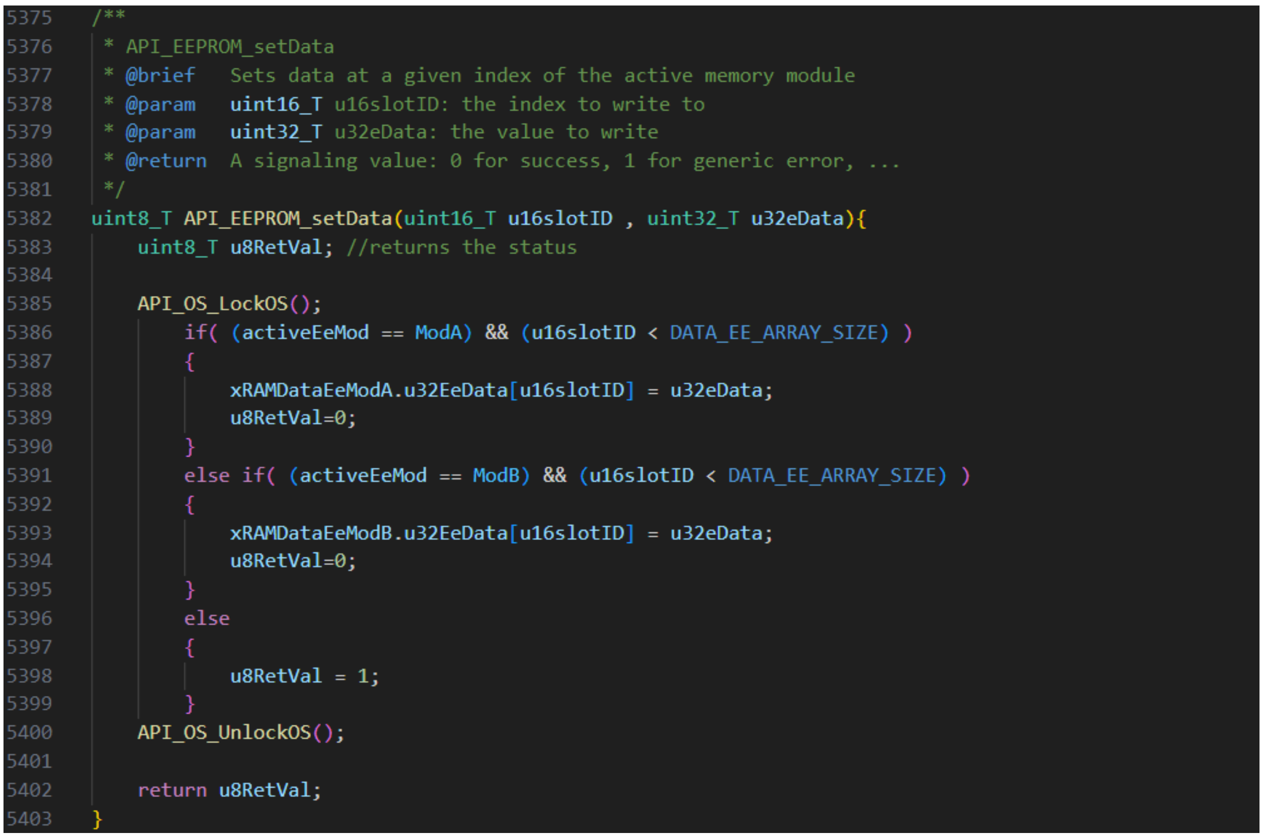
As in the getter function, this setter first locks the system, calling the OS-provided *API_OS_LockOS* to prevent races, and then starts performing the checks on the active module and the validity of the passed index.

It is important to note that the control over the validity of the data contained in the module, via the *bsDataEeModAValid* and *bsDataEeModBValid* flags, is not performed in this case. This was a deliberate choice, as that check is used to avoid trying to read corrupted data. For how this system was implemented, new data can be written using this set function, but they cannot be read until a subsequent shutdown-startup sequence is performed, in which the data have been correctly stored and read.

In the case of successful checks, the passed data is stored in the active module's user-data array at the given position. The return value is then set to 0 to mark the absence of errors.

As in the previous function, in the case of failed checks, the return value is set to 1 to indicate a general error.

The function then releases the lock via *API_OS_UnlockOs* and returns the error-signalling value.



```c
/**
 * API_EEPROM_setData
 * @brief   Sets data at a given index of the active memory module
 * @param   uint16_T u16slotID: the index to write to
 * @param   uint32_T u32eData: the value to write
 * @return  A signaling value: 0 for success, 1 for generic error, ...
 */
uint8_T API_EEPROM_setData(uint16_T u16slotID , uint32_T u32eData){
    uint8_T u8RetVal; //returns the status

    API_OS_LockOS();
        if( (activeEeMod == ModA) && (u16slotID < DATA_EE_ARRAY_SIZE) )
        {
            xRAMDataEeModA.u32EeData[u16slotID] = u32eData;
            u8RetVal=0;
        }
        else if( (activeEeMod == ModB) && (u16slotID < DATA_EE_ARRAY_SIZE) )
        {
            xRAMDataEeModB.u32EeData[u16slotID] = u32eData;
            u8RetVal=0;
        }
        else
        {
            u8RetVal = 1;
        }
    API_OS_UnlockOS();

    return u8RetVal;
}
```

Figure 4.2.7 - API_EEPROM_setData

For the aim of this Thesis, the code has been designed to work for only two modules. In the case of the expansion of the system to work with a greater number of memory modules, however, it would be possible to adapt the code shown by following some simple steps:

- Add the identifier of the new modules to the tEeMod enumerative.

- Define the corresponding tDataEeMod wrapper structures in a *#pragma session ". eeram".*
- Modify the *setData* and *getData* to perform the check on the passed index at the beginning of the function, allowing to introduce both a more specific error code and reduce the critical section protected by the OS API.
- Change the if/else constructs in *setData* and *getData* with a switch construct on the active module value.

4.2.3. Startup

To implement the defined startup strategy, it has been necessary to define a well-structured flow of calls starting from pre-existing functions and routines while adding new methods and functionalities. A simplified summary of the final flow can be seen in the picture below:

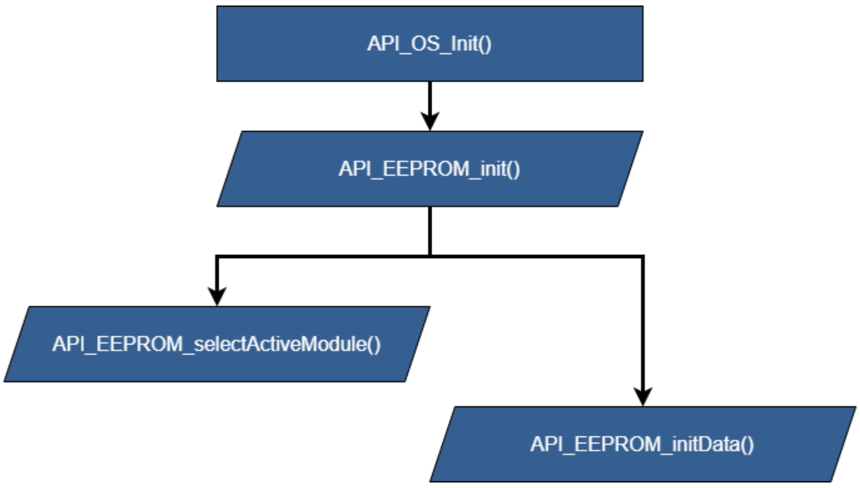
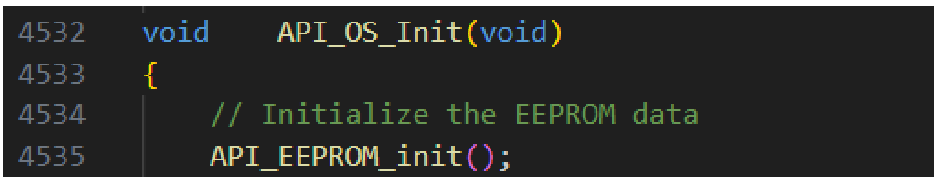


Figure 4.2.8 - NVRAM startup function calls flow

The first function called, **API_OS_Init**, is the initialisation routine of the operating system at API level.

The *API_OS_Init* starts by calling the **API_EEPROM_init**.



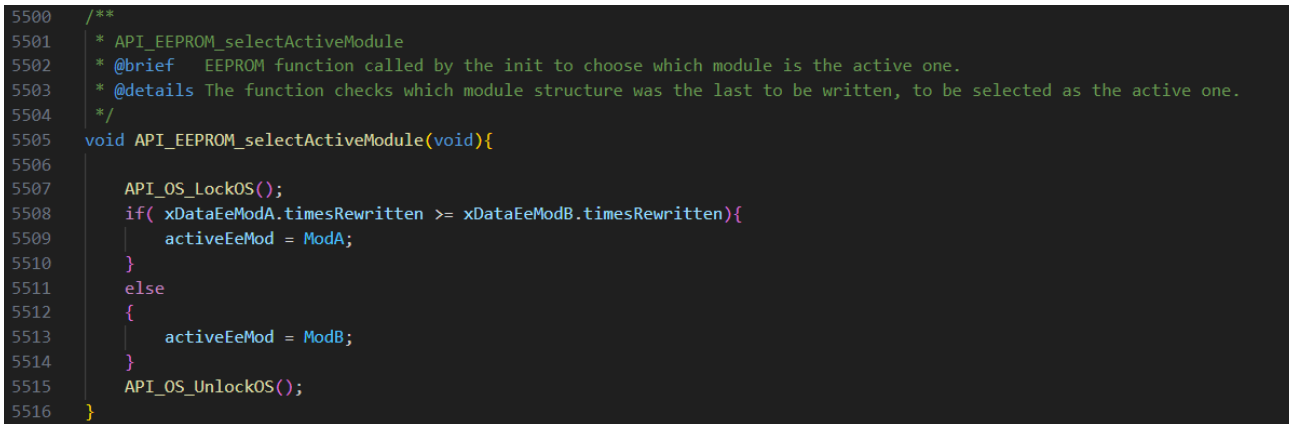
```
void    API_OS_Init(void)
{
    // Initialize the EEPROM data
    API_EEPROM_init();
```

Figure 4.2.9 - API_OS_Init calling the next function

The *API_EEPROM_init* function checks the state of the EDATA component after power-on, detecting abnormal power-offs and indicating the need to reset EEPROM data to its initial settings. The body of the function can be split into three parts, each performing different duties:

- Via some base software functions, the code sets the anomalous power-off flags and initialises the power-off counter, used by the "save" procedure.
- The system tries to read the status of the NVRAM via the BSWL function **NVRAM_Read**. In case of error (*E_NOT_OK*) or mismatch of the boot key, the system tries to rewrite the EEPROM with the **NVRAM_Write**.
- The function calls the utility for selecting the active module, **API_EEPROM_selectActiveModule()**, and then proceeds to call the function for initialising the data from the EEPROM, **API_EEPROM_initData**().

The **API_EEPROM_selectActiveModule** is a utility function for determining which of the available modules is to be selected as the active one. It is a pretty straightforward comparison between the modules' *timesRewritten* fields, although it does perform a lock to avoid possible races on those fields.



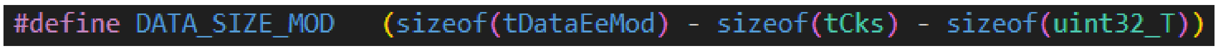
```c
/**
 * API_EEPROM_selectActiveModule
 * @brief   EEPROM function called by the init to choose which module is the active one.
 * @details The function checks which module structure was the last to be written, to be selected as the active one.
 */
void API_EEPROM_selectActiveModule(void){

    API_OS_LockOS();
    if( xDataEeModA.timesRewritten >= xDataEeModB.timesRewritten){
        activeEeMod = ModA;
    }
    else
    {
        activeEeMod = ModB;
    }
    API_OS_UnlockOS();
}
```

Figure 4.2.10 - API_EEPROM_selectActiveModule

The final procedure called by the *API_EEPROM_init* is **API_EEPROM_initData**, whose job is to validate the content of the read module, switch to the other module(s) in case of issues, and reinitialise the defined structure with the default value, if necessary.

Depending on the active module selected via the utility function, *API_EEPROM_selectActiveModule*, the code sets the module's corresponding data validity flag and then checks that the module status is valid (an additional check to those in *API_EEPROM_init*) and that the stored checksum equals the one computed at this moment.

The current checksum is calculated via BSWL function, which also takes as input the **DATA_SIZE_MOD**, defined in “api.h”. In case of an invalid state or mismatch of the checksum, the Boolean **tryOtherMod** is set to true.



```
#define DATA_SIZE_MOD   (sizeof(tDataEeMod) - sizeof(tCks) - sizeof(uint32_T))
```

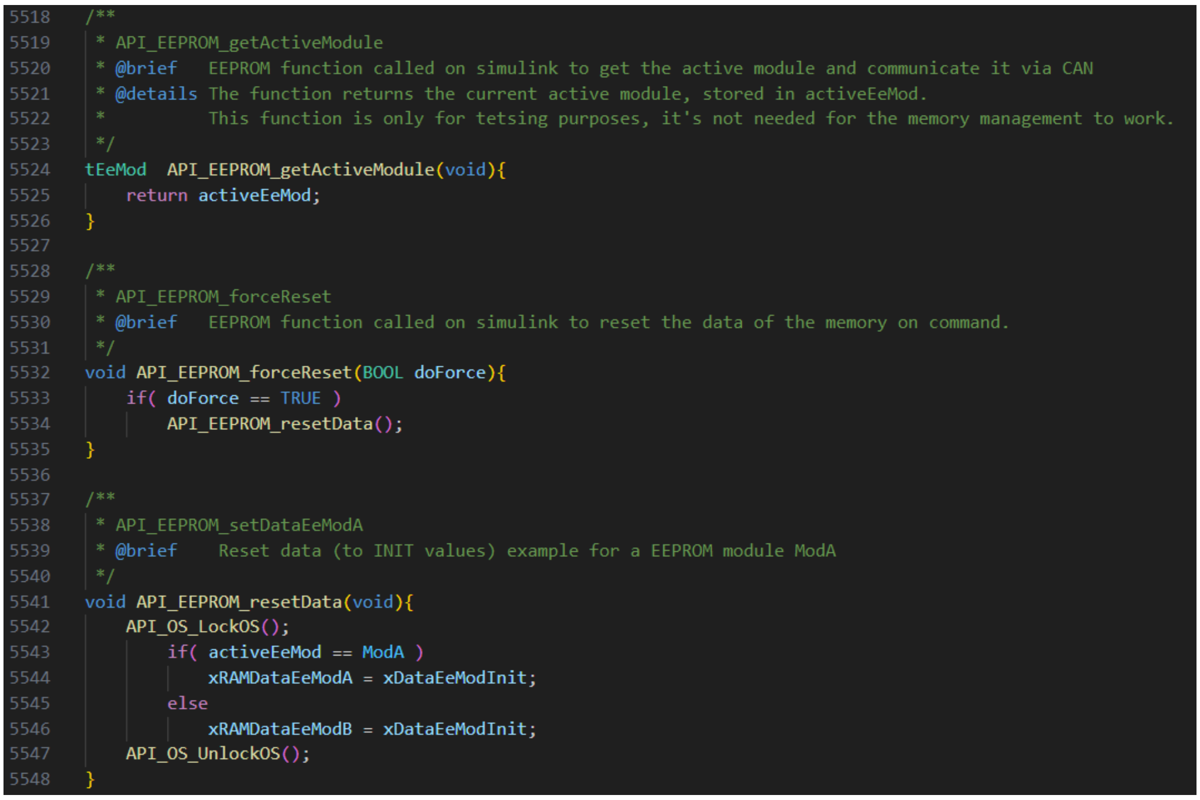
Figure 4.2.11 - DATA_SIZE_MOD

If the *tryOtherMod* flag is set to false, the procedure simply terminates, as it means that the data inside the memory module structure is valid and ready to operate.

Otherwise, we check that the other module (e.g., ModB if ModA was the active one) is operating and its data are valid, performing the same checks done on the active module. In the event of the validity of the data contained in the other module’s structure, we simply copy that backup into the active module’s in-RAM structure.

If the remaining module’s data fails the checksum comparison, all the structures are initialised with default values contained inside the **xDataEeModInit** constant, and the data validity flags for both modules are set to 0, in order to signal to read operations the presence of an error (as seen in the previous paragraphs).

These functions have been designed to be completely transparent to the user; the average customer should not be worried with how the system performs the setup of the memory and the various checks and should only interact with the NVRAM system through the *setData* and *getData* APIs.
For testing purposes, however, two more functions directly accessible at model-based software level have been developed, respectively to verify the selected module and to force the reset of the data stored in the NVRAM: **API_EEPROM_getActiveModule** and **API_EEPROM_forceReset**.



```c
/**
 * API_EEPROM_getActiveModule
 * @brief   EEPROM function called on simulink to get the active module and communicate it via CAN
 * @details The function returns the current active module, stored in activeEeMod.
 *          This function is only for tetsing purposes, it's not needed for the memory management to work.
 */
tEeMod  API_EEPROM_getActiveModule(void){
    return activeEeMod;
}

/**
 * API_EEPROM_forceReset
 * @brief   EEPROM function called on simulink to reset the data of the memory on command.
 */
void API_EEPROM_forceReset(BOOL doForce){
    if( doForce == TRUE )
        API_EEPROM_resetData();
}

/**
 * API_EEPROM_setDataEeModA
 * @brief    Reset data (to INIT values) example for a EEPROM module ModA
 */
void API_EEPROM_resetData(void){
    API_OS_LockOS();
        if( activeEeMod == ModA )
            xRAMDataEeModA = xDataEeModInit;
        else
            xRAMDataEeModB = xDataEeModInit;
    API_OS_UnlockOS();
}
```

Figure 4.2.12 - Additional functions for testing purposes

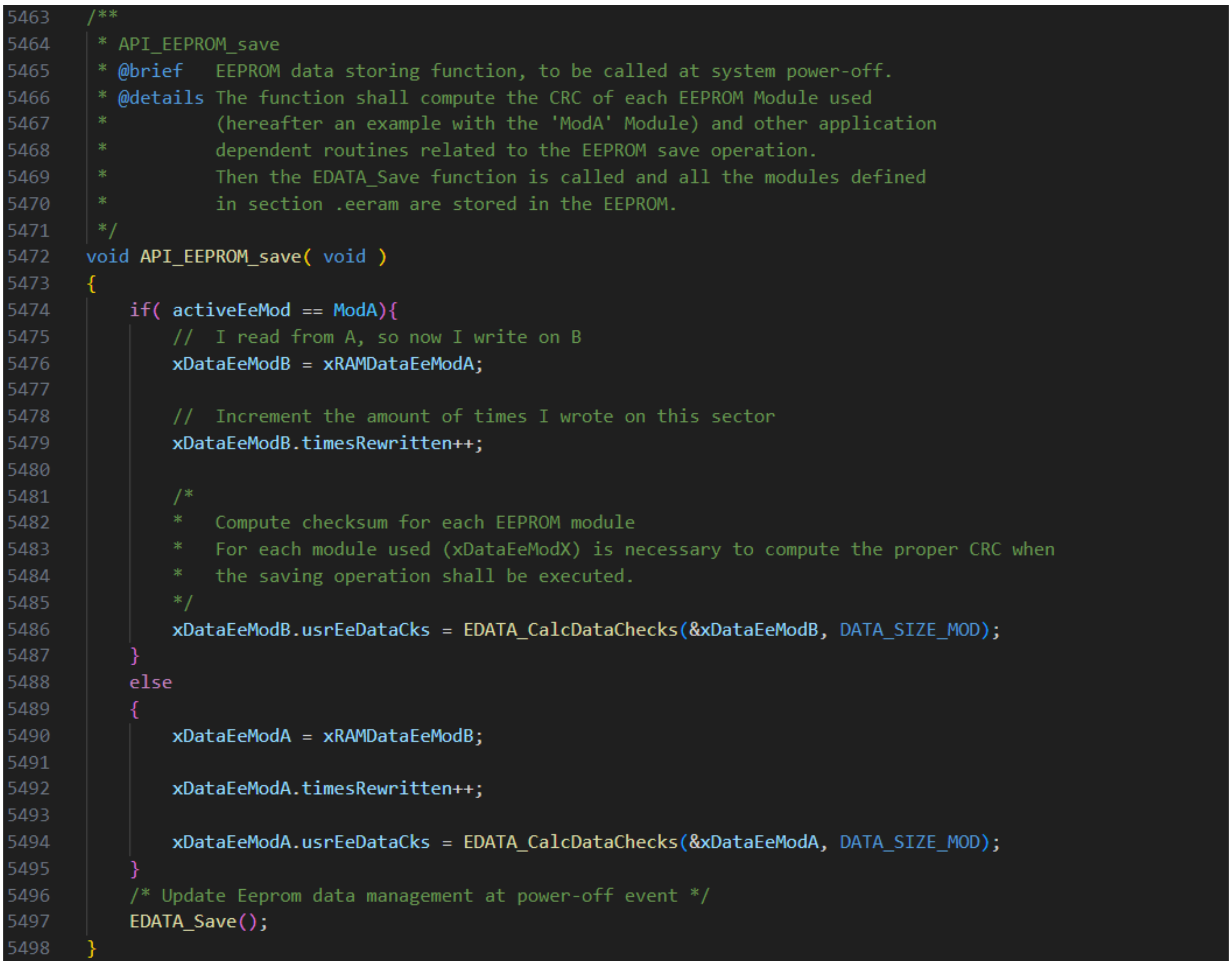
4.2.4. Shutdown

To implement the shutdown strategy, it was required to introduce a method that would be called during the system's shutdown, that is, when the Key-Off procedure is called by the OS. This procedure is in charge of performing the preparation for the power-off routines, among which **API_EEPROM_save** is the one that actually implements the shutdown strategy seen in the "Strategy" section.

The *API_EEPROM_save*, similarly to the previous methods, performs its operations based on the active module. First, it copies the current data from the active module's in-RAM structure into the other module's "*.eeram*" data structure. This way, the current active module data will be used as a backup.

The function then proceeds to increase the counter of the number of times that the other module has been rewritten and goes on to compute the checksum and store it inside the module's structure.

Once the data has been copied from the RAM into the "*.eeram*" section structure, and the fields have been updated, the system invokes the base software *EDATA_Save* method, which will then be the one to actually write the values in the ".eeram" sections into the physical modules.



```c
/**
 * API_EEPROM_save
 * @brief   EEPROM data storing function, to be called at system power-off.
 * @details The function shall compute the CRC of each EEPROM Module used
 *          (hereafter an example with the 'ModA' Module) and other application
 *          dependent routines related to the EEPROM save operation.
 *          Then the EDATA_Save function is called and all the modules defined
 *          in section .eeram are stored in the EEPROM.
 */
void API_EEPROM_save( void )
{
    if( activeEeMod == ModA){
        //  I read from A, so now I write on B
        xDataEeModB = xRAMDataEeModA;

        //  Increment the amount of times I wrote on this sector
        xDataEeModB.timesRewritten++;

        /*
        *   Compute checksum for each EEPROM module
        *   For each module used (xDataEeModX) is necessary to compute the proper CRC when
        *   the saving operation shall be executed.
        */
        xDataEeModB.usrEeDataCks = EDATA_CalcDataChecks(&xDataEeModB, DATA_SIZE_MOD);
    }
    else
    {
        xDataEeModA = xRAMDataEeModB;

        xDataEeModA.timesRewritten++;

        xDataEeModA.usrEeDataCks = EDATA_CalcDataChecks(&xDataEeModA, DATA_SIZE_MOD);
    }
    /* Update Eeprom data management at power-off event */
    EDATA_Save();
}
```

Figure 4.2.13 - API_EEPROM_save

4.3. Tests setup

After the implementation was deemed complete, it was time to run stress tests on the board mounting the new code. To do so, we decided to make use of LabVIEW for its capability to manage multiple tools that could be useful to our cause, as well as the possibility to graphically build programs that would allow us to automate this kind of testing.

In the following picture, a simplified scheme of the setup used for the tests is shown:

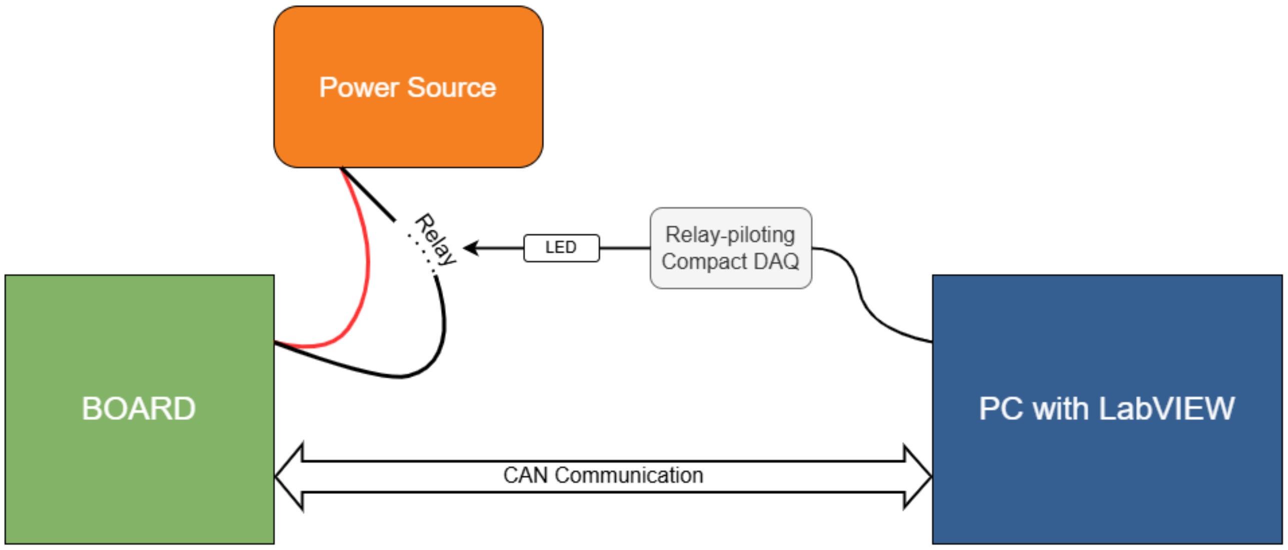


Figure 4.3.1 - Test setup schematics

The components appearing in the schematic are:

- **Power Source**: the board's power supply.
- **Board**: model HDS9 v2, mounting the NVRAM-management code in addition to the model-based generated code specific to the application.
  It presents a digital relay used to simulate the key-on/key-off command via a physical switch. For this test, the switch has been disconnected to be able to pilot the relay using the digital output of a CompactDAQ.
  It communicated via CAN with the PC on the CAN port #2.
- **PC**: a laptop running Windows 10 as the operating system, mounting LabVIEW to execute the tests' Virtual Instruments (VI).
  It also run PCAN-View by PEAK System, a software used to read and send CAN messages with ease, providing multiple representation options (decimal, hex, etc.) and other useful functionalities.
  It was connected to the board via a PEAK Dongle, a PCAN-USB adapter used for CAN communication, and to the DAQ.

- **CompactDAQ with Digital I/O**: a well-renowned data-acquisition tool, used for its portability and flexibility, produced by National Instruments.
  The module used presented various digital I/O pins, one of which was piloted by the LabVIEW VI created for the specific test.
  It is used to pilot the 'key' relay of the board to study the system's behaviour for high numbers of key-on/key-off cycles.
- **LED**: used to give immediate visual feedback of the operating state of the system. Piloted by the same output pin of the DAQ that piloted the relay.

In the following picture, it is possible to identify the power source (on the background, with a black and a red banana plugs), the board (on the left), and the open relay (on the right) with the yellow cable used to pilot it via the DAQ (not in picture).

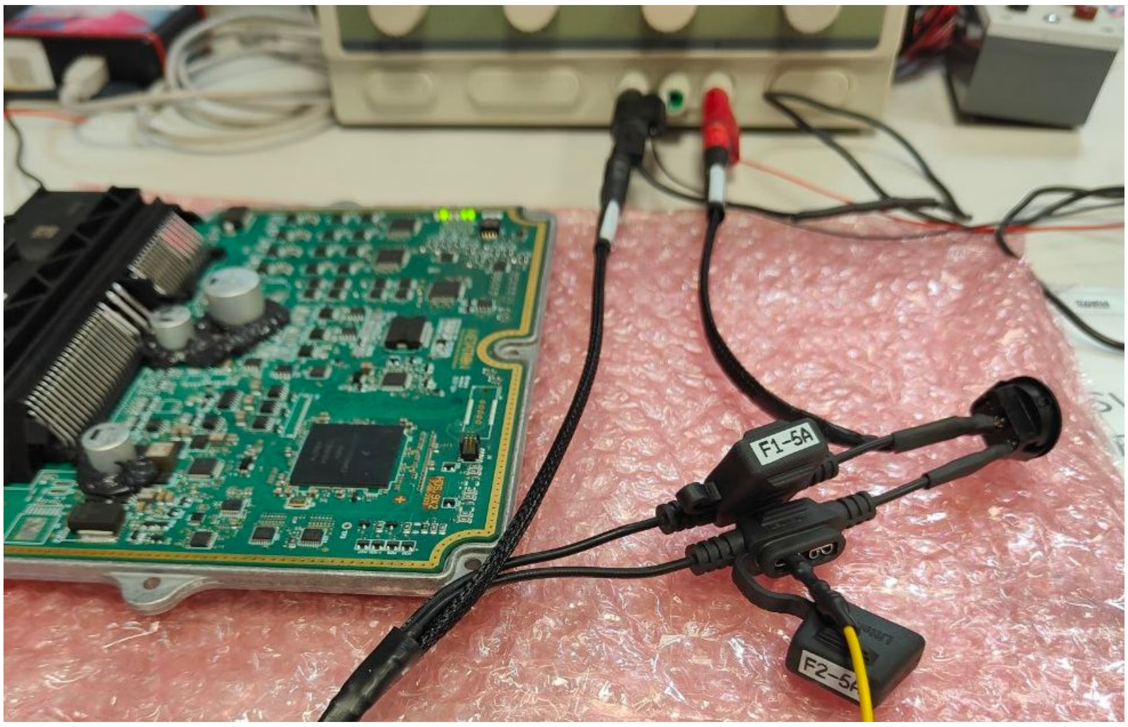


Figure 4.3.2 – Board, relay and power source

The next two sections (4.4 and 4.5) will illustrate in more detail the code for the two main tests, with detailed descriptions of both the Simulink and LabVIEW models, while also providing a general introduction to the test program's behaviour.

Both tests were designed to perform computations, store them in the memory module, shut down the system, restart it, and either directly verify the expected results or simply write the read values to a file to be verified externally (via data sheet evaluation).

4.4. Test #1 – Mem Test

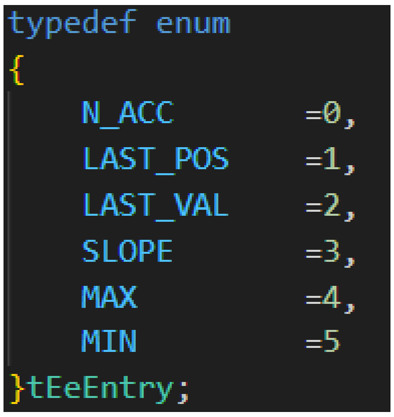
The first test, also called "Mem Test", used an application program, built via model-based design using Simulink. This application took a maximum and a minimum value via CAN, computed the next step of a function limited between these two values, and stored the current value into a new entry of the memory array, treating it as a cyclic structure.

The application then communicated, again via CAN, the last written position of the memory's array, the written value, the slope (positive or negative) of the function, and the number of startups of the system since the test started.

The six main values that the system had to store were:

- the number of startups as an integer value, **N_ACC**.
- the last written position as an integer value**, LAST_POS**.
- the last written value as a floating-point value, **LAST_VAL**.
- the slope of the step function as a Boolean value, **SLOPE**.
- the read maximum as a floating-point value, **MAX**.
- the read minimum as a floating-point value, **MIN**.

To provide easier access to these values' reserved positions in the memory array, a custom enumerative was introduced in the code, with labels associated with the values.


```
typedef enum
{
    N_ACC       =0,
    LAST_POS    =1,
    LAST_VAL    =2,
    SLOPE       =3,
    MAX         =4,
    MIN         =5
}tEeEntry;
```
Figure 4.4.1 - Enumerative of the main values for the first test

In addition, the program also computed the battery power and temperature read by the corresponding on-board sensors and transmits them via CAN, to monitor the board's operating conditions. It also read the currently active memory module, although this information was not communicated actively via CAN by the application but read directly through CANape.

Aside from simply testing the capability of the system to correctly store and read the information for large numbers of startup and shutdown cycles, both tests were also designed to study the effectiveness of the strategy selected for the storage format of the values.
In other words, we wanted to save as many different types of values as possible to be able to verify the ease of use of the data type conversion methods and ensure that it would not be a burden for the eventual users.

For the longest execution of this test, the following values were used:

- **DATA_EE_ARRAY_SIZE:** set to 256. As six of them were reserved for the values seen before, the number of steps of the function that could be stored, called **N**, equals 250.
- **MAX**: sent via CAN, set to 0.04.
- **MIN**: sent via CAN, set to 0.02.
- **Minutes between reboots**: 1 minute.

The application was executed every 100 [ms]. At each iteration, the value for the next step was computed using the formula $Step = |\left(\frac{MAX - MIN}{N}\right)|$.
To determine the slope of the function, the following equation needed to be solved at every execution:

$$\begin{aligned} Slope = {} & [(Last\,Value - Step) > MIN \;\&\&\; (Last\,Value + Step) < MAX]?\; last\_slope \\ & :\; [(Last\,Value + Step) > MAX?\; negative\_slope :\; positive\_slope]; \end{aligned}$$

where “*Last Value*” is the last computed value, “*Step*” is obtained with the previous formula, and “*positive_slope*” and “*negative_slope*” correspond to Boolean 1 and 0 values.

4.4.1. Simulink

The program for the tests was built following a model-based approach using Simulink. In this section, the key blocks of the application will be shown and described.

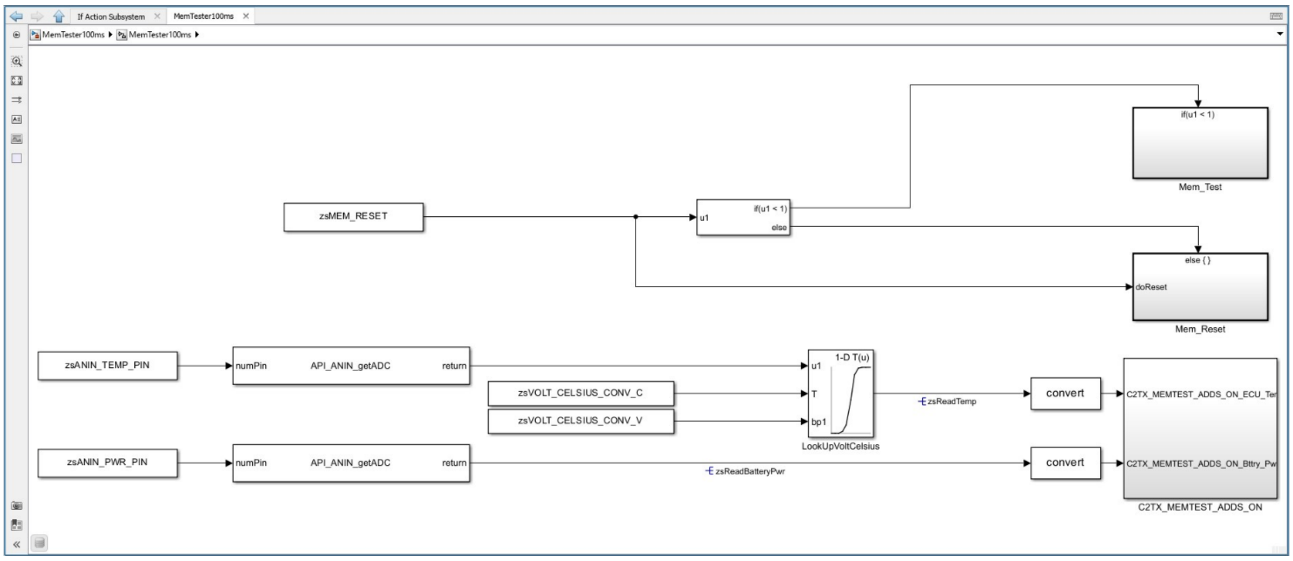


Figure 4.4.2 - Mem Test Model

The picture above shows the first level of the “Mem Test” application model, where we can vertically distinguish two parts: the section above, showing an ‘if’ construct block, piloted by a **zsMEM_RESET** constant (whose value can be changed via CANape for testing purposes), and the part below, where temperature and battery power are detected.

The section below gets the two readings as Volt values, then performs a conversion on the temperature using a lookup table from the tension to a Celsius value and writes both variables into a CAN message via the **C2TX_MEMTEST_ADD_ON** block (after having converted them in a suitable format).

The first part pilots two subsystems: the **Mem_Test** subsystem and the **Mem_Reset** one. While the *reset* subsystem simply calls the utility function **API_EEPROM_forceReset** to clean the active module’s memory structure, the *test* one contains the actual core of the application, as visible in the following pictures.

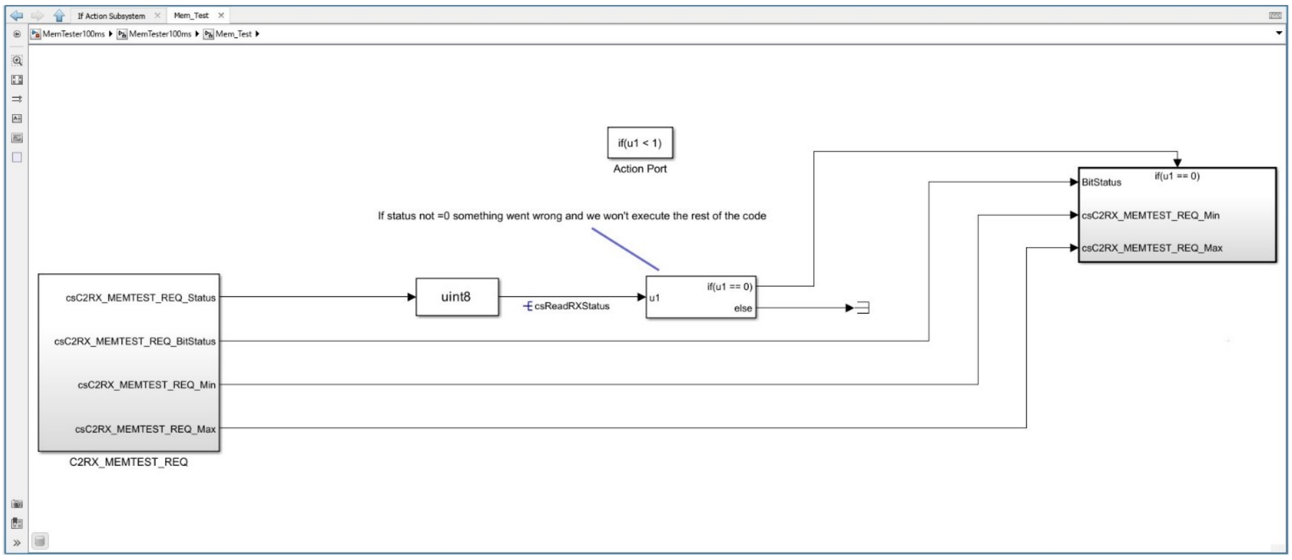


Figure 4.4.3 - Simulink subsystem Mem_Test

As shown in the picture above, the first level inside the *Mem_Test* subsystem contains the **C2RX_MEMTEST_REQ** block (on the left), used to read from the oncoming CAN messages the maximum and minimum values, along with some additional information.

One of those pieces of information, **C2RX_MEMTEST_REQ_Status**, is used to pilot an 'if' condition subsystem (on the right). This way, the subsystem will only be executed when the message is actually received, avoiding spurious executions. The remaining parts of the CAN message are directly fed to the subsystem, a full picture of which can be seen in the next image.

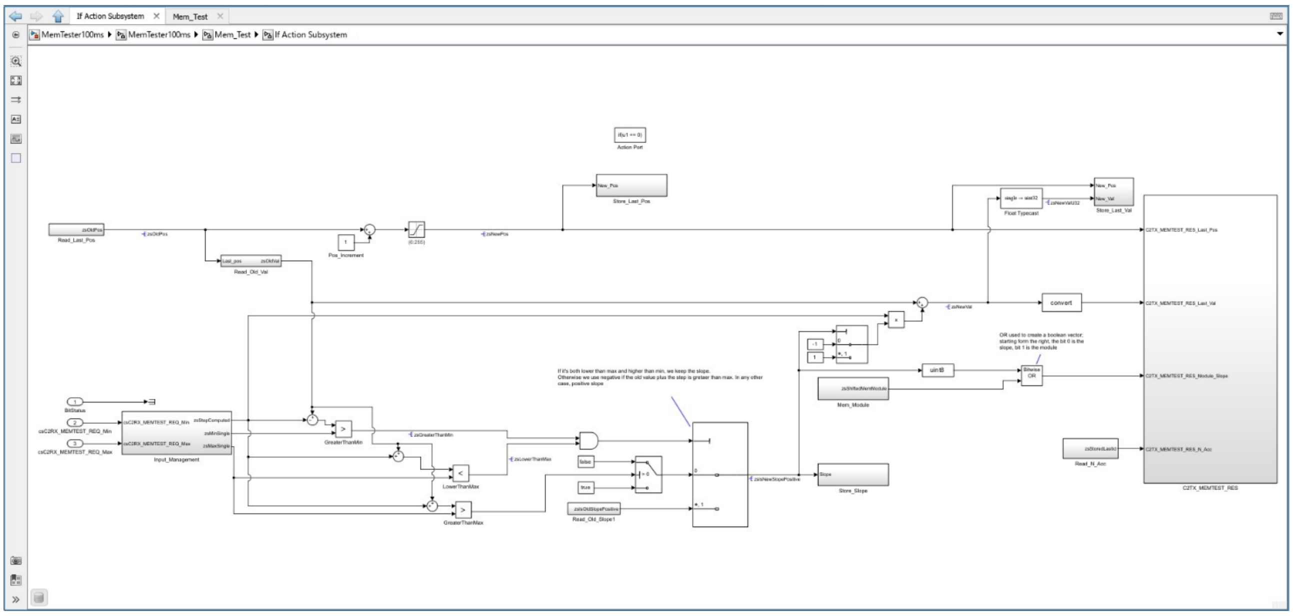

If Action Subsystem
Mem_Test
MemTester100ms ▸ MemTester100ms ▸ Mem_Test ▸ If Action Subsystem
if(u1 == 0)
Action Port
Store_Last_Pos
Read_Last_Pos
Pos_Increment
Read_Old_Val
Float Typecast
Store_Last_Val
convert
C2TX_MEMTEST_RES_Last_Pos
C2TX_MEMTEST_RES_Last_Val
C2TX_MEMTEST_RES_Module_Slope
C2TX_MEMTEST_RES_N_Acc
C2TX_MEMTEST_RES
OR used to create a boolean vector, starting form the right; the bit 0 is the slope, bit 1 is the module
If it's both lower than max and higher than min, we keep the slope. Otherwise we use negative if the old value plus the step is greater than max. In any other case, positive slope
uint8
Bitwise OR
Mem_Module
Input_Management
GreaterThanMin
LowerThanMax
GreaterThanMax
Read_Old_Slope1
Store_Slope
Read_N_Acc
BitStatus
csC2RX_MEMTEST_REQ_Min
csC2RX_MEMTEST_REQ_Max

Figure 4.4.4 - Mem_Test main subsystem

Once again, it is better to visualise the different parts that make up this model to have an easier understanding of it.

The first part is the one dedicated to the management of the CAN inputs: the Min and Max values received are stored inside the respective positions in the memory structure, using the *setData* function blocks and then the step is computed using the formula already seen.

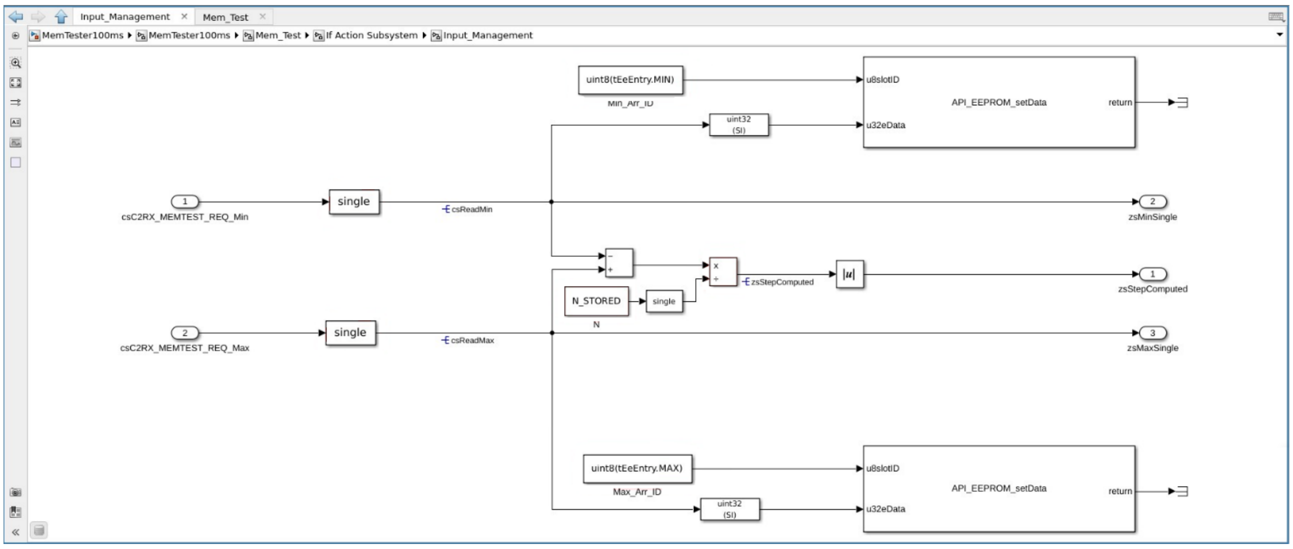


Figure 4.4.5 - Min and Max input management

All of this is contained in the **Input_Management** subsystem, in the bottom-left corner.

In the top-left part of the subsystem, the last written position is read from the memory (**Read_Last_Pos** block) using *getData*, and then its value is used to retrieve the last written value and to compute the next position to write into. The new position is then stored using *setData* in the **Store_Last_Pos** block.

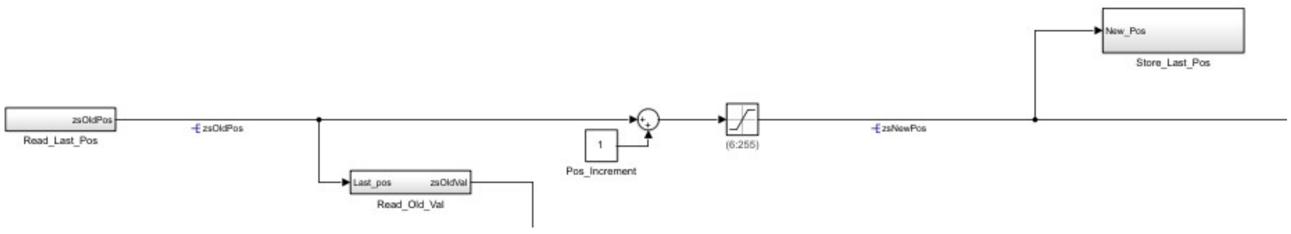


Figure 4.4.6 - Last Position retrieval part

The old value is then used, together with the step, Max, and Min computed by *Input_Management*, to obtain the new slope of the function following the formula shown before. The computed slope value is stored in memory inside the **Store_Slope** block, using a *setData* function block.

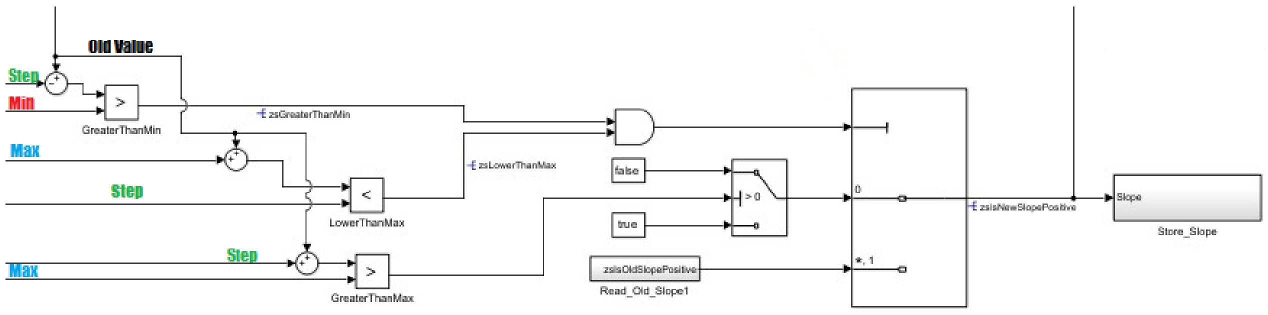


Figure 4.4.7 - Slope computation

The final part of the subsystem computes the new value, adding or subtracting (based on the slope) the step from the old value, and stores the result at the new position in the memory structure. It also compacts the slope and the memory module flag into a single 8-bit value (basically, an array of bits) after shifting the module's flag by 1 bit.

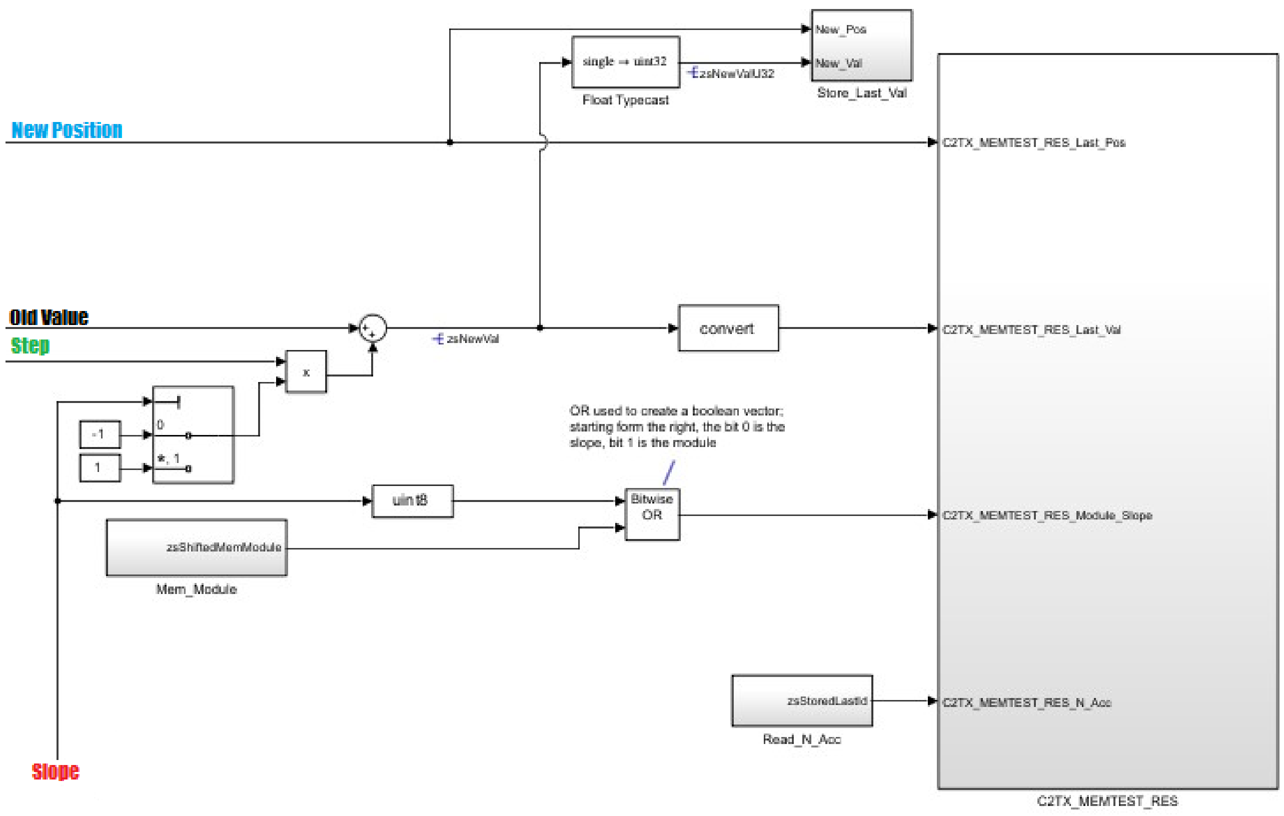


Figure 4.4.8 - Mem_Test final steps

The model then assembles the last written position, the newly computed value, the module and slope "8-bit array", and the number of startups (**N_Acc**) and writes them into a new CAN message thanks to the **C2TX_MEMTEST_RES**.

To compute the *N_Acc*, the **Read_N_Acc** subsystem checks whether this is the first time the code has been executed since the last startup. If that is the case, the model retrieves the value for the number of startups saved in memory, increments it by one, then stores it back. If this is not the first execution of the code, the system simply reads the stored value.

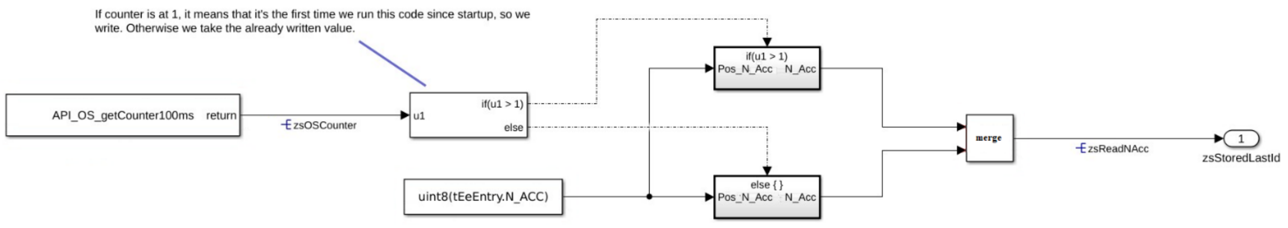
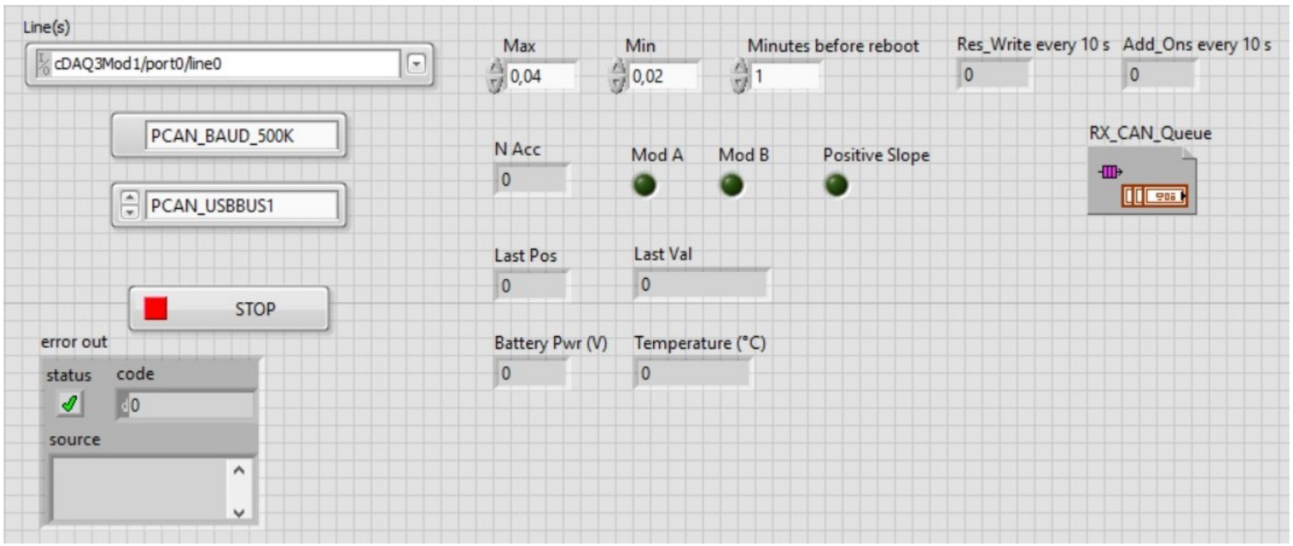

If counter is at 1, it means that it's the first time we run this code since startup, so we write. Otherwise we take the already written value.
API_OS_getCounter100ms return
zsOSCounter
u1
if(u1 > 1)
else
uint8(tEeEntry.N_ACC)
if(u1 > 1)
Pos_N_Acc N_Acc
else { }
Pos_N_Acc N_Acc
merge
zsReadNAcc
1
zsStoredLastId

Figure 4.4.9 - Subsystem for reading and computing the number of startups (N_Acc)

As can be seen from the pictures above, the model and its subsystems make use of many different data types (booleans, integers on different numbers of bits, floating and fixed-point values). This was deliberately done to test the impact of the chosen implementation strategy, which requires the homologation of the various data types into 32-bit unsigned integers to store them and their reconversion to use them.

4.4.2. LabVIEW

As previously mentioned, we have developed LabVIEW Virtual Instruments to automate the tests. This test's LabVIEW program had to produce the CAN messages containing Max and Min, store the board's responses, and manage the reboot of the board piloting the DAQ.



Line(s): cDAQ3Mod1/port0/line0
PCAN_BAUD_500K
PCAN_USBBUS1
Max: 0,04
Min: 0,02
Minutes before reboot: 1
Res_Write every 10 s: 0
Add_Ons every 10 s: 0
N Acc: 0
Mod A
Mod B
Positive Slope
Last Pos: 0
Last Val: 0
Battery Pwr (V): 0
Temperature (°C): 0
error out
status ✓
code: 0
source

Figure 4.4.10 - LabVIEW VI interface for the first test

To simplify the choice of the values and introduce a way to check the correctness of the ongoing tests, a Graphical User Interface (GUI) has been built, as visible in the previous picture.

The GUI can be used to select the *Max* and *Min* values to send, the number of minutes between reboots, the CAN baud rate and bus, and the DAQ output port. Moreover, it shows the number of startups, the last written position and value, the readings for battery power and temperature, the currently active memory module, and the slope of the function.

The virtual instrument "behind the mask" can be divided into 3 parts: an initialisation part, the main body, and a termination part.

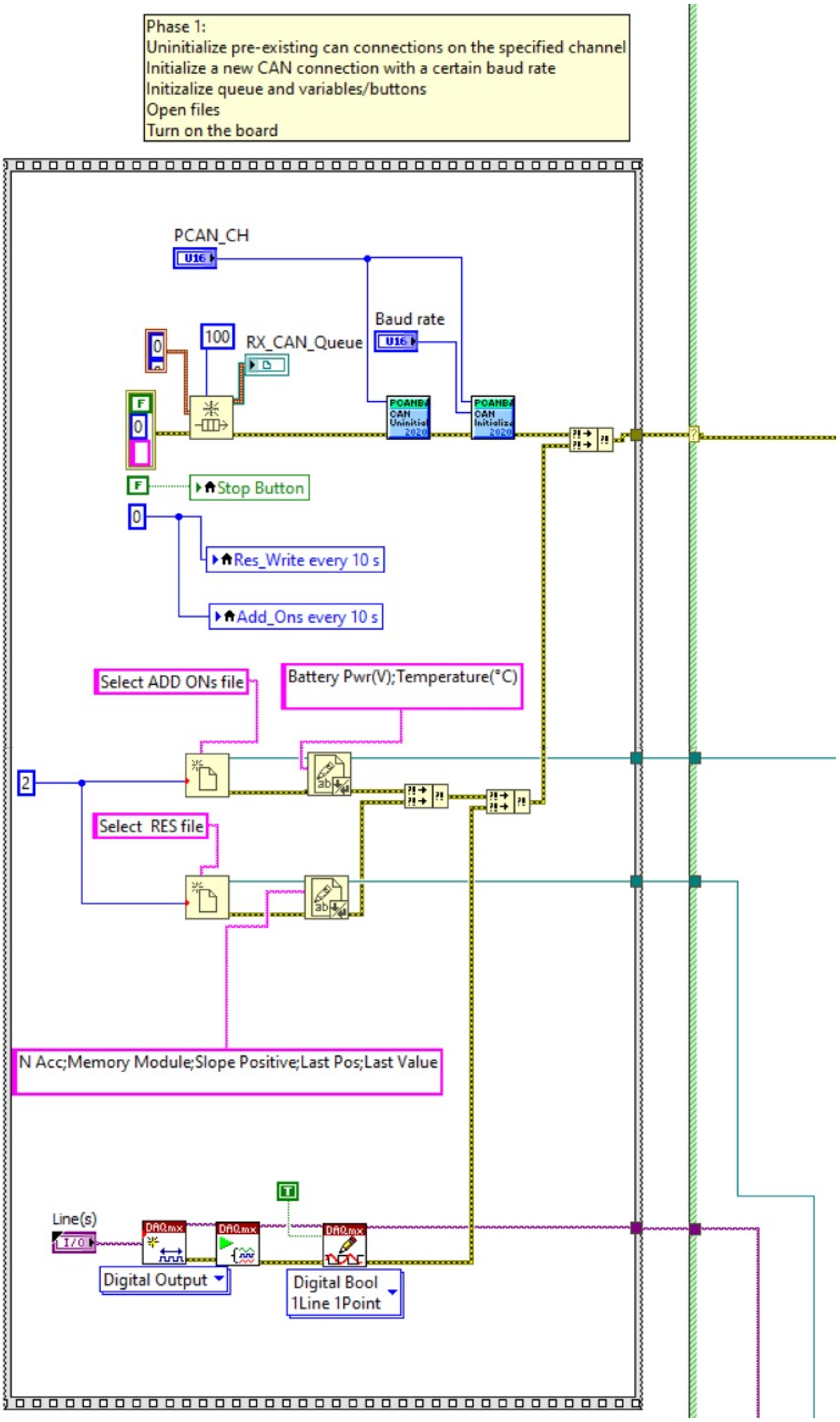
The first part is executed only once, at the beginning of the test, and has the duty to set up the system for the oncoming operations.

First of all, it creates a new, empty queue of CAN messages to store the ones received from the board. It then assigns the queue as the destination for the CAN-read instruments while uninitialising (to clear up any remnants of earlier tests) and reinitialising the chosen CAN channel with the appropriate baud rate (both of which are supplied via GUI).

The program then resets a couple of variables used by the other parts and prepares the files that will be used to record the values read by the CAN messages. It opens a "**RES**" file to record the main values read from the memory and an "**ADDS ON**" file for the values of power and temperature. It proceeds by writing on each file a header line with the columns' names. For the "RES" file, the header contained **N_Acc**, **Memory Module**, **Slope Positive**, **Last Pos**, and **Last Value**. For the other file, the only two columns were **Battery Pwr(V)** and **Temperature(°C)**.

The instrument goes on to initialise the DAQ digital output channel to control the board's switch.

Lastly, it uses the now-initialised DAQ to turn on the board.


Phase 1:
Uninitialize pre-existing can connections on the specified channel
Initialize a new CAN connection with a certain baud rate
Initizalize queue and variables/buttons
Open files
Turn on the board

PCAN_CH
U16
100
RX_CAN_Queue
Baud rate
U16
PCANBA CAN Uninitial 2020
PCANBA CAN Initialize 2020
Stop Button
Res_Write every 10 s
Add_Ons every 10 s
Select ADD ONs file
Battery Pwr(V);Temperature(°C)
2
Select RES file
N Acc;Memory Module;Slope Positive;Last Pos;Last Value
Line(s)
DAQmx
Digital Output
Digital Bool 1Line 1Point

Figure 4.4.11 - LabVIEW VI initialization part

The second part of the instrument is a set of four loops, each managing a different task.

The first loop waits for a specified amount of time (here, 150 [ms]) to avoid oversaturating the channel and then writes a message (ID: 0x170) containing *Min* and *Max*.
To do so, it takes the values from the controls as singles, applies the resolution (multiplying for 100), and splits the values into 8-bit parts, inverting the bit order (as the board expects INTEL endianness).
It then sends the CAN message on the channel initialised in the first part.

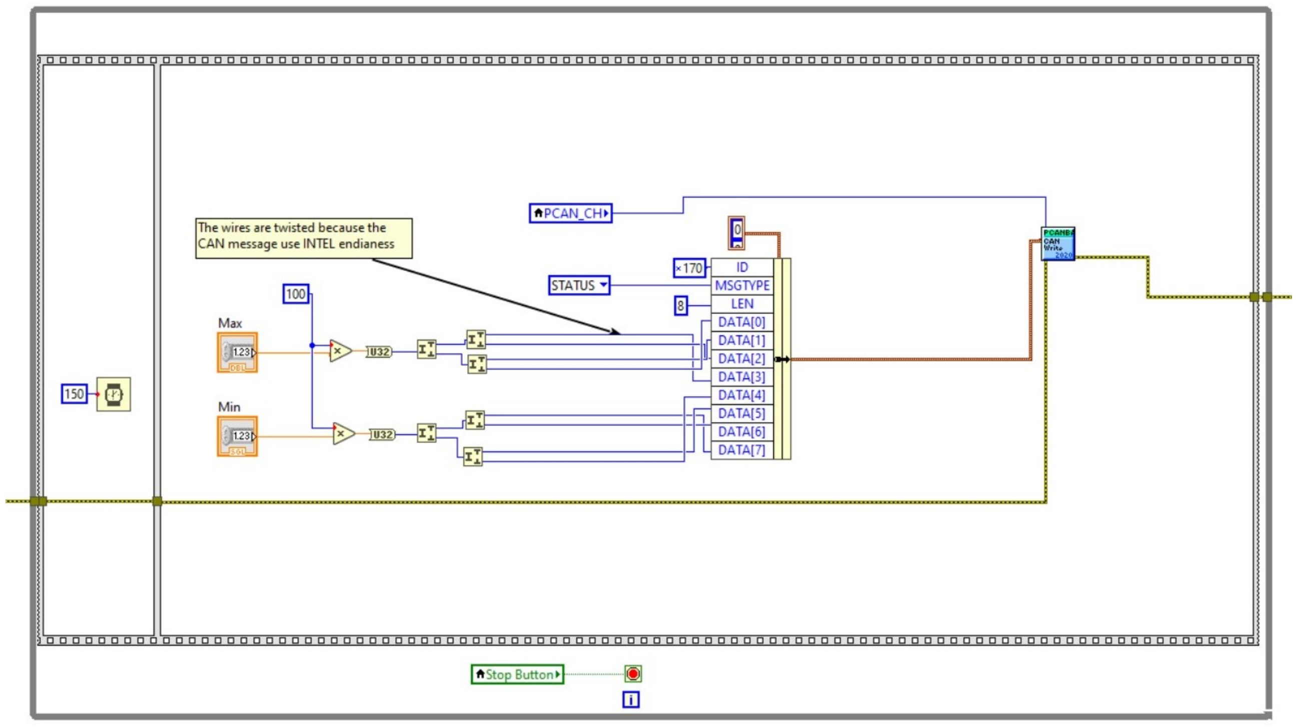


Figure 4.4.12 - First loop, to send the message

The second loop manages the enqueuing of the received messages with the expected IDs (0x171 for the “RES” messages, 0x172 for the “ADDS ON” ones).

It uses counter variables to ensure that only one message per type is stored every 10 seconds. The queue is shared between the two types of messages; however, the counters are distinct.

The current values of the counters can be seen in the GUI under the names “Res_Write every 10s” and “Add_Ons every 10s”.

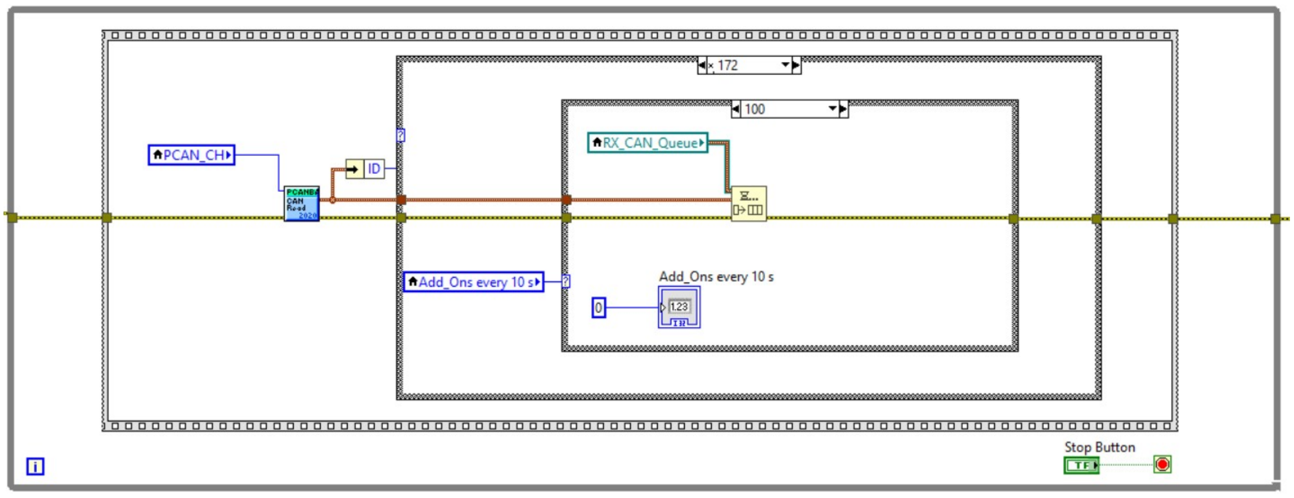


Figure 4.4.13 - Second loop, for the case of an "ADD ON" message

The third loop is the one that reads the messages from the queue, interprets the contents, converts them into the expected format, and records them in the corresponding file based on the message's ID. Depending on the message's ID, the operations performed differ. The loop is executed every 1 second.

In the case of ID: 0x171, "RES" message, it reads the first two bytes as one uint16 (N_Acc), the third byte as a boolean array (of which only the first two positions are used, for the memory module and the slope), and the fourth byte as a uint8. The last four bytes are recombined into an int32, which is then multiplied by the precision to obtain the desired value (a fixed point on 32 bits with 10^{-5} precision).

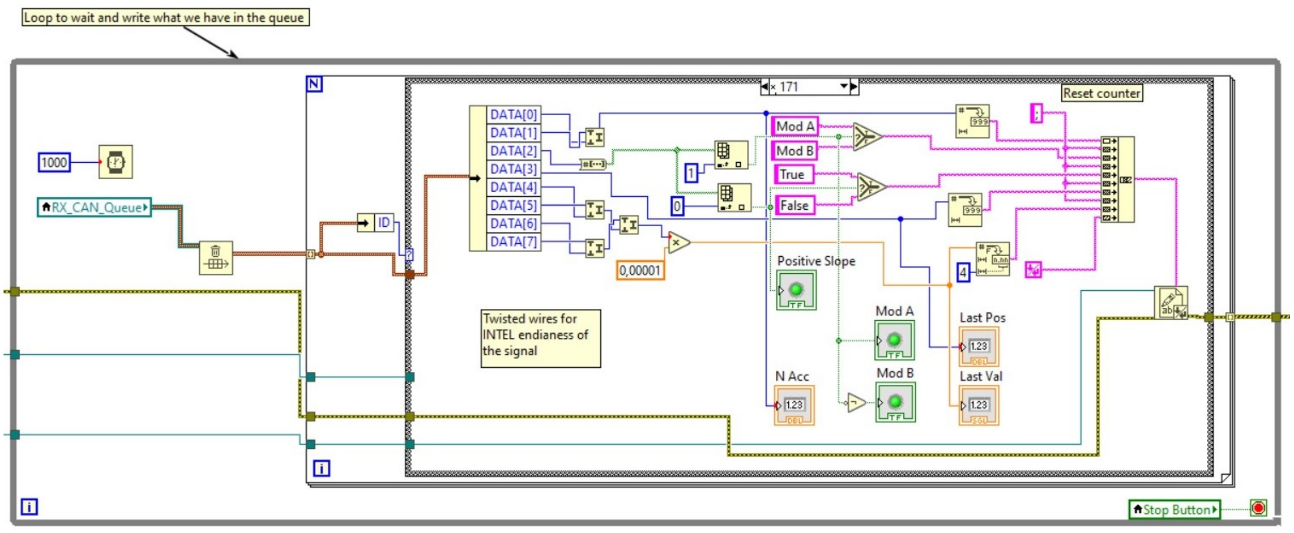


Figure 4.4.14 - Third loop, with a message with ID 0x171

For an ID equal to 0x172, the program reads the first 4 bytes as battery power (V) and the other four as temperature (°C). The bytes are recomposed as int32 and then multiplied by the desired accuracy (0.01).

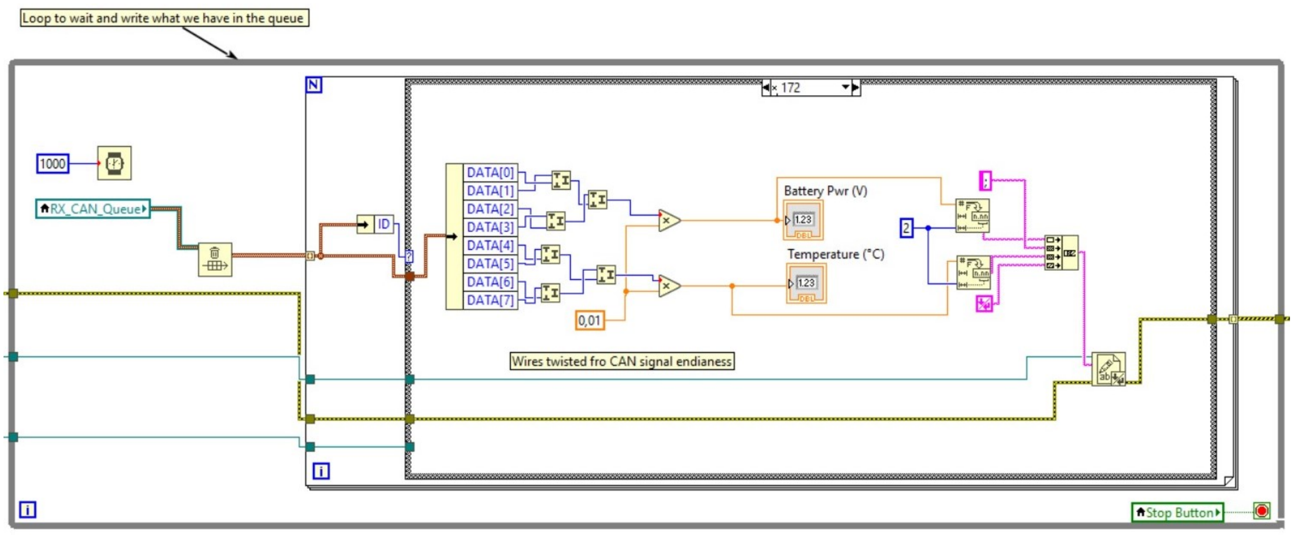


Figure 4.4.15 - Third loop, with a message with ID 0x171

In every occurrence of "byte recomposition", the values are reordered, as the board produces them in INTEL endianness while LabVIEW expects them to be Big-endian. The values obtained are then converted to strings and printed to the file.

The fourth loop is the one responsible for rebooting the board. After the period of time selected using the GUI, it sets the digital output on the DAQ to zero, awaits 5 seconds, and then sets the output back to 1, thus shutting down and turning the board on again.

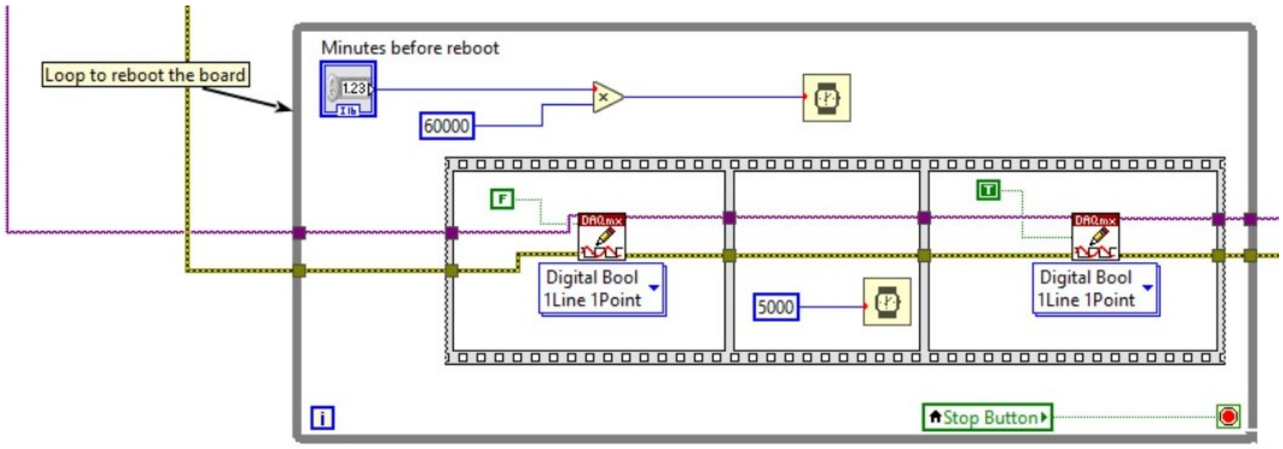


Figure 4.4.16 - Fourth loop, to reboot the board

The final part of the virtual instrument is executed only once, just like the first, at the end of the tests, that is, when the tester pushes the stop button on the GUI. This action stops all the loops seen before and moves the execution of the program to this last part, although not immediately, as the current iterations of the loops will continue normally.

The tasks of this part are all related to the clean-up of the instrument; it uninitialises the CAN channel, resets the stop button, and turns off the board. It also shows any error that has been produced so far by the system.

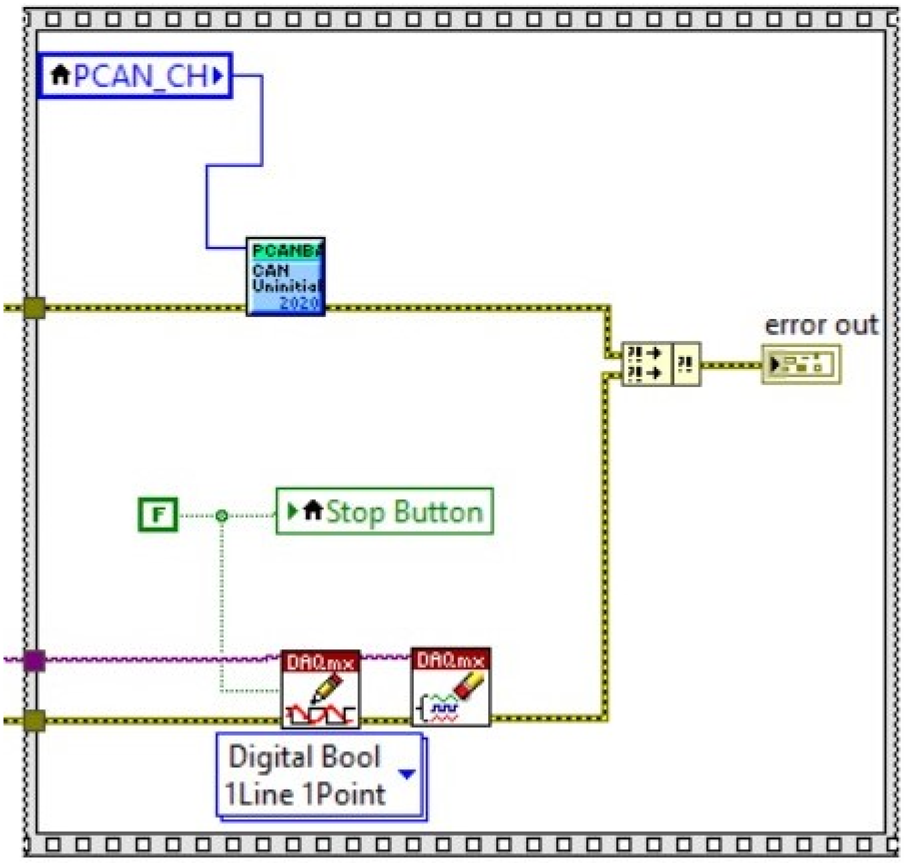


Figure 4.4.17 – LabVIEW clean-up part

4.5. Test #2 – Mem Check

As the first test, the second one, called “Mem Check”, used a combination of a program built following a model-based approach using Simulink, a LabVIEW virtual instrument specifically built to automate this test, and datasheet analysis tools such as Microsoft Excel.

As in the other test, the aim was to perform repetitive read and write memory operations, study the feasibility of the selected approach, test the burden on eventual users for the modelling process, and verify the reliability of the memory management strategy when subjected to repeated and sudden shutdowns.

The application designed for the second test takes two inputs via CAN, **VAL** and **REPS**. It then computes **N** values to store in the memory structure’s array and calculates a checksum using part of those values. Both the *N* values and the checksum are stored in memory.

In addition, the system also communicates the number of messages sent since the last startup, the currently active memory module, and the “**exceed flag**”. The last two are actually sent together as an array of bits. The *exceed flag* is set to 1 once the number of startups is greater than *N.*

As in the previous test, the system transmits an additional CAN message containing information on the battery power and the CPU’s temperature to keep the operating conditions in check.

This time, the total number of cells has been extended up to 2048, but, as in the previous test, some entries of the array are reserved for certain values and indexed using an enumerative, **tEeEntry**, for simplicity. This leaves us with *N* (= 2043) remaining cells.

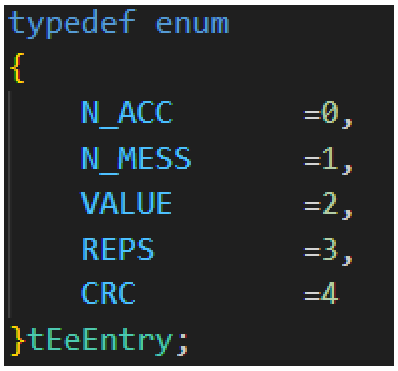

```
typedef enum
{
    N_ACC       =0,
    N_MESS      =1,
    VALUE       =2,
    REPS        =3,
    CRC         =4
}tEeEntry;
```

Figure 4.5.1 - Enumerative of the main values for the second test

The five main values that the system had to store, corresponding to the tags of the enumerative, were:

- the number of startups as integer values, **N_ACC**.
- the number of messages sent since the last startup, **N_MESS**.
- the *VAL* read via CAN as a 32-bit fixed-point signed with 10^{-5} precision, **VALUE**.
- the number of cells for block used for the checksum computation (see later), read via CAN, as a 16-bit unsigned integer **REPS**.
- the checksum value as a 32-bit fixed-point signed with 10^{-5} precision, **CRC**.

The values to store in the N memory cells are computed as follows:

- The first **OFF** $(N\_Acc \,\%\, N)$ cells are filled with their index plus *OFF*.
- The following **Q** $(N - OFF)$ cells will contain either $VAL + cell\_index * precision$ or $-(VAL + (cell\_index - REPS) * precision)$, in an alternate fashion, in groups of *REPS* values each.

The checksum **CRC** is computed on the last **X** $(Q \,\%\, (2 * REPS))$ cells as the sum of the values contained in those cells. If it is the first time that the program has been executed since the last startup, it will actually use the values stored since the previous shutdown, and thus the first computed *CRC* after a startup should equal the last one stored before turning off the board. The computation of the checksum can be summarized with the following formula:

$$\begin{cases} \displaystyle\sum_{i=N-X}^{N} VAL + i * precision, & X < REPS \\ \displaystyle\sum_{i=N-X}^{N-X+REPS} VAL + i * precision - \sum_{i=N-X}^{N-REPS} VAL + i * precision, & REPS \leq X < 2 * REPS \end{cases}$$

For the longest execution of this test, the parameters selected were:

- **VAL**: sent via CAN, set to 10^{-5}.
- **REPS**: sent via CAN, set to 3.
- **Minutes before reboot**: 1 minute.

4.5.1. Simulink

As in the previous case, the test application was built following a model-based approach using Simulink.

The first level of the model, named “MemCheck100ms” as it was executed in the 100 [ms] task of the operating system, is the same as in the previous test:

- An ‘if’ construct block, piloted by the **zsMEM_RESET** constant, whose value can be changed via CANape, leads to two subsystems: the **Mem_Check** subsystem and the **Mem_Reset** one. The Mem_Reset functions similarly to the previous test, while the Mem_Check serves as the central component of the application.
- A section identical to the one in the previous program gets the battery voltage and temperature as Volt values, performs a conversion on the temperature using a lookup table from the tension to a Celsius value, and writes both data into a CAN message via the **C2TX_MEMTEST_ADDS_ON**

The subsystem piloted by the ‘if’ case is actually a wrapper subsystem, composed of two distinct parts that converge into the real *Mem_Check* block.
The section above uses the **C2RX_MEMTEST_REQ** block to read the *VAL* and *REPS* from the oncoming CAN messages. The status of the CAN channel is used to pilot another ‘if’ statement to ensure that the rest of the system won’t operate in case of errors with the CAN.
The lower part of the subsystem performs the computation of the checksum *CRC* and the current *N_Acc* via the **CRC_Computation** subsystem and the **Read_N_Acc** one.

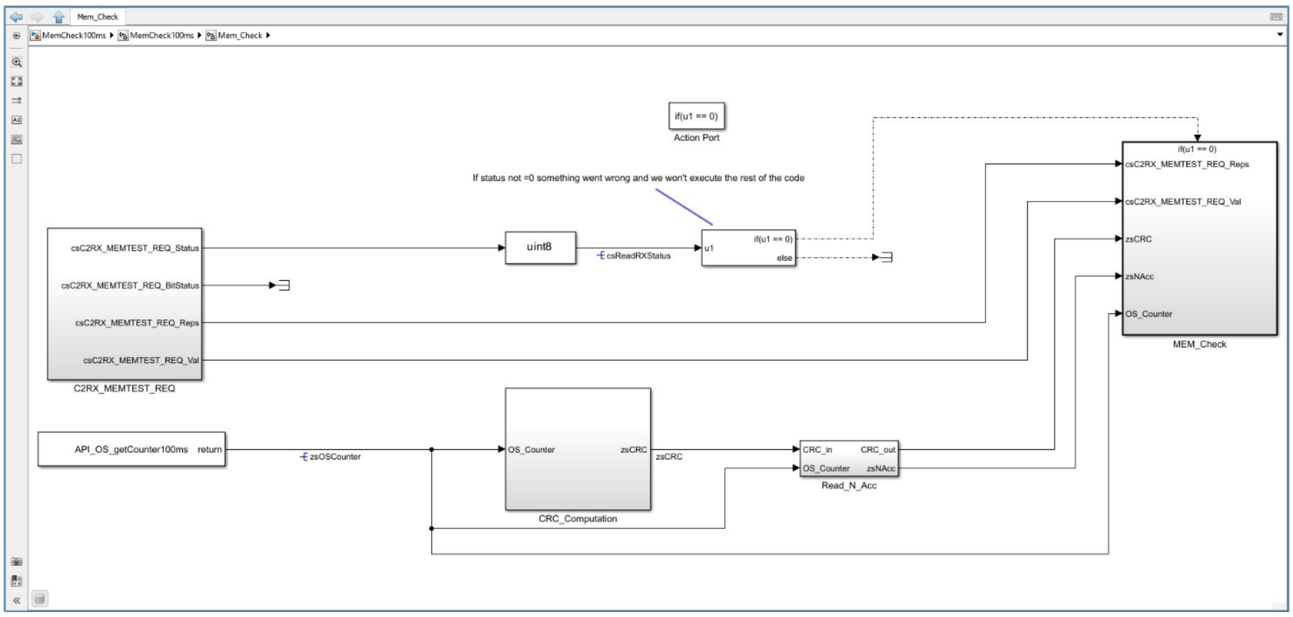


Figure 4.5.2 - Mem_Check CAN message and CRC computation wrapper level

The *CRC_Computation* subsystem takes as input the operating system's 100 [ms] counter. The value of the counter is then used to decide whether this is the first execution of the subsystem since the last startup.

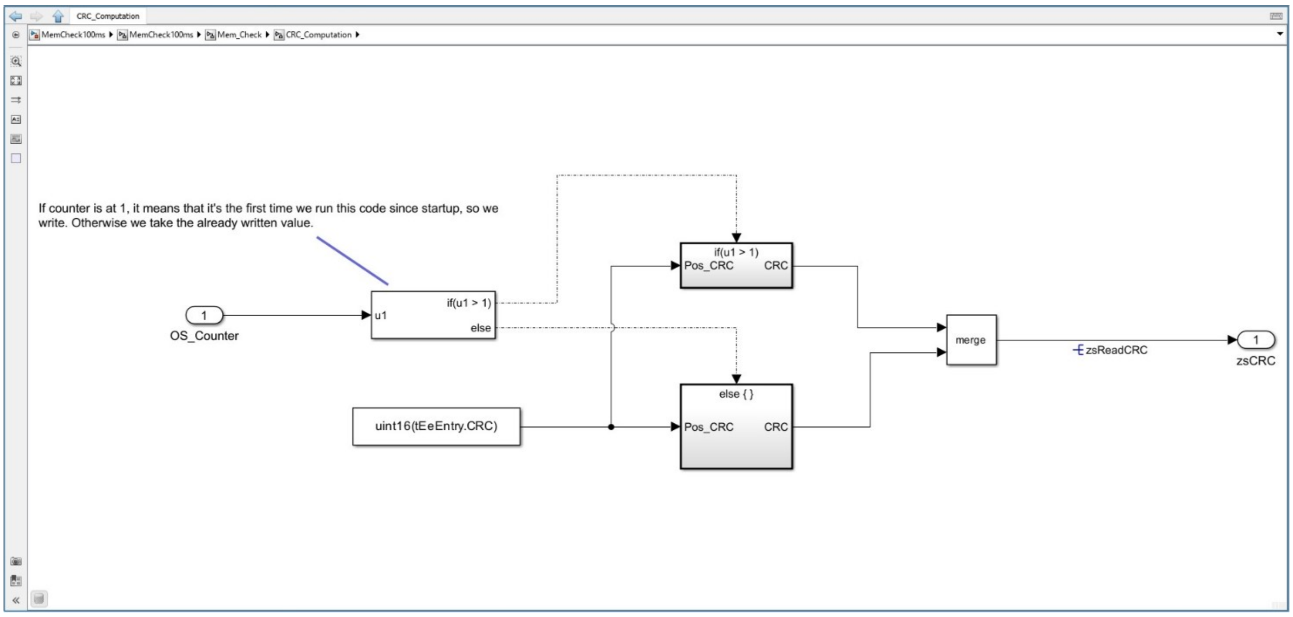

CRC_Computation
MemCheck100ms ▸ MemCheck100ms ▸ Mem_Check ▸ CRC_Computation ▸
If counter is at 1, it means that it's the first time we run this code since startup, so we write. Otherwise we take the already written value.
1
OS_Counter
u1
if(u1 > 1)
else
if(u1 > 1)
Pos_CRC
CRC
else { }
Pos_CRC
CRC
uint16(tEeEntry.CRC)
merge
zsReadCRC
1
zsCRC

Figure 4.5.3 - CRC computation subsystem

This check is made because, in the case of the first execution, the *CRC* used will be the one stored in memory before the last shutdown, retrieved via a simple *getData*.

The subsystem for executions beyond the first one is more complex, as it needs to calculate all the data required by the checksum computation formulas seen before.

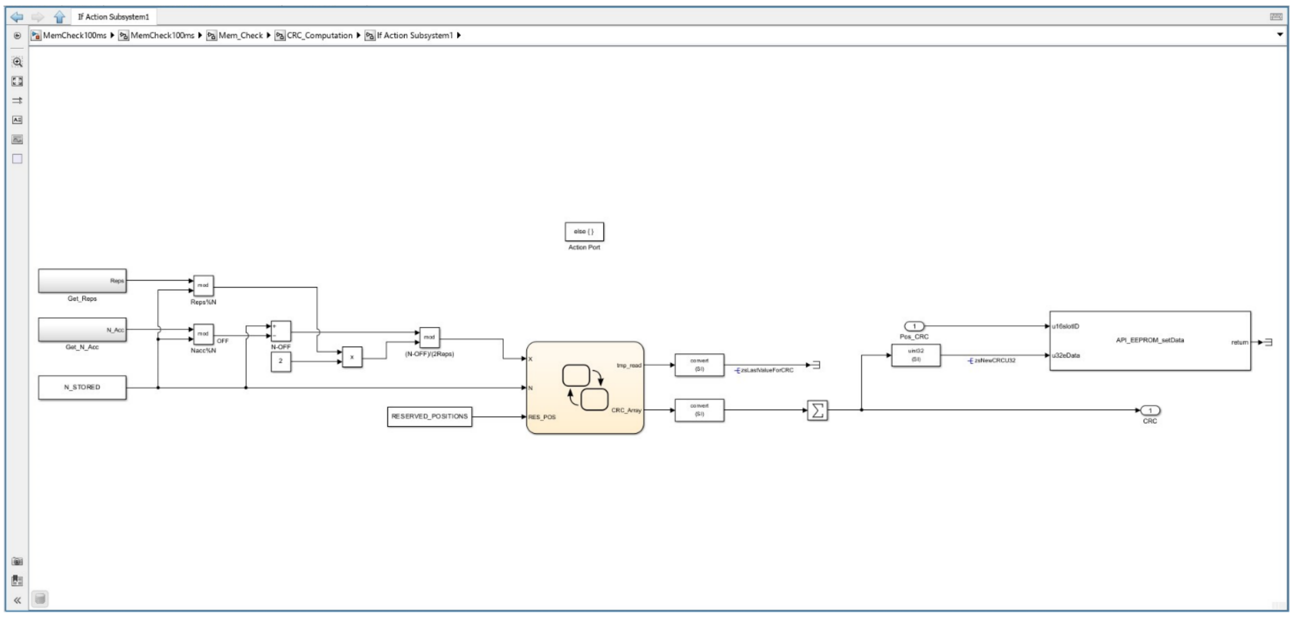


Figure 4.5.4 - CRC computation, for executions beyond the first

The subsystem can be divided into two parts: the first one, where the values needed for the formulas are computed, such as *OFF*, *Q*, and *X*, and a second part containing the finite state machine used to fill the memory cells with the values that will then be summed to obtain the *CRC*.

In the first part, *REPS* is read from the memory (rather than via CAN) and so is *N_Acc*. These two values are then used in a chain of blocks to produce *OFF*, then *Q*, and eventually *X*. This final variable, along with *N* and the number of reserved positions (a.k.a., the number of fields in the *tEeEntry* enum, called **RES_POS** in the picture below), are fed to the FSM.

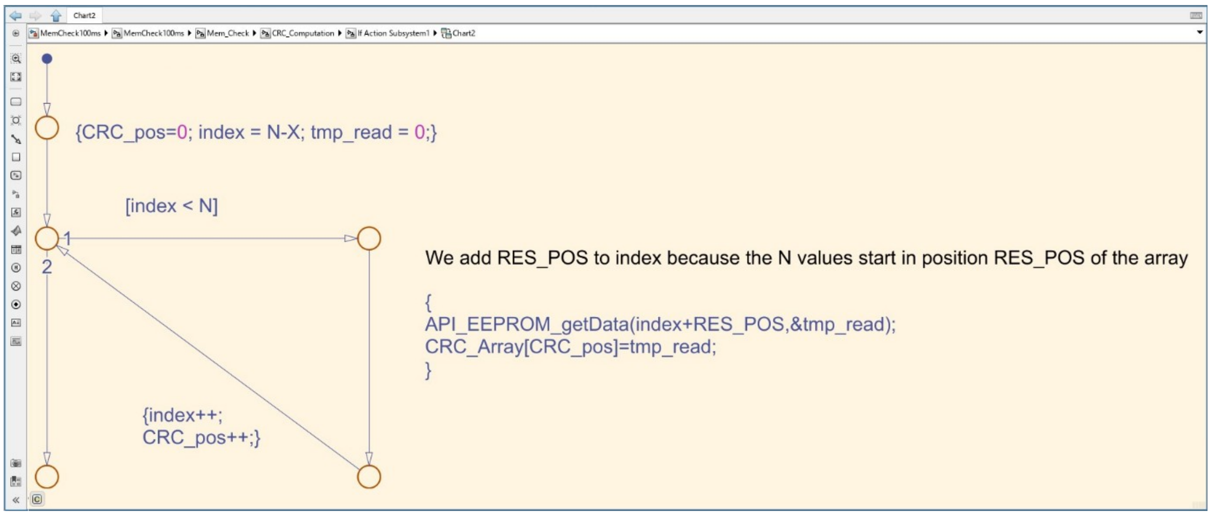


Chart2
MemCheck100ms ▸ MemCheck100ms ▸ Mem_Check ▸ CRC_Computation ▸ If Action Subsystem1 ▸ Chart2
[index < N]
1
2
We add RES_POS to index because the N values start in position RES_POS of the array

```
{CRC_pos=0; index = N-X; tmp_read = 0;}

{
API_EEPROM_getData(index+RES_POS,&tmp_read);
CRC_Array[CRC_pos]=tmp_read;
}

{index++;
CRC_pos++;}
```

Figure 4.5.5 - Finite State Machine used for the CRC values

The FSM itself is quite simple; its only task is to fill a temporary array with the values read from memory. This array will then be passed to a *Sum* Simulink block to compute the *CRC*, and the obtained value will be stored, using a *setData* block, into the reserved position of the memory structure.

Going back to the level above, the *Read_N_Acc* subsystem operates in the same way as the homonymous subsystem from the previous test program; depending on whether it is the first execution or not, it either increases the stored value *N_Acc* or simply reads it from memory. The only difference here is that the output of the *CRC_Computation* subsystem passes through this block without any changes to ensure that this block is executed after the *CRC* one.

The *N_Acc* along with the computed *CRC*, the current 100 [ms] counter value, and the *VAL* and *REPS* read from the CAN message, are sent in input to the actual **Mem_Check** block.

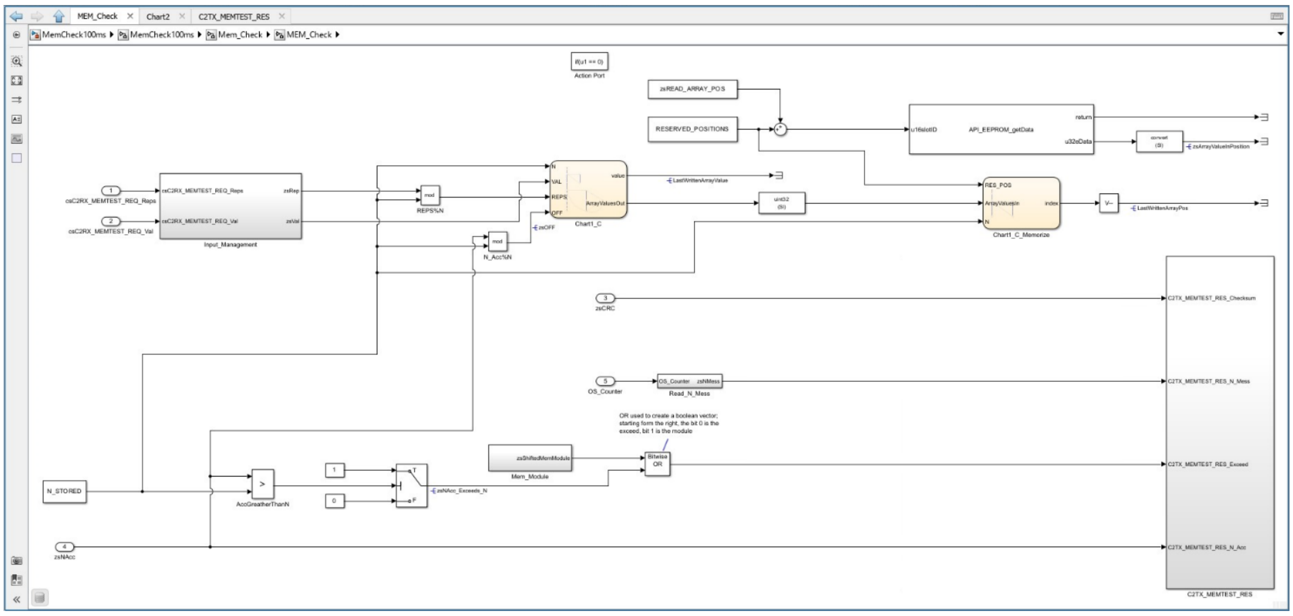


Figure 4.5.6 - Mem_Check core subsystem

This is the core subsystem of the model; it can be split into two main parts, each of which can be separated into more sections.

The upper part is the one dedicated to computing the new values to store in memory; it is comprised of an **Input_Management** block, a finite state machine to compute the array of new values, and a section to store those values using another FSM.

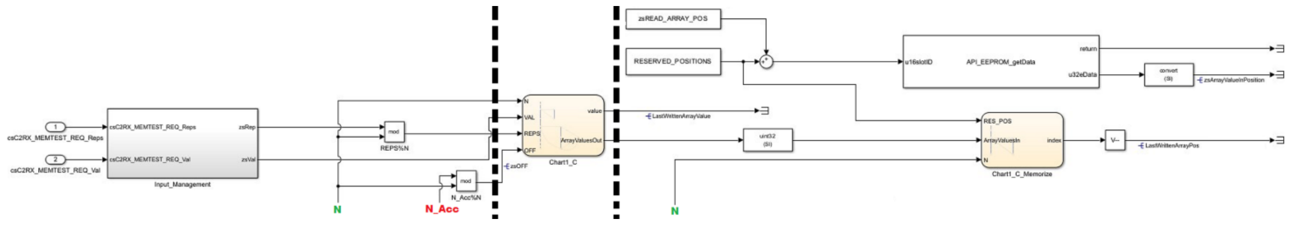
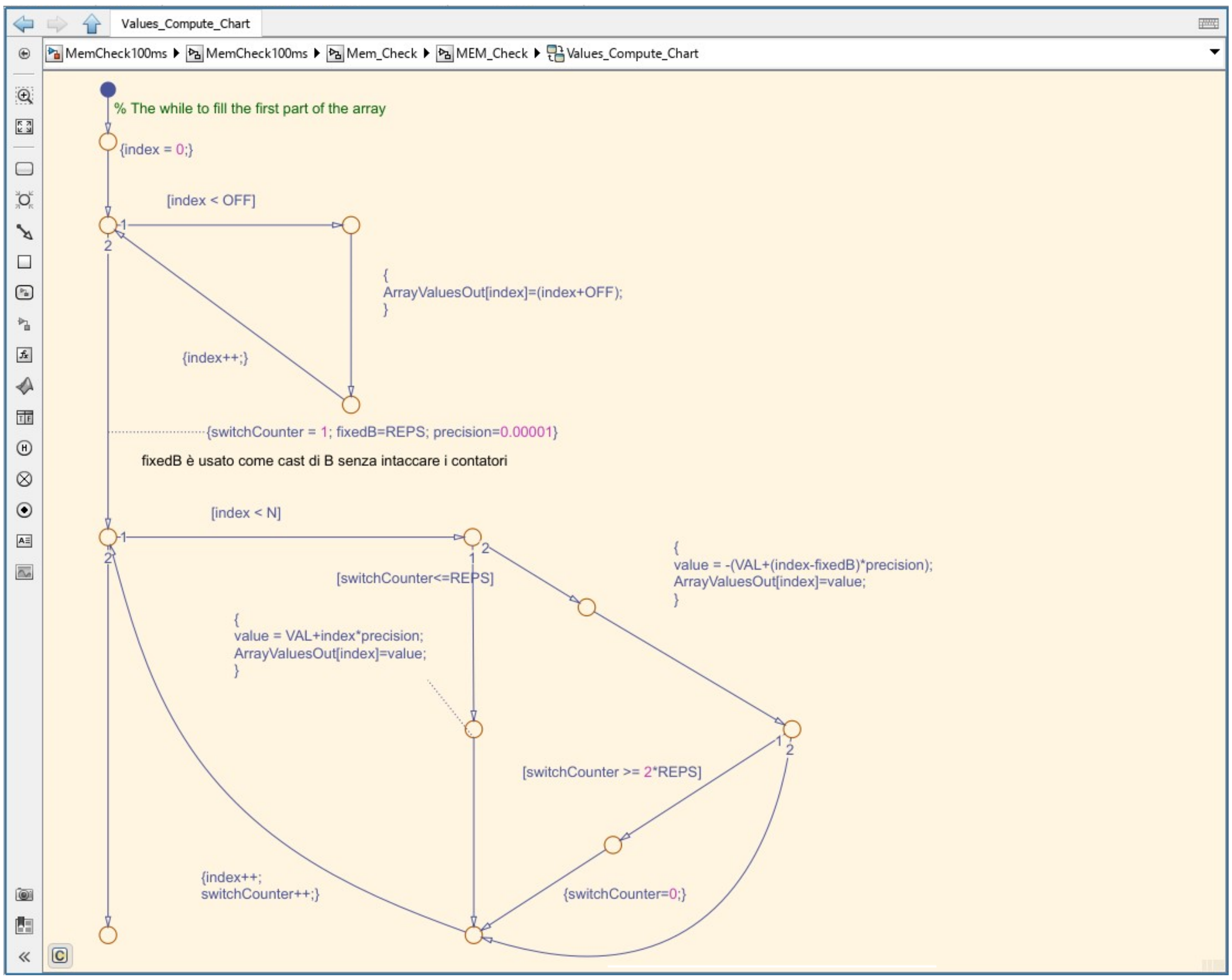


Figure 4.5.7 - The three sections of the upper part of Mem_Check

The *Input_Management* subsystem (on the left) is used to handle the conversion of *VAL* and *REPS* read from the CAN message and store them in memory. It is structured like the homonymous block in the previous test (except it does not present the *Step* computation). The system then computes *OFF* and feeds it, along with *N, REPS,* and *VAL,* to a finite state machine.

The FSM (in the middle), implemented via a Chart block, computes the *N* values to store in memory using the formulas seen before. As with the FSM in the previous test, this one is simple and mostly used to speed up the making of the program due to time constraints. It implements two loops, one to fill the first *OFF* cells and one to compute the remaining *Q* values. It does so by using the two formulas seen in the introduction to this test and a support variable **switchCounter** to decide when *REPS* cells have been filled and switch to the other formula for the values of the next *REPS* cells.


Values_Compute_Chart
MemCheck100ms ▸ MemCheck100ms ▸ Mem_Check ▸ MEM_Check ▸ Values_Compute_Chart
[index < OFF]
[index < N]
[switchCounter<=REPS]
[switchCounter >= 2*REPS]
1
2

```
% The while to fill the first part of the array
{index = 0;}
{
ArrayValuesOut[index]=(index+OFF);
}
{index++;}
{switchCounter = 1; fixedB=REPS; precision=0.00001}
fixedB è usato come cast di B senza intaccare i contatori
{
value = -(VAL+(index-fixedB)*precision);
ArrayValuesOut[index]=value;
}
{
value = VAL+index*precision;
ArrayValuesOut[index]=value;
}
{index++;
switchCounter++;}
{switchCounter=0;}
```

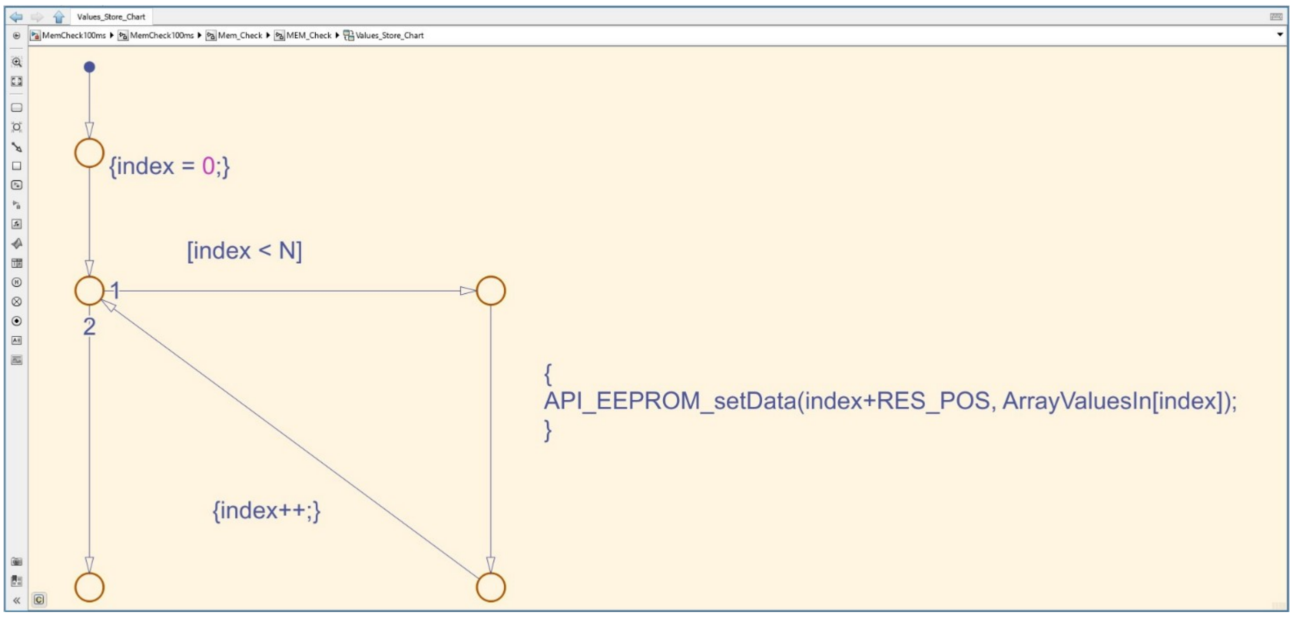
Figure 4.5.8 - FSM to compute the new values

These *N* total values computed by the FSM are stored in a temporary array that is then output by the FSM final state. The Chart block also produces the last written value of the array, as **zsLastWrittenArrayValue**, for testing purposes.

The right-most part of the section contains the FSM used to store the computed values. It also presents a set of blocks used to read a given position of the array stored in memory, used only to perform checks via CANape at runtime for debugging purposes.

The FSM implements a simple loop to call the *setData* C function on each of the *N* entries of the array. It takes as inputs the array of new values, *N*, and the number of reserved positions in memory for the main fields, *RES_POS*.

It was simply used to speed up the implementation process of the test program. It is implemented in a separate Chart block from the previous FSM to improve readability and allow for a better debugging and monitoring experience.


Values_Store_Chart
MemCheck100ms ▸ MemCheck100ms ▸ Mem_Check ▸ MEM_Check ▸ Values_Store_Chart
{index = 0;}
[index < N]
1
2
{index++;}

```
{
API_EEPROM_setData(index+RES_POS, ArrayValuesIn[index]);
}
```

Figure 4.5.9 - FSM to store the new values in memory

The second part of the *Mem_Check* core subsystem is used to compute the components of the outgoing CAN message: the checksum, *C2TX_MEMTEST_RES_Checksum*, the number of messages sent since the last startup, *C2TX_MEMTEST_RES_N_Mess*, the array of bits containing the active module flag and the *exceed flag*, *C2TX_MEMTEST_RES_Exceed*, and the number of startups, *C2TX_MEMTEST_RES_N_Acc*.

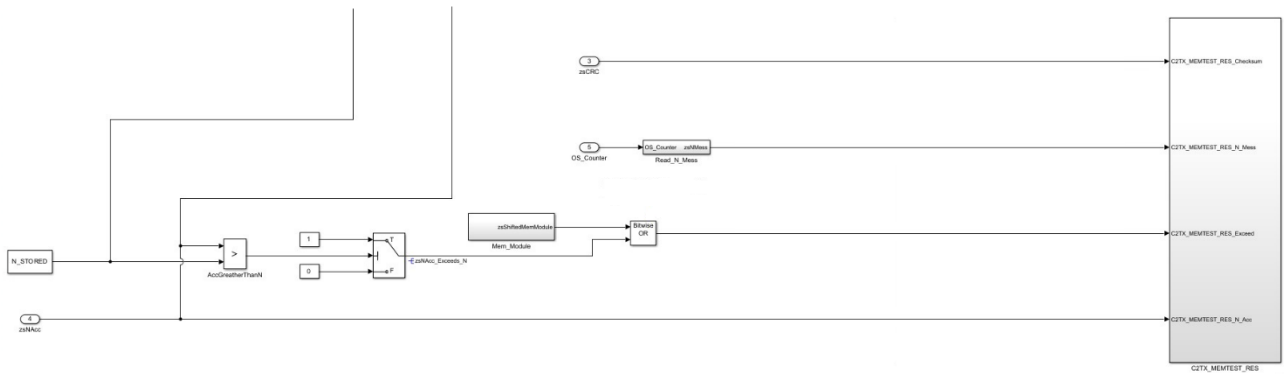


Figure 4.5.10 - Lower part of Mem_Check

The left-most blocks of this part are used to set the exceed flag by confronting *N* with the *N_Acc* read at the upper level of the model. The flag value is then OR-ed with the shifted flag for the active memory module to produce a value that will be treated as an array of bits, corresponding to the *C2TX_MEMTEST_RES_Exceed* field of the CAN message. The *N_ACC* used for these operations is also set as a field for the CAN message.

The *C2TX_MEMTEST_RES_N_Mess* field is computed by the *Read_N_Mess* block, which takes as input the value of the 100 [ms] task counter.
This block's internal structure is the same as the one presented by the *N_Acc* block: a first level containing an 'if' structure, piloting two subsystems. However, in this case, the subsystem that increments the value under consideration is the one piloted when the counter's value is above 1, as the *N_Mess* value should increase with each sent message, that is, each iteration of the system.
Moreover, the subsystem executed when the value of the counter equals 1 does not read the value previously stored in memory, as it was for *N_Acc*, but rather resets that value to 1, to signal the beginning of a new cycle of messages with the new startup.

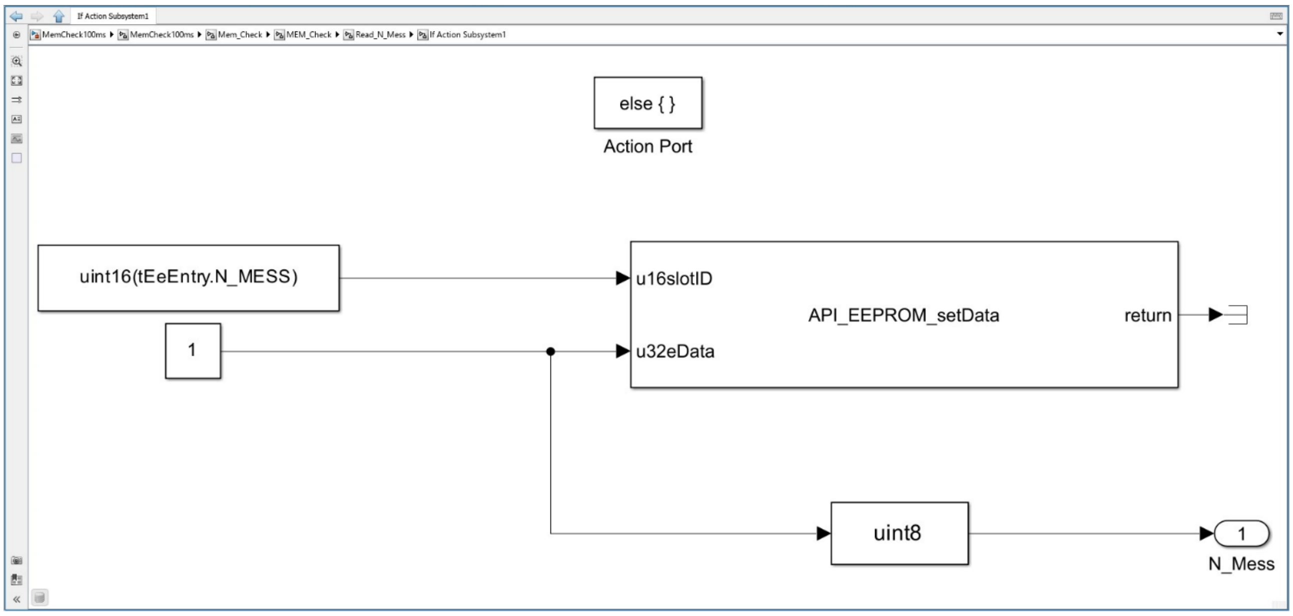


Figure 4.5.11 - Read_N_Mess subsystem's else condition block

The last field for the outgoing CAN message corresponds to the checksum, *CRC,* received from the upper level of the model.

4.5.2. LabVIEW

The LabVIEW virtual instrument designed to automate this test had to send the defined *VAL* and *REPS* to the board via CAN, store the values received from the board in the correct files, and manage the reboot cycles of the system.

As before, we designed a GUI to simplify the selection of the parameters and to have a way to check the current status of the system at a glance.

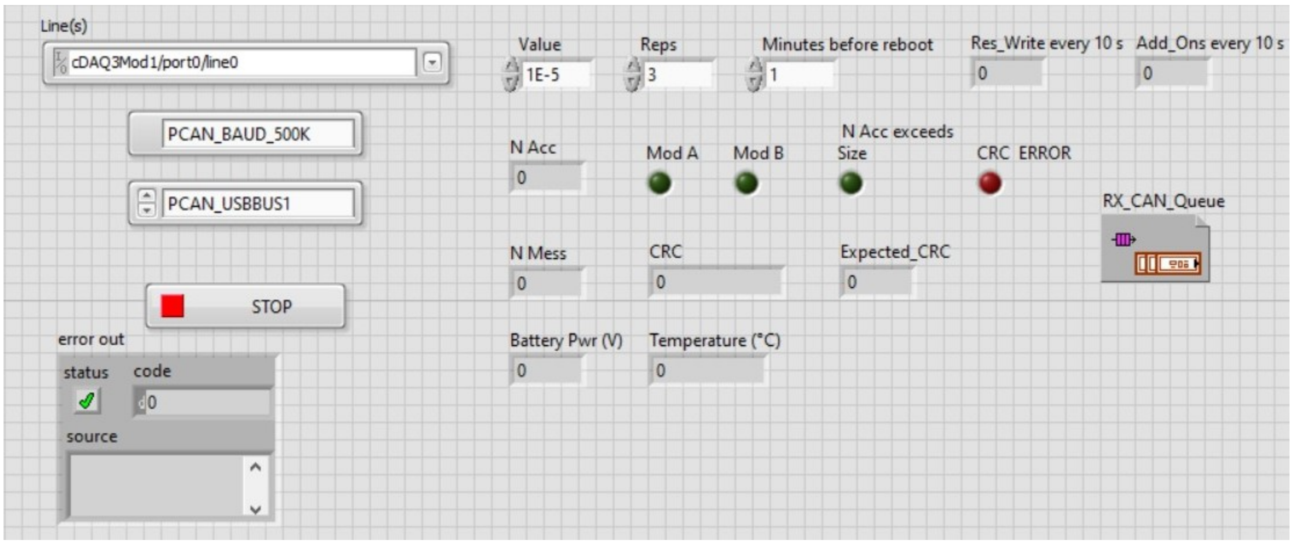

Line(s): cDAQ3Mod1/port0/line0
PCAN_BAUD_500K
PCAN_USBBUS1
STOP
error out
status ✓ code 0
source
Value 1E-5
Reps 3
Minutes before reboot 1
Res_Write every 10 s 0
Add_Ons every 10 s 0
N Acc 0
Mod A ●
Mod B ●
N Acc exceeds Size ●
CRC ERROR ●
RX_CAN_Queue
N Mess 0
CRC 0
Expected_CRC 0
Battery Pwr (V) 0
Temperature (°C) 0

Figure 4.5.12 - LabVIEW VI interface for the second test

The GUI can be used to select the *VAL* and *REPS* values to send, the number of minutes between reboots, the DAQ output port, and the CAN baud rate and bus.
Moreover, it shows the number of startups, the readings for battery power and temperature, and the number of messages sent since the last startup.
It also shows the currently active memory module and the exceed flag using LEDs. A red led marks, when turned on, the presence of a mismatch between the expected checksum and the one read from the board (both can be read using the *CRC* and *Expected_CRC* fields of the GUI).

The strategy of the virtual instrument can be split into three phases, as in the previous test: an initialization phase, a set of loops managing the actual test's body, and a clean-up phase.

The first part is executed only once, at the beginning of the test, to set up the system for the oncoming operations. This part is essential identical to the other test's, except for the different names used for the files' headers.

It starts by creating a new, empty queue of CAN messages to store the ones received from the board. It then assigns the queue as the destination for the CAN-read instruments, uninitialising (to clean up any leftovers from previous tests) and reinitialising the selected CAN channel with the appropriate baud rate (both of which are supplied via GUI).

The program proceeds to reset the counter variables used by the loops and starts operating on the files that will be used to record the values received via CAN.
It opens a “**RES**” file to record the main values read from the memory and an “**ADDS ON**” file for the values of power and temperature.
It proceeds by writing on each file a header line with the columns' names. For the “ADDS ON” file, the only two columns were **Battery Pwr(V)** and **Temperature(°C)**. For the other file, the header contained **N_Acc**, **Memory Module**, **Exceed Flag**, **N Mess**, and **CRC**.

The instrument goes on to initialise the DAQ digital output channel to control the board’s switch and uses it to turn on the board.

The main part of the virtual instrument is made of four loops: one to manage the writing of the CAN messages sent to the board, another to receive the messages from the board and insert them in the correct queue, a third to take the enqueued messages and interpret and record them, and a last loop to manage the reboot of the board.

The first loop writes a message, with ID 0x170, containing *VAL* and *REPS*, to send to the board.
It takes the *VAL* from the “value” field of the GUI as a single-precision value, applies the resolution (multiplying by 10^5, inverse of precision), and then splits the value into 8-bit parts, inverting the byte order (as the board expects INTEL endianness). It then takes the *REPS* from the corresponding GUI’s field as a 16-bit unsigned integer, splits it, and inverts the byte order.
The loop then waits a specified amount of time (here, 150 [ms]) before executing again to avoid overloading the channel.

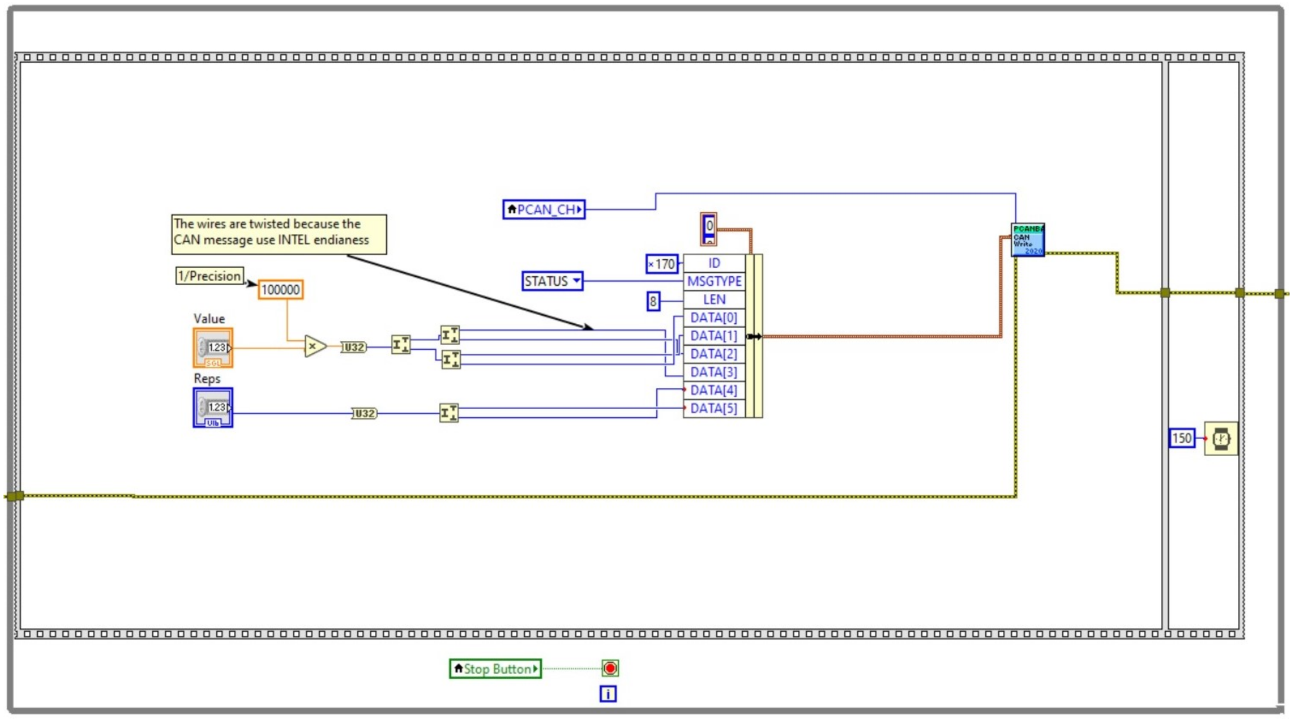


Figure 4.5.13 - First loop, to send the VAL and REPS message

The second loop manages the enqueuing of the received messages with the expected IDs (0x171, 0x172). It uses counters to ensure that only one message per type is stored every 10 seconds.

The queue is shared between the two types of messages; however, the counters are distinct.

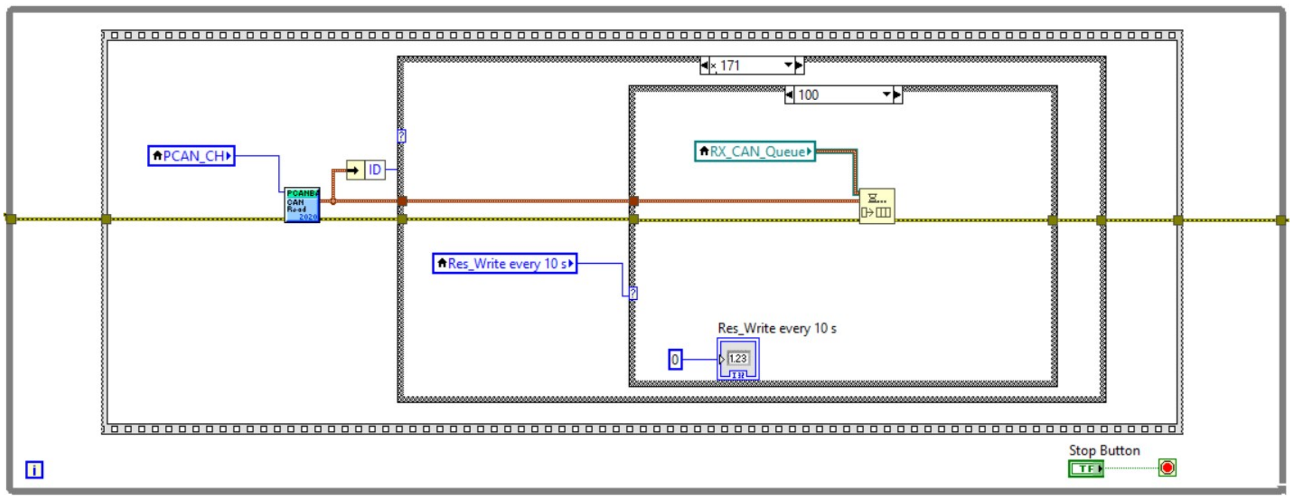


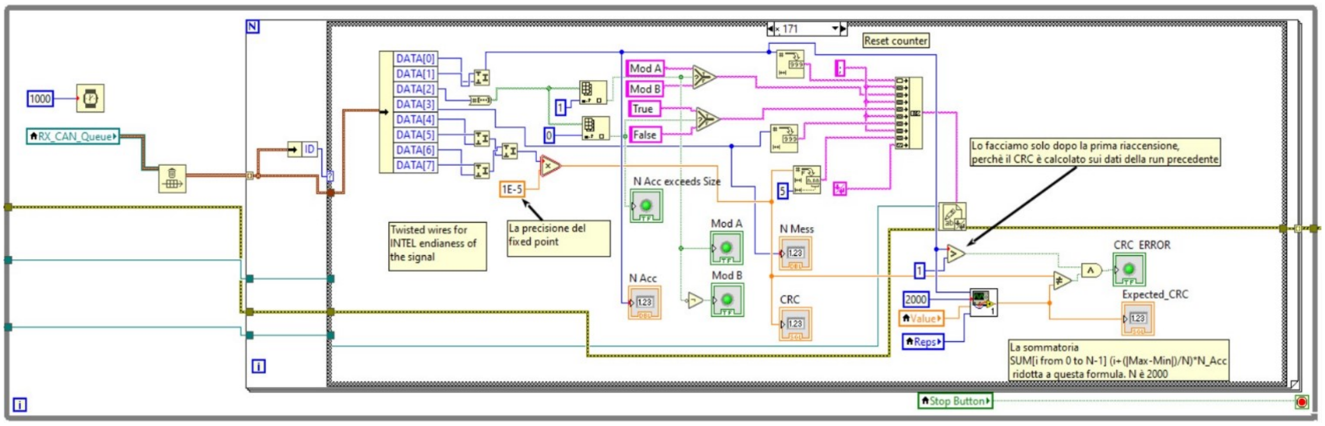
Figure 4.5.14 - Second loop, enqueuing a message with id 0x171

The third loop is executed once every second to read from the queue and write the message to the corresponding file based on the ID.

For the 0x171 messages, it reads the first two bytes as uint16 (*N_Acc*), the third byte as a boolean array (of which only the first two positions are used, for the memory module and the *exceed flag*), and the fourth byte as a uint8 *N_Mess*. These values are shown on the GUI in the form of LEDs (*exceed flag, memory module*) or directly as numeric values.

The last four bytes of the message are recomposed as a single 32-bit unsigned integer and then multiplied by the precision (here, 10^{-5}) to obtain the checksum, *CRC,* as single-precision with the desired accuracy.

In every occurrence of byte-recomposition, the bytes are reordered, as the board produces them in INTEL endianness while LabVIEW expects them as Big-endian. The values obtained are then converted to string and printed to file.



Twisted wires for INTEL endianess of the signal

La precisione del fixed point

Lo facciamo solo dopo la prima riaccensione, perchè il CRC è calcolato sui dati della run precedente

La sommatoria SUM[i from 0 to N-1] (i+(|Max-Min|)/N)*N_Acc ridotta a questa formula. N è 2000

Figure 4.5.15 - Third loop, when reading 0x171 messages

The obtained CRC is then compared with the *CRC* computed by means of a sub-virtual instrument that takes *VAL*, *REPS*, and *N* and returns the expected *CRC*.

The checksum is computed following the same process done on the board; it starts by computing *OFF* from *N* and *N_Acc*, then *Q*, and subsequently *X.* At this point, the board and the LabVIEW program diverge: while the board computes the sum of values in memory, the virtual instrument is forced to compute the remaining values one by one. It does so by implementing the formulas seen before, with a case for when *X* is lower than *REPS* and another for when *X* is included between *REPS* and twice that.

In the event of a mismatch between the received CRC and the one computed by LabVIEW, a red light on the GUI is turned on.

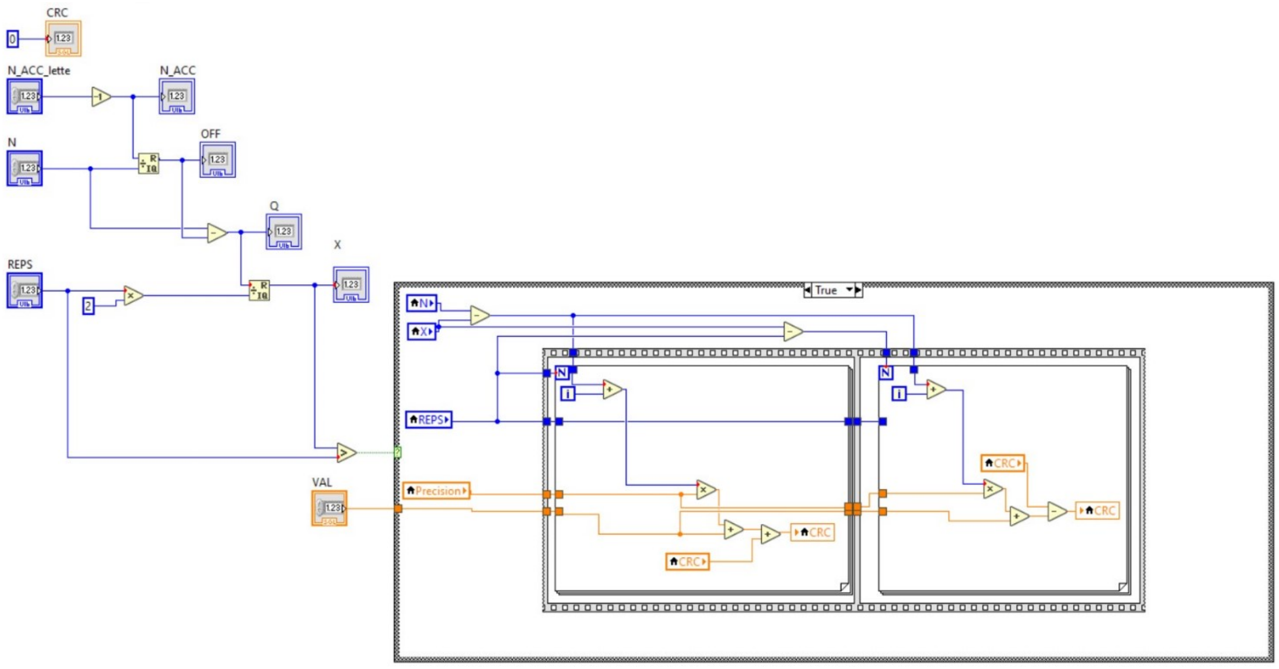


Figure 4.5.16 - Checksum computation sub-instrument

For the 0x172 ID case, it reads the first 4 bytes as battery power (V) and the other four as temperature (°C). The bytes are recomposed as 32-bit signed integers and then multiplied by the desired accuracy (0.01). Finally, they're recorded in the "*ADDS ON*" file. The scheme is identical to the one from the previous test.

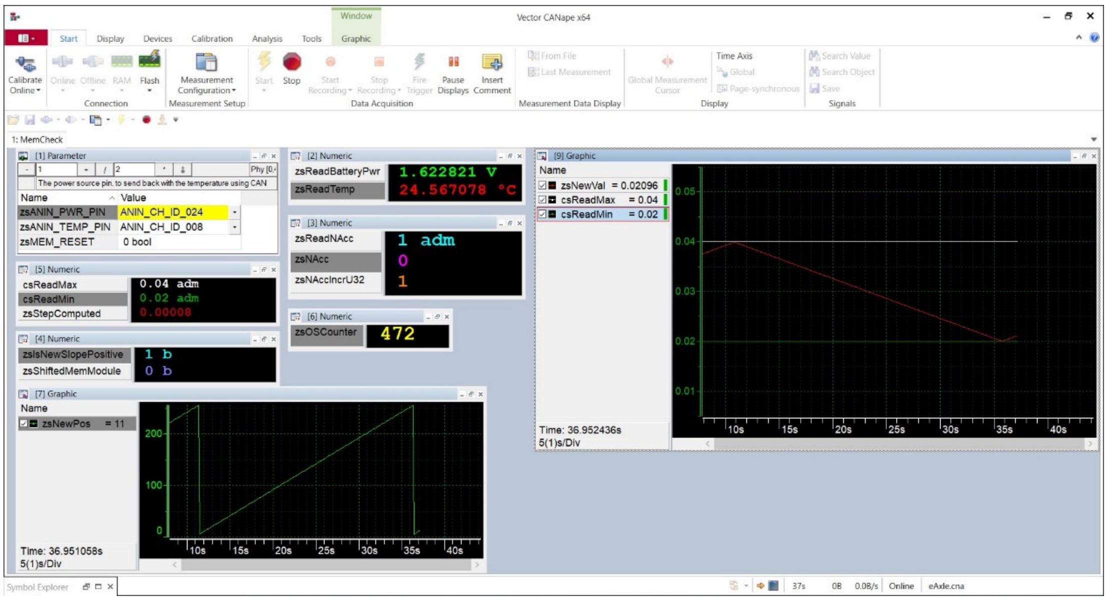
The fourth loop is the one responsible for rebooting the board. It sets the digital output on the DAQ to zero, awaits 10 seconds and then sets the output back to 1, thus shutting down and turning on again the board. Once again, it is identical to the corresponding one seen in the previous case.

As with the initialisation section, the third part of the virtual instrument is executed only once, at the end of the tests, once the tester pushes the stop button on the GUI. This will terminate all previous loops and switch the program execution to this final section, though not instantly, as the current iterations of the loops will continue normally.
This part's tasks are all connected to instrument cleanup; it uninitialises the CAN channel, resets the stop button, and powers down the board. It also displays any errors generated by the system thus far.
Just like the initialisation part, this section's scheme is identical to the corresponding one in the previous test.

4.6. Results and considerations

The two tests, “Mem Test” and “Mem Check”, were executed, respectively, for 8 hours and 4 days, for a total of around 500 and 6000 startup cycles. This roughly equals an estimated usage of 2 and 20 years’ worth of workday stress that are quite common durabilities for an Heavy Duty vehicle application.

In addition to the data saved to file by the LabVIEW virtual instruments, data recordings were also performed using CANape, which provided an additional visual representation of the running tests.

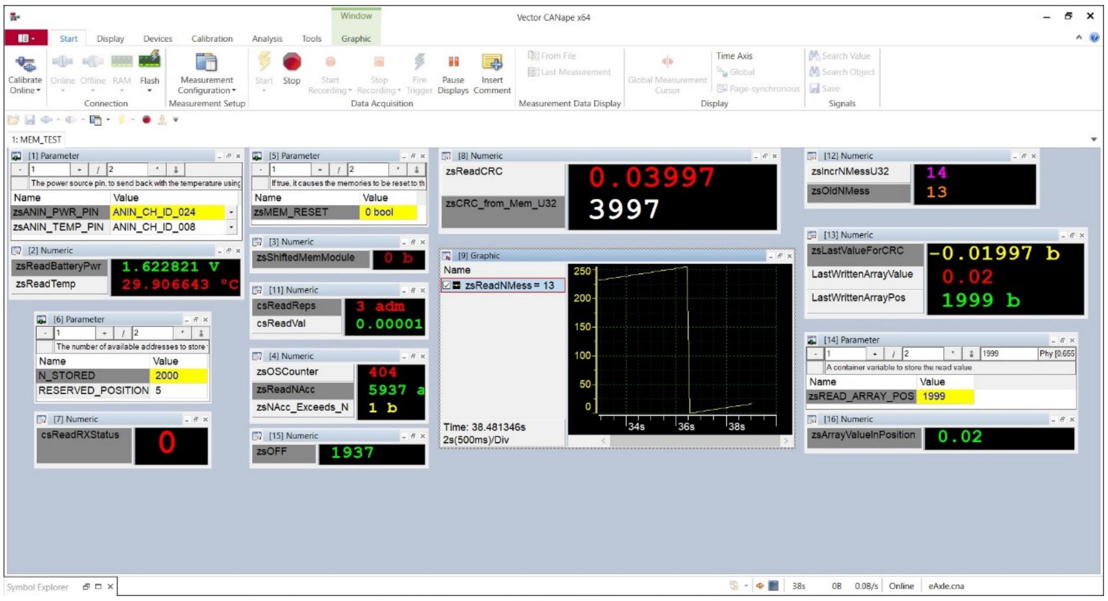


Name	Value
zsANIN_PWR_PIN	ANIN_CH_ID_024
zsANIN_TEMP_PIN	ANIN_CH_ID_008
zsMEM_RESET	0 bool
zsReadBatteryPwr	1.622821 V
zsReadTemp	24.567078 °C
zsReadNAcc	1 adm
zsNAcc	0
zsNAccIncrU32	1
zsOSCounter	472
csReadMax	0.04 adm
csReadMin	0.02 adm
zsStepComputed	0.00008
zsIsNewSlopePositive	1 b
zsShiftedMemModule	0 b
zsNewVal	= 0.02096
csReadMax	= 0.04
csReadMin	= 0.02
zsNewPos	= 11

Time: 36.952436s 5(1)s/Div

Time: 36.951058s 5(1)s/Div

Figure 4.6.1 - "Mem Test" CANape acquisition screen



Vector CANape x64
1: MEM_TEST
[1] Parameter
The power source pin, to send back with the temperature using
[5] Parameter
If true, it causes the memories to be reset to th
[6] Parameter
The number of available addresses to store
[14] Parameter
A container variable to store the read value
[9] Graphic
Symbol Explorer
Online
eAxle.cna

Name	Value
zsANIN_PWR_PIN	ANIN_CH_ID_024
zsANIN_TEMP_PIN	ANIN_CH_ID_008
zsMEM_RESET	0 bool
zsReadBatteryPwr	1.622821 V
zsReadTemp	29.906643 °C
N_STORED	2000
RESERVED_POSITION	5
csReadRXStatus	0
zsShiftedMemModule	0 b
csReadReps	3 adm
csReadVal	0.00001
zsOSCounter	404
zsReadNAcc	5937 a
zsNAcc_Exceeds_N	1 b
zsOFF	1937
zsReadCRC	0.03997
zsCRC_from_Mem_U32	3997
zsReadNMess	= 13
zsIncrNMessU32	14
zsOldNMess	13
zsLastValueForCRC	-0.01997 b
LastWrittenArrayValue	0.02
LastWrittenArrayPos	1999 b
zsREAD_ARRAY_POS	1999
zsArrayValueInPosition	0.02

Time: 38.481346s 2s(500ms)/Div

Figure 4.6.2 - "Mem Check" CANape acquisition screen

The recorded material was then used to produce graphs illustrating the behaviour of the system to analyse it and identify possible errors. Different values were studied together, like the active memory module and the *N_Acc*, to better detect issues that could go unnoticed on their own.
Save for the initial debug phase, as expected, once the programs were tuned up, the behaviour recorded during the tests matched the expected one.

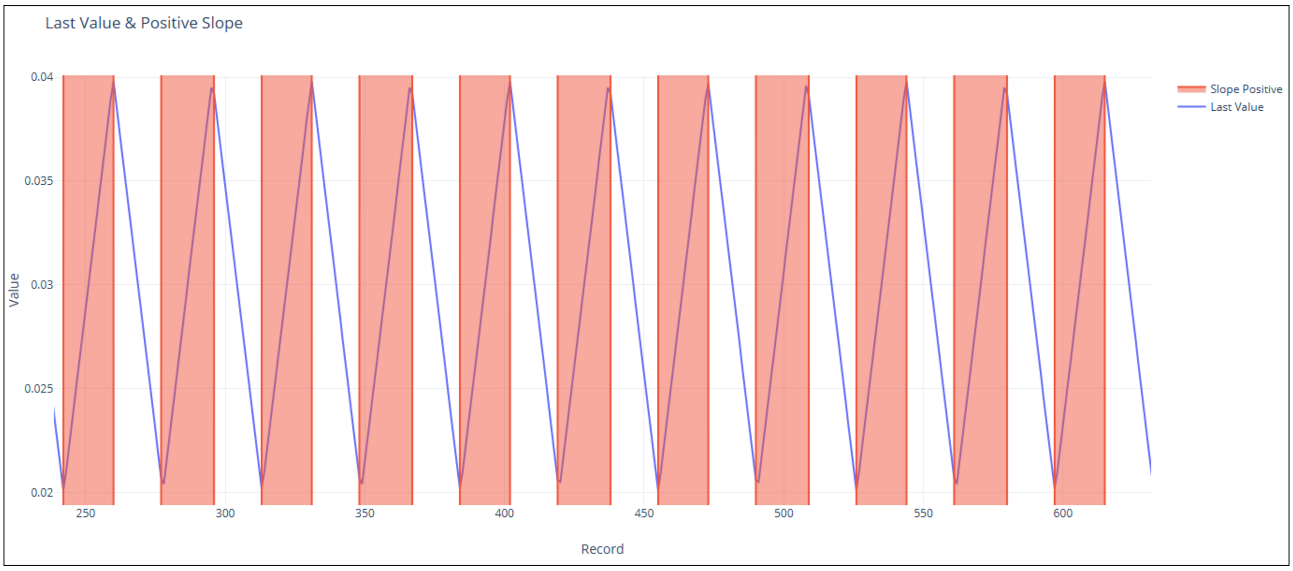


Figure 4.6.3 - Example of the extracted data in graph form, comparing the behaviour of the slope and the last value

For the aim of this Thesis these tests have been considered sufficient to provide us with a general insight into the eventual solution's strengths and shortcomings, such as the use of enumeratives to directly index the memory cells.
The "reserved positions", associated with enum labels, allowed for a mnemonic link that proved to be quite useful during the modelling phase. However, it would certainly come in handy to have a way to set up the model's components when having to work with enums, as manually inserting them could be tedious.
In the second part of this thesis, a more extensive use of enumeratives to favour immediate understandability of the code will be combined with the usage of custom-made Simulink support tools, such as *masks,* to reduce the workload linked with this strategy.

In conclusion, by working on this aspect of the thesis, it was possible to acquire a comprehensive understanding of the system, code, processes, and tools. This knowledge was essential for efficiently addressing the second and last part, centred around the On-Board Diagnostic.

5. Part 2 – Diagnostics (OBD)

On-board diagnostics (OBD) is an automotive term referring to a vehicle's self-diagnostic and reporting capability. OBD systems give the vehicle owner or repair technician access to the status of the various vehicle subsystems.

The second and main part of this thesis work was the redefinition of the on-board diagnostic strategies already present on the HDS9 board.
The goal was, once again, to produce a solution as versatile as possible and less complex than the current one while following the most prominent guidelines of the automotive sector.

The first step to reaching the goal was to analyse how the current system performs the OBD, focussing on the general adopted strategy while avoiding the actual implementation so as not to influence possible future choices.

After that, it came time to study the standards and guidelines currently in use to better define the set of characteristics and requirements that the OBD system should meet.

Once the requirements were well defined, it was time to re-design the strategy and its components, starting with the diagnosis flow.

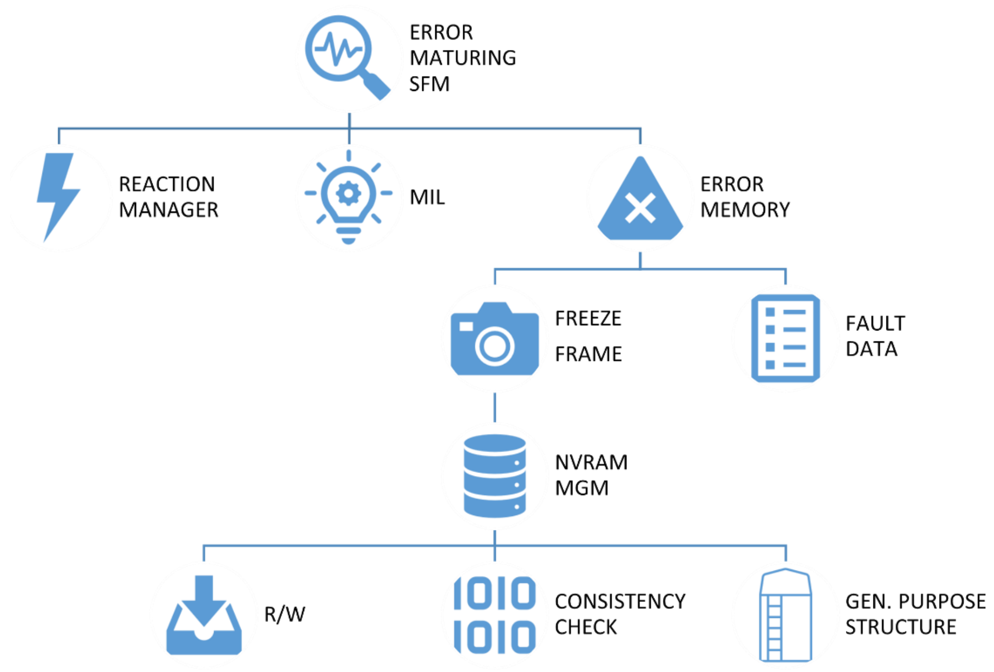


Figure 5.1 – Diagnostic System architecture showing modules interactions

After a general design of the global strategy, the work moved to the refining and implementation of the single component.
As the realisation of the various parts went on, mindful of what was learnt from the work on the NVRAM, a little time went to the design and implementation of Simulink *masks* to facilitate the work and the models' setup.

Following the well-known V-Shape approach, the components were tested as single modules (unit testing); as the components were finished, they were tested combined (integration testing). To perform the tests, various Simulink models were designed and loaded onto the board (Hardware in the Loop).
Not only that, but the core part of the OBD system, the error validation finite state machine, named **ADIA** (**A**utomatic **DIA**gnosis), was tested on pre-existing small projects thanks to the collaboration of Metatron's designers (real-case usage). This provided important information on both the behaviour of the final strategy under working-level stress and eventual compatibility issues with other base-level software and APIs of Metatron's products.

The following paragraphs will provide a more detailed description of the examined standards, the diagnosis strategy adopted, and the design and implementation of its various components.

5.1. Standards & Diagnostic Requirements Introduction

5.1.1. Introduction to the standards

On-board diagnostics has many upsides: it helps with emission controls, ensuring that vehicles comply with emission regulations by monitoring and reporting on the performance of emission-related components, supports vehicle maintenance, improving performance and longevity, and increases safety by detecting and reporting on critical issues that may affect the safety of the vehicle and its passengers.

The amount of diagnostic information provided via OBD has varied greatly since its introduction in the early 1980s versions of on-board vehicle computers; early versions of OBD would simply turn on a malfunction indicator light if a problem was detected but would not provide any additional information about the nature of the issue.

Modern OBD implementations use a standardised digital communications port to deliver real-time data, as well as a standardised series of diagnostic trouble codes, or DTCs, to quickly identify and remedy malfunctions within the vehicle.

With the growth in the amount of information that OBD systems are able to provide, the need for standardising the communication of this information has also increased.
To address this need, various standards have emerged over the years, starting with **OBD-I** in the 1980s, which laid the groundwork for electronic diagnostics in vehicles.
This was followed by **OBD-II** in the mid-1990s, born in the United States, which introduced a more uniform and comprehensive approach, including standardised connectors, protocols, and diagnostic trouble codes. **EOBD** is the European equivalent of OBD-II.

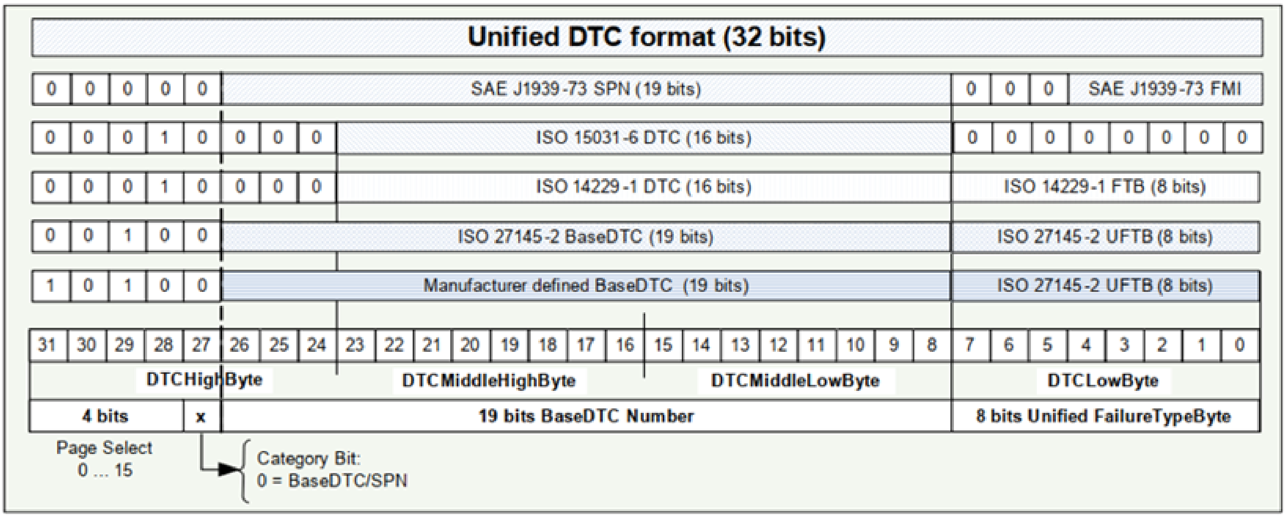
Different protocols for different markets cause incompatibilities. The same wave of standardisation of automotive components that witnessed the advent of AUTOSAR and Unification of Diagnostic Services (**UDS**), pushed by organisations like ISO and SAE, established the harmonisation of the OBD protocols as the next logical step.
The World Wide Harmonised On-Board Diagnostics (**WWH-OBD**), based on the OSI 7-layers model, was conceived as an answer to this growing need for harmonisation.

Another set of standards for diagnostics (and communication) among vehicle components is **SAE J1939**. Developed by the Society of Automotive Engineers (SAE), it is less focused on emission-related diagnostics and global standardisation than WWH-OBD and more tailored for heavy-duty vehicles with a broader scope of diagnostic capabilities.

Vehicle manufacturers are obligated to implement a WWH-OBD capable diagnostic system. As such, the HDS9 ECU implements both the OBD services, with both protocols **J1939-73** and **ISO 15765** (UDS), and the EOBD ones, with the WWH-OBD protocol (**ISO 17145**). However, only the WWH-OBD protocol is fully available, while UDS and J1939 (specific for heavy-duty systems) have only the main services implemented.

A fundamental concept of on-board diagnostics used for the before-mentioned protocols are the **diagnosis lines**: electrical or functional diagnoses associated with a system component or system functionality. A diagnosis line is identified by a Diagnostic Trouble Code, usually referring to the component or function under test (**SAE code number**) and the kind of malfunction/ symptom (**FTB**). WWH-OBD introduces the Unified

DTC format and provides ways to map from other DTC formats into this one in the ISO 27145-2 document.


Unified DTC format (32 bits)
0 0 0 0 0 | SAE J1939-73 SPN (19 bits) | 0 0 0 | SAE J1939-73 FMI
0 0 0 1 0 0 0 0 | ISO 15031-6 DTC (16 bits) | 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 | ISO 14229-1 DTC (16 bits) | ISO 14229-1 FTB (8 bits)
0 0 1 0 0 | ISO 27145-2 BaseDTC (19 bits) | ISO 27145-2 UFTB (8 bits)
1 0 1 0 0 | Manufacturer defined BaseDTC (19 bits) | ISO 27145-2 UFTB (8 bits)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DTCHighByte
DTCMiddleHighByte
DTCMiddleLowByte
DTCLowByte
4 bits
x
19 bits BaseDTC Number
8 bits Unified FailureTypeByte
Page Select
0 ... 15
Category Bit:
0 = BaseDTC/SPN
Figure 5.1.1 - Mappings to Unified DTC format

The J1939 protocol also uses DTCs, although with a slightly different structure from the one seen until now, as they are typically 4 bytes divided as follows:

- Bytes 1-2: Suspect Parameter Number (SPN), which identifies the specific parameter or component that has an issue.
- Byte 3: Failure Mode Indicator (FMI), which describes the type of failure (e.g., short circuit, open circuit).
- Byte 4: Occurrence count or additional context about the fault.

As with OBD and EOBD, WWH-OBD implements diagnostic services using UDS protocols over CAN. The reserved CAN channel for this kind of communication on the HDS9 is CAN3, and the HDS9 is able to automatically detect the protocol to be used once a diagnostic tool is connected to CAN3 and the connection is started.
The HDS9 also implements some diagnostic services for DM standards using the J1939 bus on the HDS9-CAN2 channel to make some diagnostic information available.

In addition to the mandatory implementation of WWH-OBD, all the new vehicle registrations for heavy-duty vehicle must conform to the requirements of the Euro-VI (European market) or China-VI (Chinese market) emissions standard starting in 2014.

Euro VI and China VI are emission standards aimed at reducing pollutants from heavy-duty vehicles. Euro VI, implemented in the European Union, sets strict limits on nitrogen oxides (NOx) and particulate matter (PM) emissions.

It incorporates real-driving emissions (RDE) testing and portable emissions measurement systems (PEMS) to ensure compliance under real-world conditions.

China VI, on the other hand, is a two-phase standard implemented nationwide in China. The first phase, VI-a, is largely equivalent to Euro VI, while the second phase, VI-b, introduces even more stringent requirements.
China VI combines best practices from both European and U.S. standards (such as CARB, California Air Resources Board), with additional measures like remote emission monitoring.

Both standards aim to address severe air pollution issues by enforcing stricter limits on NOx and PM emissions and promoting the use of cleaner technologies in heavy-duty vehicles. To do so, these standards include comprehensive OBD guidelines.
The On-Board Diagnostics systems in Euro VI vehicles are designed to ensure compliance with emission standards throughout the vehicle's lifespan by continuously monitoring the performance of emission-related components and, if a fault is detected, triggering a warning light on the dashboard while storing a Diagnostic Trouble Code (DTC) that can be read with an OBD scanner.
China VI has OBD guidelines comparable to Euro VI but with additional features customised to local standards, such as remote emission monitoring capabilities that allow authorities to follow vehicle emissions in real time.

5.1.2. Introduction to the requirements

The before-mentioned standards are made of multiple sections, called in different ways (e.g., part, annex), and exist in different versions as they are periodically updated.
For this thesis, the parts and versions that were studied to design the proposed strategy and its implementation were:

- WWH-OBD: **ISO 27145-4:2016**
- J1939: **J1939DA, January 2020 revision**
- Euro VI: **Addendum 48, Regulation No. 49, Revision 7**
- China VI: **GB 17691-2018, Annex F**

Out of all of them, the main sources for the requirements were the Euro VI and China VI documents.

The main requirements extracted from the documents and used as the basis for the design of the system were inherent to:

- DTCs formats

- Classifications of different fault types
- Differential memorisation and management of the various classes of faults
- Mandatory engine data to store in the presence of certain fault types (**freeze frame**)
- Mandatory memory operations, both automatic and on external tool requests
- Memory behaviour for certain requests based on the stored fault classes
- Timings and periods of the Malfunction Indicator Lamps (**MIL**) for the different types of faults and depending on the engine status

These must be added to all the requirements proper to the actual Metatron system, that is to say, all those linked to the pre-existing code structure (architectural requirements), the underlaying base-level software, and the final process with which the users are able to utilise the provided APIs for the model-based design of their own applications.

Further elaboration on the requirements will be included in the subsequent sections in which they have been implemented.

5.2. Diagnosis Flow

As stated in this chapter’s introduction, this part's work started with the analysis of the OBD as it already existed on the HDS9 system, without looking at its implementation but analysing the general strategy adopted.

The existing strategy adopted a well-refined flow of actions, from the detection of a possible issue to the activation of recovery actions. For each diagnosis line, corresponding to a possible fault (a combination of a component and a specific malfunction), fault management followed this process, with each part tailored to the specific parameters of the fault.

The diagnosis flow started with the board’s hardware sensors and actuators that read the physical values and could communicate them to the **fault test check routines**.
These user-defined routines, which could be enabled via certain customised **conditions**, took those values, compared them with the user-defined expected values, and produced an **error code**.
These error codes were then fed to the **finite state machine for the filtering and validation of the error code**, the ADIA. The automaton updated its status and managed the activation of the **recovery** and the management of the **error memory** and the **lamps**.
The actual implementation of the recovery strategies was delegated to the user, with the ADIA simply setting the global flags (**recovery lines**) to pilot those recovery functions.
The error memory and the MILs were comprised in the same step of the flow. Moreover, they were actually embedded into the ADIA, rather than simply having them piloted by it.

To this flow of actions conducted at runtime by the system to perform the OBD, we must also add the tedious and time-consuming steps that were needed by the customers to define the parameters for the system to use:

1. Update the various spreadsheets containing the parameters of the diagnosis line to modify and the ones related to the recovery lines to activate when that fault is detected.
2. Execute the macros on those spreadsheets to produce parameter “dataset” files, used to update the calibration dataset of the ECU.
3. Load those files onto the board and test them.

Aside from the parameters set, the actual diagnosis flow as defined was a solid starting point that only needed a few tweaks.
As always, the focus was on improving the modularity of the system and increasing its usability. To do so, the flow was changed to split the error memory and the MIL management. The separation between the finite state machine and these two was also increased, although this can only be seen at the implementation level.

The resulting diagnosis flow can be seen in the following picture, where the parts that include user-defined functions are encircled by a dotted line:

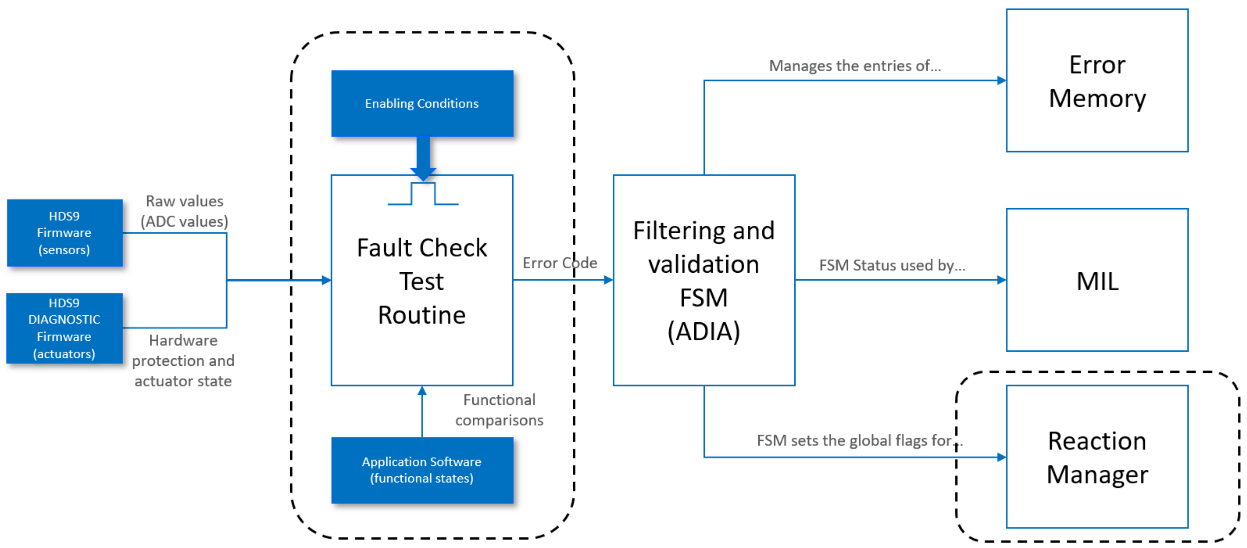

HDS9 Firmware (sensors)
HDS9 DIAGNOSTIC Firmware (actuators)
Raw values (ADC values)
Hardware protection and actuator state
Enabling Conditions
Fault Check Test Routine
Application Software (functional states)
Functional comparisons
Error Code
Filtering and validation FSM (ADIA)
Manages the entries of...
Error Memory
FSM Status used by...
MIL
FSM sets the global flags for...
Reaction Manager

Figure 5.2.1 - Diagnosis flow

The process of defining the parameters for the diagnosis lines has also been simplified with the introduction of Simulink mask blocks that can be used to intuitively select the preferred values for the single fault.
This solution removed the need for external spreadsheets to produce the “dataset” files to map the parameters to the various automata and instead incorporated their selection at the model-based design level.

5.3. ADIA Finite State Machine

The filtering and validation finite state automaton is the core of the whole diagnostic process, as it performs several tasks ranging from the corroboration of the fault's presence to the management of the memorisation of its information and the activation of the recovery procedures.

Due to the strategic importance of this component, to protect the company's know-how, many implementation details of this section will be left out.

5.3.1. Requirements

Starting from the documents (both regulations and Metatron's own internal documentation), we collected the following set of requirements linked to the diagnosis line:

- Each fault (seen as a combination of the component and the specific malfunction) should have its own associated DTC.
- Each detected malfunction should exist in a defined state; according to the WWH-OBD standard, the available states are:
    - **NO ERROR**: the fault is not active, nor has its presence been detected.
    - **PENDING**: the fault's presence has been detected, but the system is not yet sure whether it has been a false positive due to some electrical spikes. Some filtering/debouncing is needed.
    - **CONFIRMED AND ACTIVE**: the system has determined the detected fault to be actually present after a certain time in the *pending* status has passed. The fault has been stored in the **error memory** along with a snapshot of the current engine status, called **freeze frame**.
    - **PREVIOUSLY ACTIVE**: the fault is no longer active, but its presence is still recorded in memory. A malfunction must remain in this status for a determined number of warm-up cycles before being automatically erased from the *error memory*.
- A fault determined to be in the *active state* must be stored in the *error memory*. This memory must be of the non-volatile type.
- Certain faults must be permanently stored in memory once detected as active. This means they shall not be erased via external tools or on memory reset until the system itself has determined that the fault is no more.
- Each fault must be categorized into one of the available classes: **A**, **B1**, **B2**, and **C**:

    - Class A faults are linked to emissions-related problems, signal malfunctions that could raise emissions levels, and are typically required to be monitored continuously. These codes are crucial for meeting environmental regulations.
    - Class B1 DTCs focus on performance issues that could affect the vehicle's drivability or safety. Although less critical than Class A, they still need attention and may involve problems with sensors, actuators, or other performance-related components.
    - Class B2 DTCs address non-critical issues that don't immediately impact safety or emissions but may point to potential problems, assisting in diagnosing and tracking the vehicle's condition over time and supporting proactive maintenance.
    - Class C DTCs are manufacturer-specific codes offering additional diagnostic details. Unlike standardised codes, they vary across manufacturers and usually pertain to features or systems unique to a specific brand or model.
- Faults belonging to different classes will make the system behave differently. More details on this topic can be found in the requirements subsections of the Error Memory and Freeze Frames and MIL Management paragraphs.

5.3.2. Strategy

As previously stated, to perform the validation of the presence of a fault, the selected strategy was to use an automaton. For each diagnosis line, then, a set of parameters had to be defined to allow the correct functioning of the line's FSM.

Originally, the strategy made use of a spreadsheet with a row for each diagnosis line. The columns of these rows included various fields connected to the fault at various degrees, ranging from the global ID to the lamp settings when the fault was active. Among these parameters, there were many connected to the ADIA, and even some that were used for the symbolic mapping once the spreadsheet had been converted and loaded on the board.

The first step to outline the new strategy was to discard all those parameters that were only loosely related to the ADIA or the fault's classification and keep the most important ones; this would help to increase the modularity of the system by better dividing the various parts of the diagnostic flow.

From the original parameters present in the spreadsheets (~30), the only ones that directly belonged to the diagnosis line and thus were preserved were:

- The identifier of the fault inside the system, a.k.a. the name of the diagnosis line
- The class of the fault
- The fault's severity
- Whether to make it a permanent entry in memory or not
- The DTC code, in J1939 and WWH-OBD format
- The list of recovery lines to activate when the fault is detected

Other parameters that were preserved, although separately from the diagnosis line, were the thresholds used by the fault's ADIA to change from one state to another, that is, the number of times that the ADIA must receive an "error signal" or must pass without such a signal to change the ADIA's internal status.
In addition to those, the parameter to select the step by which to increase the threshold's counter was also kept. The presence of this parameter allows the customer to better regulate the amount of time the fault has to be detected to change status.

Originally, the ADIAs had many more parameters. This overabundance of parameters was partially due to the fact that the previous strategy was comprised of different types of ADIAs, and each diagnosis line had a corresponding ADIA of one of those types (A, B, C, D).

The size of these finite state machines grew from the D class moving to the A, with a growing number of available states used to perform debouncing and refiltering operations.
As the states used by the ADIAs did not always correspond to those required by the guidelines, a mapping was introduced to convert the additional states to the expected ones, increasing the complexity of the system.
For example, the largest FSM, those of class A, were composed of nine states, that is, five intermediate states in addition to the ones provided for by the standards.

To reduce the complexity of the original strategies, it was decided to converge all the different types of ADIAs into a single one that was able to manage all the requirements expressed by the consulted guidelines. As such, the starting point was not the existing ADIAs but the Euro-VI and China-VI documents that were used to determine the states, the conditions to move from one state to the other, and the events that such transitions brought along.

After producing a basic finite state machine with this procedure, we went back to reanalyse the pre-existing ADIAs types to look for specifications related to Metatron's projects that were not present in the standards, selecting only those needed and discarding remnants from previous projects that went unused for a long time.

The final automata designed to perform the filtering and validation of the fault codes, manage the events triggered by the detection of such codes, and overall make up the core of the whole diagnostic system has four states: **NO_FAULT**, **PENDING**, **CONFIRMED_AND_ACTIVE**, and **PREVIOUSLY_ACTIVE**.

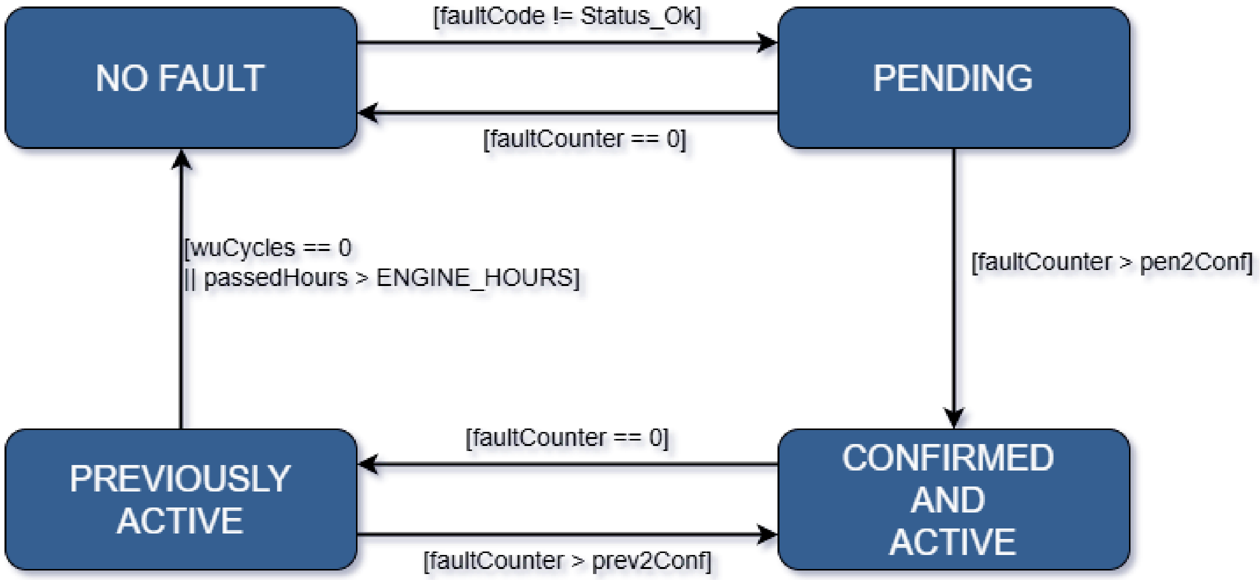


Figure 5.3.1 - New FSM scheme, extremely simplified to protect the actual know-how

Initially, the ADIA begins in the *NO_FAULT* state. When the ADIA receives in input a message with a fault code, it increases its own fault counter and then moves to the *PENDING* state.

From here, if it receives a message containing a *STATUS_OK* code, it decreases the counter and, if it reaches zero, goes back to the *NO_FAULT* state.
If instead the ADIA receives messages containing fault codes, it increases its counter and, if it reaches a first threshold, tries to store the fault in the error memory and to move on to the *CONFIRMED_AND_ACTIVE* state.
During this transition, the recovery actions related to the ADIA's fault are activated, if any, and the counter value is set to a second threshold.

Once the ADIA is in the *CONFIRMED_AND_ACTIVE* state, on each message received containing an error code, the counter will be reset to the value of the second threshold.
If the received message does not contain an error code, the *faultCounter* will be decreased by one unit, and if the counter reaches zero, the ADIA's state will be updated to *PREVIOUSLY_ACTIVE,* and the error memory entry for the fault will be updated with the new status of the malfunction. Moreover, all the currently set recovery lines for the fault should be turned off after having made sure that no other active fault concerns them.

While the ADIA is in the *PREVIOUSLY_ACTIVE* state, receiving a sufficient number of error codes will bring it back to the previous state.
From the *PREVIOUSLY_ACTIVE* state, the ADIA has only two ways to go back to the *NO_FAULT* state: by external intervention with a diagnostic tool or when the system itself decides that the fault is no more. This second event happens when either a certain number of hours with the engine running (**engine hours**) without incurring the fault again have passed or after a given number of **warm-up cycles** have been completed without the fault returning active. According to the Euro-VI and China-VI standards, these equal 200 hours and 40 cycles.
When the ADIA moves to the *NO_FAULT* state, the corresponding entry in the error memory is erased.

It is important to note that, although each ADIA could theoretically receive in input different error codes and thus manage different malfunctions for a single component, the adopted strategy was to instantiate an ADIA for each specific combination of components plus malfunction. A diagnosis line in the system describes a component and one of the fault modes associated with it.
This choice was made because, while the other option saved more memory, this was closer to what the analysed standards and guidelines described. It also comes with the added benefits of making it easier to add a new diagnosis line and making it simpler to manage the identification of faults at the code level. It also simplified the task of discerning when two distinct faults occurred for the same component.

After the system's startup, the ADIAs had to restart where they stopped last time; thus, there was a need for a way to save the data of the ADIA at shutdown and reload them afterwards. Moreover, as we strived for the increase of the degrees of separation between the various components of the diagnostic system, we had to find a way to maintain the link between the diagnosis line and its ADIA, which was originally established via the parameters of the two. To solve these two needs, we came up with the idea of a single structure containing all the ADIAs that could be indexed using the global IDs of the diagnosis lines.

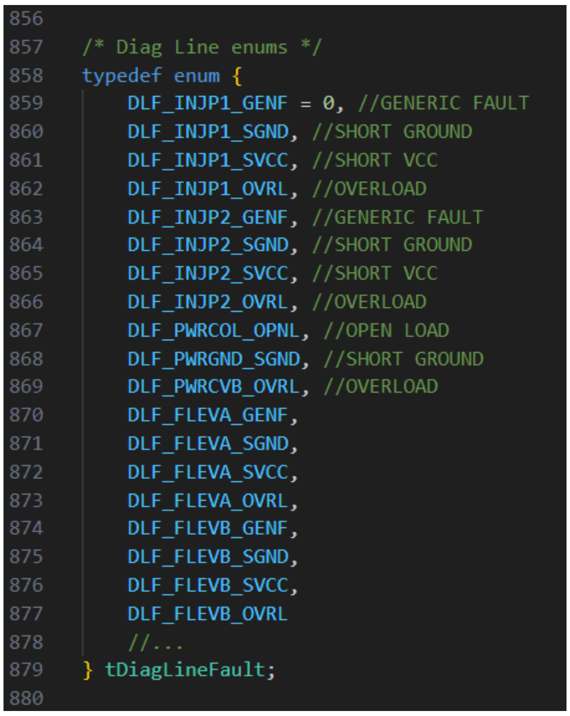
Studying the Euro-VI and China-VI guidelines’ documents, we also determined that the FSM should be in charge of managing the counters linked to the different classes of active faults; thus, the implementation of the ADIAs would also have to manage this task. These counters are used for the management of the malfunction indicator lamps and for some diagnostic requirements dictated by the documents.

Before moving on to the “implementation” part, the strategy defined until now can be summed up as:

- A diagnosis line is defined as a component and a single, specific malfunction.
- Each diagnosis line has a class, a severity, and other details. It is identified with its global ID.
- For each diagnosis line, there is an automaton, ADIA, in charge of filtering and validating the presence of the fault. These ADIAs should be instantiated at model level and executed on differently timed tasks depending on the needed update frequency.
- The input to the ADIA is obtained by a user-defined function that confronts the expected value of a certain measure with the actual one detected by sensors and produces an error code if this value is out of the intended range.
- The ADIA receives this value and updates its internal state, which corresponds to the fault state and can be any of four options.
- While updating its state, the ADIA also manages the error memory and the recovery lines (through APIs). It also manages certain counters used, for instance, by the MIL management and for other safety requirements linked to the classification of the active faults.
- All the ADIAs data are contained in a single structure, and the ADIA is linked to its diagnosis line via indexing.

5.3.3. Implementation

The first step of the implementation was the definition of the memory structures to host all the diagnosis line data and their corresponding ADIA's. To follow the same simple but efficient approach, more important than ever seen the number of R/W operations done on these structures at runtime, we opted to go with two parallel arrays, indexed using the global ID of the fault. The size of these arrays is determined by the #define directive **N_DIAGLINES**, which must be equal to the number of labels of the **tDiagLineFault** enumerative. This enum's values correspond to the global IDs and internal names of the faults.



```c

/* Diag Line enums */
typedef enum {
    DLF_INJP1_GENF = 0, //GENERIC FAULT
    DLF_INJP1_SGND, //SHORT GROUND
    DLF_INJP1_SVCC, //SHORT VCC
    DLF_INJP1_OVRL, //OVERLOAD
    DLF_INJP2_GENF, //GENERIC FAULT
    DLF_INJP2_SGND, //SHORT GROUND
    DLF_INJP2_SVCC, //SHORT VCC
    DLF_INJP2_OVRL, //OVERLOAD
    DLF_PWRCOL_OPNL, //OPEN LOAD
    DLF_PWRGND_SGND, //SHORT GROUND
    DLF_PWRCVB_OVRL, //OVERLOAD
    DLF_FLEVA_GENF,
    DLF_FLEVA_SGND,
    DLF_FLEVA_SVCC,
    DLF_FLEVA_OVRL,
    DLF_FLEVB_GENF,
    DLF_FLEVB_SGND,
    DLF_FLEVB_SVCC,
    DLF_FLEVB_OVRL
    //...
} tDiagLineFault;

```

Figure 5.3.2 – Tentative faults of the tDiagLineFault enumerative, in api.h

Both *N_DIAGLINES* and the entries of the enumerative can be modified by the users to suit their needs, with the only requirement being that the #define equals the enum's size. As such, there are no strict rules to define the format of the labels' names, but the form used for this thesis work was "DLF" (as in diagnosis line fault), followed by an indication of the component (e.g., INJP1 standing for injector 1), again followed by a term to define the malfunction (e.g., OVRL to indicate an overload), with the three terms connected by underscores.

The use of this enumerative to index both ADIAs and diagnosis lines' data sprouted from the observations on the results of the first part of the thesis and facilitate the management of the system at model level.

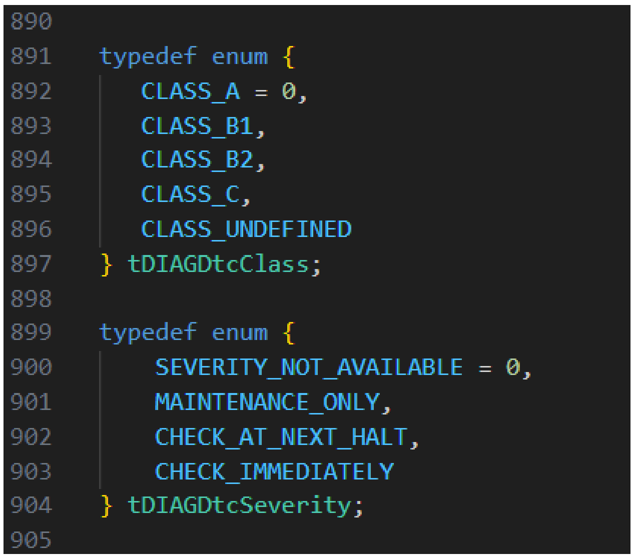
The **tDiagLine** structure defines the items that make up the array of diagnostic lines, each of which contains data from a single diagnosis line.

The first field of the struct is a boolean which determines whether the cell of the array is currently being used by the system. This field is needed because there could be instances in which the entry of the enumerative does exist, and thus so does the array cell, but in the system the fault is not being used and there is no diagnosis flow associated with that fault. This simple flag can then be used to skip these unused entries for certain computationally expensive operations, optimising the process.

The second field simply contains the global ID, the *tDiagLineFault* enum label, to identify the entry when the indexing correspondence cannot be used.

While the two fields above are determined by the system and cannot be modified by the user, the remaining six are calibratable and correspond to the parameters of the diagnosis line obtained by the requirements studies.
The first of them can assume one of the values defined in the **tDIAGDtcClass** enumerative. It defines the class of the fault, and it is used to perform some class-specific operations, such as certain lamp activation or different behaviours in the error memory.

The second calibratable field introduces the severity level of the fault, which can be used to prioritise a fault instead of another, for instance, to decide which one to preserve in the memory error. The possible values are listed in the **tDIAGDtcSeverity** enumerative.



```
typedef enum {
    CLASS_A = 0,
    CLASS_B1,
    CLASS_B2,
    CLASS_C,
    CLASS_UNDEFINED
} tDIAGDtcClass;

typedef enum {
     SEVERITY_NOT_AVAILABLE = 0,
     MAINTENANCE_ONLY,
     CHECK_AT_NEXT_HALT,
     CHECK_IMMEDIATELY
} tDIAGDtcSeverity;
```

Figure 5.3.3 - Class and severity enumeratives, in api.h

A fourth, boolean field is used to mark the fault as a permanent entry in the error memory, regardless of its class. It can only be used to force a fault to be stored as permanent, not to prevent it; as per Euro-VI and China-VI guidelines, class A entries will always be saved as permanent faults, and so will class B1 faults after a certain amount of active time.

The next two fields are used to store the DTC of the fault as read using external diagnostic tools. The final decision was to provide support for both *J1939* and *WWH-OBD* standards, as the first is the main standard for heavy-duty vehicles, while the other is the harmonised standards and provides a well-defined way to map these DTCs to other standards.

The last field is an array of bytes used to list the flags of the recovery lines to activate once the fault passes in the *confirmed and active* status. More information will be provided in the "Reaction Manager" section.

Each entry of the FSM array is a structure containing the information of the ADIA. Each of those ADIA is associated with the diagnosis line in the previous array at the same index as the FSM.

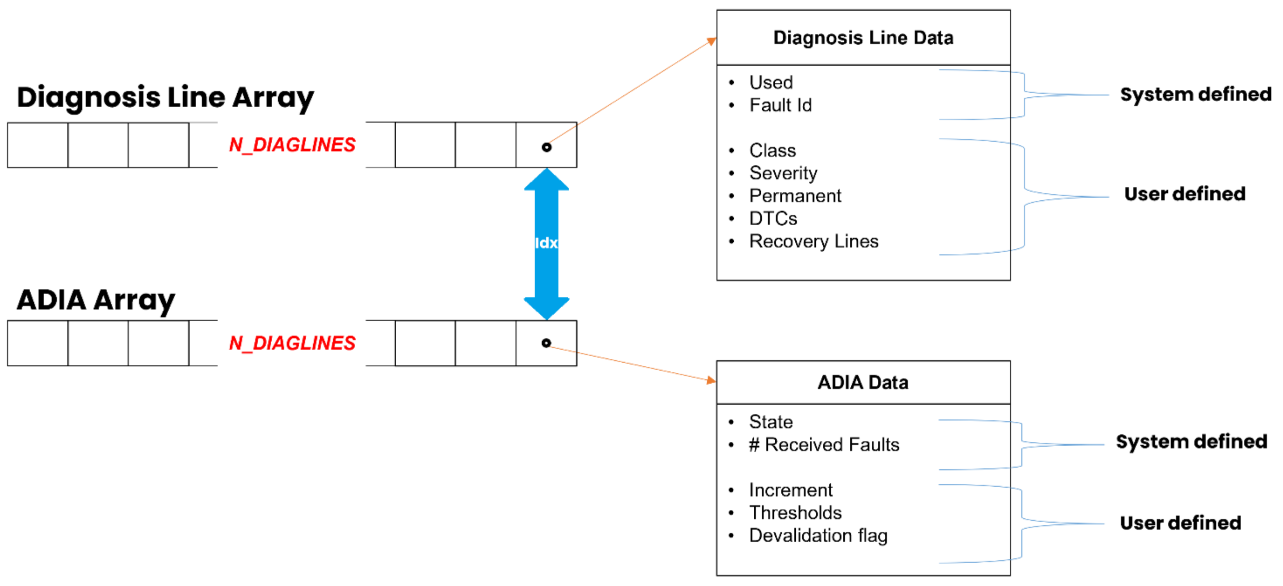
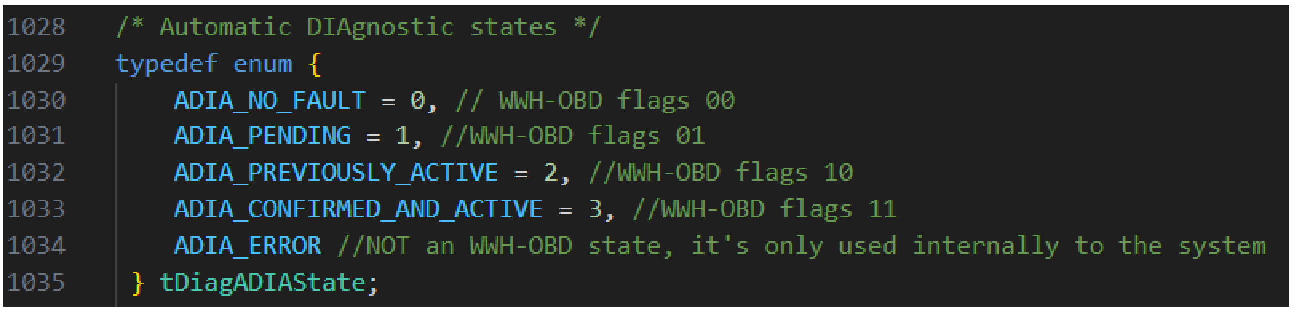


Figure 5.3.4 - Diagnosis lines and ADIAs arrays relation

The first field is the internal state of the ADIA, which corresponds to the fault's status. Its possible values are defined in the **tDiagADIAState** enumerative and correspond to the ones seen in the "Strategy" section, with the addition of an "error state".



```c
/* Automatic DIAgnostic states */
typedef enum {
    ADIA_NO_FAULT = 0, // WWH-OBD flags 00
    ADIA_PENDING = 1, //WWH-OBD flags 01
    ADIA_PREVIOUSLY_ACTIVE = 2, //WWH-OBD flags 10
    ADIA_CONFIRMED_AND_ACTIVE = 3, //WWH-OBD flags 11
    ADIA_ERROR //NOT an WWH-OBD state, it's only used internally to the system
 } tDiagADIAState;
```

Figure 5.3.5 - ADIA states

The order of the labels in the enum is not random but follows the *WWH-OBD* guidelines for the values of the state on two bits.

The second field is the counter variable, which is increased or decreased depending on the message received by the ADIA. Its value is used to determine whether the ADIA should change state by comparing it with the thresholds seen in the “Strategy” section.

While the two mentioned fields are values strictly internal to and manipulated by the ADIA, the remaining five are calibrations corresponding to the parameters of the ADIA.

The first of these values is the user step, which can be used by the calibrators to define, when combined with the thresholds and the periodic task of the board’s OS in which the ADIA is executed, the time required for a detected fault to enter a certain status.

The next three fields are the thresholds used by the ADIA to move from a state to another, while the last field is a flag to determine whether an active fault should be devaluated (i.e., moved to the *previously active* status) after a whole operating sequence without the error presenting itself. When this flag is set to true, a *confirmed and active* fault will be set in the *previously active* status at the system’s startup rather than restarting in the *confirmed* state.

The contents of the diagnosis lines and ADIAs arrays are initialised at startup with their respective initialisation functions, invoked by an **ON Init** wrapper procedure once the method responsible for the initialisation and the checks on the system NVRAM, *API_EEPROM_init*, has terminated. This order of execution is necessary as the wrapper function also reads from the NVRAM for the error memory and the initialisation of the ADIAs.
The init procedure is in turn called by the operating system initialisation function.

The diagnosis lines initialisation function simply sets up the cells of the respective array with default values. All the actual values of the fields are read at runtime from the calibrations of the model used to set up the diagnosis lines, through *set* APIs, and facilitated by the *mask*. As none of these values may change during the operational life cycle of the system, they do not require to be stored in NVRAM to perform Read/Write operations.

On the other hand, the ADIAs do require to have some information stored in the NVRAM, mainly the current state and the last value of the fault counter. The initialisation function of the ADIAs reads these two values from the ones stored in memory, while the other fields are initialised with default values and will receive the actual values at runtime, similarly to the diagnosis lines.

The runtime management of the diagnosis lines and the associated ADIAs is performed through the wrapper function **API_DIAG_DiagLine_ADIA_Mngr**. This API is the single access point through which a user can model the behaviour of a single ADIA and its diagnosis line. Although it is designed to be used inside the *mask* block, it is also made available directly as a Simulink block in case the user might want higher control over its inputs (e.g., to use calibrations during the test phases).

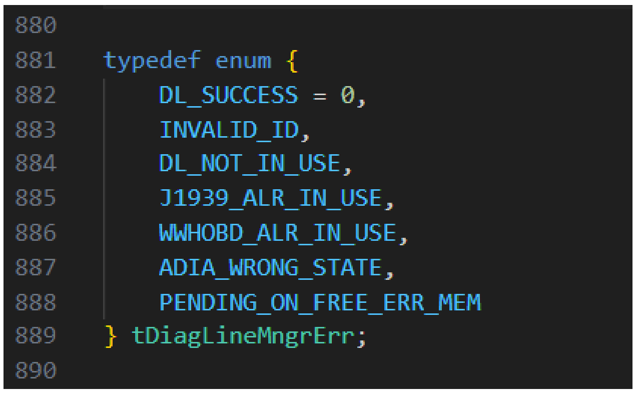
The function takes in as parameters all the calibratable values of the diagnosis line and ADIA structures and returns the state of the ADIA computed during this execution of the wrapper or a generic error (**ADIA_ERROR**). Additionally, the function requires as inputs the global ID of the diagnosis line, the ‘enable automaton’ flag to disable the execution of the FSM when required, and of course the fault code produced by the fault check test routine.

Although the function only returns a generic error to avoid bloating the ADIA states enumeratives and does not require the use of additional variables for each ADIA at the model-based design level, it also stores the more precise error inside a global array to improve the debugging and testing experience. This array has been defined using certain directives that make it possible to observe it at runtime using CANape.

The function’s code can be divided into two parts for easier understanding. The first part starts by checking the validity of the given ID (immediately returning an error if the position is invalid) and then cleans up any previous error in that position of the global array.

It then goes on to mark the entry in the diagnosis line array as being used and calls a 'set' function to store the diagnosis line's parameters in the same position of the array. In case this function returns an error, it is stored in an entry of the before-mentioned global error array, and the manager function returns the generic error.

In the second part of the function, the 'set' procedure for the ADIA is invoked, and then a check is performed on the passed 'enable automaton' parameter; if the flag is set to true, the **Automa** function is called to actually execute the FSM with the given error code as input.
Finally, if the function arrived here without encountering errors, it retrieves the current state of the ADIA (and thus, of the fault) and returns it to the caller.



```

typedef enum {
    DL_SUCCESS = 0,
    INVALID_ID,
    DL_NOT_IN_USE,
    J1939_ALR_IN_USE,
    WWHOBD_ALR_IN_USE,
    ADIA_WRONG_STATE,
    PENDING_ON_FREE_ERR_MEM
} tDiagLineMngrErr;

```

Figure 5.3.6 - Specific errors produced by the manager's inner functions, in api.h

The first function called by the manager to set the values of the diagnosis line performs some additional checks: it verifies that the given ID is valid, that the selected entry is being used, and that no other entry already uses the given DTC codes.

The function setting the ADIA values follows the same principle, copying the given values into the entry's fields while performing additional checks.

The *Automa*, called by the manager function after having set both the ADIA and the diagnosis line's values, implements the finite state machine whose state graph was shown in the "Strategy" section. It is the core of the fault management system, as it not only performs filtering and validation on the "punctual faults" it receives as input, updating its and the fault's state, but it also calls the APIs for the management of the recovery actions and the error memory, in addition to managing the system variables used for the MIL.

The automaton function takes in input just two parameters: the global ID of the diagnosis line and the fault message. The first operation the function performs is verifying that the received ID is a valid one. It then retrieves the corresponding ADIA's information.

The second part of the function is where the finite state machine is actually implemented, performing different operations depending on the state of the ADIA and the received input. There are a total of five cases: no-fault, pending, confirmed and active, previously active, and the default error state.

Barring the default case, which is used to manage unforeseen errors and simply returns an error message, the first case is also the simplest and corresponds to the *no-fault* state of the FSM model. In case the received message shows that a punctual error was detected, it increases the automaton's internal counter of fault detections by a user-defined amount and updates the state to the *Pending* one.

The *Pending* case reacts to an 'error-free' input message by decreasing the inner counter and moving the ADIA's state back to *No fault* once it reaches zero.

In case of a different message, the counter is increased, and its value is confronted with the first threshold. If the value equals or exceeds the threshold, a key set of instructions is executed to set the state to *confirmed and active* and insert the fault in the error memory (more information on how this works can be found in the implementation part of the "Error memory" paragraph).

If it fails to insert the fault in the error memory, e.g., because it was full, the event is signalled updating the entry of the global error array, and a fallback strategy is adopted to allow, on the next execution of the automaton, to immediately retry the insertion in memory without having to wait again to reach the first threshold.

In case the insertion was successful, the function calls three other APIs: one that manages the activation of the recovery lines, one to fill the freeze frame associated with the newly inserted fault (more information on this can be found in later paragraphs), and one to manage the activation of the flags of faults happening in the current operating sequence. As required by the Euro-VI and China-VI guidelines, the activation (even momentary) of certain classes of faults during an operating sequence is information to be shared via external tools and used internally for the management of the OBD system (e.g., for the MIL).

The third case of the automaton function, associated with the *Confirmed and active* state, presents an 'if-else' structure like the previous case, with calls to the recovery and error memory APIs.

If the received message does not contain an error code, the function decreases the counter, and after a number of calls without errors equal to the second threshold, it calls the function to update the error memory entry with the new state. It also updates the ADIA's internal state, moving to the *Previously active* one. At this point, if the memory update was successful, the system updates the counters of active class B1 and B2 faults, if needed. The automaton then calls the API to update the recovery lines to remove the contribution of this fault.

In case the received message did contain an error code, the error memory entry is still updated to keep track of the inner counter value, but there is neither a state change nor an update of the recovery lines.

The last case corresponds to the *Previously* active state of the automaton.

If the received message is not an error one, the function retrieves the position in the error memory of the fault and then goes on to check if the amount of time passed since the fault entered the *Previously active* state is greater than the standard-defined amount saved in the **bsENGINE_HOURS** calibration or if the remaining number of warm-up cycles that the fault has to spend in the error memory is zero.

The number of hours passed is computed using the timestamp of when the fault reached the state, stored in the error memory, and compared with the current time value maintained at the system level. This time value was implemented using a global variable and by creating specific APIs to manage its updating and readings. These functions were needed as it was peremptory to avoid race conditions on this variable, so a simple direct access to it wouldn't suffice. To compute the passing of a warm-up cycle, as the definition might differ from one manufacturer to another, a simple set of variables and APIs, called by the user-generated code, has also been implemented.

The result of these computations is stored in a global variable and is conducted in such a way to take eventual overflows into consideration.

If the condition on the warm-up cycles or the one on the engine hours is verified, the state is reverted to the *No fault* one, and the entry in the error memory is cleaned using the proper API.

When the received fault message contains an error code, the function increases the counter, then checks whether its new value equals or surpasses the third threshold. In that case the state is set back to *Confirmed and active*, and the error memory entry is updated with the new state.

Then, if the error memory update was successful, the same operations seen in the transition between the *Pending* and *Confirmed and active* states are called, updating system counters, freeze frames, and active recovery lines.

To recap, the strategy was implemented by defining two parallel arrays containing the diagnosis line and the ADIA information. These arrays are indexed using the global ID of the fault, defined as labels of an enumerative.

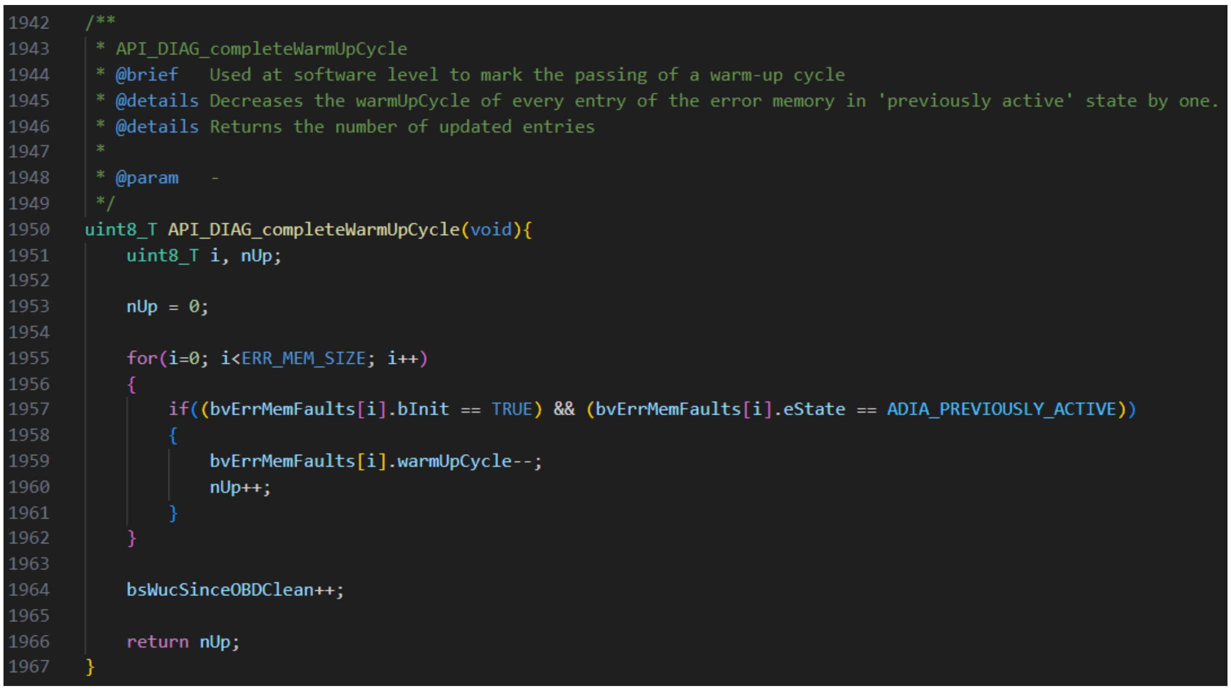
The user-modifiable data of both the diagnosis line and the ADIA are read at runtime from the model-based generated code and are passed to the manager function, which returns the new ADIA's state, calls two other APIs to set the received data for the diagnosis line and the ADIA, and then calls the function implementing the finite state machine.

The automaton function implements the various states of the FSM and calls the APIs related to the management of the recovery lines and the error memory.

The system keeps track of the errors happening during this whole process with a global array called that can be checked using CANape to have a visual, real-time feedback during testing and debugging.

Both the diagnosis lines and the ADIA's structure are initialised with default values during the system's startup. The ADIA's state and inner counter are retrieved from the non-volatile memory instead.

In addition, system-level variables and related APIs have been implemented for the management of the time passed for a fault in a certain state, the completion of a warm-up cycle, and other standard-required information such as the number of active faults of a certain class (also useful for the MIL management).



```c
/**
 * API_DIAG_completeWarmUpCycle
 * @brief   Used at software level to mark the passing of a warm-up cycle
 * @details Decreases the warmUpCycle of every entry of the error memory in 'previously active' state by one.
 * @details Returns the number of updated entries
 *
 * @param   -
 */
uint8_T API_DIAG_completeWarmUpCycle(void){
    uint8_T i, nUp;

    nUp = 0;

    for(i=0; i<ERR_MEM_SIZE; i++)
    {
        if((bvErrMemFaults[i].bInit == TRUE) && (bvErrMemFaults[i].eState == ADIA_PREVIOUSLY_ACTIVE))
        {
            bvErrMemFaults[i].warmUpCycle--;
            nUp++;
        }
    }

    bsWucSinceOBDClean++;

    return nUp;
}
```

Figure 5.3.7 - One of the additional functions implemented, her the one for the Warm-up cycle completion, in api_diagobd.c

5.3.4. Mask

Although the original amount of information required to the customer to define the parameters of a diagnosis line and its ADIA was greatly reduced, it was still a lot of values to set, especially considering the large number of fault lines a system might have.

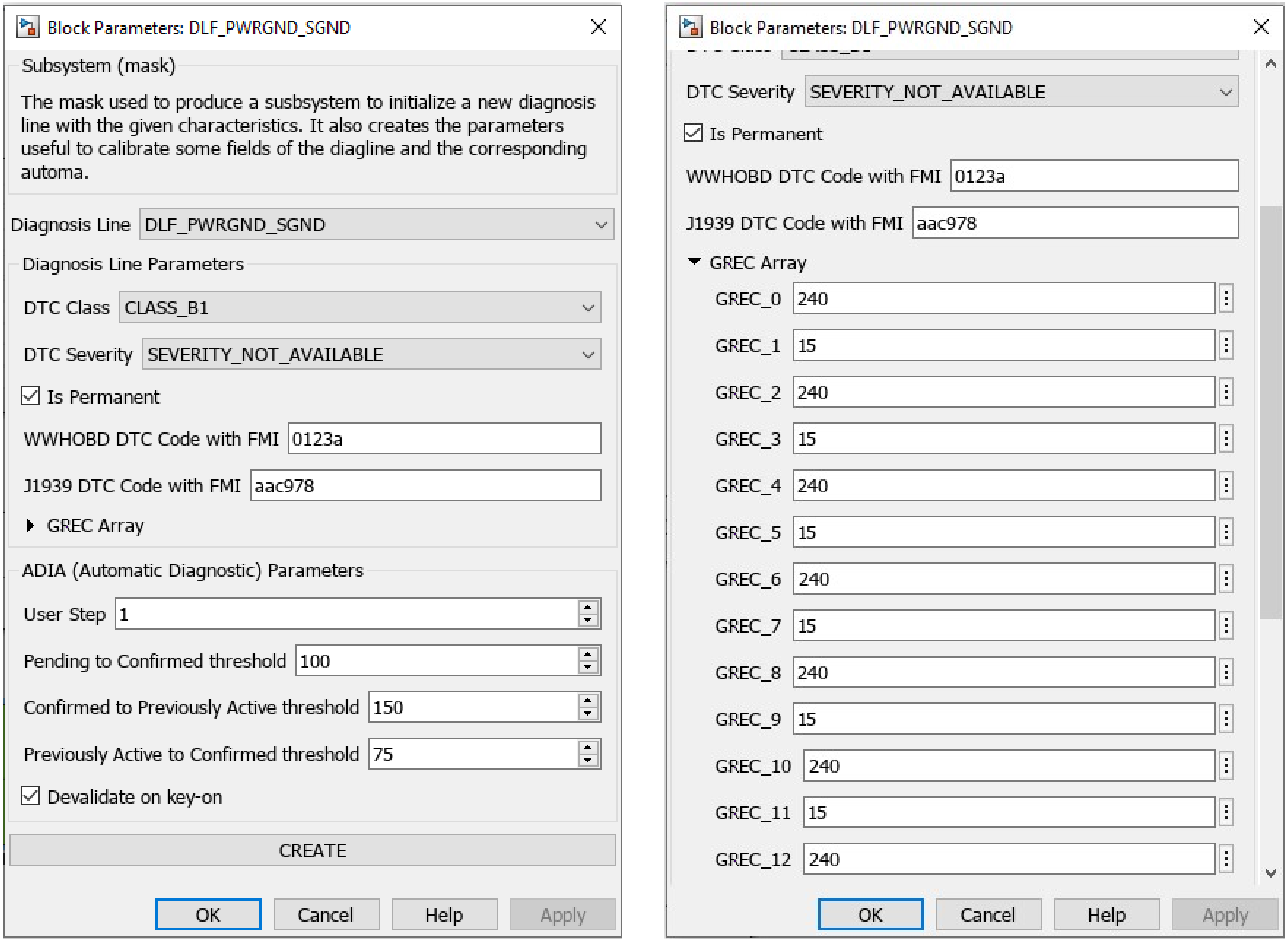
Had we kept the original strategy of using external spreadsheets with macros to compile the parameters in a suitable format for the board, our goal of simplifying the system could not have been considered to be achieved. There was a need for a way to easily set the required values that was already integrated into the design workflow.

The solution was to design a certain kind of Simulink block, a *mask*, to then insert into Metatron's library for the customers to use.

A Simulink mask is a block that provides a graphical interface to the user to define the block's parameters. Depending on how the mask code was implemented, these parameters will then be used to generate or update the inner model of the mask.

The mask's GUI is split into three parts. The first one is a drop-down menu to select the fault that shows the list of the IDs available in the system. The list is actually the tDiagLineFault enumerative, which can be modified to include new diagnosis lines or remove preexisting ones. The script used for the mask reads directly from the *api.c* file that contains the enumerative. The second part contains all the parameters seen for the diagnosis line: class, severity, permanent, DTCs, and recovery lines array (here called GRECs, from **Global RECovery**). The third part allows to set the characteristics of the ADIA, that is, the user step, the three thresholds, and whether to "devalidate" the fault after a key cycle.

The mask generates or updates the model upon clicking the *CREATE* button of the GUI.



Block Parameters: DLF_PWRGND_SGND

Subsystem (mask)

The mask used to produce a susbsystem to initialize a new diagnosis line with the given characteristics. It also creates the parameters useful to calibrate some fields of the diagline and the corresponding automa.

Diagnosis Line: DLF_PWRGND_SGND

Diagnosis Line Parameters

DTC Class: CLASS_B1

DTC Severity: SEVERITY_NOT_AVAILABLE

☑ Is Permanent

WWHOBD DTC Code with FMI: 0123a

J1939 DTC Code with FMI: aac978

GREC Array

ADIA (Automatic Diagnostic) Parameters

User Step: 1

Pending to Confirmed threshold: 100

Confirmed to Previously Active threshold: 150

Previously Active to Confirmed threshold: 75

☑ Devalidate on key-on

Block Parameters: DLF_PWRGND_SGND

DTC Severity: SEVERITY_NOT_AVAILABLE

☑ Is Permanent

WWHOBD DTC Code with FMI: 0123a

J1939 DTC Code with FMI: aac978

GREC Array

GREC_0	240
GREC_1	15
GREC_2	240
GREC_3	15
GREC_4	240
GREC_5	15
GREC_6	240
GREC_7	15
GREC_8	240
GREC_9	15
GREC_10	240
GREC_11	15
GREC_12	240

Figure 5.3.8 - The diagnosis line and ADIA mask, with (right) and without (left) the recovery lines array open

The designed mask had to take the values previously mentioned for both the diagnosis line and the ADIA and generate a model, internal to the mask.

The model had to contain the inserted values in the form of Simulink signals that could be interacted with, as calibrations, using CANape, and the function block for the manager API seen before.

The generated model also contains the blocks needed to perform certain conversion operations on the inserted data to make them suitable for the API, as the data format used by Simulink does not always directly correspond to an actual *C* data type.

The generation of the model would also change the mask block, creating two input ports to read the enabling flag and the fault message and an output port for a signal containing the current state of the ADIA.

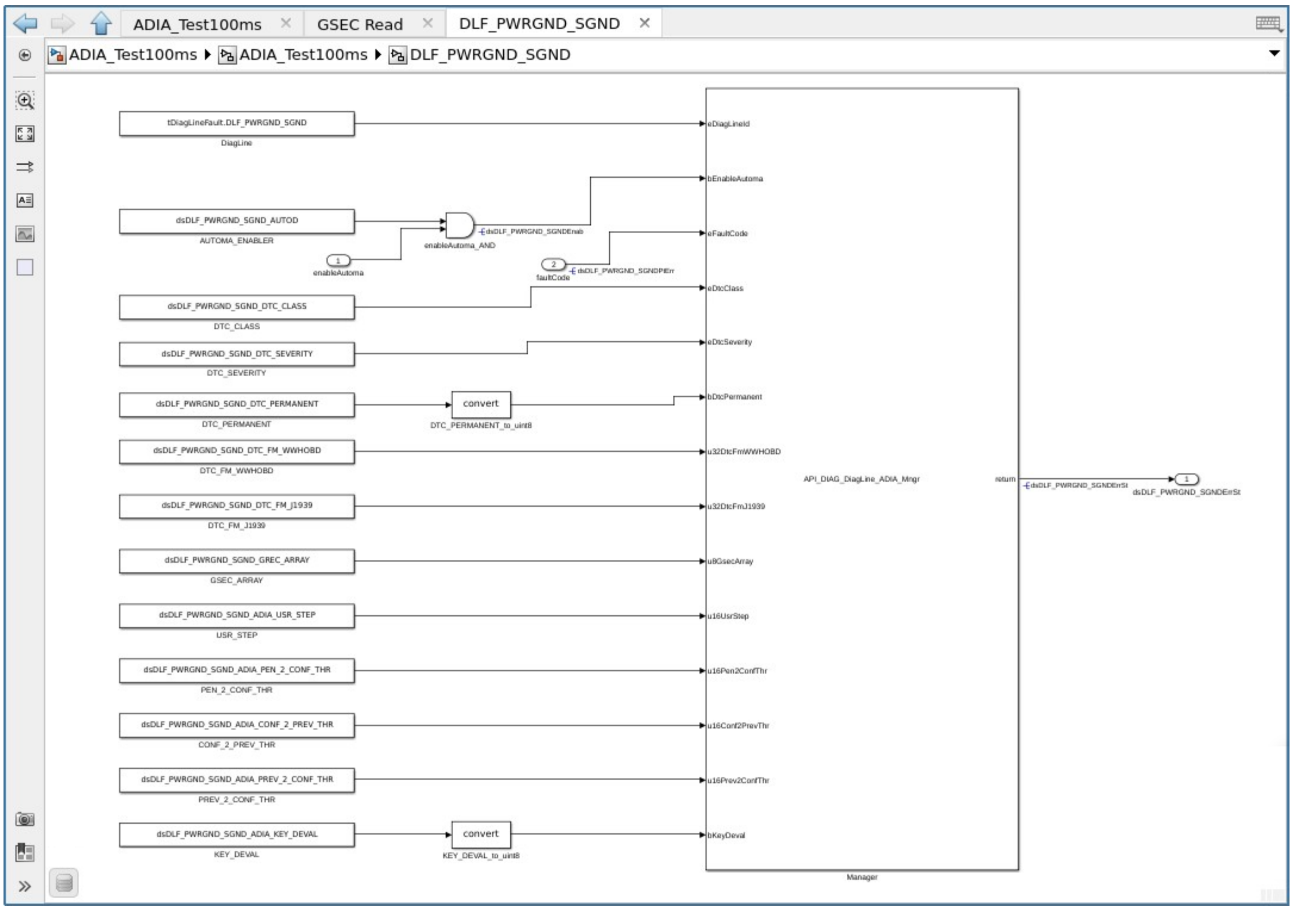


Figure 5.3.9 - The internal model of the mask

To define the behaviour of the mask, a MATLAB script, **DIAG_LINE_MANAGER.m**, has been designed.

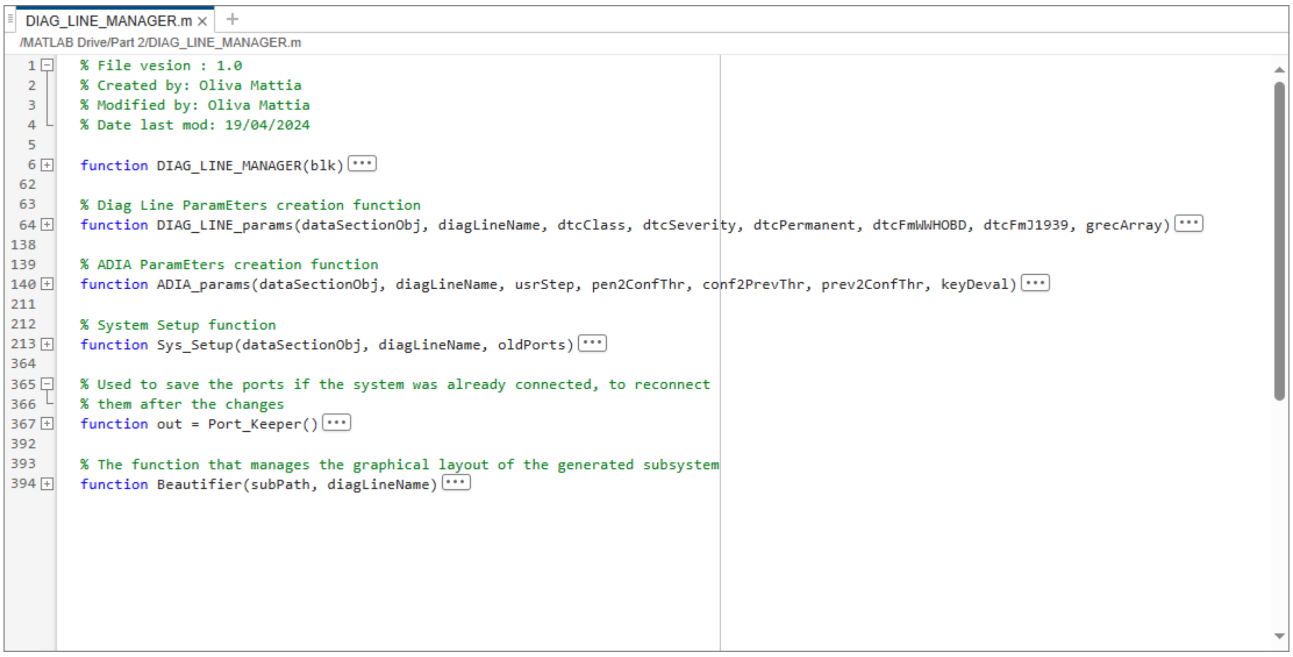
The mask block contains a few lines of code that reads the data inserted into the mask's fields and prepares them, then calls the first function of the script.

The main function of the script, DIAG_LINE_MANAGER, receives as inputs the block and the parent block, performs various checks on the inserted parameters, detaches the mask's block from the library in order to modify the block's contents, and then calls the remaining functions, in this order: *Port_keeper, DIAG_LINE_params, ADIA_params, Sys_Setup*.

The *Port_keeper* function is used to temporarily save the current ports connected to the block and restore the connections after the remodelling has happened.

The *DIAG_LINE_params* and *ADIA_params* functions are used to generate the Simulink signals and store them in the model's data dictionary. The format for the variable names will be "ds" plus the ID plus "_" followed by the parameter's name, or "ds" plus the ID plus "_" followed by "ADIA" and then by "_" and the parameter's name. Thanks to the hierarchical structure used by the data dictionaries in Metatron's projects, this standard naming ensures that if the same diagnosis line is being defined somewhere else in the various models, an error will be generated.

The *Sys_Setup* function is the one in charge of actually building the model. It starts by disconnecting the ports and clearing up the model, and then goes on to create each block and connect them. Once the building phase is finished, it reconnects the ports saved using the *Port_keeper* function and then calls the *Beautifier* function, which repositions the various components one-by-one to obtain a more comprehensible model (visible in *Figure 5.3.9*).



DIAG_LINE_MANAGER.m

/MATLAB Drive/Part 2/DIAG_LINE_MANAGER.m

```
% File vesion : 1.0
% Created by: Oliva Mattia
% Modified by: Oliva Mattia
% Date last mod: 19/04/2024

function DIAG_LINE_MANAGER(blk) ...

% Diag Line ParamEters creation function
function DIAG_LINE_params(dataSectionObj, diagLineName, dtcClass, dtcSeverity, dtcPermanent, dtcFmWWHOBD, dtcFmJ1939, grecArray) ...

% ADIA ParamEters creation function
function ADIA_params(dataSectionObj, diagLineName, usrStep, pen2ConfThr, conf2PrevThr, prev2ConfThr, keyDeval) ...

% System Setup function
function Sys_Setup(dataSectionObj, diagLineName, oldPorts) ...

% Used to save the ports if the system was already connected, to reconnect
% them after the changes
function out = Port_Keeper() ...

% The function that manages the graphical layout of the generated subsystem
function Beautifier(subPath, diagLineName) ...
```

Figure 5.3.10 - The DIAG_LINE_MANAGER.m file, with the various functions used by the script

5.4. Reaction Manager

5.4.1. Requirements

Differently from the previous modules, the reaction manager does not derive from external guidelines nor standards. The set of requirements upon which the reaction strategy has been designed all came from Metatron's internal documents and previous strategies.

The only actual requirement was the need for simplification of the pre-existing implementation, in line with the rest of the work done on the system.

5.4.2. Strategy

The adopted strategy consists in entrusting the implementation of the recovery functions to the users, who will design them at higher levels, while the system simply activates the flags (**recovery lines**) to indicate the need for those functions to operate.

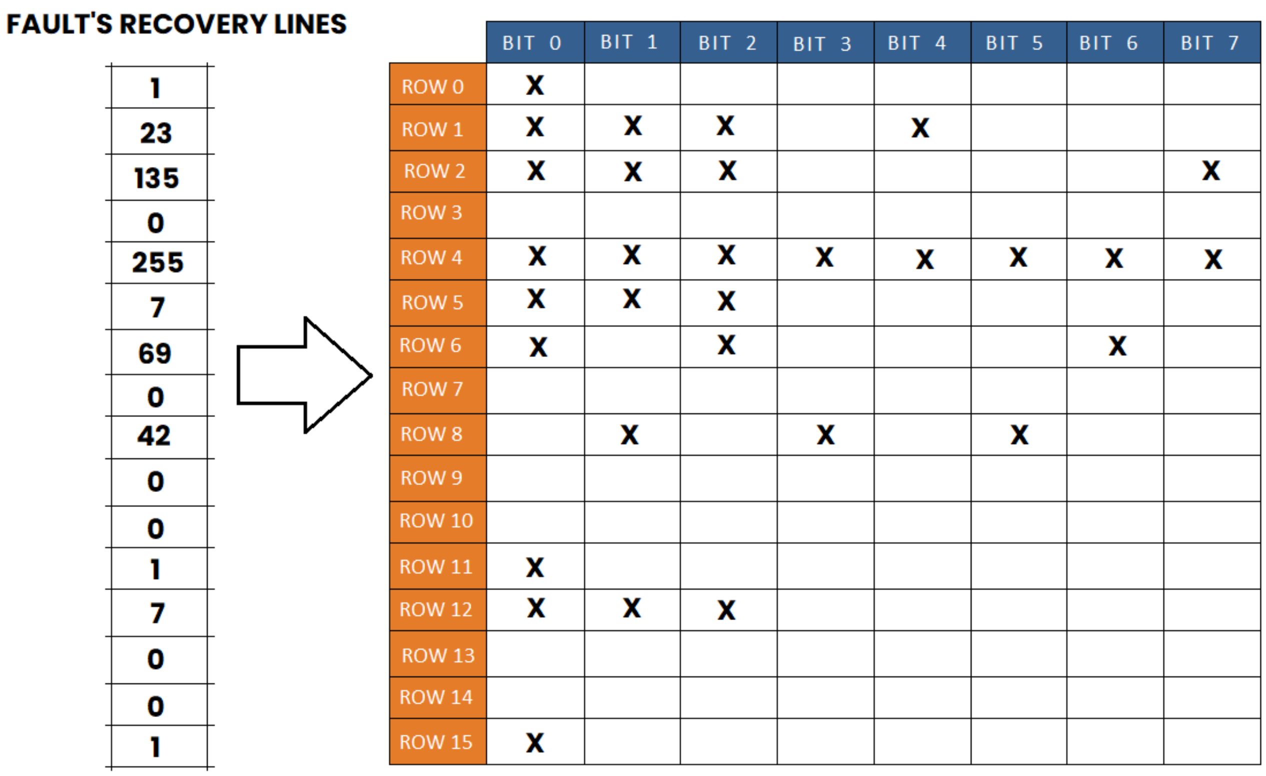
To do so, two things are needed: a way to associate a certain fault to one or more recovery lines and a way to manage the activation and deactivation of the correct lines depending on the various faults' states.

The association between a diagnosis line and its activated recoveries is made possible at the time of the definition of the Simulink mask for the fault and the ADIA, where a set of fields are available to select the recovery lines (*Figure 5.3.8*).

This set of fields is a transposition of the structure used for the management of the various lines: a table composed of 16 rows, each of which contains 8 cells. This table is a single, global structure that can be seen as a matrix of bits, with each cell corresponding to a flag that marks whether that recovery line should be active or not.

The mask can be used to select the cells of this global table when the fault is active by inserting a value between 0 and 255 on each line. The '1's of the binary form of this value should be seen as the active recovery lines of the table's row.

The activation of the recovery lines should happen while the associated fault is determined to be *confirmed and active*. Once the fault changes state, those lines should be turned off if no other currently active fault is associated with them. As the activation and deactivation of the recoveries is heavily dependent on the transition between the states of the faults, their management has been assigned to the ADIAs.



FAULT'S RECOVERY LINES
1
23
135
0
255
7
69
0
42
0
0
1
7
0
0
1

	BIT 0	BIT 1	BIT 2	BIT 3	BIT 4	BIT 5	BIT 6	BIT 7
ROW 0	X							
ROW 1	X	X	X		X			
ROW 2	X	X	X					X
ROW 3								
ROW 4	X	X	X	X	X	X	X	X
ROW 5	X	X	X					
ROW 6	X		X				X	
ROW 7								
ROW 8		X		X		X		
ROW 9								
ROW 10								
ROW 11	X							
ROW 12	X	X	X					
ROW 13								
ROW 14								
ROW 15	X							

Figure 5.4.1 - Examples of active recovery lines flags ('x') in the global structure when a fault with the given lines is active

At model level, it will be necessary to have a way to read a given flag from the global structure in order to activate the associated recovery function.

5.4.3. Implementation

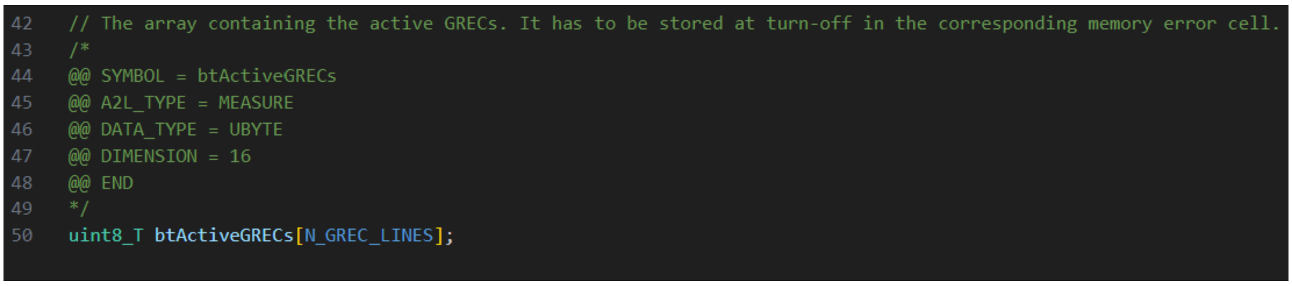
All the code produced for this part was contained in the **api.h**, **api_diagobd.h**, and **api_diagobd.c** files.

The structures to associate the diagnosis lines with their recovery lines have already been shown in the previous chapters: they've been implemented as an array of **N_GREC_LINES** (here, 16) 8-bit entries, the **GREC_Array** field of the *tDiagLine* structure.

The global table of the recovery lines flags has been implemented in a similar fashion, except it has been declared as a global variable, **btActiveGRECs**, rather than a field of a struct.
As some recovery functions might have to operate before the models that define the values of the diagnosis lines (which include the recovery lines to activate), this global

table has an in-NVRAM counterpart, where it is copied from and to at startup and shutdowns, stored in the error memory.

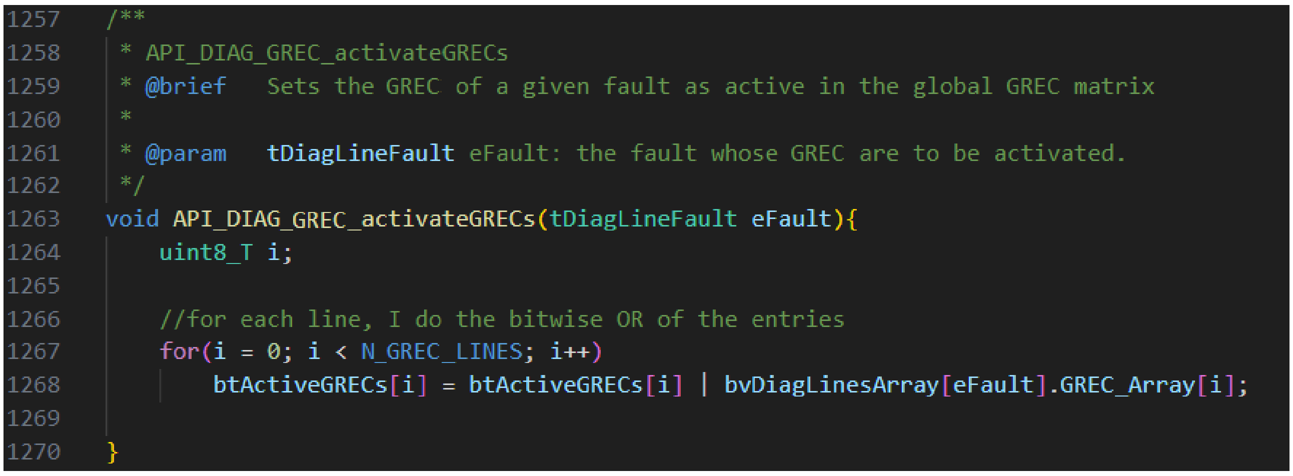
The number of rows of the table is determined by the #define *N_GREC_LINES*, while the number of cells in each row, i.e., the number of bits, is written in the #define **N_GRECS_PER_LINE** (here, 8).



```
// The array containing the active GRECs. It has to be stored at turn-off in the corresponding memory error cell.
/*
@@ SYMBOL = btActiveGRECs
@@ A2L_TYPE = MEASURE
@@ DATA_TYPE = UBYTE
@@ DIMENSION = 16
@@ END
*/
uint8_T btActiveGRECs[N_GREC_LINES];
```

Figure 5.4.2 - The global recovery lines table, with special comments to make it visible on CANape, in api_diagobd.c

The first function related to the recovery lines is **API_DIAG_GREC_activateGRECs**, which takes the ID of the fault whose lines must be activated. It is called by the ADIA when it moves from the *Pending* state to the *Confirmed and active* one.
The function simply performs a bitwise 'or' between the global table and the diagnosis line's field.

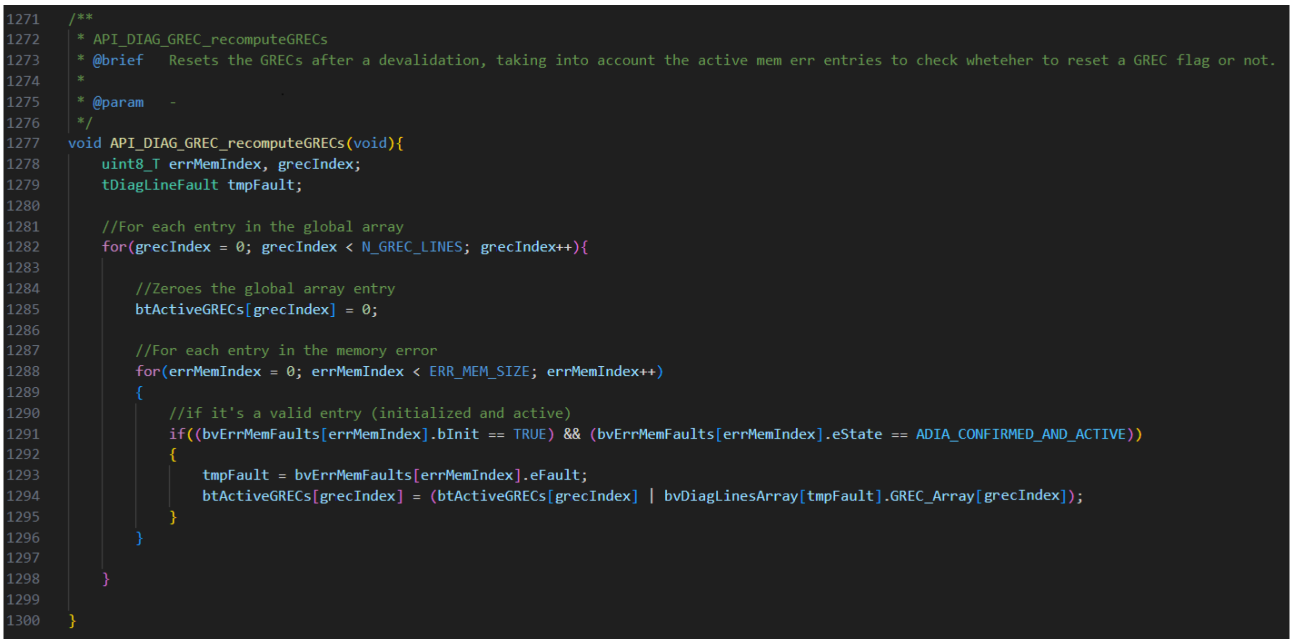


```
/**
 * API_DIAG_GREC_activateGRECs
 * @brief   Sets the GREC of a given fault as active in the global GREC matrix
 *
 * @param   tDiagLineFault eFault: the fault whose GREC are to be activated.
 */
void API_DIAG_GREC_activateGRECs(tDiagLineFault eFault){
    uint8_T i;

    //for each line, I do the bitwise OR of the entries
    for(i = 0; i < N_GREC_LINES; i++)
        btActiveGRECs[i] = btActiveGRECs[i] | bvDiagLinesArray[eFault].GREC_Array[i];

}
```

Figure 5.4.3 - API_DIAG_GREC_activateGRECs, in api_diagobd.c

When an ADIA moves to the *Previously active* state, all the recovery lines that were activated by the fault must be turned off, except for those that are being piloted by other currently active faults. To do this, the ADIA invokes the **API_DIAG_GREC_recomputeGRECs** function, passing it the fault's ID.

This function zeroes the global table, then scans the error memory array looking only for the currently used entries, i.e., those with the **bInit** flag at true, whose ADIA is currently in the *Confirmed and active* state. Whenever the function finds one of these entries, it updates the global table, as seen in the *API_DIAG_GREC_activateGRECs* function. We scan the error memory rather than the diagnosis lines array, as it contains fewer entries and only the faults currently in the memory can activate the recovery lines.



```c
/**
 * API_DIAG_GREC_recomputeGRECs
 * @brief   Resets the GRECs after a devalidation, taking into account the active mem err entries to check whetheher to reset a GREC flag or not.
 *
 * @param   -
 */
void API_DIAG_GREC_recomputeGRECs(void){
    uint8_T errMemIndex, grecIndex;
    tDiagLineFault tmpFault;

    //For each entry in the global array
    for(grecIndex = 0; grecIndex < N_GREC_LINES; grecIndex++){

        //Zeroes the global array entry
        btActiveGRECs[grecIndex] = 0;

        //For each entry in the memory error
        for(errMemIndex = 0; errMemIndex < ERR_MEM_SIZE; errMemIndex++)
        {
            //if it's a valid entry (initialized and active)
            if((bvErrMemFaults[errMemIndex].bInit == TRUE) && (bvErrMemFaults[errMemIndex].eState == ADIA_CONFIRMED_AND_ACTIVE))
            {
                tmpFault = bvErrMemFaults[errMemIndex].eFault;
                btActiveGRECs[grecIndex] = (btActiveGRECs[grecIndex] | bvDiagLinesArray[tmpFault].GREC_Array[grecIndex]);
            }
        }

    }

}
```

Figure 5.4.4 - API_DIAG_GREC_recomputeGRECs, in api_diagobd.c

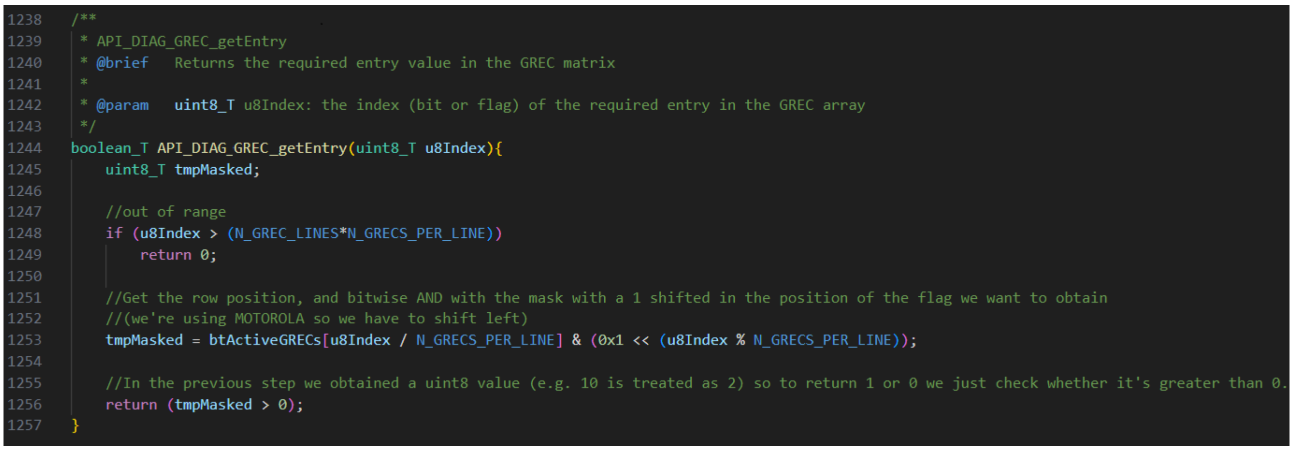
The remaining function associated with the recovery lines is also the only one that is exported to model level and made available in the Simulink block library: **API_DIAG_GREC_getEntry**. This API allows to retrieve the value of a single entry of the table, passed as a parameter.

The function starts by checking that the given index is a valid one, that is, included between zero and *N_GREC_LINES* N_GREC_PER_LINES*.
It then goes on to store in the support variable **tmpMasked** the result of a logical operation: the bitwise 'and' between the indexed value and a '1' left shifted by a certain amount.

The correct flag position is computed by performing an integer division (given index over *N_GRECS_PER_LINE*), while the amount to shift the '1' is obtained using the modulo operator on the same values.

This way, *tmpMasked* will contain a non-zero value if and only if the chosen cell contained a ‘1’.

The function then returns the result of the comparison, ‘*tmpMasked* > 0’.



```c
/**
 * API_DIAG_GREC_getEntry
 * @brief   Returns the required entry value in the GREC matrix
 *
 * @param   uint8_T u8Index: the index (bit or flag) of the required entry in the GREC array
 */
boolean_T API_DIAG_GREC_getEntry(uint8_T u8Index){
    uint8_T tmpMasked;

    //out of range
    if (u8Index > (N_GREC_LINES*N_GRECS_PER_LINE))
        return 0;

    //Get the row position, and bitwise AND with the mask with a 1 shifted in the position of the flag we want to obtain
    //(we're using MOTOROLA so we have to shift left)
    tmpMasked = btActiveGRECs[u8Index / N_GRECS_PER_LINE] & (0x1 << (u8Index % N_GRECS_PER_LINE));

    //In the previous step we obtained a uint8 value (e.g. 10 is treated as 2) so to return 1 or 0 we just check whether it's greater than 0.
    return (tmpMasked > 0);
}
```

Figure 5.4.5 - API_DIAG_GREC_getEntry, in api_diagobd.c

The returned value can then be used to pilot the recovery functions defined by the user. In the following picture, an example of implementation using Simulink that pilots the *enabled subsystems* implementing the functions, using the value returned by the *API_DIAG_GREC_getEntry* block.

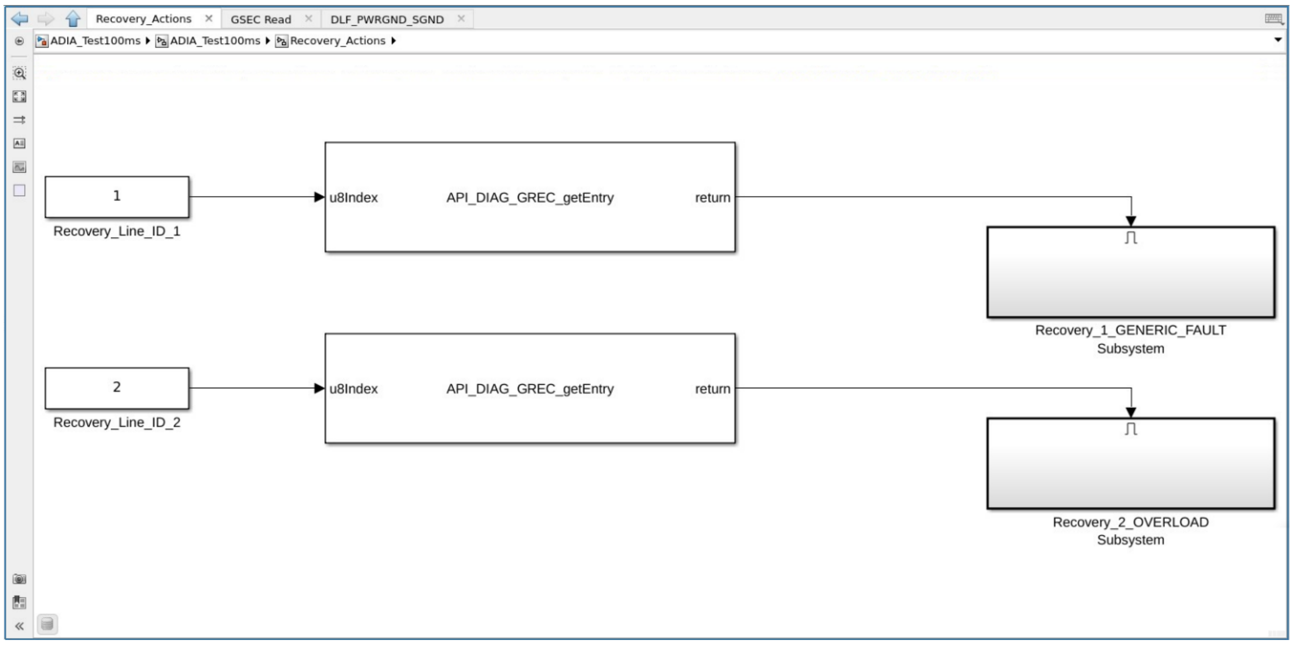


Figure 5.4.6 - Example of usage of the recovery lines system with Simulink

5.5. Error Memory and Freeze Frames

Detecting the presence of a fault is not sufficient: some faults might happen only under rare conditions, some might be flickering, others might be due to multiple factors, yet another might prevent the vehicle from circulating safely, and so on.

Recovery actions performed onboard might help mitigate the issue, but as the complexity of the systems grows, so does that of the faults.

For these reasons, it is important to gather as much information as possible whenever a fault presents itself and to store that information in a permanent way so that it could be used afterwards to perform checks and reparations.

5.5.1. Requirements

Both Euro-VI and China-VI directives include a set of various information that an OBD system should provide when interrogated with external tools. These details are related to the single fault, to the whole system, and to the engine status when a fault is confirmed.

We call the set of this information "Error Memory" and the subset of it corresponding to the snapshot of the engine when a fault is inserted into the error memory "Freeze Frames". These data must be stored in a non-volatile memory, as they must be accessible even after a system's shutdown and must also follow certain rules for what concerns their storage and availability.

As said, the guidelines define the information to store for three main categories: system, fault, and freeze frame.

For the system, the regulations required to be available were:

- The amount of time since the last reset (clean-up) of the system, in hours.
- The number of warm-up cycles completed since the last clean-up.
- The number of hours during which the engine has operated while a Class B1 malfunction was active.
- The number of engine hours passed with a continuous MIL and the cumulative number of hours of continuous MIL.

For the faults, the information to be made available were:

- The number of active faults of a certain class.
- All the currently active faults of a class.
- All the previously active faults of a given class.

When it came to a single fault, the required details were:

- The fault's DTCs.
- The class of the fault.
- The current state of the fault.
- The number of times the fault went back to *Confirmed and active* after having become *Previously active*.
- Whether the fault happened in the last operating sequence.
- The number of consecutive warm-up cycles since the fault entered the devalidated state.
- The freeze frame associated with the fault.

The Euro-Vi and Cina-VI guidelines also define various rules on the behaviour of the system for what concerns the mentioned information; for instance, "if and when" a value stored in memory can be reset, via external tools or directly by the system. The B1 hour counter, for example, increments itself every 1 hour of uptime as long as a class B1 fault is active. If it passes a certain number of hours and there are no more active B1 faults (due to being erased by tools or no longer existing), the guidelines require the counter to be set to a given value. The standard also defines the number of continuous operating sequences without B1 faults needed to reset the counter.

Another rule concerns how to behave when trying to store a new fault in case of full memory, based on the class of the faults and other factors. Different fault classes have different priorities during memorisation; for instance, a class B2 entry in memory shall not be replaced by a class C fault. In cases of equal class, the diagnosis line's defined severity shall be used. If the severities are equal too, the oldest entry shall be preserved.

The documents also include various sets of information about the engine to be stored along with a fault in the error memory, the freeze frame. This information is divided into mandatory and optional and can be seen as a snapshot of the system at the time of the fault's confirmation. This set of data must be retrieved when the OBD system is required to provide the information of a fault using external tools.

While for the previous requirements, such as the fault classes, the number of hours to reset the B1 counters, or the number of warm-up cycles that a fault must remain in the *Pending* status before being automatically erased from the error memory (40 cycles), in the case of the fields of the freeze frames the two documents had different requests, at least at first glance.

Before moving on to the design of the strategy to manage the error memory, a work of unification of the requests had to be carried out, with particular attention to freeze frames. The final set of engine information is the following:

Mandatory

Calculated Load *(engine torque as % of maximum torque available at current speed)*
Rotatory Engine speed
Engine coolant temperature
Barometric pressure *(either directly measured or estimated)*
Air intake volume *(read by air mass flow sensor)* *****
DPF differential pressure *****
SCR catalyst inlet temperature *****

*Only if the system is equipped with the needed sensor. Otherwise, it could be possible to use the indirect, calculated value.

Optional

- Rotary speed and load information of the selected engine

Driver's demand engine torque *(as % of maximum engine torque)*
Actual engine torque *(as % of max engine torque, e.g. calculated from commanded injection fuel quantity)*
Time elapsed since engine start

- Any information of validate or invalidate emission or OBD system

Fuel level *(e.g. % of the nominal capacity of the tank)* or tank fuel pressure
Engine oil temperature
Vehicle speed
Engine control computer system voltage *(for the main control chip)*

- Information for determination or calculation, if installed on the engine

Absolute throttle position / intake air throttle position
Diesel fuel control system status in case of a close loop system
Fuel rail pressure
Injection control pressure
Representative fuel injection timing *(beginning of first main injection)*
Commanded fuel rail pressure *(the set value)*
Commanded injection control pressure *(the set value)*
Intake air temperature
Environmental air temperature
Super/Turbocharger inlet temperature
Super/Turbocharger outlet pressure
Change air temperature *(if post intercooler)*
Actual boost pressure *(supercharger)*
Air flow rate from mass air flow sensor
Commanded EGR valve duty cycle/position
Actual EGR valve duty cycle/position
PTO status *(active or not active)*
Accelerator pedal position
Redundant absolute pedal position
Instantaneous fuel consumption
Commanded/target boost pressure *(if boost pressure used to control turbo ops)*
DPF inlet pressure
DPF outlet pressure
DPF delta pressure
Engine-out exhaust pressure
DPF inlet temperature
DPF outlet temperature
Engine-out exhaust gas temperature
Rotatory Turbocharger/turbine speed
Variable geometry/section turbocharger position
Commanded variable geometry/section turbocharger position
Wastegate valve position

To recap, the study of the guidelines document of Euro-VI and China-VI determined the set of requirements on both the data to store and the behaviour of the system regarding those data and laid the foundations for the design of the error memory strategy.

5.5.2. Strategy

As expected, this section was probably the one that took the most from the first part of this thesis work on the NVRAM management. Many of the strategies and the insight developed during that phase were improved and reused here, starting with the separation between the data stored in the NVRAM and the one directly manipulated during the system's operation.

All the main structures and variables used had to be duplicated or at least compacted into secondary structures that were only used to store the data at the shutdown and reload them at startup.

For this reason, the strategy adopted had to be split into three phases: startup, operating, and shutdown.

At startup, it was essential to ensure that the stored data were not corrupted before restoring them as needed. Only the data necessary to meet the requirements were saved from the diagnosis line; not all data were stored. The values that needed to be reloaded into the ADIA included only the state and the fault counter. Additionally, other data that had to be read from memory and restored included various time counters, the counter for operating sequences, and the active recovery lines.

The 'operating' part of the strategy is the most complex, as it not only defines the behaviour of the system for the single faults to insert, update, or remove from the error memory but also the management of all other 'accessory' data required by the guidelines and Metatron's documentation: active fault counters, timers, freeze frames, key devalidation, and so on.

The insertion of a new fault in the error memory happens when the state of the fault's ADIA passes from *Pending* to *Confirmed and active*.
If the error memory is full, we look for an entry with lower class than the new fault's, or lower priority in case of class equivalence, to substitute.
The timestamp is used to ensure that, *ceteris paribus*, in case of a substitution of an entry, the newest entries are swapped out rather than the oldest, as per guidelines.
If it fails to insert, the ADIA remains in the *Pending* state to try again next time. While this could create some potential issues, the relatively contained number of entries in the error memory and the low probability in normal working conditions of having too many contemporary faults mitigate the drawbacks of this solution. When inserting the new

fault, it must be marked as a **permanent failure code** if the permanent flag is set for that specific diagnosis line, if the fault is of class A, or if it's of class B1 but it has been active for more than 200 hours.

When the fault passes from the *Confirmed and active* state to the *Pending* one, its corresponding entry in the error memory is updated; a timestamp must be associated with it, and we must also mark the number of warm-up cycles that the fault spends in the memory while in the *Previously active* state.
The warm-up cycles counter will start at the maximum value (by the guidelines, 40) and will be decreased by one each time the system completes a warm-up cycle while the fault is in the mentioned state.
Whenever the fault’s ADIA moves back to the *Confirmed and active* state, the warm-up cycle counter will freeze, and the counter for the occurrences of 'going back to active' is updated.
The counter will be reset to the maximum value once the fault moves back to the *Previously active* state. At the same time, the timestamp will be updated with the most recent value.

For the removal of an entry in the error memory, if the entry is a Permanent Failure Code, it cannot be removed by means of external tools, but only after the system itself has deemed the fault repaired.
Non-permanent failure codes can always be removed by external tools.
Any type of failure code can be erased automatically if the fault has been in the *Previously active* status for at least 40 warm-up cycles or 200 engine operating hours. Rather than increasing a time counter for each entry of the memory with every hour passed, the timestamp associated with the entry will be confronted with the system’s current time.

As the mentioned insertion, update, and deletion operations are highly dependent on the faults’ states and their transition, the management of such operations will be conducted by the ADIAs.

The standards define the possibility to devalidate an entry (from *Confirmed and active* to *Previously active*) after a whole operating sequence without the error presenting itself has passed.
The pre-existing strategy of allowing this for certain entries using a flag 'keyDeval' for the diagnosis line has been kept. This requires ways to compute the passing of an operating

sequence; as the definition of it is not standardised, the most functional solution is to let the users determine when one has been completed, simply providing them with an API able to coordinate the set of operations related to its passing.

For each fault inserted into the error memory, a corresponding freeze frame is stored when that fault passes to the *Confirmed and active* status (either from *Pending* or *Previously active*). The implemented solution consists of a global freeze frame that is constantly updated with the needed information, and once there's the need to store a snapshot of the system for a fault's insertion, the global freeze frame is stored in the memory structure associated with the fault. As the faults' memory error entries are separated from their freeze frames, the APIs will need to 'reconnect' them before returning them neatly packed to the users.
This general freeze frame must be updated before the execution of any ADIA and should be updated using a function generated via model-based design by the customers rather than with a function directly implemented by Metatron at lower levels.
The necessity of leaving the updating of the freeze frame to the users comes from the fact that while the guidelines define the various fields of the freeze frames, not all those data are available on every system, and thus the selection of the data is left to those who better know the vehicles where the code will run: the vehicle's manufactures.
To simplify the management of the freeze frame and its fields at the model level, a Simulink mask, similar to the one used for the diagnosis lines and the ADIAs, has been designed.

For the management of counters linked to the B1 class faults, the strategy follows step by step what required by the Euro-Vi and China-VI documents: when a B1 fault is stored, if it is the first, a counter must be initialized to record the number of hours during which the engine has operated while a class B1 malfunction was present.
If no B1 are detected or all have been erased the counter will freeze, otherwise the counter will increment the counter every 1 hour of operativity.
If the counter passed 200h and there are no more B1 (due to being erased by tools or no longer existing), set it to 190h.
When no Class B1 failure has been detected in 3 continuous operation sequences (this value is defined by the standard, but for the thesis scope it has been implemented as a parameter that can be defined by the customer), the B1 counter shall be reset to 0.

Given the need for precise time counters, the designed strategy must also include ways to update and read these timers, taking into account events like overflows.

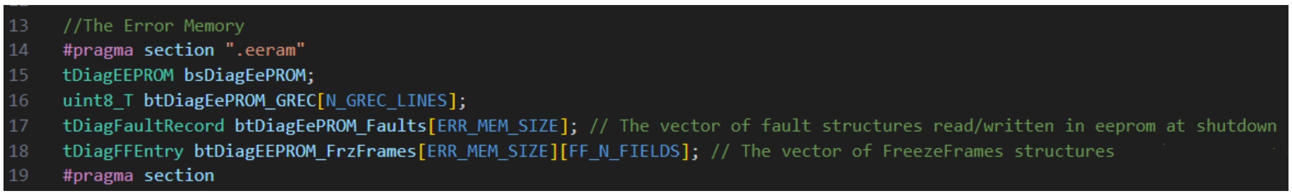
In addition to the various ways to create, update, and erase the data, the system must provide interfaces to allow the users, or external diagnostic tools, to access these data.

For the last part of the strategy, the 'shutdown', the system simply has to copy the contents of the structures and variables from the volatile memory into others created especially for the NVRAM.

5.5.3. Implementation

All the code produced for this part was contained in the **api.h**, **api_diagobd.h**, and **api_diagobd.c** files.

The data structure of this part is similar to the one adopted in the first part of this thesis, with a mirroring of variables and structures between the NVRAM and the volatile memory. To implement the non-volatile part, we made use of the '*#pragma section ".eeram"*' directive, as seen before.



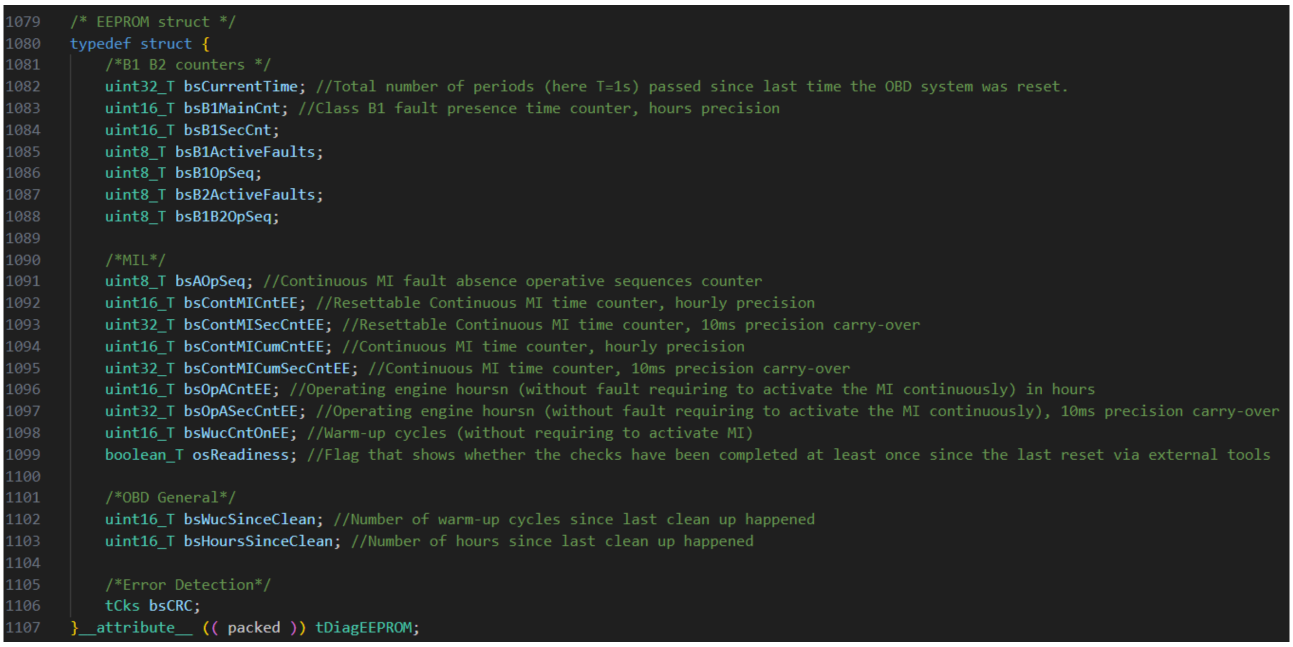
```
//The Error Memory
#pragma section ".eeram"
tDiagEEPROM bsDiagEePROM;
uint8_T btDiagEePROM_GREC[N_GREC_LINES];
tDiagFaultRecord btDiagEePROM_Faults[ERR_MEM_SIZE]; // The vector of fault structures read/written in eeprom at shutdown
tDiagFFEntry btDiagEEPROM_FrzFrames[ERR_MEM_SIZE][FF_N_FIELDS]; // The vector of FreezeFrames structures
#pragma section
```

Figure 5.5.1 - Data structures in the ".eeram" section, in api_diagobd.c

The **bsDiagEePROM** is a struct of type **tDiagEEPROM** that is used to store in NVRAM the set of system variables like counters and timers that must be restored during the initialisation of the system. This struct does not present a direct correspondence with a variable in the volatile memory but rather encapsulates many different values with its fields (whose names correspond to the variables in the volatile memory).

This solution has the double advantage of being both more compact and easier to read while analysing the code and allowing for the computation of a checksum value on the whole set of variables to control the validity of the data at initialisation time. In case of mismatch of the checksum, the choice can be to either simply replace the whole contents of the struct with default values or perform more sophisticated strategies like the one explored in the first part of this thesis.

The fields of the *tDiagEEPROM* can be divided into four groups, depending on where their volatile counterparts are used.



```c
/* EEPROM struct */
typedef struct {
    /*B1 B2 counters */
    uint32_T bsCurrentTime; //Total number of periods (here T=1s) passed since last time the OBD system was reset.
    uint16_T bsB1MainCnt; //Class B1 fault presence time counter, hours precision
    uint16_T bsB1SecCnt;
    uint8_T bsB1ActiveFaults;
    uint8_T bsB1OpSeq;
    uint8_T bsB2ActiveFaults;
    uint8_T bsB1B2OpSeq;

    /*MIL*/
    uint8_T bsAOpSeq; //Continuous MI fault absence operative sequences counter
    uint16_T bsContMICntEE; //Resettable Continuous MI time counter, hourly precision
    uint32_T bsContMISecCntEE; //Resettable Continuous MI time counter, 10ms precision carry-over
    uint16_T bsContMICumCntEE; //Continuous MI time counter, hourly precision
    uint32_T bsContMICumSecCntEE; //Continuous MI time counter, 10ms precision carry-over
    uint16_T bsOpACntEE; //Operating engine hoursn (without fault requiring to activate the MI continuously) in hours
    uint32_T bsOpASecCntEE; //Operating engine hoursn (without fault requiring to activate the MI continuously), 10ms precision carry-over
    uint16_T bsWucCntOnEE; //Warm-up cycles (without requiring to activate MI)
    boolean_T osReadiness; //Flag that shows whether the checks have been completed at least once since the last reset via external tools

    /*OBD General*/
    uint16_T bsWucSinceClean; //Number of warm-up cycles since last clean up happened
    uint16_T bsHoursSinceClean; //Number of hours since last clean up happened

    /*Error Detection*/
    tCks bsCRC;
}__attribute__ (( packed )) tDiagEEPROM;
```

Figure 5.5.2 - tDiagEEPROM structure, in api.h

The first group is used to manage the information on the B1 and B2 classes. This group's fields include:

- **bsCurrentTime**, keeps track of the seconds passed since the last reset of the OBD system.
- **bsB1MainCnt** and **bsB1SecCnt**, the timers to count the time passed while a B1 fault was active. The *bsB1SecCnt* has a 10 [ms] precision, and it's used to update the *bsB1MainCnt*, which has a precision of 1 hour.
- **bsB1ActiveFaults** and **bsB2ActiveFaults**, the counter that marks the number of currently active faults of class B1 or B2.
- **bsB1OpSeq** and **bsB1B2OpSeq**, keep track of the number of continuous operating sequences without active faults of B1 or B2 classes.

The second group is the one gathering the counters and timers used for the MIL management. More detailed information will be provided in the MIL chapter.

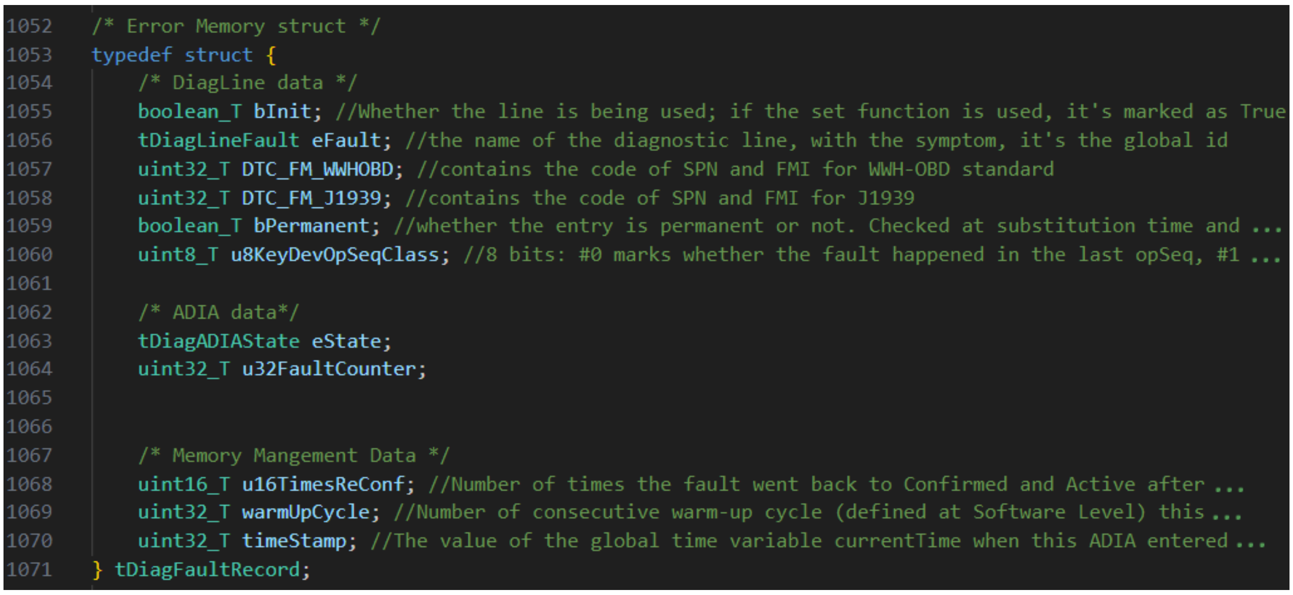
The third group contains two values that have general validity for the OBD system: the counter of the warm-up cycles, **bsWucSinceClean**, and the timer to keep track of the working engine hours, **bsHoursSinceClean**. Both entries restart to count whenever a clean-up happens.

The final group, made up of a single entry, does not actually correspond to any system variable but rather is the field where the checksum computed at the shutdown is stored. This value will then be read at startup time and compared with the new checksum computed over the readings of the other fields of the struct. A mismatch between the two checksums indicates a corruption of the stored data.

In the *".eeram"* section, after the error memory structure, we can find the corresponding NVRAM entry for the recovery lines table, **btDiagEePROM_GREC**, whose utility has already been discussed in the previous chapter.

After that array, we find another one, **btDiagEePROM_Faults**, used to store the data of the faults inserted into the error memory. Its volatile memory counterpart is **bvErrMemFaults**. Both arrays have **ERR_MEM_SIZE** entries, where the value of *ERR_MEM_SIZE* is given by a *#define* directive in *api_diagobd.h* that has been put equal to 16 based on the average number of entries of OBD error memories.

Each entry is of **tDiagFaultRecord** type and contains information from the diagnosis line, the ADIA, and some useful for the management of the entry in memory.



```c
/* Error Memory struct */
typedef struct {
    /* DiagLine data */
    boolean_T bInit; //Whether the line is being used; if the set function is used, it's marked as True
    tDiagLineFault eFault; //the name of the diagnostic line, with the symptom, it's the global id
    uint32_T DTC_FM_WWHOBD; //contains the code of SPN and FMI for WWH-OBD standard
    uint32_T DTC_FM_J1939; //contains the code of SPN and FMI for J1939
    boolean_T bPermanent; //whether the entry is permanent or not. Checked at substitution time and ...
    uint8_T u8KeyDevOpSeqClass; //8 bits: #0 marks whether the fault happened in the last opSeq, #1 ...

    /* ADIA data*/
    tDiagADIAState eState;
    uint32_T u32FaultCounter;


    /* Memory Mangement Data */
    uint16_T u16TimesReConf; //Number of times the fault went back to Confirmed and Active after ...
    uint32_T warmUpCycle; //Number of consecutive warm-up cycle (defined at Software Level) this ...
    uint32_T timeStamp; //The value of the global time variable currentTime when this ADIA entered ...
} tDiagFaultRecord;
```

Figure 5.5.3 - tDiagFaultRecord structure, in api.h

The majority of the fields connected to the diagnosis line and the ADIA can be found with the same name in the respective structures.
The **u8KeyDevOpSeqClass** field, instead, is a new 8-bit unsigned value that is used to encapsulate different information: the bit #0 is used to mark whether the fault happened

in the last operating sequence, the bit #1 marks whether the entry can be devalidated at key on, following the previously explained strategy, and the remaining bits are used to store the class (3 bits should suffice, as there are 5 possible values for the class enumerative). It's used to manage the Key Devalidation procedure.
The last three fields of the struct are used to keep count the times the fault went back to be *Confirmed* after having entered the *Previously active* state (**u16TimesReConf** field), keep track of the number of warm-up cycles spent in the *Previously active* state (**warmUpCycle** field), and keep track of when the fault entered the *Previously active* state (**timestamp** field).

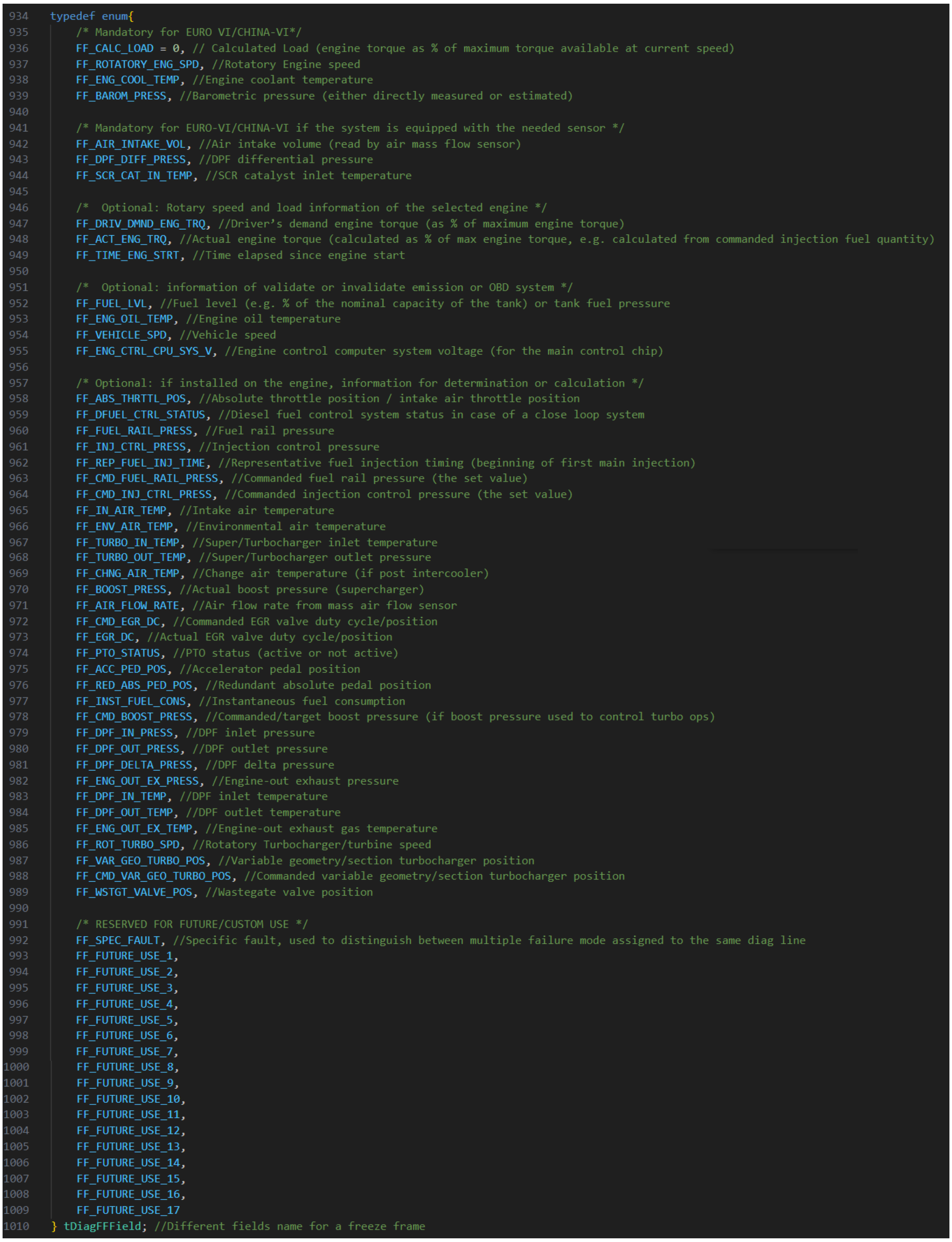
The **btDiagEePROM_FrzFrames** is a parallel array to the faults one that contains the freeze frames associated with the fault with the same index in *btDiagEePROM_Faults*. Its volatile memory counterpart is **btErrMemFrzFrames**.

Each entry of *btDiagEePROM_FrzFrames* is a freeze frame, implemented as an array of **tDiagFFEntry.** The *btDiagEePROM_FrzFrames* array's size is the same as the faults array's, *ERR_MEM_SIZE*, while the single freeze frame is **FF_N_FIELDS** long. The freeze frame length selected for this thesis work, 64 cells, corresponds to the number of labels of the **tDiagFFField** enumerative.

This enumerative contains the totality of the possible fields required by the Euro-VI and China-VI guidelines, including both mandatory and optional entries. In addition to those fields, seventeen additional labels have already been placed in anticipation of future additions wanted by the customers. This enumerative should be used to allow for easier indexing of the fields of a single freeze frame.

The *tDiagFFEntry*, which implements a single field of a freeze frame, is a struct with only two fields: a boolean flag **bAvailable** and an unsigned 16-bit value, **u16Value**. The flag ensures that the freeze frame's field is actively being used and does not just contain random values. This is useful, as although there are many fields available for the freeze frame, not all the systems are able to provide the required information for all those fields; through the use of the appropriate API and a Simulink mask (see later), it is possible to select only the desired fields.

As all the freeze frames contained in the error memory are copies of the global freeze frame, **bvGlobalFF**, it is sufficient to manipulate the entries of that freeze frame to ensure that the ones stored in the error memory will contain the desired fields with the correct values.



```c
typedef enum{
    /* Mandatory for EURO VI/CHINA-VI*/
    FF_CALC_LOAD = 0, // Calculated Load (engine torque as % of maximum torque available at current speed)
    FF_ROTATORY_ENG_SPD, //Rotatory Engine speed
    FF_ENG_COOL_TEMP, //Engine coolant temperature
    FF_BAROM_PRESS, //Barometric pressure (either directly measured or estimated)

    /* Mandatory for EURO-VI/CHINA-VI if the system is equipped with the needed sensor */
    FF_AIR_INTAKE_VOL, //Air intake volume (read by air mass flow sensor)
    FF_DPF_DIFF_PRESS, //DPF differential pressure
    FF_SCR_CAT_IN_TEMP, //SCR catalyst inlet temperature

    /*  Optional: Rotary speed and load information of the selected engine */
    FF_DRIV_DMND_ENG_TRQ, //Driver’s demand engine torque (as % of maximum engine torque)
    FF_ACT_ENG_TRQ, //Actual engine torque (calculated as % of max engine torque, e.g. calculated from commanded injection fuel quantity)
    FF_TIME_ENG_STRT, //Time elapsed since engine start

    /*  Optional: information of validate or invalidate emission or OBD system */
    FF_FUEL_LVL, //Fuel level (e.g. % of the nominal capacity of the tank) or tank fuel pressure
    FF_ENG_OIL_TEMP, //Engine oil temperature
    FF_VEHICLE_SPD, //Vehicle speed
    FF_ENG_CTRL_CPU_SYS_V, //Engine control computer system voltage (for the main control chip)

    /* Optional: if installed on the engine, information for determination or calculation */
    FF_ABS_THRTTL_POS, //Absolute throttle position / intake air throttle position
    FF_DFUEL_CTRL_STATUS, //Diesel fuel control system status in case of a close loop system
    FF_FUEL_RAIL_PRESS, //Fuel rail pressure
    FF_INJ_CTRL_PRESS, //Injection control pressure
    FF_REP_FUEL_INJ_TIME, //Representative fuel injection timing (beginning of first main injection)
    FF_CMD_FUEL_RAIL_PRESS, //Commanded fuel rail pressure (the set value)
    FF_CMD_INJ_CTRL_PRESS, //Commanded injection control pressure (the set value)
    FF_IN_AIR_TEMP, //Intake air temperature
    FF_ENV_AIR_TEMP, //Environmental air temperature
    FF_TURBO_IN_TEMP, //Super/Turbocharger inlet temperature
    FF_TURBO_OUT_TEMP, //Super/Turbocharger outlet pressure
    FF_CHNG_AIR_TEMP, //Change air temperature (if post intercooler)
    FF_BOOST_PRESS, //Actual boost pressure (supercharger)
    FF_AIR_FLOW_RATE, //Air flow rate from mass air flow sensor
    FF_CMD_EGR_DC, //Commanded EGR valve duty cycle/position
    FF_EGR_DC, //Actual EGR valve duty cycle/position
    FF_PTO_STATUS, //PTO status (active or not active)
    FF_ACC_PED_POS, //Accelerator pedal position
    FF_RED_ABS_PED_POS, //Redundant absolute pedal position
    FF_INST_FUEL_CONS, //Instantaneous fuel consumption
    FF_CMD_BOOST_PRESS, //Commanded/target boost pressure (if boost pressure used to control turbo ops)
    FF_DPF_IN_PRESS, //DPF inlet pressure
    FF_DPF_OUT_PRESS, //DPF outlet pressure
    FF_DPF_DELTA_PRESS, //DPF delta pressure
    FF_ENG_OUT_EX_PRESS, //Engine-out exhaust pressure
    FF_DPF_IN_TEMP, //DPF inlet temperature
    FF_DPF_OUT_TEMP, //DPF outlet temperature
    FF_ENG_OUT_EX_TEMP, //Engine-out exhaust gas temperature
    FF_ROT_TURBO_SPD, //Rotatory Turbocharger/turbine speed
    FF_VAR_GEO_TURBO_POS, //Variable geometry/section turbocharger position
    FF_CMD_VAR_GEO_TURBO_POS, //Commanded variable geometry/section turbocharger position
    FF_WSTGT_VALVE_POS, //Wastegate valve position

    /* RESERVED FOR FUTURE/CUSTOM USE */
    FF_SPEC_FAULT, //Specific fault, used to distinguish between multiple failure mode assigned to the same diag line
    FF_FUTURE_USE_1,
    FF_FUTURE_USE_2,
    FF_FUTURE_USE_3,
    FF_FUTURE_USE_4,
    FF_FUTURE_USE_5,
    FF_FUTURE_USE_6,
    FF_FUTURE_USE_7,
    FF_FUTURE_USE_8,
    FF_FUTURE_USE_9,
    FF_FUTURE_USE_10,
    FF_FUTURE_USE_11,
    FF_FUTURE_USE_12,
    FF_FUTURE_USE_13,
    FF_FUTURE_USE_14,
    FF_FUTURE_USE_15,
    FF_FUTURE_USE_16,
    FF_FUTURE_USE_17
} tDiagFFField; //Different fields name for a freeze frame
```

Figure 5.5.4 – The tDiagFFField enumerative used to index the fields of the freeze frames, in api.h

The choice of reducing the connection between a fault in the error memory and its freeze frame to a purely logical one through indexing rather than encapsulating the freeze frame inside of the *tDiagFaultRecord* structure was made with the idea of improving the access speed of the system in mind.

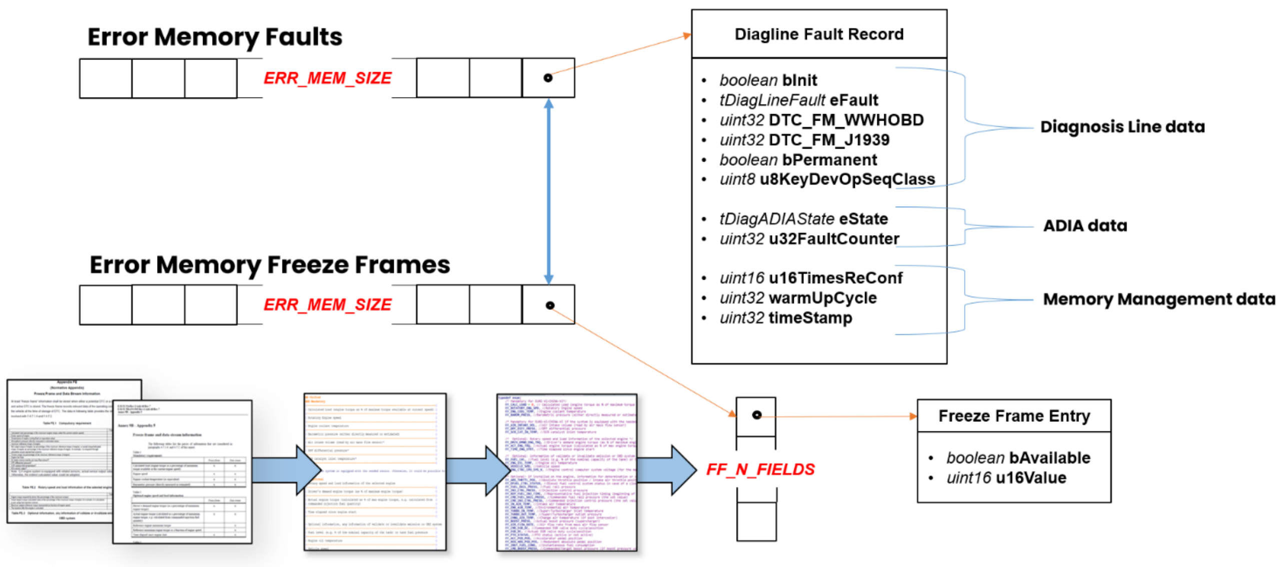


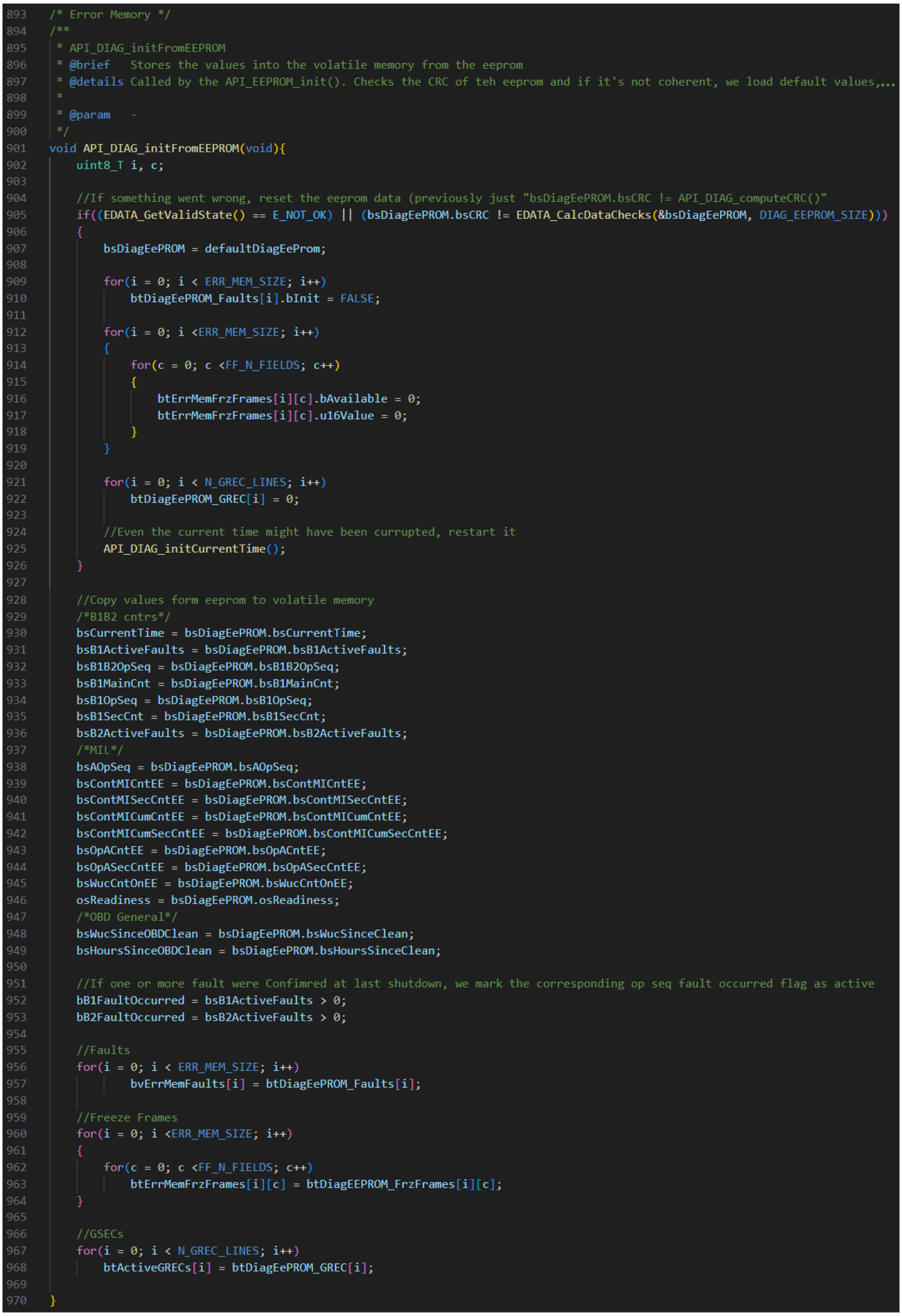
Figure 5.5.5 - Visual representation of the logical link between the faults and the freeze frames structures

Similarly, the idea behind storing the global recovery lines table rather than the single table for each diagnosis line was made to both speed up the process and reduce the amount of non-volatile memory required.

Now that we've seen the data structures used to implement the strategy, let's examine the functions that operate on these structures.

The initialisation of the structures starts with the already mentioned *API_DIAG_OnInit*. This function begins by calling the **API_DIAG_initFromEEPROM** procedure, which loads the values into the volatile memory from the NVRAM correspondents.

It operates by computing the checksum on the fields of the error memory structure and comparing it with the checksum stored in the structure. The function verifies the validity of the EEPROM entries using the base-level software APIs, **EDATA_GetValidState** and **EDATA_CalcDataChecks**. If the two values do not correspond, it loads the default values in the NVRAM structures. It then goes on to copy the data from the NVRAM into the volatile memory.



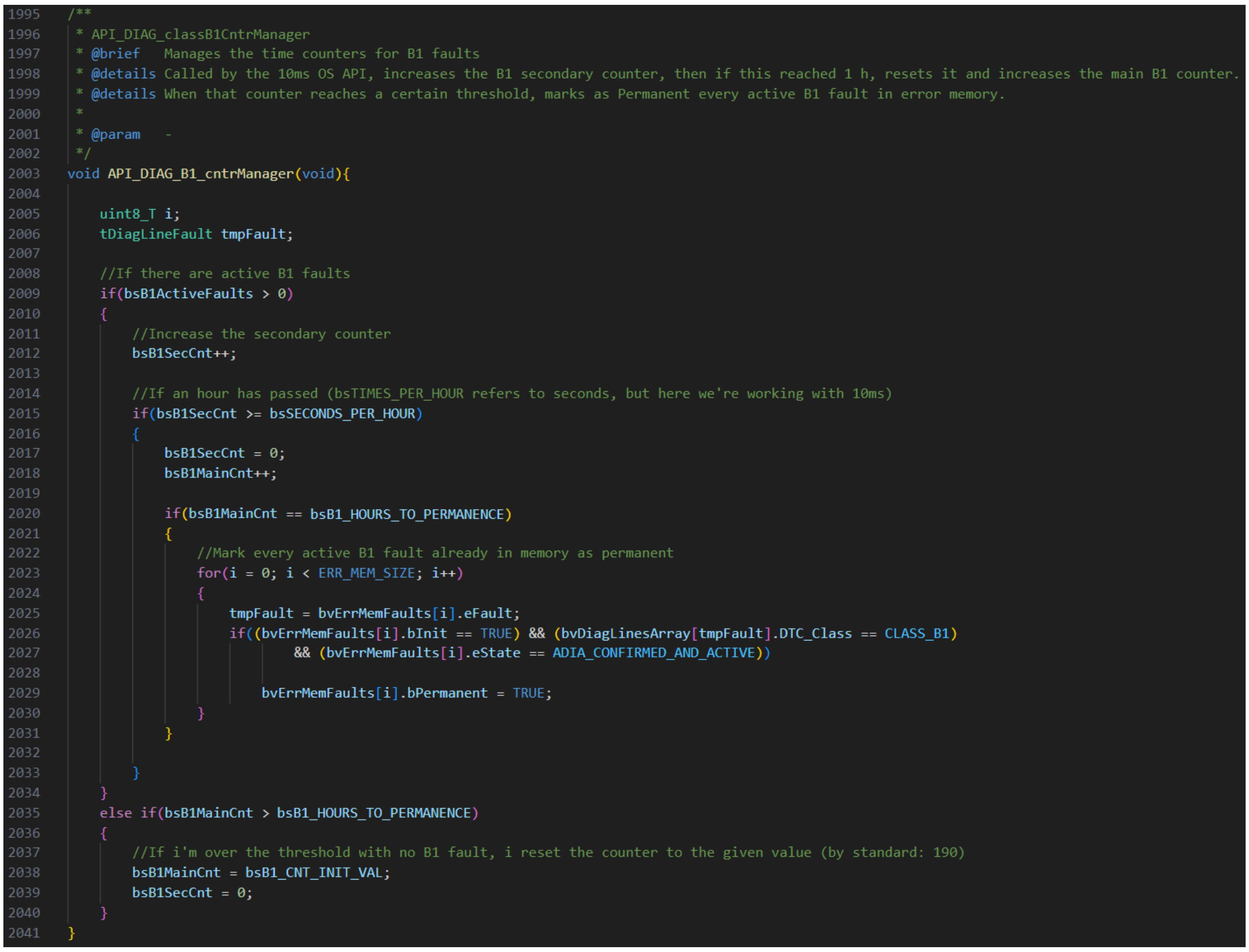
```c
/* Error Memory */
/**
 * API_DIAG_initFromEEPROM
 * @brief   Stores the values into the volatile memory from the eeprom
 * @details Called by the API_EEPROM_init(). Checks the CRC of teh eeprom and if it's not coherent, we load default values,...
 *
 * @param   -
 */
void API_DIAG_initFromEEPROM(void){
    uint8_T i, c;

    //If something went wrong, reset the eeprom data (previously just "bsDiagEePROM.bsCRC != API_DIAG_computeCRC()"
    if((EDATA_GetValidState() == E_NOT_OK) || (bsDiagEePROM.bsCRC != EDATA_CalcDataChecks(&bsDiagEePROM, DIAG_EEPROM_SIZE)))
    {
        bsDiagEePROM = defaultDiagEeProm;

        for(i = 0; i < ERR_MEM_SIZE; i++)
            btDiagEePROM_Faults[i].bInit = FALSE;

        for(i = 0; i <ERR_MEM_SIZE; i++)
        {
            for(c = 0; c <FF_N_FIELDS; c++)
            {
                btErrMemFrzFrames[i][c].bAvailable = 0;
                btErrMemFrzFrames[i][c].u16Value = 0;
            }
        }

        for(i = 0; i < N_GREC_LINES; i++)
            btDiagEePROM_GREC[i] = 0;

        //Even the current time might have been currupted, restart it
        API_DIAG_initCurrentTime();
    }

    //Copy values form eeprom to volatile memory
    /*B1B2 cntrs*/
    bsCurrentTime = bsDiagEePROM.bsCurrentTime;
    bsB1ActiveFaults = bsDiagEePROM.bsB1ActiveFaults;
    bsB1B2OpSeq = bsDiagEePROM.bsB1B2OpSeq;
    bsB1MainCnt = bsDiagEePROM.bsB1MainCnt;
    bsB1OpSeq = bsDiagEePROM.bsB1OpSeq;
    bsB1SecCnt = bsDiagEePROM.bsB1SecCnt;
    bsB2ActiveFaults = bsDiagEePROM.bsB2ActiveFaults;
    /*MIL*/
    bsAOpSeq = bsDiagEePROM.bsAOpSeq;
    bsContMICntEE = bsDiagEePROM.bsContMICntEE;
    bsContMISecCntEE = bsDiagEePROM.bsContMISecCntEE;
    bsContMICumCntEE = bsDiagEePROM.bsContMICumCntEE;
    bsContMICumSecCntEE = bsDiagEePROM.bsContMICumSecCntEE;
    bsOpACntEE = bsDiagEePROM.bsOpACntEE;
    bsOpASecCntEE = bsDiagEePROM.bsOpASecCntEE;
    bsWucCntOnEE = bsDiagEePROM.bsWucCntOnEE;
    osReadiness = bsDiagEePROM.osReadiness;
    /*OBD General*/
    bsWucSinceOBDClean = bsDiagEePROM.bsWucSinceClean;
    bsHoursSinceOBDClean = bsDiagEePROM.bsHoursSinceClean;

    //If one or more fault were Confimred at last shutdown, we mark the corresponding op seq fault occurred flag as active
    bB1FaultOccurred = bsB1ActiveFaults > 0;
    bB2FaultOccurred = bsB2ActiveFaults > 0;

    //Faults
    for(i = 0; i < ERR_MEM_SIZE; i++)
            bvErrMemFaults[i] = btDiagEePROM_Faults[i];

    //Freeze Frames
    for(i = 0; i <ERR_MEM_SIZE; i++)
    {
        for(c = 0; c <FF_N_FIELDS; c++)
            btErrMemFrzFrames[i][c] = btDiagEEPROM_FrzFrames[i][c];
    }

    //GSECs
    for(i = 0; i < N_GREC_LINES; i++)
        btActiveGRECs[i] = btDiagEePROM_GREC[i];

}
```

Figure 5.5.6 - API_DIAG_initFromEEPROM, in api_diagobd.c

Once the variables and structures have been initialised with the data read from memory, the system can go on and operate normally. The 'operating' part of the strategy includes both ADIA-related tasks and periodical tasks.

The first of the periodic tasks is **API_DIAG_updateCurrentTime**, which is called by the operating system's *API_OS_Task1s*. The function simply increases the **bsCurrentTime** global variable, avoiding race conditions. Along with the *update* function, two other methods have been defined to instantiate and read the variable's value: **API_DIAG_initCurrentTime** (used when resetting the system after a memory failure) and **API_DIAG_getCurrentTime**.

Another periodical task is the one that manages the counters and timers related to the class B1 faults. The **API_DIAG_B1_cntrManager** is called by the operating system's *API_OS_Task10ms* task.



```c
/**
 * API_DIAG_classB1CntrManager
 * @brief   Manages the time counters for B1 faults
 * @details Called by the 10ms OS API, increases the B1 secondary counter, then if this reached 1 h, resets it and increases the main B1 counter.
 * @details When that counter reaches a certain threshold, marks as Permanent every active B1 fault in error memory.
 *
 * @param   -
 */
void API_DIAG_B1_cntrManager(void){

    uint8_T i;
    tDiagLineFault tmpFault;

    //If there are active B1 faults
    if(bsB1ActiveFaults > 0)
    {
        //Increase the secondary counter
        bsB1SecCnt++;

        //If an hour has passed (bsTIMES_PER_HOUR refers to seconds, but here we're working with 10ms)
        if(bsB1SecCnt >= bsSECONDS_PER_HOUR)
        {
            bsB1SecCnt = 0;
            bsB1MainCnt++;

            if(bsB1MainCnt == bsB1_HOURS_TO_PERMANENCE)
            {
                //Mark every active B1 fault already in memory as permanent
                for(i = 0; i < ERR_MEM_SIZE; i++)
                {
                    tmpFault = bvErrMemFaults[i].eFault;
                    if((bvErrMemFaults[i].bInit == TRUE) && (bvDiagLinesArray[tmpFault].DTC_Class == CLASS_B1)
                            && (bvErrMemFaults[i].eState == ADIA_CONFIRMED_AND_ACTIVE))

                        bvErrMemFaults[i].bPermanent = TRUE;
                }
            }

        }
    }
    else if(bsB1MainCnt > bsB1_HOURS_TO_PERMANENCE)
    {
        //If i'm over the threshold with no B1 fault, i reset the counter to the given value (by standard: 190)
        bsB1MainCnt = bsB1_CNT_INIT_VAL;
        bsB1SecCnt = 0;
    }
}
```

Figure 5.5.7 - API_DIAG_B1_cntrManager, in api_diagobd.c

The function starts by checking whether there are any active B1 faults.

If that's the case, it increases the secondary timer, **bsB1SecCnt**, by one. Then, if the counter equals or exceeds the value of **bsSECONDS_PER_HOUR** (defined as a calibration for testing purposes) and thus an hour has passed, the secondary counter is reset while the main one, **bsB1MainCnt**, is increased by one unit. After that, the bsB1MainCnt value is compared to the **bsB1_HOURS_TO_PERMANENCE** threshold, by standards 200 hours, and if the two values are the same, all the currently active B1 faults in the error memory are marked as permanent failure codes.

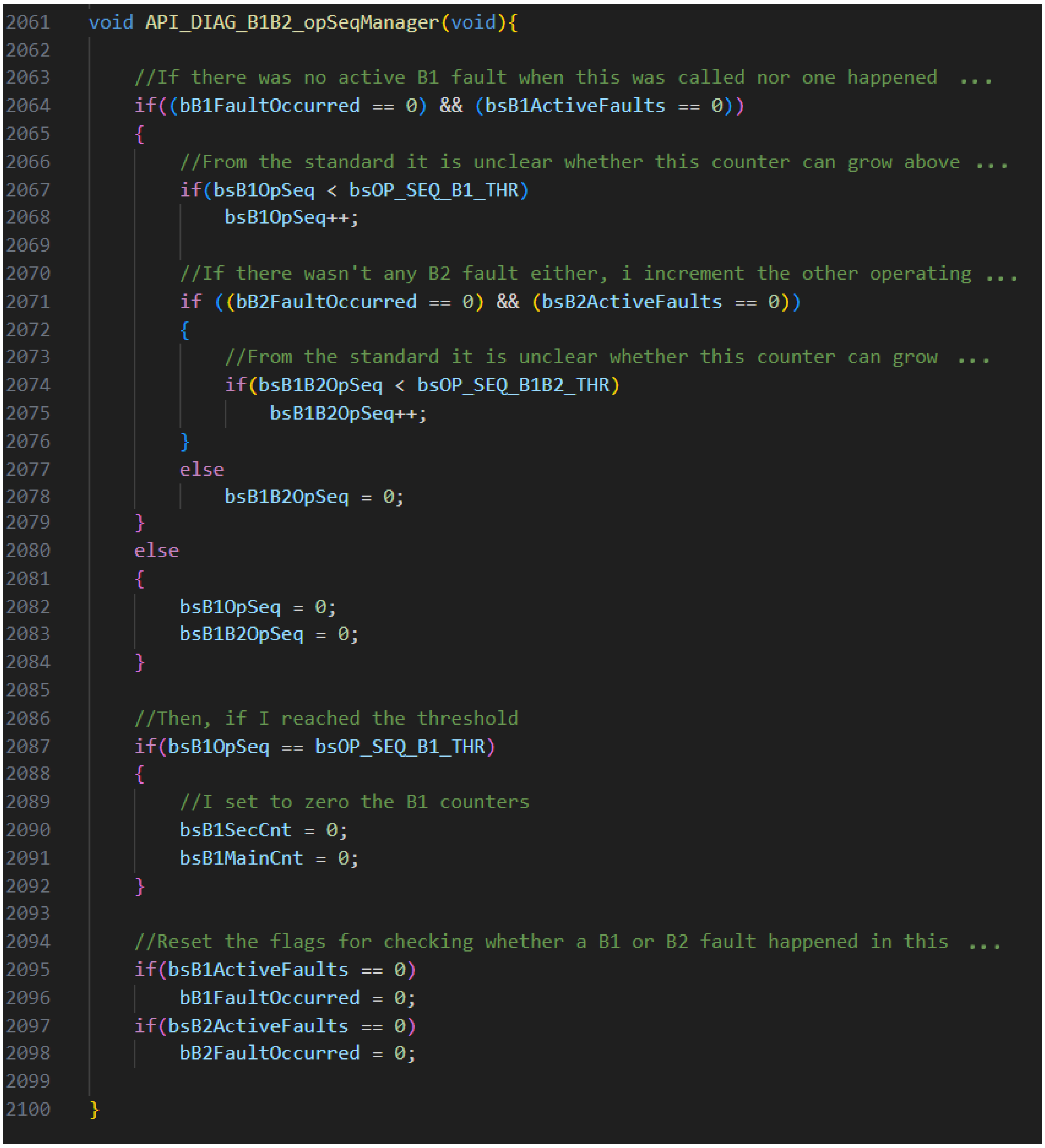
In case that no B1 fault is currently active, the function directly checks whether the *bsB1MainCnt* counter exceeds the *bsSECONDS_PER_HOUR* threshold and, in that case, resets the secondary counter and sets the main counter to **bsB1_CNT_INIT_VAL**, which by standards should be 190 hours.

Another function that allows to manage different variables of the system, but is not part of the periodic tasks, is the one to signal the completion of a single operating sequence: **API_DIAG_completeOperatingSequence**. This API is exported at model level to allow the users to design the signalling of the passing of an operating sequence, as the definition of it is not standardised and thus must be delegated to the manufacturer.

The *API_DIAG_completeOperatingSequence* is actually a wrapper function that calls three other methods: **API_DIAG_A_opSeqManager**, **API_DIAG_B1B2_opSeqManager**, and **API_DIAG_KeyDevalManager**.

API_DIAG_A_opSeqManager is used to manage the counters for operating sequences without continuous MI faults for the MIL. A more detailed description will be provided in the lamp management chapter.

API_DIAG_opSeqManagerB1B2 is used to manage the counter of operating sequences without B1 and B2 faults. It checks that no active B1 faults are present and that none occurred since the last time the operating sequence function has been called.
It does so by checking that the counter **dsB1ActiveFaults** equals 0 and the same is true for the flag **bB1FaultOccurred**.



```c
void API_DIAG_B1B2_opSeqManager(void){

    //If there was no active B1 fault when this was called nor one happened  ...
    if((bB1FaultOccurred == 0) && (bsB1ActiveFaults == 0))
    {
        //From the standard it is unclear whether this counter can grow above ...
        if(bsB1OpSeq < bsOP_SEQ_B1_THR)
            bsB1OpSeq++;

        //If there wasn't any B2 fault either, i increment the other operating ...
        if ((bB2FaultOccurred == 0) && (bsB2ActiveFaults == 0))
        {
            //From the standard it is unclear whether this counter can grow  ...
            if(bsB1B2OpSeq < bsOP_SEQ_B1B2_THR)
                bsB1B2OpSeq++;
        }
        else
            bsB1B2OpSeq = 0;
    }
    else
    {
        bsB1OpSeq = 0;
        bsB1B2OpSeq = 0;
    }

    //Then, if I reached the threshold
    if(bsB1OpSeq == bsOP_SEQ_B1_THR)
    {
        //I set to zero the B1 counters
        bsB1SecCnt = 0;
        bsB1MainCnt = 0;
    }

    //Reset the flags for checking whether a B1 or B2 fault happened in this  ...
    if(bsB1ActiveFaults == 0)
        bB1FaultOccurred = 0;
    if(bsB2ActiveFaults == 0)
        bB2FaultOccurred = 0;

}
```

Figure 5.5.8 - API_DIAG_B1B2_opSeqManager, in api_diagobd.c

If the conditions hold true, it increases **dsB1OpSeq** by one, and it also checks if no B2 happened nor is any active (checking the **bB2FaultOccurred** flag and the **dsB2ActiveFaults** counter), and in that case increases the **dsB1B2OpSeq** (that will be used for the lamps); otherwise, it sets **dsB1B2OpSeq** to 0.

If any of the conditions is false, the function sets the **dsB1OpSeq** and **dsB1B2OpSeq** to 0.

If the **dsB1OpSeq** has reached the calibratable threshold **dsOP_SEQ_B1_THR** (by standards, 3), the function resets the B1 time counters to 0.
Finally, the API sets the two flags **bB1FaultOccurred** and **bB1B2FaultOccurred** to 0 if the corresponding counter is equal to 0 (it is implied that no faults of that type are currently active).

The last function called by the *API_DIAG_completeOperatingSequence* wrapper, *API_DIAG_KeyDevalManager*, manages the devalidation of error memory entries whose ADIAs have the *bKeyDeval* flag set to true.

The 'key devalidation' strategy works like this:

1. When a fault is added to the error memory, the *u8KeyDevOpSeqClass* field is set to |000|DTC_CLASS(3bit)|KeyDev(1bit)|1|, taking the *DTC_CLASS* from the diagnosis line and the *KeyDev* bit from the ADIA. This happens inside the function to insert a new entry in the error memory, **API_DIAG_ErrMem_insert**.
2. When an entry goes back from *Previously active* to *Confirmed and active,* or when it remains in *Confirmed*, the API for the update of an entry is called, and the *u8KeyDevOpSeqClass* field is reset to |000|DTC_CLASS(3bit)|KeyDev(1bit)|1|.
3. When an operating sequence has passed (signalled using the *completeOpSeq* API) or when the key-off routine is executed, the system calls the *API_DIAG_KeyDevalManager* function.

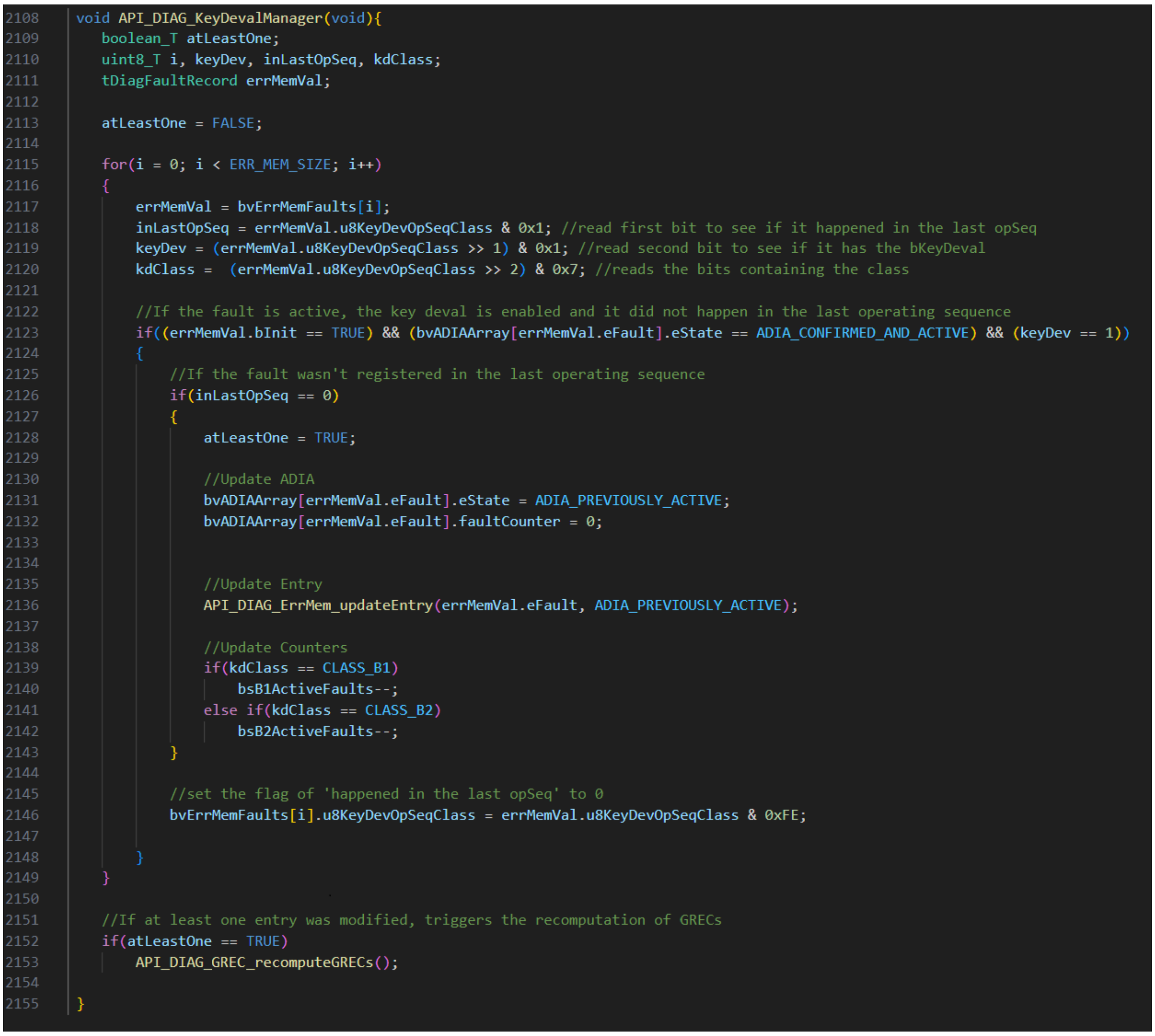
This function runs through the array of faults in the error memory, and for each of the active entries, it checks the *u8KeyDevOpSeqClass* of that entry. It reads each subfield of the value by performing a logical operation:

- Bitwise 'and' with 0x01 to read the value that indicates whether the fault happened in the last operating sequence. The result is stored in the support variable **inLastOpSeq**.
- Right shift by 1 position followed by a bitwise 'and' with 0x01 to check that the entry has an active 'KeyDev' flag. The result is stored in **keyDev**.
- Right shift by two positions followed by a bitwise 'and' with 0x07 to check the three bits that indicate the fault's class. The obtained value is stored in **kdClass**.

The function then checks which of the entries is currently active and that *keyDev* is set to one before proceeding. If that's the case, the function sets the first bit of the *u8KeyDevOpSeqClass* to 0, to mark that the fault hasn't happened yet in this new operating sequence, by applying a bitwise 'and' between the field and the value 0xFE.

In addition, if *inLastOpSeq* was already at zero, which means that the fault did not present itself in the last operating sequence, the function sets the **atLeastOne** variable to *true* and updates the state to *Previously active* and the *faultCounter* to 0 for both the entry and the ADIA. It then updates the active fault counter of the devalidate fault's class.

If the state of one or more entries might have changed, and thus *atLeastOne* is set to *true*, the function calls the API to recompute the active recovery lines, *API_DIAG_recomputeGRECs*.



```
void API_DIAG_KeyDevalManager(void){
    boolean_T atLeastOne;
    uint8_T i, keyDev, inLastOpSeq, kdClass;
    tDiagFaultRecord errMemVal;

    atLeastOne = FALSE;

    for(i = 0; i < ERR_MEM_SIZE; i++)
    {
        errMemVal = bvErrMemFaults[i];
        inLastOpSeq = errMemVal.u8KeyDevOpSeqClass & 0x1; //read first bit to see if it happened in the last opSeq
        keyDev = (errMemVal.u8KeyDevOpSeqClass >> 1) & 0x1; //read second bit to see if it has the bKeyDeval
        kdClass =  (errMemVal.u8KeyDevOpSeqClass >> 2) & 0x7; //reads the bits containing the class

        //If the fault is active, the key deval is enabled and it did not happen in the last operating sequence
        if((errMemVal.bInit == TRUE) && (bvADIAArray[errMemVal.eFault].eState == ADIA_CONFIRMED_AND_ACTIVE) && (keyDev == 1))
        {
            //If the fault wasn't registered in the last operating sequence
            if(inLastOpSeq == 0)
            {
                atLeastOne = TRUE;

                //Update ADIA
                bvADIAArray[errMemVal.eFault].eState = ADIA_PREVIOUSLY_ACTIVE;
                bvADIAArray[errMemVal.eFault].faultCounter = 0;


                //Update Entry
                API_DIAG_ErrMem_updateEntry(errMemVal.eFault, ADIA_PREVIOUSLY_ACTIVE);

                //Update Counters
                if(kdClass == CLASS_B1)
                    bsB1ActiveFaults--;
                else if(kdClass == CLASS_B2)
                    bsB2ActiveFaults--;
            }

            //set the flag of 'happened in the last opSeq' to 0
            bvErrMemFaults[i].u8KeyDevOpSeqClass = errMemVal.u8KeyDevOpSeqClass & 0xFE;

        }
    }

    //If at least one entry was modified, triggers the recomputation of GRECs
    if(atLeastOne == TRUE)
        API_DIAG_GREC_recomputeGRECs();

}
```

Figure 5.5.9 - API_DIAG_KeyDevalManager, in api_diagobd.c

An additional API used at the model level to signal a specific event is the one responsible for managing warm-up cycles: **API_DIAG_completeWarmUpCycle**. This function simply sifts through the *bvErrMemFaults* array looking for faults in the *Previously active* state. Once one of such faults has been found, the function decreases the entry's field that indicates the number for the remaining warm-up cycle before being removed from the error memory. Once it finishes analysing the entries, the function increments the global counter **bsWucSinceOBDClean,** marking another completed warm-up cycle since the last OBD reset, and then returns the number of modified entries.

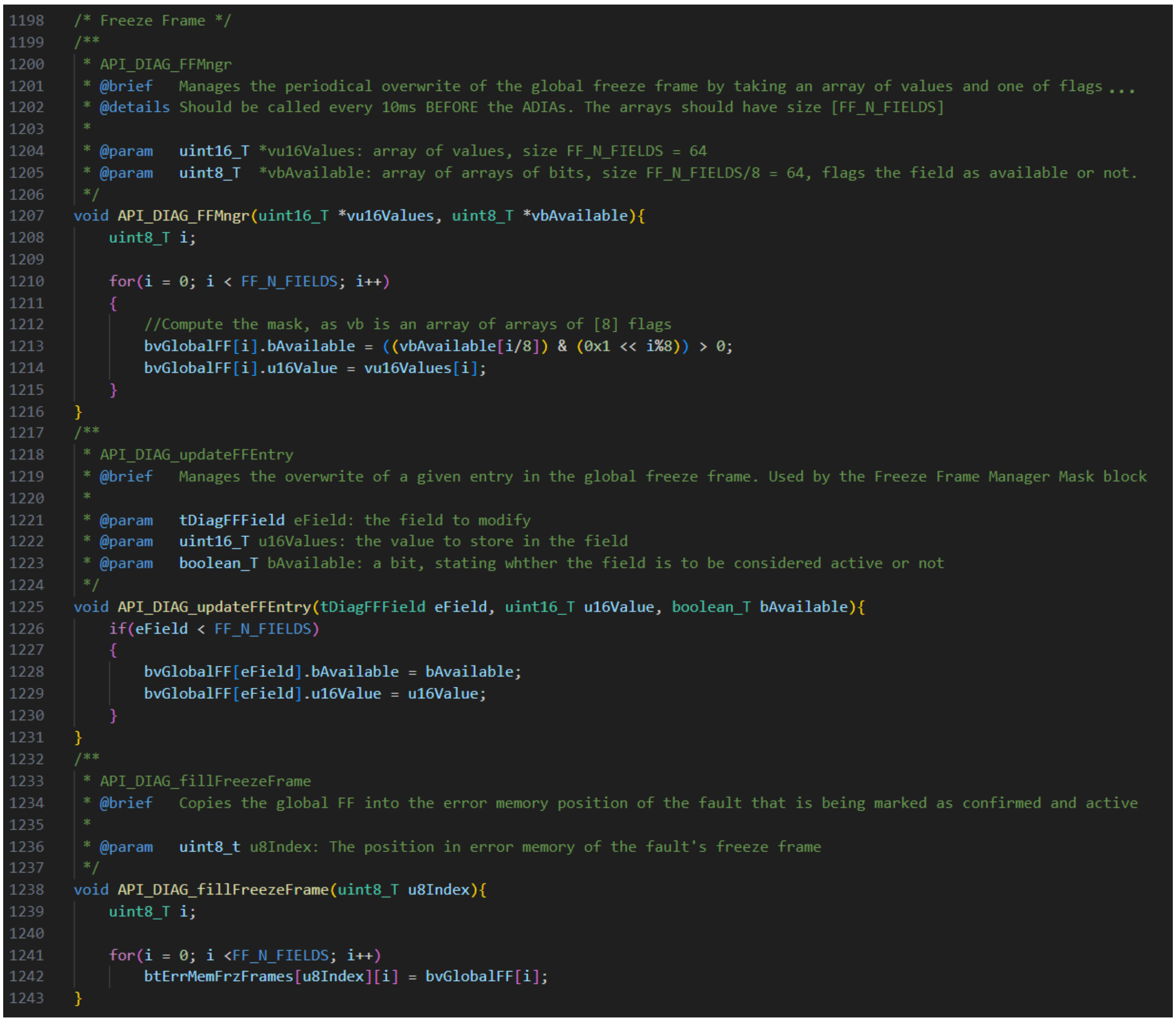
Other functions that could be considered directly under control of the user, i.e., APIs exported at model level, are those connected to the management of the freeze frames. More precisely, the functions that can be used to update the contents of the global freeze frame, whose data will be copied in the error memory when a fault reaches the *Confirmed and active* state.

The available functions to update the contents of the global freeze frame are **API_DIAG_FFMngr** and **API_DIAG_updateFFEntry**. The first one allows to update the whole freeze frame with a single call, while the latter works on a single field of the freeze frame.

API_DIAG_FFMngr takes as inputs an array of *FF_N_FIELDS* 16-bit unsigned values and an array of FF_N_FIELDS/8 8-bit unsigned entries. The second array works as an array of arrays of flags, with 8 flags per row. This was originally done to save space. The function then updates every field of the freeze frame with the value and the availability bit read from the entries of the input arrays in the same position (after having computed the value in the 'availability flags' array applying a set of bitwise transformations).

API_DIAG_updateFFEntry provides a more direct and punctual approach, and it's the function used by the 'Freeze Frame mask'. The function takes as parameters the value of the field and the validity flag, along with the enumerative to index the freeze frame field, and then simply assigns the first two to the corresponding fields of the indexed *tDiagFFEntry*.

The last function related to the freeze frames is the one used to copy the global freeze frame into the new freeze frame in the error memory, **API_DIAG_fillFreezeFrame**, called by the ADIA when moving to the *Confirmed and active* status.



```c
/* Freeze Frame */
/**
 * API_DIAG_FFMngr
 * @brief   Manages the periodical overwrite of the global freeze frame by taking an array of values and one of flags ...
 * @details Should be called every 10ms BEFORE the ADIAs. The arrays should have size [FF_N_FIELDS]
 *
 * @param   uint16_T *vu16Values: array of values, size FF_N_FIELDS = 64
 * @param   uint8_T  *vbAvailable: array of arrays of bits, size FF_N_FIELDS/8 = 64, flags the field as available or not.
 */
void API_DIAG_FFMngr(uint16_T *vu16Values, uint8_T *vbAvailable){
    uint8_T i;

    for(i = 0; i < FF_N_FIELDS; i++)
    {
        //Compute the mask, as vb is an array of arrays of [8] flags
        bvGlobalFF[i].bAvailable = ((vbAvailable[i/8]) & (0x1 << i%8)) > 0;
        bvGlobalFF[i].u16Value = vu16Values[i];
    }
}
/**
 * API_DIAG_updateFFEntry
 * @brief   Manages the overwrite of a given entry in the global freeze frame. Used by the Freeze Frame Manager Mask block
 *
 * @param   tDiagFFField eField: the field to modify
 * @param   uint16_T u16Values: the value to store in the field
 * @param   boolean_T bAvailable: a bit, stating whther the field is to be considered active or not
 */
void API_DIAG_updateFFEntry(tDiagFFField eField, uint16_T u16Value, boolean_T bAvailable){
    if(eField < FF_N_FIELDS)
    {
        bvGlobalFF[eField].bAvailable = bAvailable;
        bvGlobalFF[eField].u16Value = u16Value;
    }
}
/**
 * API_DIAG_fillFreezeFrame
 * @brief   Copies the global FF into the error memory position of the fault that is being marked as confirmed and active
 *
 * @param   uint8_t u8Index: The position in error memory of the fault's freeze frame
 */
void API_DIAG_fillFreezeFrame(uint8_T u8Index){
    uint8_T i;

    for(i = 0; i <FF_N_FIELDS; i++)
        btErrMemFrzFrames[u8Index][i] = bvGlobalFF[i];
}
```

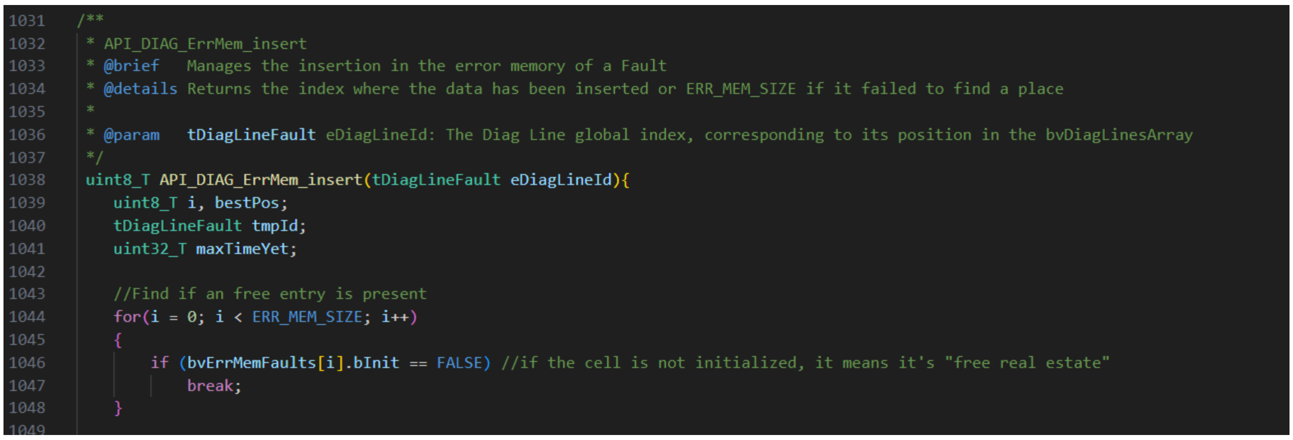
Figure 5.5.10 - API_DIAG_FFMngr, API_DIAG_updateFFEntry, and API_DIAG_fillFreezeFrame, in api_diagobd.c

As stated in the ‘strategy’ section of this chapter, the ADIA doesn’t simply manage the copy of the global freeze frame but is also responsible for the insertion, update, and erasure of the faults into the error memory.

The function for the insertion of the faults is **API_DIAG_ErrMem_insert** and is called by the ADIA when passing from the *Pending* status to the *Confirmed and active* one. It takes as input the ID of the fault to insert and returns either the position where it has been inserted or the size of the error memory faults array, *ERR_MEM_SIZE*, if it failed to find an entry.

The *API_DIAG_ErrMem_insert* function can be separated into three parts: the part where we look for an empty entry, the case in which we have to try to find an occupied entry, and the part where the fault is inserted into the array.

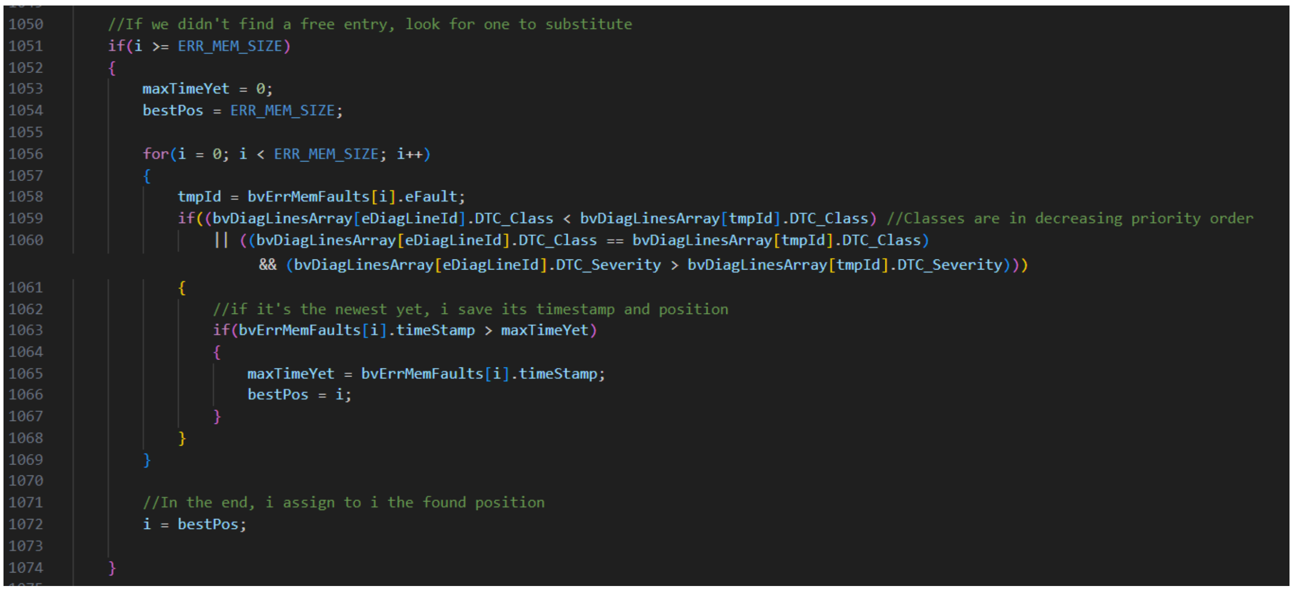
The first part simply consists in a ‘for’ loop that sifts through the error memory faults array and stops whenever it finds an empty entry, i.e., an entry whose *bInit* field is set to *false*.



```
/**
 * API_DIAG_ErrMem_insert
 * @brief   Manages the insertion in the error memory of a Fault
 * @details Returns the index where the data has been inserted or ERR_MEM_SIZE if it failed to find a place
 *
 * @param   tDiagLineFault eDiagLineId: The Diag Line global index, corresponding to its position in the bvDiagLinesArray
 */
uint8_T API_DIAG_ErrMem_insert(tDiagLineFault eDiagLineId){
    uint8_T i, bestPos;
    tDiagLineFault tmpId;
    uint32_T maxTimeYet;

    //Find if an free entry is present
    for(i = 0; i < ERR_MEM_SIZE; i++)
    {
        if (bvErrMemFaults[i].bInit == FALSE) //if the cell is not initialized, it means it's "free real estate"
            break;
    }
```

Figure 5.5.11a - First part of API_DIAG_ErrMem_insert, in api_diagobd.c

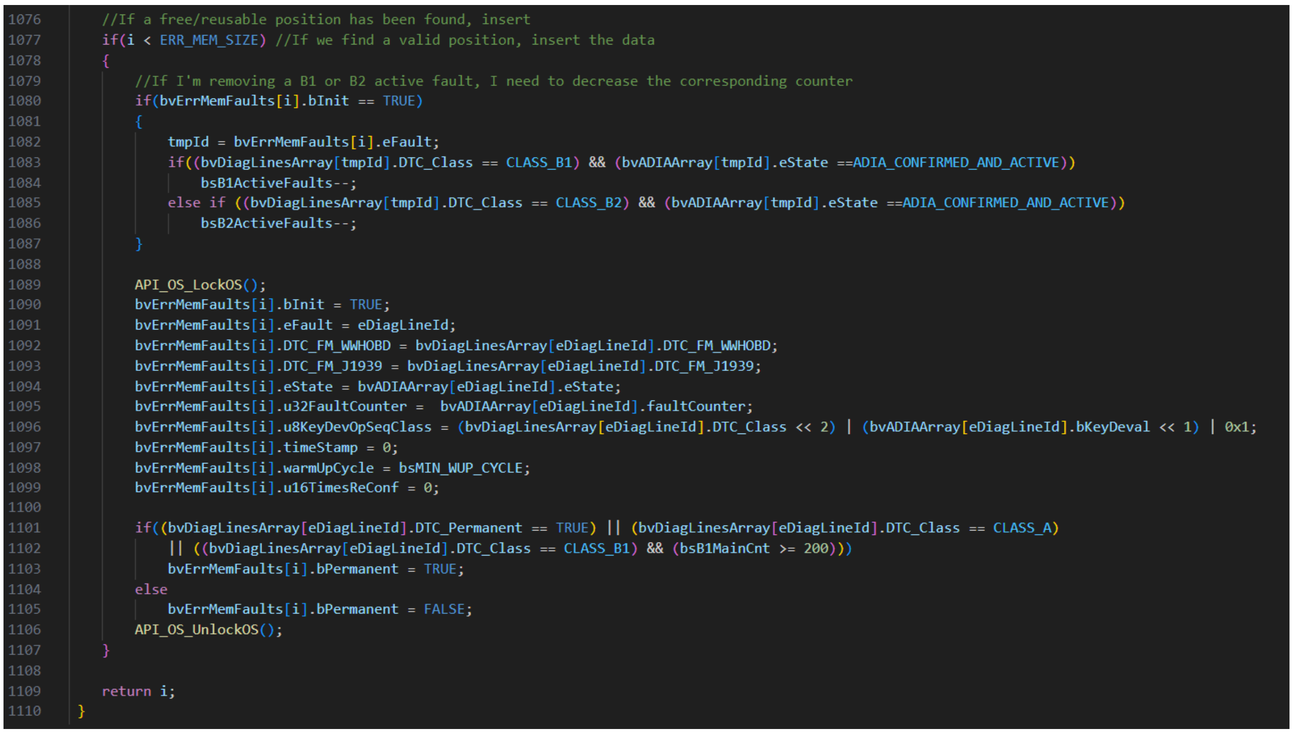
The second part is executed only if no empty slot was found, as that means that the counter used in the first part’s loop reached *ERR_MEM_SIZE*. In this case, we look for the best position to replace; we try to find either an entry with a lower class or one with the same class but with higher severity.
Whenever a suitable entry is found, we check that its timestamp is the greatest found so far. In this way, we’ll replace the newest suitable entry, preserving the older ones as per guidelines.
At the end of this part, we’ll either have found the best position where to insert the new fault, or we’ll still have the loop index equal to *ERR_MEM_SIZE*.



```
    //If we didn't find a free entry, look for one to substitute
    if(i >= ERR_MEM_SIZE)
    {
        maxTimeYet = 0;
        bestPos = ERR_MEM_SIZE;

        for(i = 0; i < ERR_MEM_SIZE; i++)
        {
            tmpId = bvErrMemFaults[i].eFault;
            if((bvDiagLinesArray[eDiagLineId].DTC_Class < bvDiagLinesArray[tmpId].DTC_Class) //Classes are in decreasing priority order
                || ((bvDiagLinesArray[eDiagLineId].DTC_Class == bvDiagLinesArray[tmpId].DTC_Class)
                     && (bvDiagLinesArray[eDiagLineId].DTC_Severity > bvDiagLinesArray[tmpId].DTC_Severity)))
            {
                //if it's the newest yet, i save its timestamp and position
                if(bvErrMemFaults[i].timeStamp > maxTimeYet)
                {
                    maxTimeYet = bvErrMemFaults[i].timeStamp;
                    bestPos = i;
                }
            }
        }

        //In the end, i assign to i the found position
        i = bestPos;

    }
```

Figure 5.5.11b - Second part of API_DIAG_ErrMem_insert, in api_diagobd.c

The last part of the function is only executed if a valid position has been found in the previous steps. The function checks whether the found position contains a B1 or B2 class fault in the *Confirmed and active* state, and in that case decreases the corresponding counter.
The function then goes on to acquire the lock on the critical section, where it will copy or create the required data to store in the error memory entry.
It then releases the lock and returns the found position.



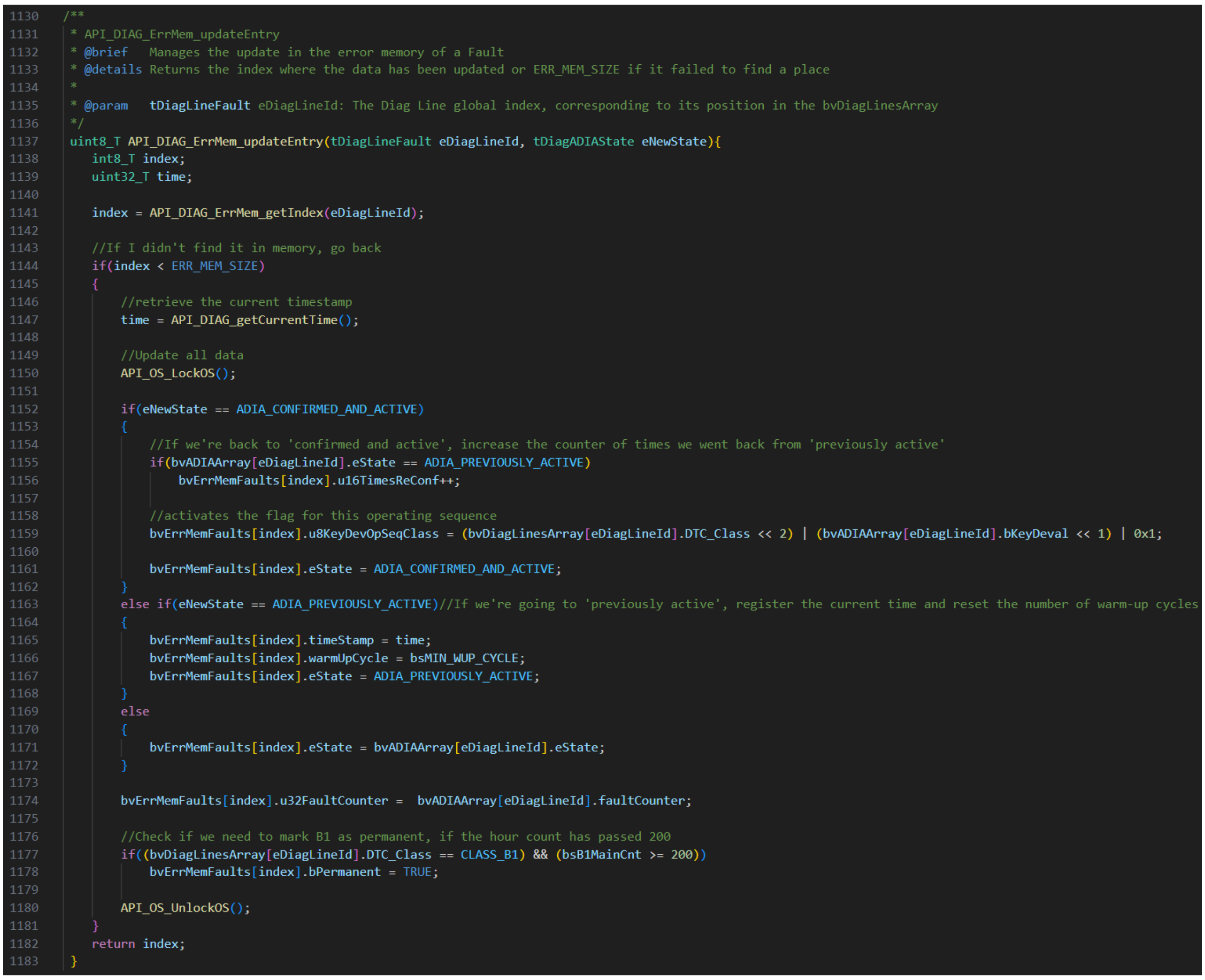
```c
    //If a free/reusable position has been found, insert
    if(i < ERR_MEM_SIZE) //If we find a valid position, insert the data
    {
        //If I'm removing a B1 or B2 active fault, I need to decrease the corresponding counter
        if(bvErrMemFaults[i].bInit == TRUE)
        {
            tmpId = bvErrMemFaults[i].eFault;
            if((bvDiagLinesArray[tmpId].DTC_Class == CLASS_B1) && (bvADIAArray[tmpId].eState ==ADIA_CONFIRMED_AND_ACTIVE))
                bsB1ActiveFaults--;
            else if ((bvDiagLinesArray[tmpId].DTC_Class == CLASS_B2) && (bvADIAArray[tmpId].eState ==ADIA_CONFIRMED_AND_ACTIVE))
                bsB2ActiveFaults--;
        }

        API_OS_LockOS();
        bvErrMemFaults[i].bInit = TRUE;
        bvErrMemFaults[i].eFault = eDiagLineId;
        bvErrMemFaults[i].DTC_FM_WWHOBD = bvDiagLinesArray[eDiagLineId].DTC_FM_WWHOBD;
        bvErrMemFaults[i].DTC_FM_J1939 = bvDiagLinesArray[eDiagLineId].DTC_FM_J1939;
        bvErrMemFaults[i].eState = bvADIAArray[eDiagLineId].eState;
        bvErrMemFaults[i].u32FaultCounter =  bvADIAArray[eDiagLineId].faultCounter;
        bvErrMemFaults[i].u8KeyDevOpSeqClass = (bvDiagLinesArray[eDiagLineId].DTC_Class << 2) | (bvADIAArray[eDiagLineId].bKeyDeval << 1) | 0x1;
        bvErrMemFaults[i].timeStamp = 0;
        bvErrMemFaults[i].warmUpCycle = bsMIN_WUP_CYCLE;
        bvErrMemFaults[i].u16TimesReConf = 0;

        if((bvDiagLinesArray[eDiagLineId].DTC_Permanent == TRUE) || (bvDiagLinesArray[eDiagLineId].DTC_Class == CLASS_A)
            || ((bvDiagLinesArray[eDiagLineId].DTC_Class == CLASS_B1) && (bsB1MainCnt >= 200)))
            bvErrMemFaults[i].bPermanent = TRUE;
        else
            bvErrMemFaults[i].bPermanent = FALSE;
        API_OS_UnlockOS();
    }

    return i;
}
```

Figure 5.5.11c - Final part of API_DIAG_ErrMem_insert, in api_diagobd.c

The next function called from the ADIA is the one responsible for the updating of an entry in the error memory. The **API_DIAG_ErrMem_updateEntry** takes in input the ID of the fault and its new state and returns the position of the updated entry (or *ERR_MEM_SIZE* if something went wrong).

Before doing anything, the function calls another method, **API_DIAG_ErrMem_getIndex**, which returns the position of a fault in the error memory given the ID.
This simple function cycles through the error memory faults array and returns the first position containing a used entry with the given ID or *ERR_MEM_SIZE* if it fails to find one.
It is also used in the function to initialise the ADIAs at startup, by the ADIA while in the *Previously active* state before trying to move to the *No error* state, and in the API used to retrieve the freeze frame of a given fault.

Moving back to *API_DIAG_ErrMem_updateEntry*, if the *getIndex* returned a valid value, the function goes on reading the current system time calling *API_DIAG_getCurrentTime* and then proceeds to obtain a lock to safely update the entry.
The update depends on the new state of the fault; if it is *Confirmed and active*, the function first checks that the old state was *Previously active*, in which case increases the entry's *u16TimesReConf* field, then goes on to update the *u8KeyDevOpSeqClass* field and the entry's state.
If the new state is *Previously active*, the function simply assigns the read time value as the new timestamp, resets the counter for the entry's warm-up cycles, and updates the entry's state.
In any other case, it simply updates the state of the entry. The function then updates the entry's fault counter and, if the fault class is B1 and the *bsB1MainCnt* is greater than 200 hours, updates the 'permanent' flag.
The function then releases the lock and returns the updated index.



```c
/**
 * API_DIAG_ErrMem_updateEntry
 * @brief   Manages the update in the error memory of a Fault
 * @details Returns the index where the data has been updated or ERR_MEM_SIZE if it failed to find a place
 *
 * @param   tDiagLineFault eDiagLineId: The Diag Line global index, corresponding to its position in the bvDiagLinesArray
 */
uint8_T API_DIAG_ErrMem_updateEntry(tDiagLineFault eDiagLineId, tDiagADIAState eNewState){
   int8_T index;
   uint32_T time;

   index = API_DIAG_ErrMem_getIndex(eDiagLineId);

   //If I didn't find it in memory, go back
   if(index < ERR_MEM_SIZE)
   {
       //retrieve the current timestamp
       time = API_DIAG_getCurrentTime();

       //Update all data
       API_OS_LockOS();

       if(eNewState == ADIA_CONFIRMED_AND_ACTIVE)
       {
           //If we're back to 'confirmed and active', increase the counter of times we went back from 'previously active'
           if(bvADIAArray[eDiagLineId].eState == ADIA_PREVIOUSLY_ACTIVE)
               bvErrMemFaults[index].u16TimesReConf++;

           //activates the flag for this operating sequence
           bvErrMemFaults[index].u8KeyDevOpSeqClass = (bvDiagLinesArray[eDiagLineId].DTC_Class << 2) | (bvADIAArray[eDiagLineId].bKeyDeval << 1) | 0x1;

           bvErrMemFaults[index].eState = ADIA_CONFIRMED_AND_ACTIVE;
       }
       else if(eNewState == ADIA_PREVIOUSLY_ACTIVE)//If we're going to 'previously active', register the current time and reset the number of warm-up cycles
       {
           bvErrMemFaults[index].timeStamp = time;
           bvErrMemFaults[index].warmUpCycle = bsMIN_WUP_CYCLE;
           bvErrMemFaults[index].eState = ADIA_PREVIOUSLY_ACTIVE;
       }
       else
       {
           bvErrMemFaults[index].eState = bvADIAArray[eDiagLineId].eState;
       }

       bvErrMemFaults[index].u32FaultCounter =  bvADIAArray[eDiagLineId].faultCounter;

       //Check if we need to mark B1 as permanent, if the hour count has passed 200
       if((bvDiagLinesArray[eDiagLineId].DTC_Class == CLASS_B1) && (bsB1MainCnt >= 200))
           bvErrMemFaults[index].bPermanent = TRUE;

       API_OS_UnlockOS();
   }
   return index;
}
```

Figure 5.5.12 - API_DIAG_ErrMem_updateEntry, in api_diagobd.c

The last function used by the ADIA to manage the faults in the error memory is the one to erase them, **API_DIAG_ErrMem_eraseEntry**. This function directly takes in input the index of the error memory entry to remove. To optimise the process, rather than cleaning up the whole structure, the function simply sets the *bInit* field of the entry to *false*, as it is the flag that is checked from all the other functions to see whether the entry is valid or not.
As always, before modifying the structure in the error memory, the function obtains a lock on the critical session and releases it once done.

Along with all these functions that implement the correct behaviour of the error memory in the OBD system, other APIs have also been created to satisfy the requirements defined by the Euro-VI and China-VI on the possible requests from external tools.

The list includes:

- **API_DIAG_OBDClean**, which cleans up the error memory and certain counters. It erases (i.e., sets *bInit* to *false*) all the non-permanent entries of the error memory, recomputes the recovery lines table, resets the readiness and continuous MI counter (not the cumulative one) used by the MIL management, resets the counters for the hours and the warm-up cycles since the last cleanup, and updates the counters for the active faults calling the **API_DIAG_MIL_readDiagInfo10ms** function.
- **API_DIAG_getB1HourCntr**, which returns the hour counter of B1 faults.
- **API_DIAG_getHoursSinceLastOBDClean**, which retrieves the hours passed since the last OBD cleanup by an external tool.
- **API_DIAG_getWUCSinceLastOBDClean**, which returns the number of warm-up cycles that have passed since the OBD was last reset with an external tool.
- **API_DIAG_getActiveFaultsCntOfClass**, which computes the count of active faults of a given class.
- **API_DIAG_getActiveFaultsOfClass**, which returns the active faults of a given class and stores them in the array passed by parameter, returning the number of read elements.
- **API_DIAG_getPreviouslyActiveFaults**, which retrieves the previously active faults and stores them in the array passed by parameter, returning the number of read elements.
- **API_DIAG_getFreezeFrame**, which returns the freeze frame (all fields) of the fault with the given ID, storing it in an array passed as a parameter, and returns whether the fault and the corresponding freeze frame were in memory.

The last set of functions created for the management of the error memory were those regarding the 'shutdown' strategy. Similarly to the 'startup' functions, the first link of the function chain is called by a task of the operating system, **API_OS_KeyOffRoutine**. This OS procedure calls the **API_DIAG_OnKeyOff** function before invoking the already seen *API_EEPROM_save*.

The *API_DIAG_OnKeyOff* is a wrapper for both the before-mentioned *API_DIAG_KeyDevalManager* and the **API_DIAG_prepareStoreEEPROM** function. The latter is the one responsible for copying the values from the volatile variables and structure into their NVRAM counterparts. It also computes the checksum and stores it into the *bsDiagEePROM* structure's **bsCRC** field, using the *EDATA_CalcDataChecks* base-software function.

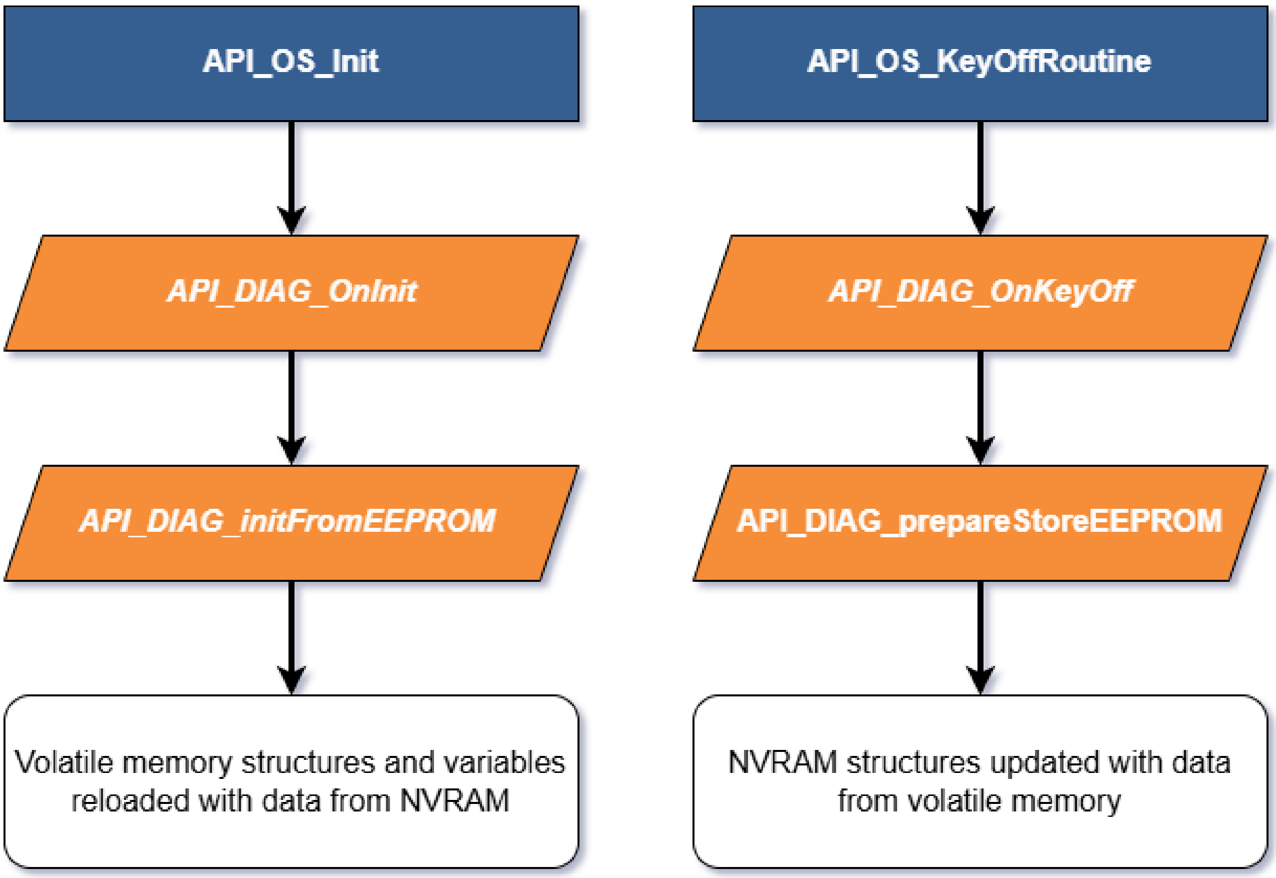

API_OS_Init
API_DIAG_OnInit
API_DIAG_initFromEEPROM
Volatile memory structures and variables reloaded with data from NVRAM
API_OS_KeyOffRoutine
API_DIAG_OnKeyOff
API_DIAG_prepareStoreEEPROM
NVRAM structures updated with data from volatile memory

Figure 5.5.13 - Parallels between the startup and shutdown procedures for what concerns the error memory

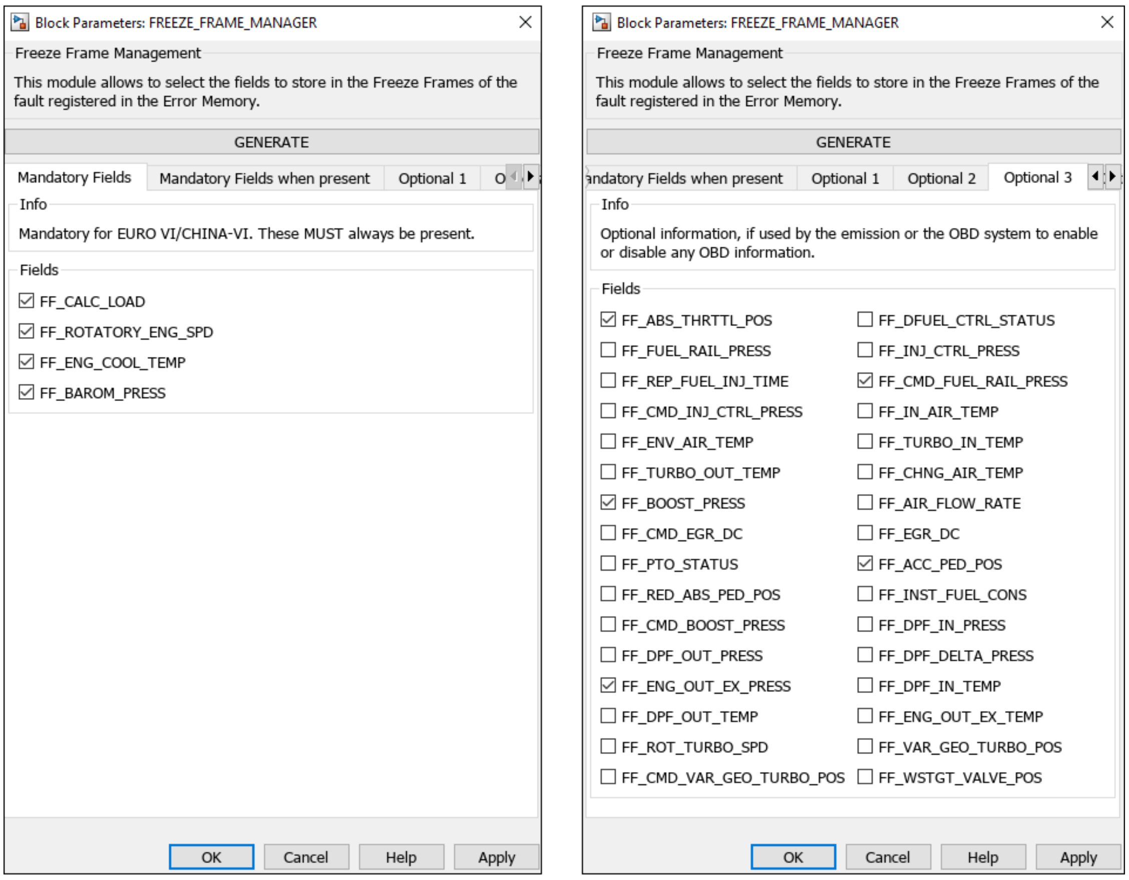
5.5.4. Freeze Frame Mask

To manage the global freeze frame at model-based software level, a new Simulink mask has been implemented. The mask can be used to select the fields of the freeze frame that shall be filled by the system and will produce a block that will take as input the values for the selected fields, and only those.

The GUI of the mask is divided into six tabs:

- Mandatory fields
- Mandatory fields when present[1]
- Optional 1, 2, and 3
- Optional custom

Each tab contains fields belonging to a different category seen in the requirements part, to facilitate the selection of distinct entries. A field can be selected or deselected simply by checking the box next to the name.


Block Parameters: FREEZE_FRAME_MANAGER

Freeze Frame Management

This module allows to select the fields to store in the Freeze Frames of the fault registered in the Error Memory.

GENERATE

Mandatory Fields | Mandatory Fields when present | Optional 1 | O

Info

Mandatory for EURO VI/CHINA-VI. These MUST always be present.

Fields

☑ FF_CALC_LOAD
☑ FF_ROTATORY_ENG_SPD
☑ FF_ENG_COOL_TEMP
☑ FF_BAROM_PRESS

OK | Cancel | Help | Apply

Block Parameters: FREEZE_FRAME_MANAGER

Freeze Frame Management

This module allows to select the fields to store in the Freeze Frames of the fault registered in the Error Memory.

GENERATE

andatory Fields when present | Optional 1 | Optional 2 | Optional 3

Info

Optional information, if used by the emission or the OBD system to enable or disable any OBD information.

Fields

☑ FF_ABS_THRTTL_POS	☐ FF_DFUEL_CTRL_STATUS
☐ FF_FUEL_RAIL_PRESS	☐ FF_INJ_CTRL_PRESS
☐ FF_REP_FUEL_INJ_TIME	☑ FF_CMD_FUEL_RAIL_PRESS
☐ FF_CMD_INJ_CTRL_PRESS	☐ FF_IN_AIR_TEMP
☐ FF_ENV_AIR_TEMP	☐ FF_TURBO_IN_TEMP
☐ FF_TURBO_OUT_TEMP	☐ FF_CHNG_AIR_TEMP
☑ FF_BOOST_PRESS	☐ FF_AIR_FLOW_RATE
☐ FF_CMD_EGR_DC	☐ FF_EGR_DC
☐ FF_PTO_STATUS	☑ FF_ACC_PED_POS
☐ FF_RED_ABS_PED_POS	☐ FF_INST_FUEL_CONS
☐ FF_CMD_BOOST_PRESS	☐ FF_DPF_IN_PRESS
☐ FF_DPF_OUT_PRESS	☐ FF_DPF_DELTA_PRESS
☑ FF_ENG_OUT_EX_PRESS	☐ FF_DPF_IN_TEMP
☐ FF_DPF_OUT_TEMP	☐ FF_ENG_OUT_EX_TEMP
☐ FF_ROT_TURBO_SPD	☐ FF_VAR_GEO_TURBO_POS
☐ FF_CMD_VAR_GEO_TURBO_POS	☐ FF_WSTGT_VALVE_POS

OK | Cancel | Help | Apply

Figure 5.5.14 - Two tabs of the mask's GUI

[1] It could be a measure from a real sensor (if present) or from a virtual one (by model).

The fields' names are taken directly from the *api.h* file, from the labels of the *tDiagFFField*. To add a new field, one can simply go and change the reserved positions of the enum (those named **FF_FUTURE_USE_N**) or directly add others. In case of addition of entries, rather than replacing them, the user must also remember to update the *#define FF_N_FIELDS* with the new number of fields.

Once the desired fields have been selected, pressing the *GENERATE* button on the GUI will update the mask's block to include the new input ports, one for each chosen field.

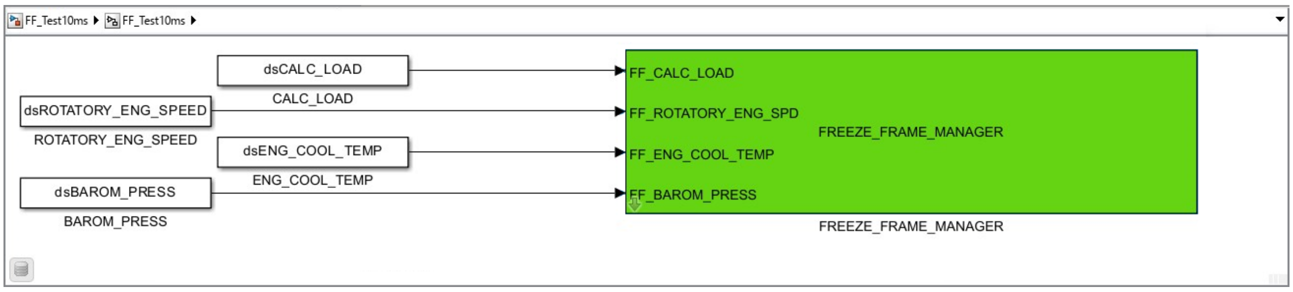


Figure 5.5.15 - Example of block with four fields selected, with its input ports connected with the appropriate sources

The mask uses three scripts: one for the initialisation, one to generate the block's internal model, and one to update the block's appearance based on the selected fields.

The initialisation script, **mskFreezeFrameInit**, reads the fields from the library file and automatically checks the mandatory fields.

The script used to generate the block's subsystem, **mskFreezeFrameGenerate**, is made up of five different functions. The homonymous *mskFreezeFrameGenerate* is the one called from the block and pilots the execution of the other four.
It starts by invoking the *Port_Keeper* function to temporarily save the already existing connections.
Then computes the array of flags for the selected fields using the **computeAvailableArrays** function, returning a boolean array with '*true*' for the selected fields' positions.
The main function will then clear the block's internal model and call the **ManageEntries** function. For each '*true*' flag in the array, the function will create four components: a constant block containing the enumerative label of the field, an input port to read the field's value from, a constant block for the availability flag of the freeze frame entry, and the API block of the *API_DIAG_updateFFEntry*. It will then connect each of the first three blocks to the correct input port of the API block.

Once the ManageEntries has finished, the main function will call the Beautifier method, which will improve the placement of the generated blocks in order to make the model more pleasing to the human eye for understandability reasons.

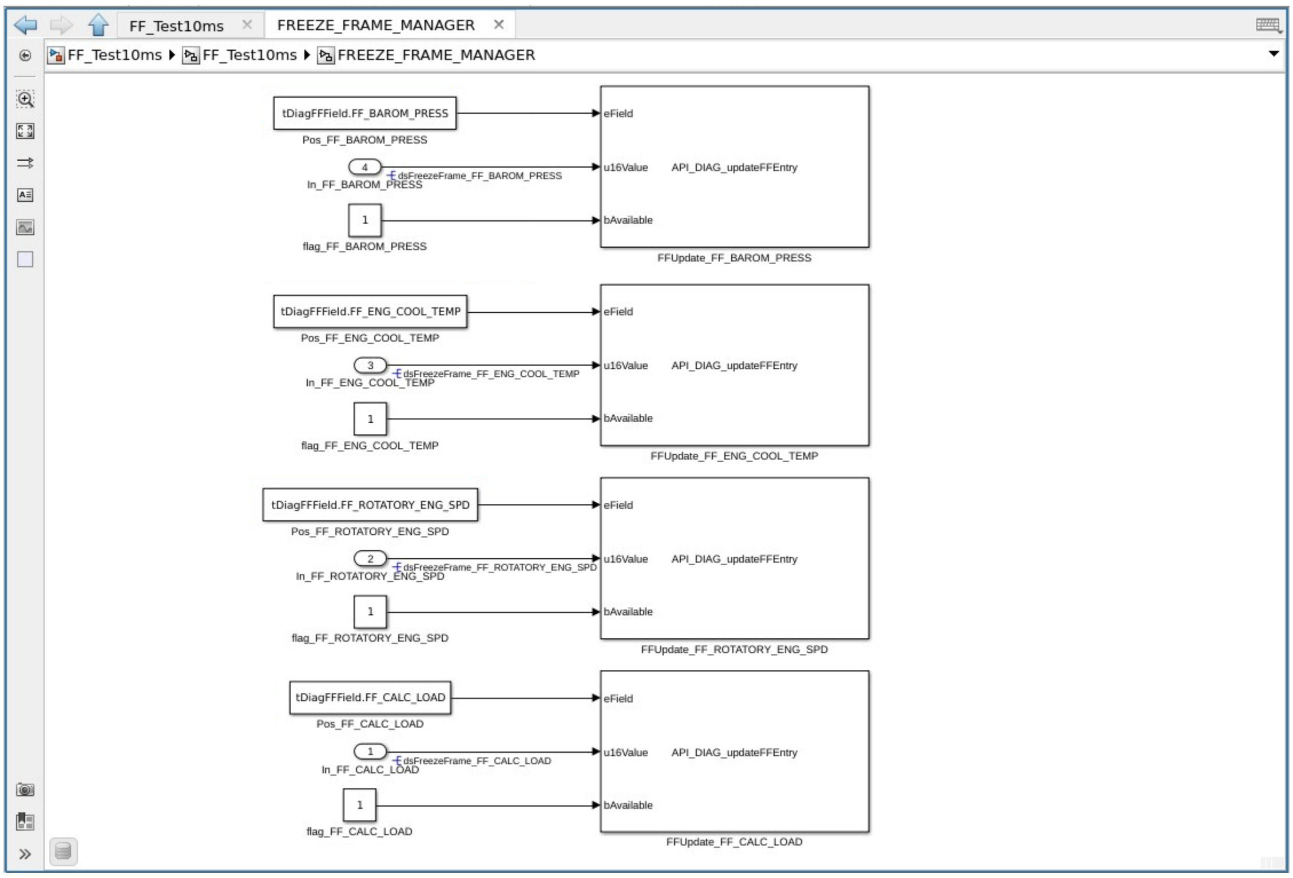


Figure 5.5.16 - Example of the generated internal model

For what concerns the usage of the freeze frame management system, the suggested strategy consists in designing a single model containing the mask with the selected fields and converging the various input signals for the freeze frame entries to that single model. The model's code should then be placed into a periodic task of the operating system, making sure that it gets executed before any ADIA does, as to ensure that the freeze frame is updated in case of insertion in the error memory.

This isn't the only way, as many strategies are possible using the provided APIs, but it should be considered the most efficient one, as it introduces a single point of management. It is also the strategy adopted in the sample application developed for testing purposes.

5.6. MIL Management

The Malfunction Indicator Light is a key component of OBD, as it not only allows the drivers to immediately notice if a fault presents itself, but it is also useful to the technical personnel to discern the severity of the fault even before the usage of external tools.

5.6.1. Requirements

For the definition of the requirements for this topic, the focus went to the *WWH-OBD*, *Euro-VI* and *China-VI* documents.

The extracted requirements included constraints on the timing and behaviour of the lamp, the conditions for its activation and deactivation, and the information on the MIL status that the system must be able to provide.
Other requirements, such as the colour of the indicator, went ignored as not inherently regarding the software part.

The *WWH-OBD* MIL management defines the behavioural constraints of the indicator differently depending on whether the **engine** is **on** or **off**.

Another distinction in the indicator functioning is given by the **readiness status** of the system. The readiness of the OBD (On-Board Diagnostics) system refers to the status of various self-diagnostic tests, also known as *readiness monitors*, that the computer performs to ensure the emissions control systems are functioning correctly. Although the documents do not specify any constraint on these monitors, they do define the different behaviour of the MIL depending on the completion of such tests.

Each combination of engine status, readiness status, and active fault classes determines the MIL status and produces a different on/off pattern of the light, whose timings and durations are precisely defined by both *Euro-VI* and *China-VI* guidelines.

The regulations define four operating modes for the MIL: **no active DTC**, **On-demand**, **Short**, and **Continuous**.

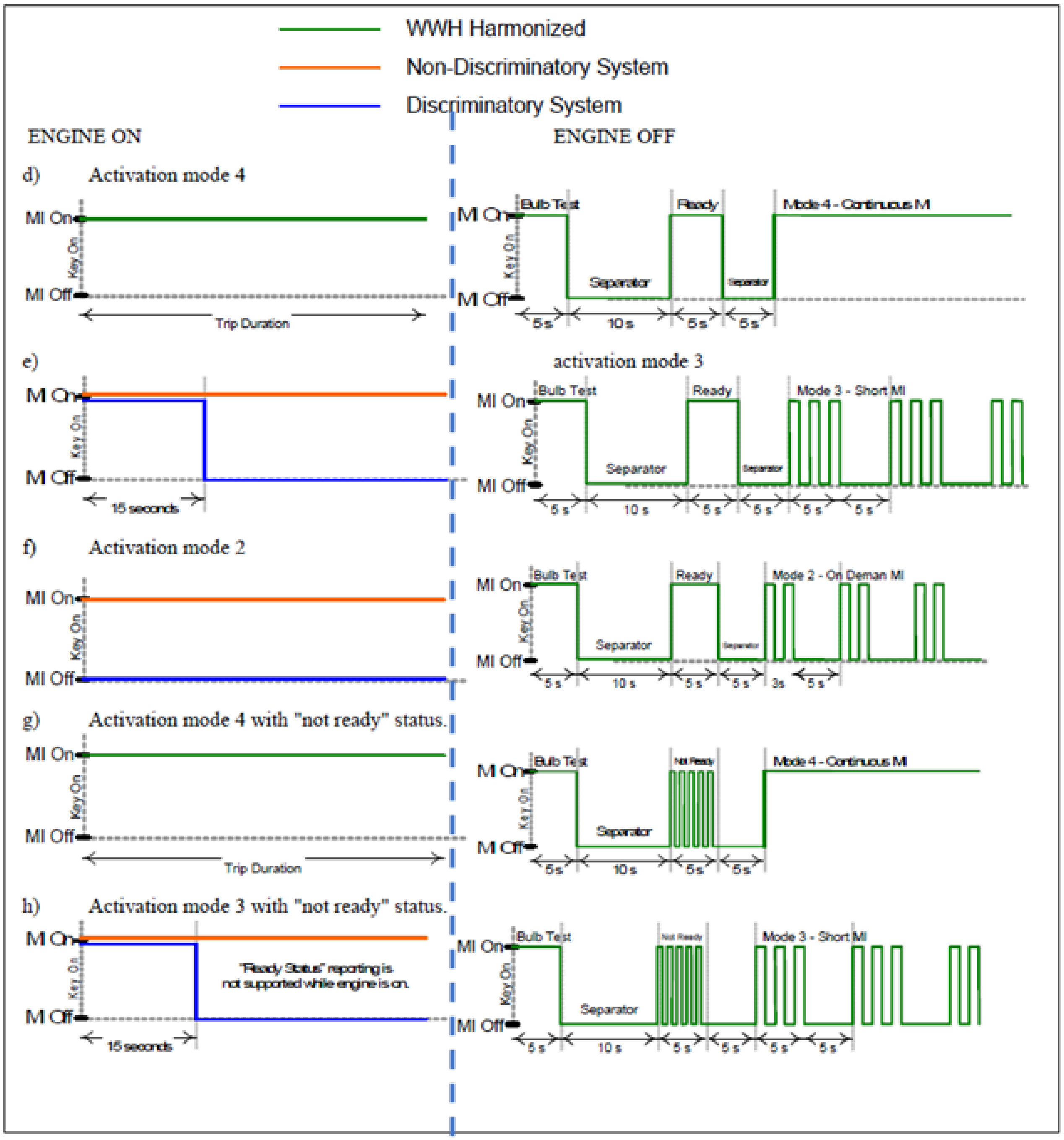


Figure 5.6.1 - Examples of the required behaviours and timings of the MI under different conditions

As the base for the requirements was the harmonised OBD, the final implementation doesn’t make any distinction between discriminatory and non-discriminatory systems and uses the discriminatory approach for the *engine on* status. For non-discriminatory screens, the *Continuous* mode is applied regardless of the fault's class.

A summary of the lamp's timing for each activation mode is the following (higher number equals higher precedence over the other modes):

- Mode 1: *Absence of malfunction (No Dtc MI)*
    - The MIL will flash 1 time (1s): ON(1s) + OFF(5s) + …
- Mode 2: *On-demand MI*
    - Engine Off: the MIL will flash 2 times: ON(1s) + OFF(1s) + ON(1s) + OFF(5s).
    - Engine On: the MIL will stay on (non-discriminatory systems) or off (discriminatory systems).
- Mode 3: *Short MI*
    - Engine Off: the MIL will flash 3 times: ON(1s) + OFF(1s) + ON(1s) + OFF(1s) + ON(1s) + OFF(5s).
    - Engine On: the MIL will stay on (non-discriminatory systems) or will be ON (15s) and then stay off (discriminatory systems).
- Mode 4: *Continuous MI*
    - Engine Off: the MIL will stay on.
    - Engine On: the MIL will stay on.

For what concerns the required information related to the MI that the system must provide, the list includes:

- MIL status.
- Readiness of the diagnostic system.
- Number of hours passed with a continuous MI since the last time that the MIL was activated.
- Cumulative number of engine hours with an active continuous MI.
- Confirmed and active faults of every class.
- Permanent failure codes.

5.6.2. Strategy

The original logic for the management of the MIL went basically untouched, as it already followed almost step-by-step what was required from the regulations. The main focus was once again to simplify it while still satisfying all the requirements.

Due to the complexity of the system caused by the high number of variables and different conditions to keep track, the adopted strategy made use of finite state machines. The first step was to redraw these automa, discarding the unneeded intermediate states.

The final result is a hierarchical structure of FSMs that is able to manage the behaviour of the malfunction indicator just as required by the guidelines.

The hierarchy starts with the automaton used to discern between the engine's status. The FSM contains three states: an initialisation state, used only to set up the variables before entering the actual states, an '***engine not ignited***' state, and an '***engine ignited***' state. Both the engine states correspond to an automaton.
The FSM enters one of the two engine states depending on whether a global flag is set. The flag must be set at MBS level through the use of an API.
The automaton moves from one state to the other with the changing of the global flag. In addition to the first flag, a second one is needed for the transition from the '*ignited*' to the '*not ignited*' state to avoid resetting the MIL for '*Start and Stop*' systems.

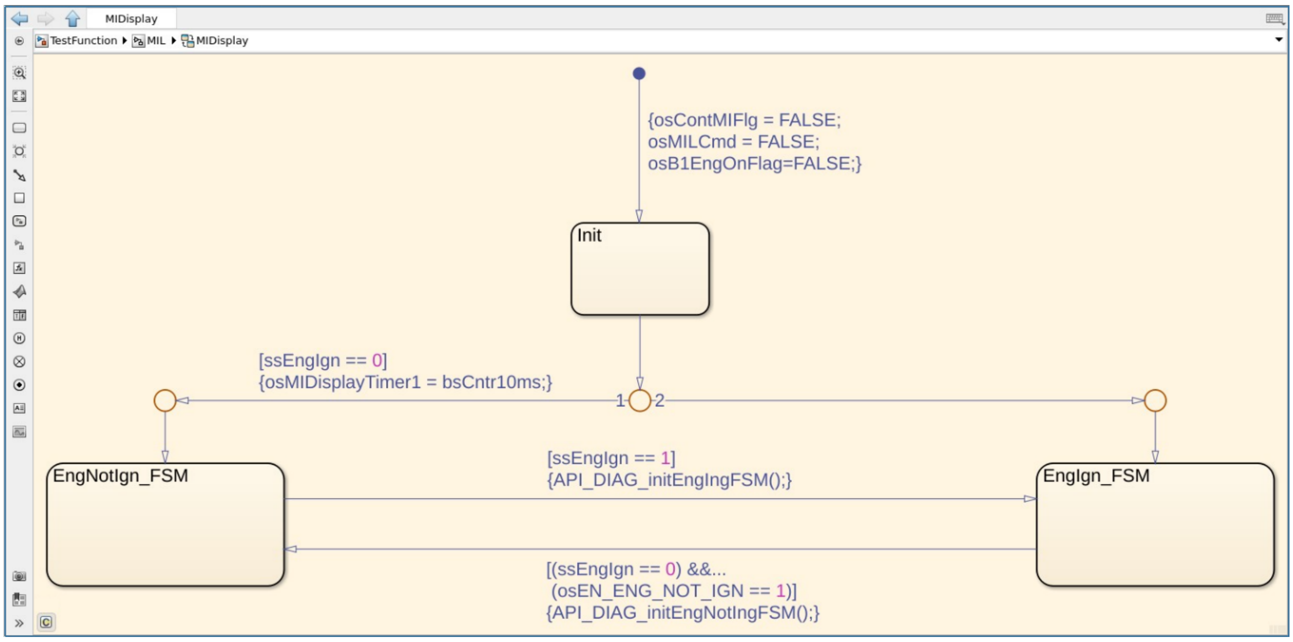
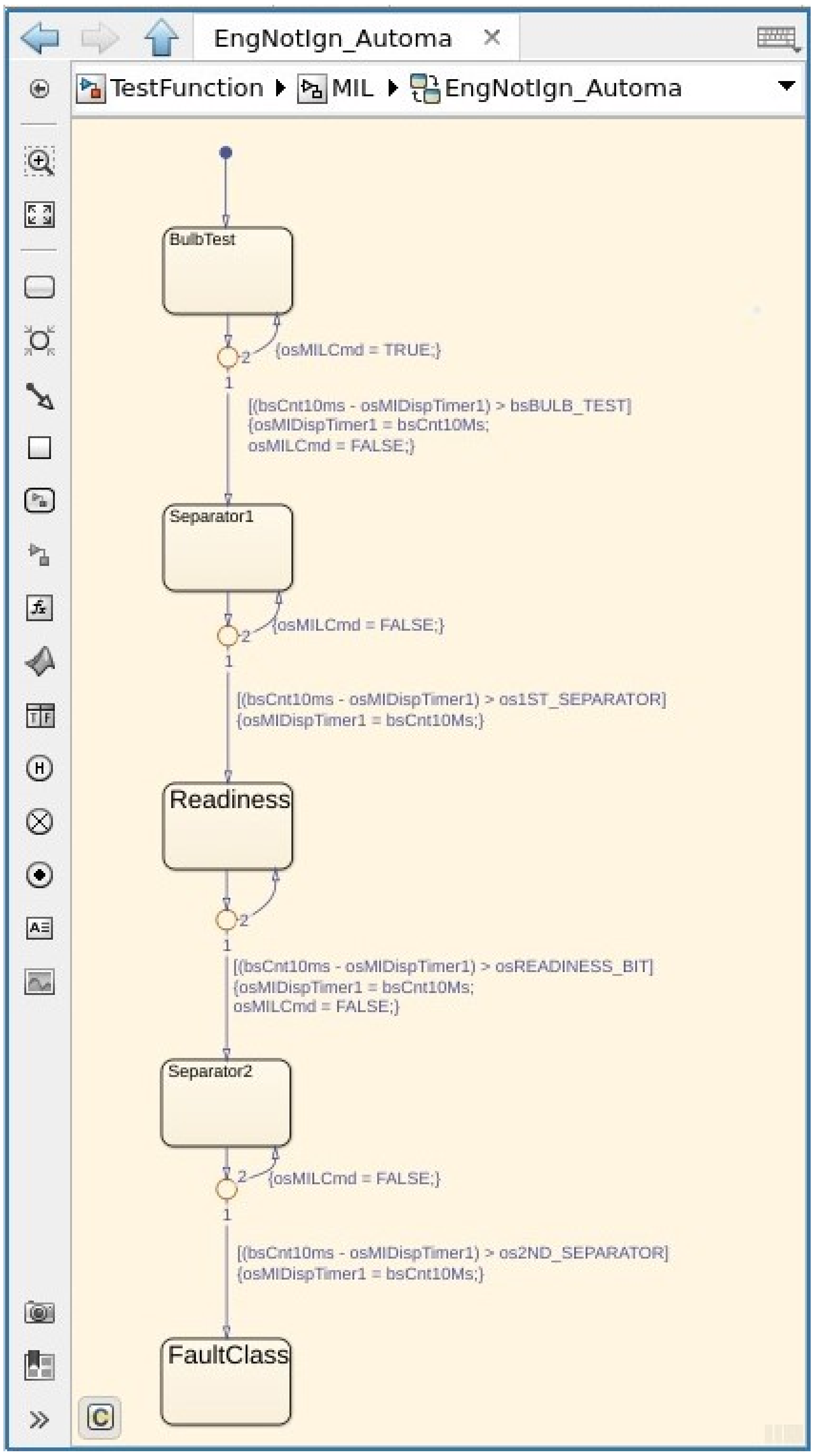


Figure 5.6.2 - FSM for the engine states

The '*engine off*' FSM is the largest of the bunch, as it counts a total of five states, with two of them being automata themselves. The flow of the automaton is linear, with each state being reached sequentially without backward branches from a state to the previous ones.



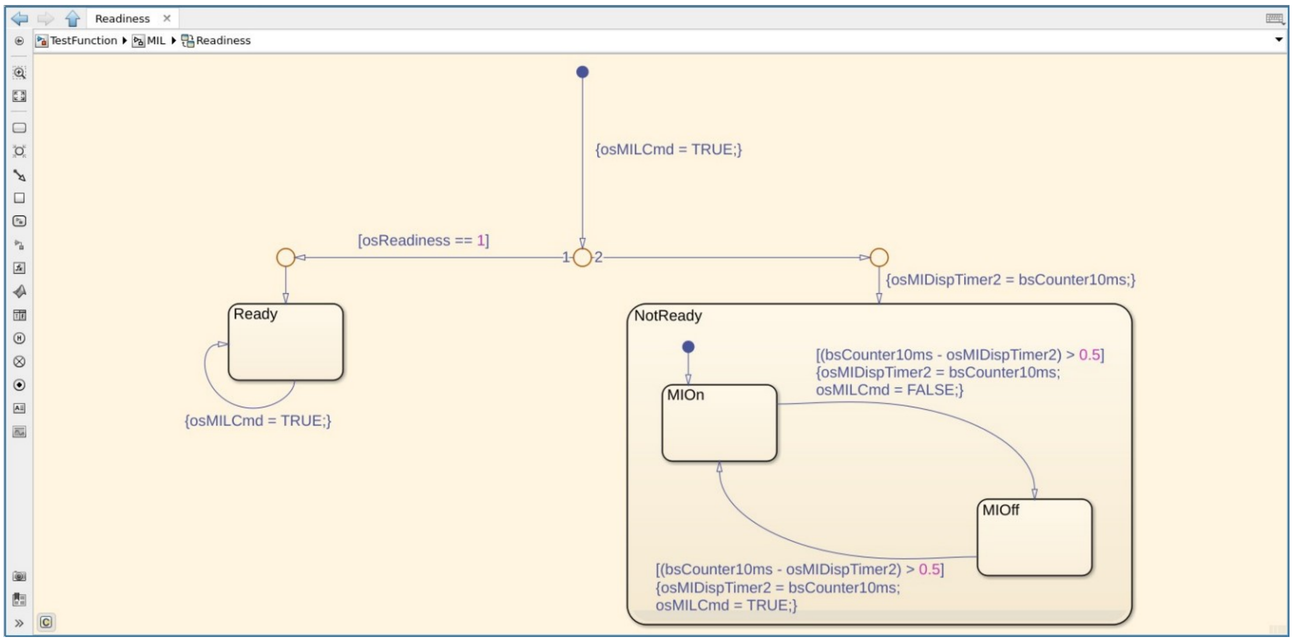
```
{osMILCmd = TRUE;}
[(bsCnt10ms - osMIDispTimer1) > bsBULB_TEST]
{osMIDispTimer1 = bsCnt10Ms;
osMILCmd = FALSE;}

{osMILCmd = FALSE;}
[(bsCnt10ms - osMIDispTimer1) > os1ST_SEPARATOR]
{osMIDispTimer1 = bsCnt10Ms;}

[(bsCnt10ms - osMIDispTimer1) > osREADINESS_BIT]
{osMIDispTimer1 = bsCnt10Ms;
osMILCmd = FALSE;}

{osMILCmd = FALSE;}
[(bsCnt10ms - osMIDispTimer1) > os2ND_SEPARATOR]
{osMIDispTimer1 = bsCnt10Ms;}
```

Figure 5.6.3 - FSM for the engine not ignited

The first state is the **BulbTest**, where the MIL is kept on for 5 seconds. Once the timer reaches the correct value, the FSM moves on to the next state.

The second state is **Separator1**, where the light is kept off. Once the timer registers that 10 seconds have passed, the next state is reached.

The third state, **Readiness**, is actually a finite state machine with two possible states, **Ready** and **NotReady**. The state of this FSM is determined based on the value of the global *readiness* flag. If the flag is set to 1, the *Ready* state is selected, and the light is kept on for 5 seconds. Otherwise, the *NotReady* state, which implements the flickering of the light thanks to another sub-FSM, is chosen.



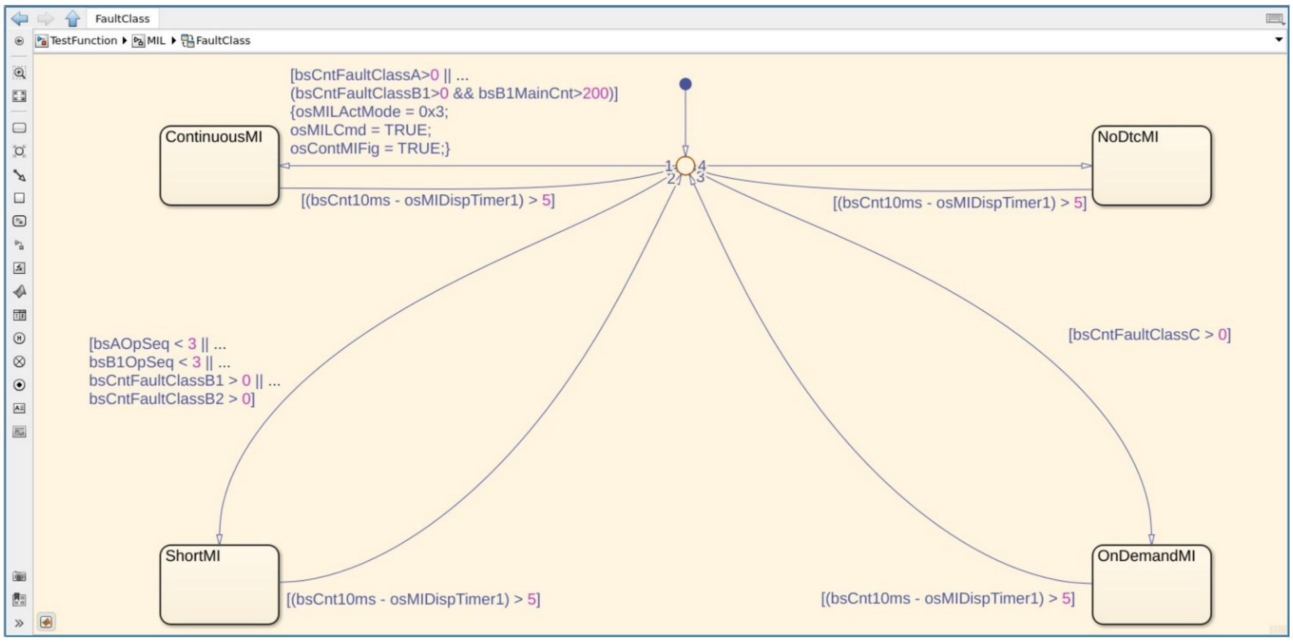
Readiness
TestFunction ▸ MIL ▸ Readiness
{osMILCmd = TRUE;}
[osReadiness == 1]
1
2
{osMIDispTimer2 = bsCounter10ms;}
Ready
{osMILCmd = TRUE;}
NotReady
[(bsCounter10ms - osMIDispTimer2) > 0.5]
{osMIDispTimer2 = bsCounter10ms;
osMILCmd = FALSE;}
MIOn
MIOff
[(bsCounter10ms - osMIDispTimer2) > 0.5]
{osMIDispTimer2 = bsCounter10ms;
osMILCmd = TRUE;}

Figure 5.6.4 - FSM for the Readiness state

After 5 seconds have passed, the '*not ignited*' FSM enters the **Separator2** state, which will turn off the light for 5 seconds before moving to the last state.

The final state, **FaultClass**, is the FSM that governs all the other FSMs implementing the MIL behaviour based on the active faults: in case of class A faults or class B1 faults with more than 200 engine hours, the selected state will be **ContinuousMI**.
When the previous conditions are not verified but the last active class A or B1 fault happened in the last 3 operating sequences (as per the guidelines), or if there are currently active B1 or B2 faults, the **ShortMI** state will be selected.

If all the before-mentioned cases failed and there is any active class C fault, the **OnDemandMI** state will be chosen.
Lastly, if none of the previous conditions are true and there is no currently active fault, the **NoDtcMI** state will be the active one.
Every 5 seconds, the state will be recomputed to be up to date with the current system's status.


FaultClass
TestFunction ▸ MIL ▸ FaultClass
[bsCntFaultClassA>0 || ...
(bsCntFaultClassB1>0 && bsB1MainCnt>200)]
{osMILActMode = 0x3;
osMILCmd = TRUE;
osContMIFig = TRUE;}
ContinuousMI
NoDtcMI
1
2
3
4
[(bsCnt10ms - osMIDispTimer1) > 5]
[(bsCnt10ms - osMIDispTimer1) > 5]
[bsAOpSeq < 3 || ...
bsB1OpSeq < 3 || ...
bsCntFaultClassB1 > 0 || ...
bsCntFaultClassB2 > 0]
[bsCntFaultClassC > 0]
ShortMI
OnDemandMI
[(bsCnt10ms - osMIDispTimer1) > 5]
[(bsCnt10ms - osMIDispTimer1) > 5]
Figure 5.6.5 - FSM for the FaultClass state

The *ContinuousMI* state does not correspond to an FSM, as it simply means that the light will be kept on.

The *ShortMI* state is instead implemented with an FSM with two states.
The first state, **Blinking**, is also a finite state machine that alternates between the two internal states **MIOn** and **MIOff** to blink the lamp, passing from one state to the other every second.
After 5 seconds have passed, the *ShortMI* automaton moves to the **Off** state, which lasts for 5 seconds, in which the indicator remains off.

The *OnDemand* FSM works in the same way as the *ShortMI* one, but the *Blinking* state only lasts for 3 seconds. This means that in a cycle there will be only two 'on' peaks, against the three of the *short* activation mode.

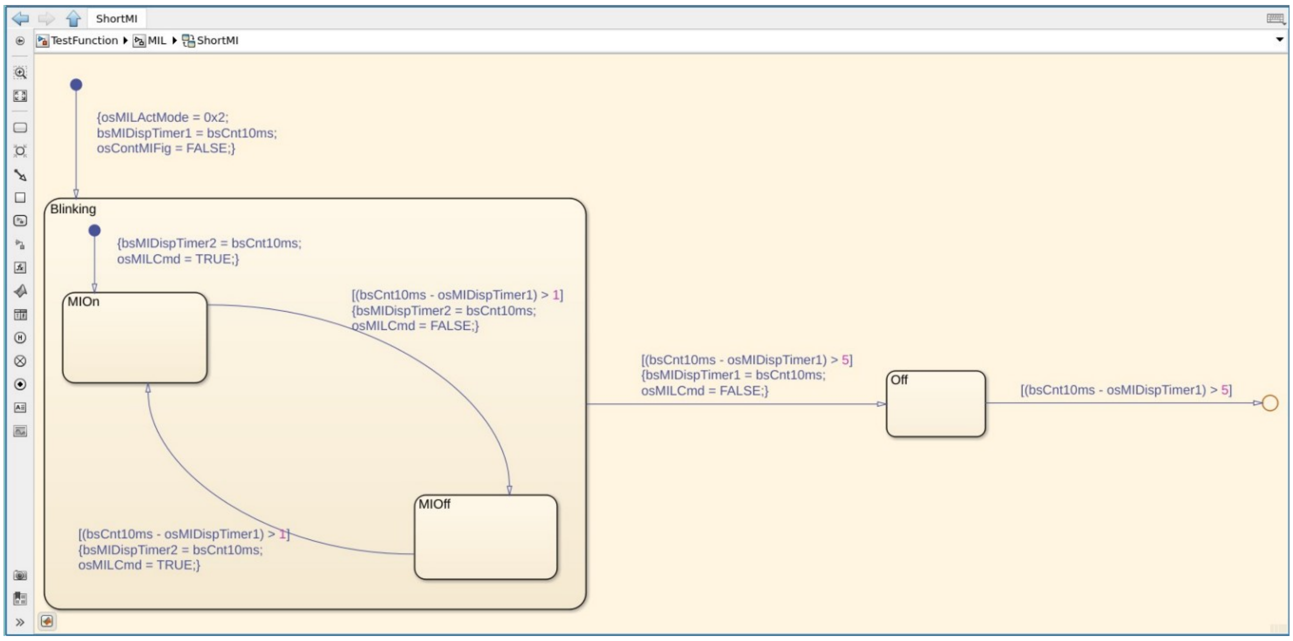

ShortMI
TestFunction ▸ MIL ▸ ShortMI
{osMILActMode = 0x2;
bsMIDispTimer1 = bsCnt10ms;
osContMIFig = FALSE;}
Blinking
{bsMIDispTimer2 = bsCnt10ms;
osMILCmd = TRUE;}
MIOn
[(bsCnt10ms - osMIDispTimer1) > 1]
{bsMIDispTimer2 = bsCnt10ms;
osMILCmd = FALSE;}
MIOff
[(bsCnt10ms - osMIDispTimer1) > 1]
{bsMIDispTimer2 = bsCnt10ms;
osMILCmd = TRUE;}
[(bsCnt10ms - osMIDispTimer1) > 5]
{bsMIDispTimer1 = bsCnt10ms;
osMILCmd = FALSE;}
Off
[(bsCnt10ms - osMIDispTimer1) > 5]

Figure 5.6.6 - FSM for the ShortMI state, almost identical to the OnDemandMI automaton

The remaining state, *NoDtcMI*, also corresponds to a simple FSM with two states: **Blink**, lasting for 1 second, with the light on, and *Off*.

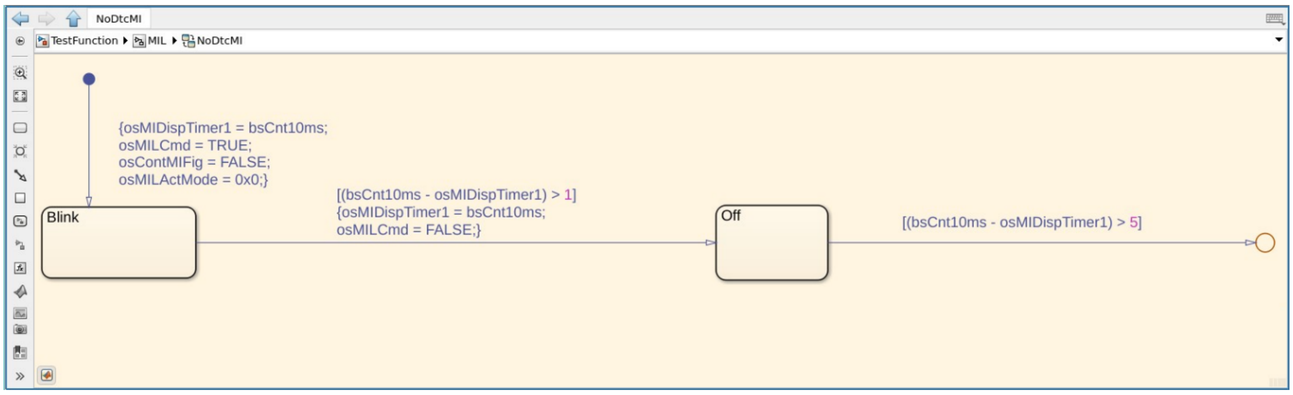

NoDtcMI
TestFunction ▸ MIL ▸ NoDtcMI
{osMIDispTimer1 = bsCnt10ms;
osMILCmd = TRUE;
osContMIFig = FALSE;
osMILActMode = 0x0;}
Blink
[(bsCnt10ms - osMIDispTimer1) > 1]
{osMIDispTimer1 = bsCnt10ms;
osMILCmd = FALSE;}
Off
[(bsCnt10ms - osMIDispTimer1) > 5]

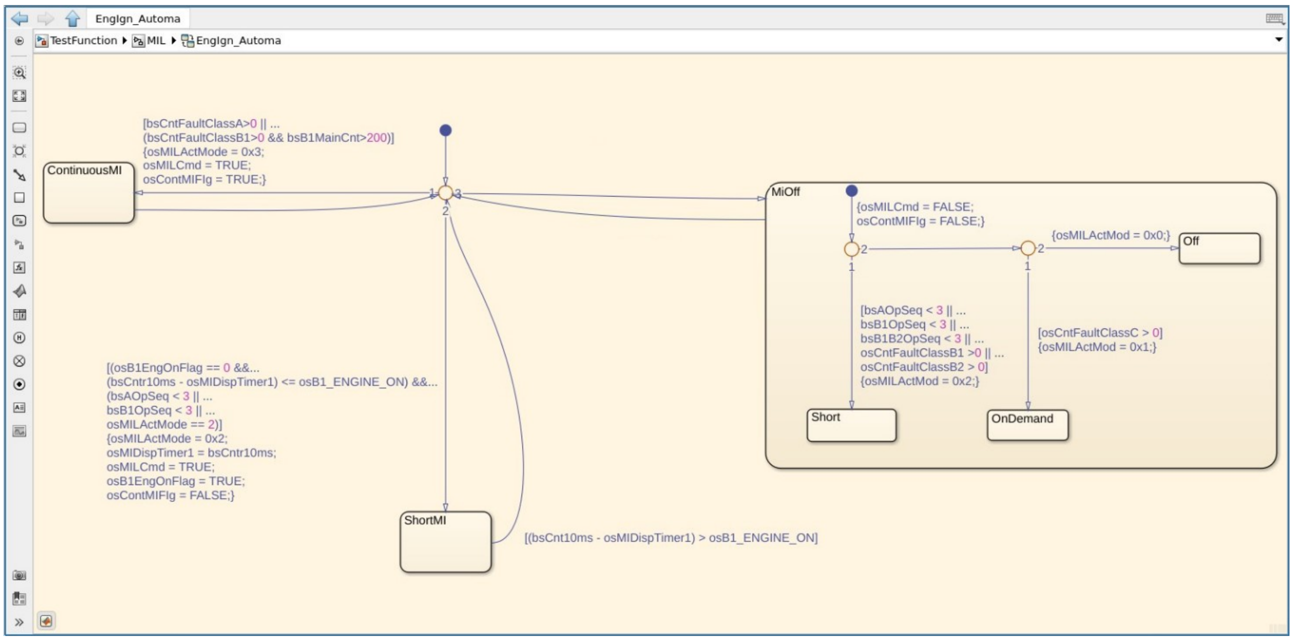
Figure 5.6.7 - FSM for the NoDtcMI state

The last automa is the one describing the behaviour of the system when the engine is already ignited. The automa works as follows:

- If there are class A errors or class B1 errors and the B1 meter has reached 200 hours, the indicator light mode is the fourth, that is *Continuous MI*, and the indicator light is continuously activated. The automaton state is called **ContinuosMI**.
- If the *Short MI* mode has not been activated yet and 15 seconds still haven't passed since the engine started: if in the *not ignited* phase the indicator was in the *Short* state, or if 3 operating sequences have not yet elapsed since the last continuous mode MI was switched on or since the last de-validation of a class B1 or B2 fault,

then the indicator light mode is the third one, i.e., *Short MI*, and the indicator is kept active for 15 seconds. This corresponds to the **ShortMI** state of the automa.

- If the short MI mode has already been activated, or type B1 or B2 faults are present, or if 3 operating sequences have not yet elapsed since the last continuous mode MI was switched on or a class B1 or B2 fault was last de-validated, then the indicator is set to the third mode, but the lamp is still kept off because the first 15 seconds since the engine started have already passed. This corresponds to the sub-state **Short** of the **MIOff**.
- If class C faults are present, the indicator shall be set to the second mode, namely *On-demand MI*, but the lamp shall still be kept off. This can be seen in the **OnDemand** sub-state.
- If none of the previous cases happened, the indicator mode is the first, a.k.a., *No Dtc MI*, while the lamp is kept off.



```
[bsCntFaultClassA>0 || ...
(bsCntFaultClassB1>0 && bsB1MainCnt>200)]
{osMILActMode = 0x3;
osMILCmd = TRUE;
osContMIFlg = TRUE;}
```

```
[(osB1EngOnFlag == 0 &&...
(bsCntr10ms - osMIDispTimer1) <= osB1_ENGINE_ON) &&...
(bsAOpSeq < 3 || ...
bsB1OpSeq < 3 || ...
osMILActMode == 2)]
{osMILActMode = 0x2;
osMIDispTimer1 = bsCntr10ms;
osMILCmd = TRUE;
osB1EngOnFlag = TRUE;
osContMIFlg = FALSE;}
```

```
[(bsCnt10ms - osMIDispTimer1) > osB1_ENGINE_ON]
```

```
{osMILCmd = FALSE;
osContMIFlg = FALSE;}
```

```
{osMILActMod = 0x0;}
```

```
[bsAOpSeq < 3 || ...
bsB1OpSeq < 3 || ...
bsB1B2OpSeq < 3 || ...
osCntFaultClassB1 >0 || ...
osCntFaultClassB2 > 0]
{osMILActMod = 0x2;}
```

```
[osCntFaultClassC > 0]
{osMILActMod = 0x1;}
```

Figure 5.6.8 - FSM for the engine ignited

The designed finite state machines require support for the counters and the checks on the active faults to work. Additionally, the strategy requires functions to set the state of the engine and mark the passing of the operating sequences. As already shown, the operating sequence management has been handed to the users; both the engine status and the system readiness have been treated similarly, providing APIs to manage them at the model-based software level.

5.6.3. Implementation

While the implementation of this part did not require any particular data structure, it does make use of many variables, and they also occupy some of the entries in the error memory *tDiagEEPROM* structure. The values that must be stored in order to retrieve them after a shutdown are:

- The continuous MI fault absence operative sequences counter, **bsAOpSeq**.
- The resettable Continuous MI time counter with hourly precision, **bsContMICntEE**, and its support counter with 10 [ms] precision with carry-over, **bsContMISecCntEE**.
- The non-resettable cumulative MI hour counter, **bsContMICumCntEE**, and its support counter with 10 [ms] precision and carry-over, **bsContMICumSecCntEE**.
- The counter for the operating engine hours without continuous MI, **bsOpACntEE**, and its support counter with 10 [ms] precision and carry-over, **bsOpASecCntEE**.
- The flag that shows whether the *readiness monitors* have been completed at least once since the last reset via external tools, **osReadiness**.

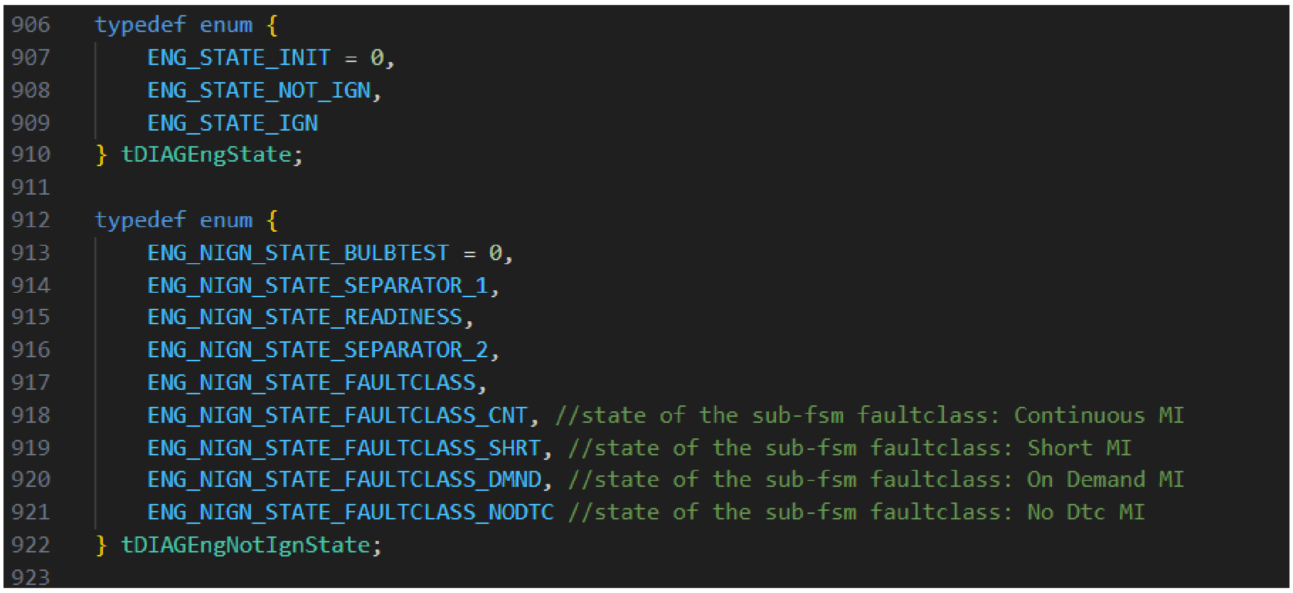
In addition to these fields and their volatile memory counterparts, there are other important variables for the implementation of the strategy, namely the counters of the number of active faults for each class, the flags to mark whether a fault of a certain class happened in the current operating sequence, the flag for the engine ignition, and the variables containing the states of the automata used.

The state variables are the one for the engine state FSM, **osDiagEngFSMState**, the one for the *engine not ignited* FSM and for its sub-FSM *FaultClass,* **osDiagEngNotIgnFSMState** and **osDiagEngNotIgnFltClss**.

The first state variable's type is the enumerative **tDIAGEngState,** which simply contains the three states *init*, *ignited*, and *not ignited*.

The second and third variables share the **tDIAGEngNotIgnState** enum as their type. The labels of this enumerative are split between the four states of the main FSM and the states of the sub-FSM.

Although an enumerative for the *engine* ignited FSM states was originally designed, the implementation chosen for that automaton does not require a state variable, and the enumerative was consequently discarded.



```
typedef enum {
    ENG_STATE_INIT = 0,
    ENG_STATE_NOT_IGN,
    ENG_STATE_IGN
} tDIAGEngState;

typedef enum {
    ENG_NIGN_STATE_BULBTEST = 0,
    ENG_NIGN_STATE_SEPARATOR_1,
    ENG_NIGN_STATE_READINESS,
    ENG_NIGN_STATE_SEPARATOR_2,
    ENG_NIGN_STATE_FAULTCLASS,
    ENG_NIGN_STATE_FAULTCLASS_CNT, //state of the sub-fsm faultclass: Continuous MI
    ENG_NIGN_STATE_FAULTCLASS_SHRT, //state of the sub-fsm faultclass: Short MI
    ENG_NIGN_STATE_FAULTCLASS_DMND, //state of the sub-fsm faultclass: On Demand MI
    ENG_NIGN_STATE_FAULTCLASS_NODTC //state of the sub-fsm faultclass: No Dtc MI
} tDIAGEngNotIgnState;

```

Figure 5.6.9 - Enumeratives for the types of the state variables, in api.c

Other support variables include the timers used by the automa, **osMIDisplayTimer1** and **osMIDisplayTimer2**.

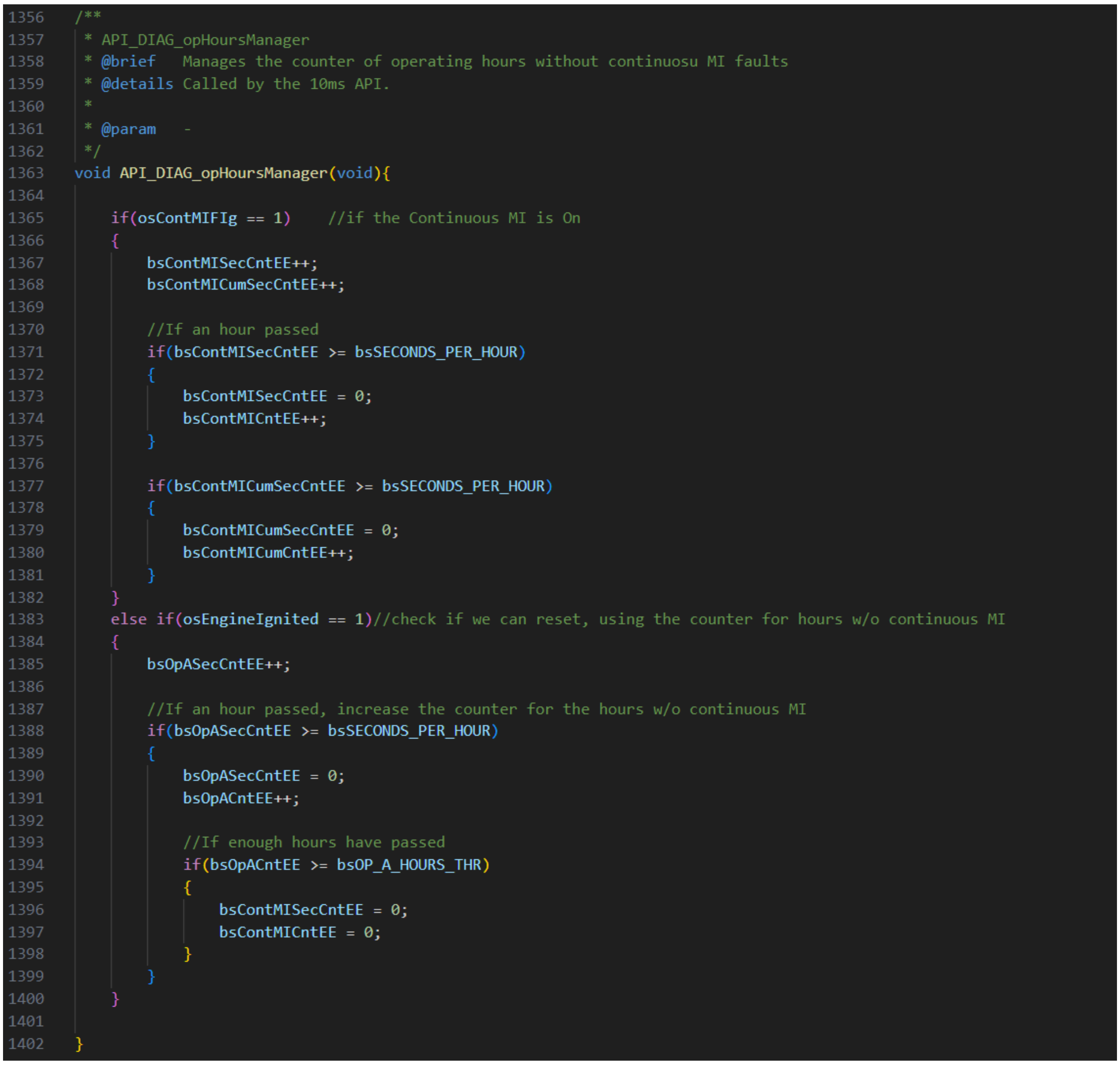
Besides the error memory structure, another piece of the implementation that includes a part useful to the MIL management has already been shown, the *API_DIAG_A_opSeqManager*. This wrapper API includes a call to the function that manages the counter of operating sequences without continuous MI activations, **API_DIAG_A_opSeqManager**.

The function increments the *bsAOpSeq* counter when the engine is ignited and the MIL has not been activated in continuous mode during the current operating sequence, up to a certain threshold (according to the standards, 3). If the presence of a continuous MIL was registered (the **bsContMILWentOn** flag was set), instead it resets the counter. In addition, it also resets *bsContMILWentOn* if the continuous MIL is currently off.

Another function that has already appeared but is mostly important for the MIL management is the **API_DIAG_MIL_readDiagInfo10ms** function, that is called by *API_DIAG_OBDClean*. The function is used to fill the counters of the numbers of different types of *Currently active* faults in the error memory. It works by simply sifting through the error memory fault array and updating the counter based on the current entry class and state.

Apart from being called by the *clean* function, this method is also periodically invoked by the OS routine *API_OS_Task10ms* to keep the counters updated for the MIL.

Another function called by the *API_OS_Task10ms* task, after *API_DIAG_MIL_readDiagInfo10ms*, is **API_DIAG_opHoursManager**. This procedure is designed to manage the counters used as timers for the engine hours passed with and without continuous MI.



```c
/**
 * API_DIAG_opHoursManager
 * @brief   Manages the counter of operating hours without continuosu MI faults
 * @details Called by the 10ms API.
 *
 * @param   -
 */
void API_DIAG_opHoursManager(void){

    if(osContMIFIg == 1)    //if the Continuous MI is On
    {
        bsContMISecCntEE++;
        bsContMICumSecCntEE++;

        //If an hour passed
        if(bsContMISecCntEE >= bsSECONDS_PER_HOUR)
        {
            bsContMISecCntEE = 0;
            bsContMICntEE++;
        }

        if(bsContMICumSecCntEE >= bsSECONDS_PER_HOUR)
        {
            bsContMICumSecCntEE = 0;
            bsContMICumCntEE++;
        }
    }
    else if(osEngineIgnited == 1)//check if we can reset, using the counter for hours w/o continuous MI
    {
        bsOpASecCntEE++;

        //If an hour passed, increase the counter for the hours w/o continuous MI
        if(bsOpASecCntEE >= bsSECONDS_PER_HOUR)
        {
            bsOpASecCntEE = 0;
            bsOpACntEE++;

            //If enough hours have passed
            if(bsOpACntEE >= bsOP_A_HOURS_THR)
            {
                bsContMISecCntEE = 0;
                bsContMICntEE = 0;
            }
        }
    }

}
```

Figure 5.6.10 - API_DIAG_opHoursManager, in api_diagobd.c

The function is split into two parts with an ‘if-else’ construct:

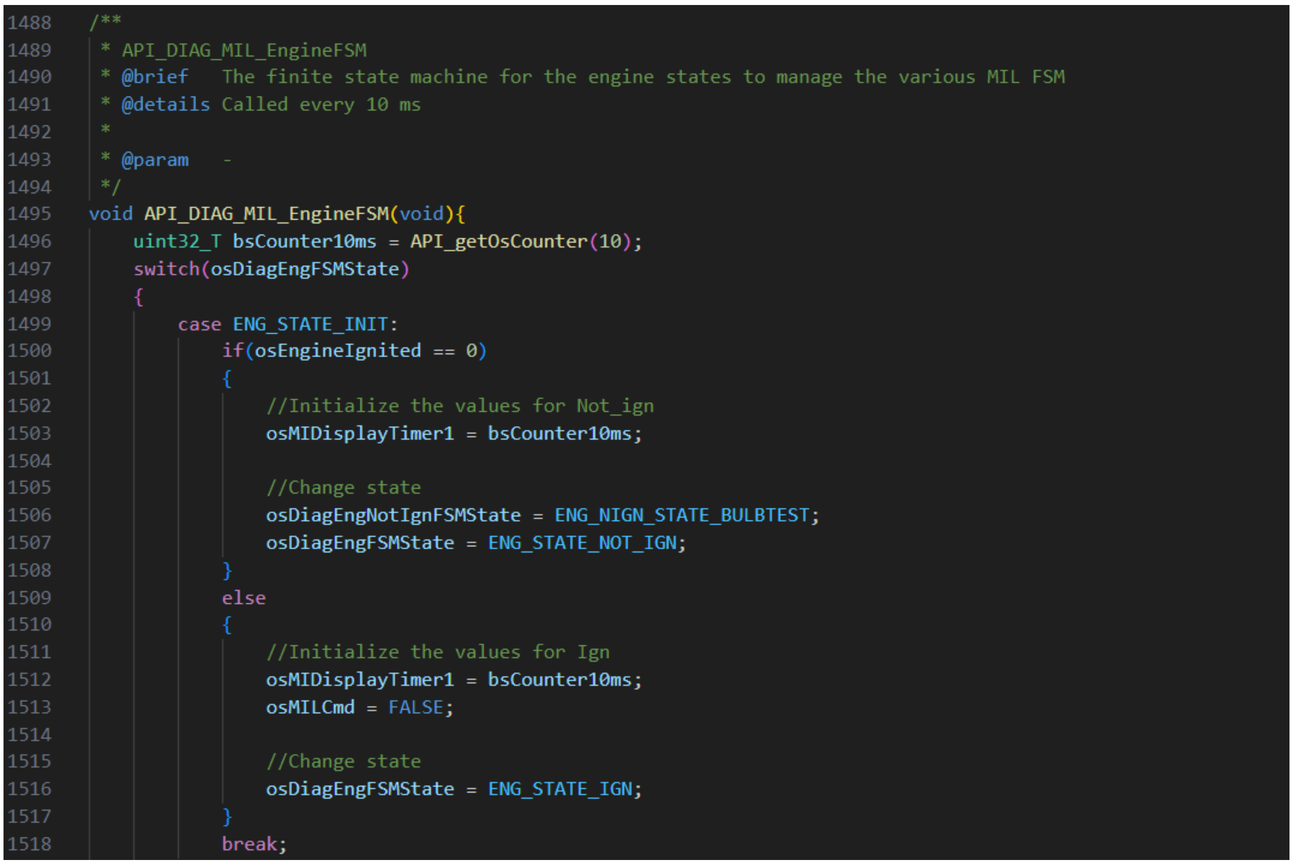
- If the continuous MI is on, i.e., if the **osContMIFlg** flag is set, the function increases the secondary counters *bsContMISecCntEE* and *bsContMICumSecCntEE*. It then goes on to check if either of them reached an hour, and in that case resets the counter in question and increases the corresponding primary counter (either the resettable *bsContMICntEE* or the cumulative *bsContMICumCntEE*).
- If the continuous MIL isn’t on and the engine is ignited, the function increases the secondary counter *bsOpASecCntEE* and then checks whether an hour has passed on it. If that is the case, the secondary counter is reset and the main counter *bsOpACntEE* is increased; then, if enough hours have passed (by regulations, 200 hours), the function resets the counters for the operating hours with continuous MI, *bsContMISecCntEE* and *bsContMICntEE*.

All the previous functions are used to support the ones implementing the FSM shown in the ‘Strategy’ section.

The function **API_DIAG_MIL_EngineFSM** implements the first state machine, which switches between engine ignited and non-ignited states. Differently from what happened with the ADIAs, here there is a single call to this function, and it is embedded in the code and not up to the user. The function is called by the operating system’s task *API_OS_Task10ms* after all the other procedures have been executed in order to have all the variables (timers, counters) correctly updated.

Moreover, the *API_DIAG_MIL_EngineFSM* function has the task of invoking the other two main FSMs and does so depending on its internal state. The three possible states of the machine have been implemented using a ‘switch’ construct on the state variable *osDiagEngFSMState*.

If the variable’s value is **ENG_STATE_INIT**, which corresponds to the default initialisation value at startup, the function checks the flag **osEngineIgnited**, which marks whether the engine is ignited or not, and that can be set via the **API_DIAG_setEngineIgnited** API.
If the flag is set to zero, the function initialises the *osMIDisplayTimer1* with the 10 [ms] OS timer read at the beginning of the function, then sets its state variable to the **ENG_STATE_NOT_IGN** and initialises the state variable of the *not ignited* FSM with the **ENG_NIGN_STATE_BULBTEST** state.
If the flag is equal to 1, instead sets the *osMIDisplayTimer1*, changes its state variable to **ENG_STATE_IGN**, and turns off the lamp.

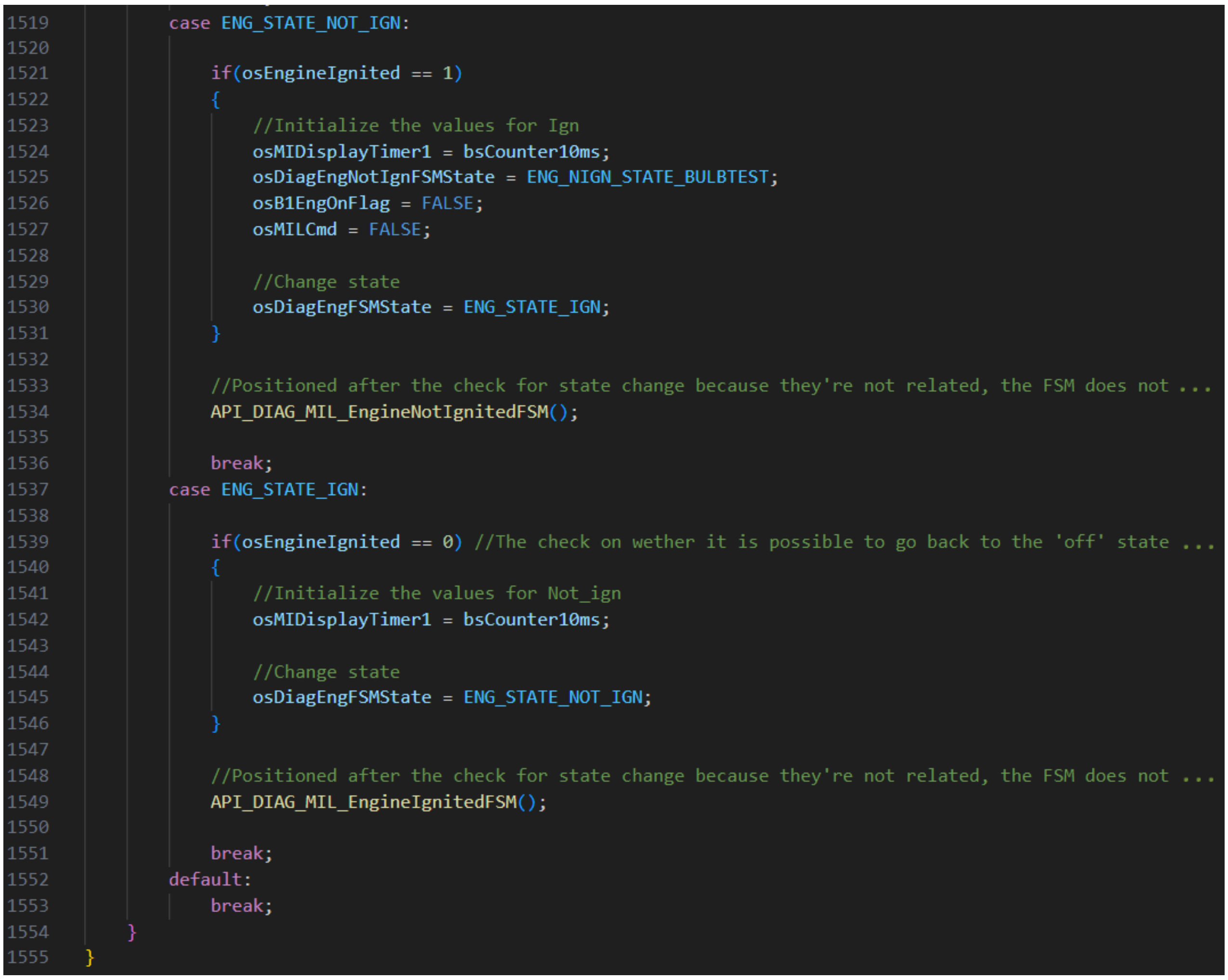


```c
/**
 * API_DIAG_MIL_EngineFSM
 * @brief   The finite state machine for the engine states to manage the various MIL FSM
 * @details Called every 10 ms
 *
 * @param   -
 */
void API_DIAG_MIL_EngineFSM(void){
    uint32_T bsCounter10ms = API_getOsCounter(10);
    switch(osDiagEngFSMState)
    {
        case ENG_STATE_INIT:
            if(osEngineIgnited == 0)
            {
                //Initialize the values for Not_ign
                osMIDisplayTimer1 = bsCounter10ms;

                //Change state
                osDiagEngNotIgnFSMState = ENG_NIGN_STATE_BULBTEST;
                osDiagEngFSMState = ENG_STATE_NOT_IGN;
            }
            else
            {
                //Initialize the values for Ign
                osMIDisplayTimer1 = bsCounter10ms;
                osMILCmd = FALSE;

                //Change state
                osDiagEngFSMState = ENG_STATE_IGN;
            }
            break;
```

Figure 5.6.11a - The 'init state' case of API_DIAG_MIL_EngineFSM, in api_diagobd.c

If the state variable is set to *ENG_STATE_NOT_IGN*, it starts by checking the *engine ignited* flag and, if it has been set, updates the *osMIDisplayTimer1* counter, resets the *engine not ignited* state variable to the *ENG_NIGN_STATE_BULBTEST* value, turns off the lamp setting **osMILCmd** to *false*, and finally changes its own state variable's value to *ENG_STATE_IGN*. Regardless of whether the previous actions have been taken or not, the function then calls the **API_DIAG_MIL_EngineNotIgnitedFSM** method, implementing the *not ignited* FSM.

The *ENG_STATE_IGN* switch case starts by checking if the *engine ignited* flag has been reset to zero, and in that case initialises the *osMIDisplayTimer1* with the current value of the 10 [ms] timer, then updates the FSM state variable to *ENG_STATE_NOT_IGN*. Note that the additional check done for '*Start and Stop*' systems is not done in the state-switch condition, but rather it is implemented in the API for the *engine ignited* flag, *API_DIAG_setEngineIgnited*, that checks the value of the calibratable flag **osEN_ENG_NOT_IGN** to decide if it is possible to go back to the *not ignited* status. Independently from the results of the flag check, the FSM executes the function implementing the *engine ignited* state machine, **API_DIAG_MIL_EngineIgnitedFSM**.



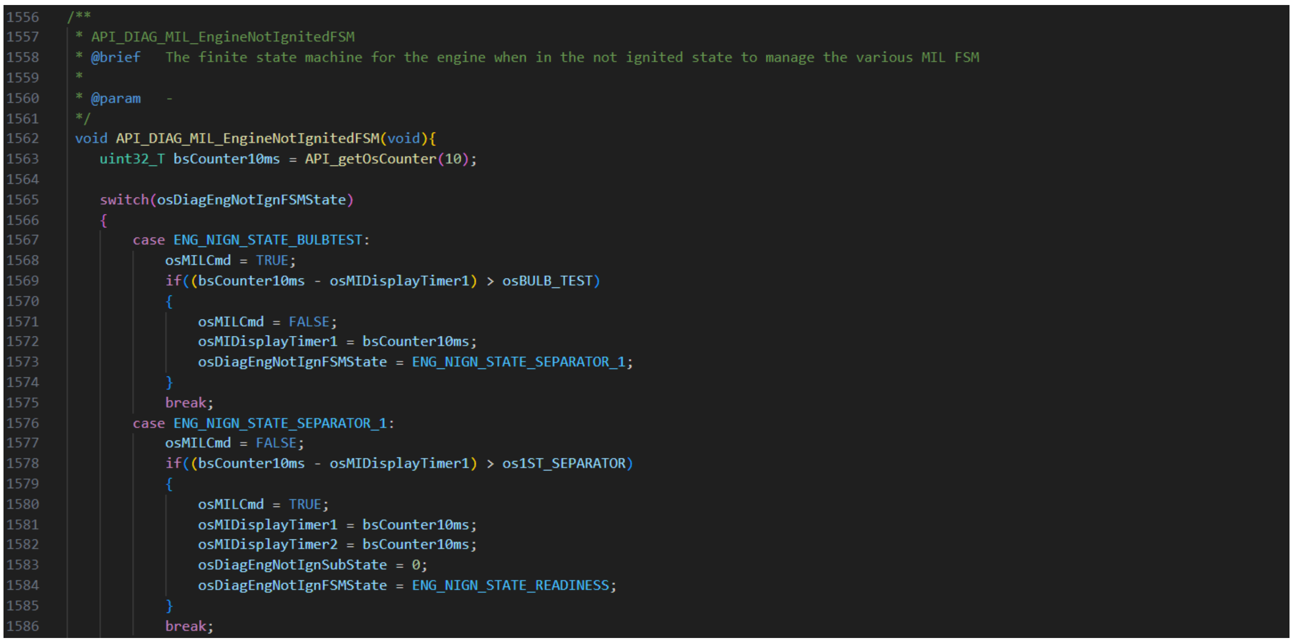
```c
        case ENG_STATE_NOT_IGN:

            if(osEngineIgnited == 1)
            {
                //Initialize the values for Ign
                osMIDisplayTimer1 = bsCounter10ms;
                osDiagEngNotIgnFSMState = ENG_NIGN_STATE_BULBTEST;
                osB1EngOnFlag = FALSE;
                osMILCmd = FALSE;

                //Change state
                osDiagEngFSMState = ENG_STATE_IGN;
            }

            //Positioned after the check for state change because they're not related, the FSM does not ...
            API_DIAG_MIL_EngineNotIgnitedFSM();

            break;
        case ENG_STATE_IGN:

            if(osEngineIgnited == 0) //The check on wether it is possible to go back to the 'off' state ...
            {
                //Initialize the values for Not_ign
                osMIDisplayTimer1 = bsCounter10ms;

                //Change state
                osDiagEngFSMState = ENG_STATE_NOT_IGN;
            }

            //Positioned after the check for state change because they're not related, the FSM does not ...
            API_DIAG_MIL_EngineIgnitedFSM();

            break;
        default:
            break;
    }
}
```

Figure 5.6.11b – The 'engine state' cases of API_DIAG_MIL_EngineFSM, in api_diagobd.c

The *API_DIAG_MIL_EngineNotIgnitedFSM* function implements the *engine not ignited* state machine using a 'switch' construct with five cases plus the default one. The FSM moves from one state to the next once the difference between the current OS 10 [ms] time counter and the *osMIDisplayTimer1* is greater than a certain threshold, which depends on the current state. Each of these thresholds is a calibratable variable with the default value provided for by the guidelines.

The first case, *ENG_NIGN_STATE_BULBTEST*, simply keeps the MIL on, setting *osMILCmd* to *true.* Once the threshold **osBULB_TEST** has been reached, it turns off the MIL, updates the time counter with the current value, and moves to the next state, **ENG_NIGN_STATE_SEPARATOR_1**.

The *ENG_NIGN_STATE_SEPARATOR_1* case keeps the MIL turned off and, once reached the **os1ST_SEPARATOR** time threshold, updates the main time counter and the secondary one (*osMIDisplayTimer2*), turns on the MIL, and prepares the FSM for the next state, updating the state variable's value to **ENG_NIGN_STATE_READINESS**. Additionally, it also prepares the *readiness* sub-FSM by setting its state variable, ***osDiagEngNotIgnSubState***.

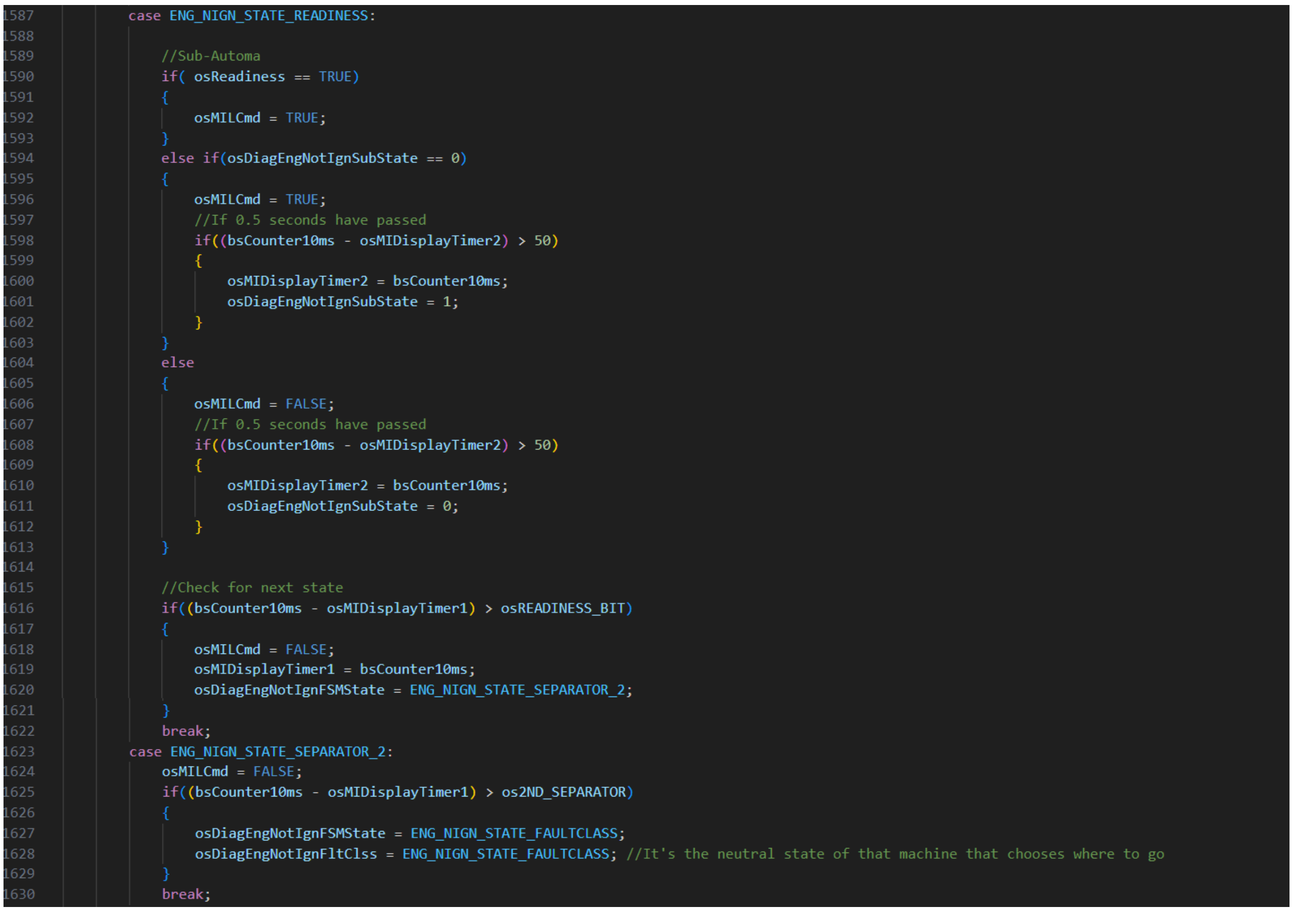


```
/**
 * API_DIAG_MIL_EngineNotIgnitedFSM
 * @brief   The finite state machine for the engine when in the not ignited state to manage the various MIL FSM
 *
 * @param   -
 */
void API_DIAG_MIL_EngineNotIgnitedFSM(void){
   uint32_T bsCounter10ms = API_getOsCounter(10);

   switch(osDiagEngNotIgnFSMState)
   {
       case ENG_NIGN_STATE_BULBTEST:
           osMILCmd = TRUE;
           if((bsCounter10ms - osMIDisplayTimer1) > osBULB_TEST)
           {
               osMILCmd = FALSE;
               osMIDisplayTimer1 = bsCounter10ms;
               osDiagEngNotIgnFSMState = ENG_NIGN_STATE_SEPARATOR_1;
           }
           break;
       case ENG_NIGN_STATE_SEPARATOR_1:
           osMILCmd = FALSE;
           if((bsCounter10ms - osMIDisplayTimer1) > os1ST_SEPARATOR)
           {
               osMILCmd = TRUE;
               osMIDisplayTimer1 = bsCounter10ms;
               osMIDisplayTimer2 = bsCounter10ms;
               osDiagEngNotIgnSubState = 0;
               osDiagEngNotIgnFSMState = ENG_NIGN_STATE_READINESS;
           }
           break;
```

Figure 5.6.12a - First two cases of the API_DIAG_MIL_EngineNotIgnitedFSM function, in api_diagobd.c

The *ENG_NIGN_STATE_READINESS* case implements a sub-automaton with three states. The first state is reached if the *readiness* flag has been set using the **API_DIAG_setDiagsReady** API, and it simply turns on the MIL.
The other two states are used to produce a blinking effect, turning the MIL on and off and moving from one state to the other when the difference between the secondary time counter and the current OS time is greater than 0.5 seconds.
The function then uses the main counter to check if the **osREADINESS_BIT** threshold has been reached before moving to the next state.

The following case, **ENG_NIGN_STATE_SEPARATOR_2**, works exactly as the previous separator state except for the use of a different threshold, **os2ND_SEPARATOR**.
The lamp remains off for the whole duration of the state, then the machine moves to the next state, which implements the *FaultClass* sub-FSM, setting both the *engine not ignited* state variable and the sub-automaton state variable's values to **ENG_NIGN_STATE_FAULTCLASS**.



```
        case ENG_NIGN_STATE_READINESS:

            //Sub-Automa
            if( osReadiness == TRUE)
            {
                osMILCmd = TRUE;
            }
            else if(osDiagEngNotIgnSubState == 0)
            {
                osMILCmd = TRUE;
                //If 0.5 seconds have passed
                if((bsCounter10ms - osMIDisplayTimer2) > 50)
                {
                    osMIDisplayTimer2 = bsCounter10ms;
                    osDiagEngNotIgnSubState = 1;
                }
            }
            else
            {
                osMILCmd = FALSE;
                //If 0.5 seconds have passed
                if((bsCounter10ms - osMIDisplayTimer2) > 50)
                {
                    osMIDisplayTimer2 = bsCounter10ms;
                    osDiagEngNotIgnSubState = 0;
                }
            }

            //Check for next state
            if((bsCounter10ms - osMIDisplayTimer1) > osREADINESS_BIT)
            {
                osMILCmd = FALSE;
                osMIDisplayTimer1 = bsCounter10ms;
                osDiagEngNotIgnFSMState = ENG_NIGN_STATE_SEPARATOR_2;
            }
            break;
        case ENG_NIGN_STATE_SEPARATOR_2:
            osMILCmd = FALSE;
            if((bsCounter10ms - osMIDisplayTimer1) > os2ND_SEPARATOR)
            {
                osDiagEngNotIgnFSMState = ENG_NIGN_STATE_FAULTCLASS;
                osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS; //It's the neutral state of that machine that chooses where to go
            }
            break;
```

Figure 5.6.12b - Readiness FSM and the second separator case of API_DIAG_MIL_EngineNotIgnitedFSM, in api_diagobd.c

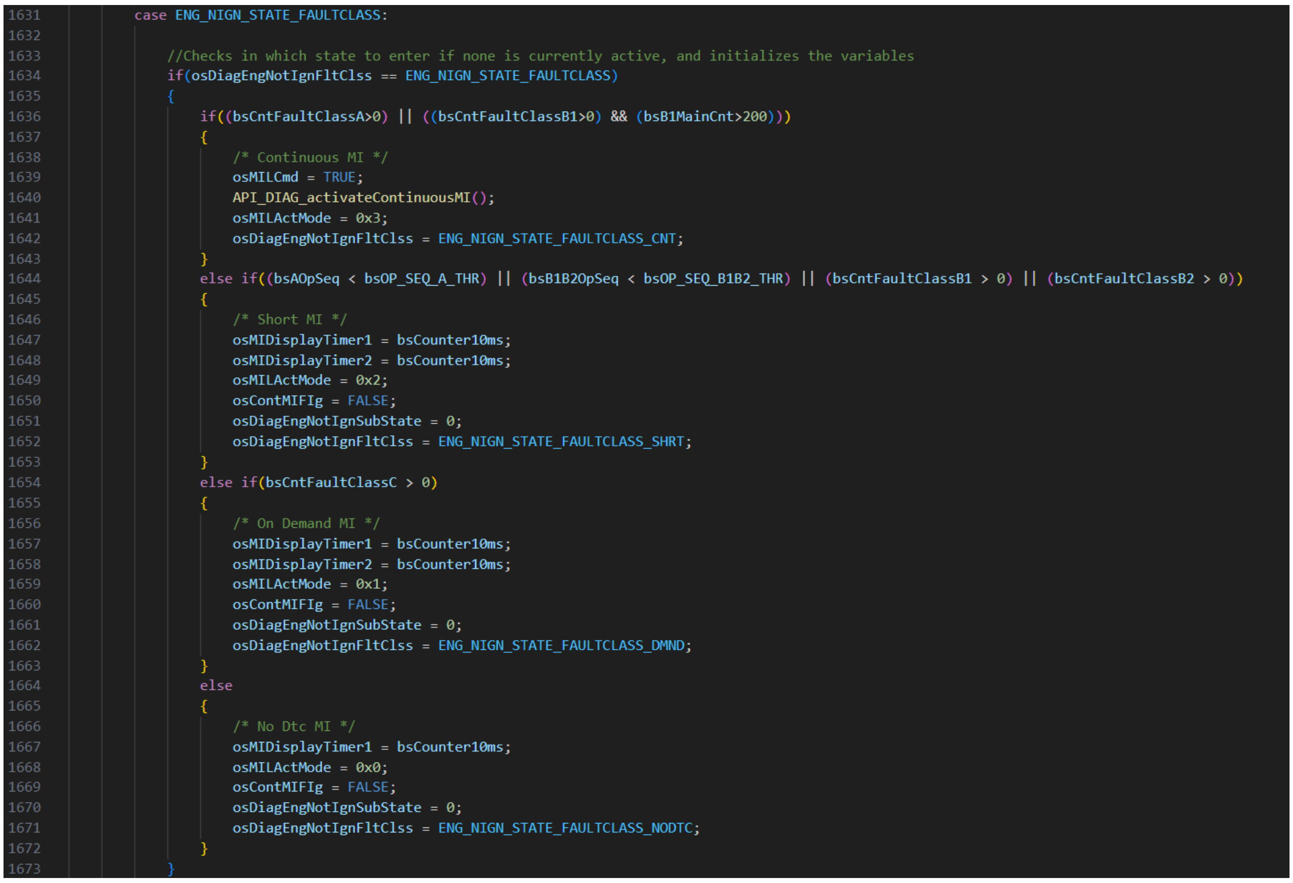
The *ENG_NIGN_STATE_FAULTCLASS* case implements the *FaultClass* FSM seen in the ‘Strategy’ section. To determine which of the possible states the machine should enter, a first part performs a set of checks on various conditions, using an ‘if-then-else if’ structure to ensure that the most severe conditions have priority.

The first check is performed for the *Continuous MI* mode, and if it is successful, the activation mode (**osMILActMode**) is set to 0x3, the lamp is turned on, the sub-machine state variable is set to **ENG_NIGN_STATE_FAULTCLASS_CNT**, and the **API_DIAG_activateContinuousMI** function is called. This function is used to reset all the various counters that kept track of certain events *without* continuous MI activations and sets the *osContMIFlg* and *bsContMILWentOn* flags.

The next check is for the *Short MI* activation mode, and if the conditions are verified, the function turns off the MIL, sets the activation mode to 0x2, updates *osMIDisplayTimer1* and *osMIDisplayTimer2* to use them in the *Short MI* FSM, and sets the sub-FSM state variable to *ENG_NIGN_STATE_FAULTCLASS_SHRT*.

The third check is done for the *On Demand MI* mode and initialises the same variables but sets the activation mode to 0x1 and the sub-state variable to *ENG_NIGN_STATE_FAULTCLASS_DMND*.

If none of the previous checks passes, the function updates *osMIDisplayTimer1*, turns off the lamp, sets the activation mode to 0x0 (*No DTC*), and sets the sub-state to *ENG_NIGN_STATE_FAULTCLASS_NODTC*.



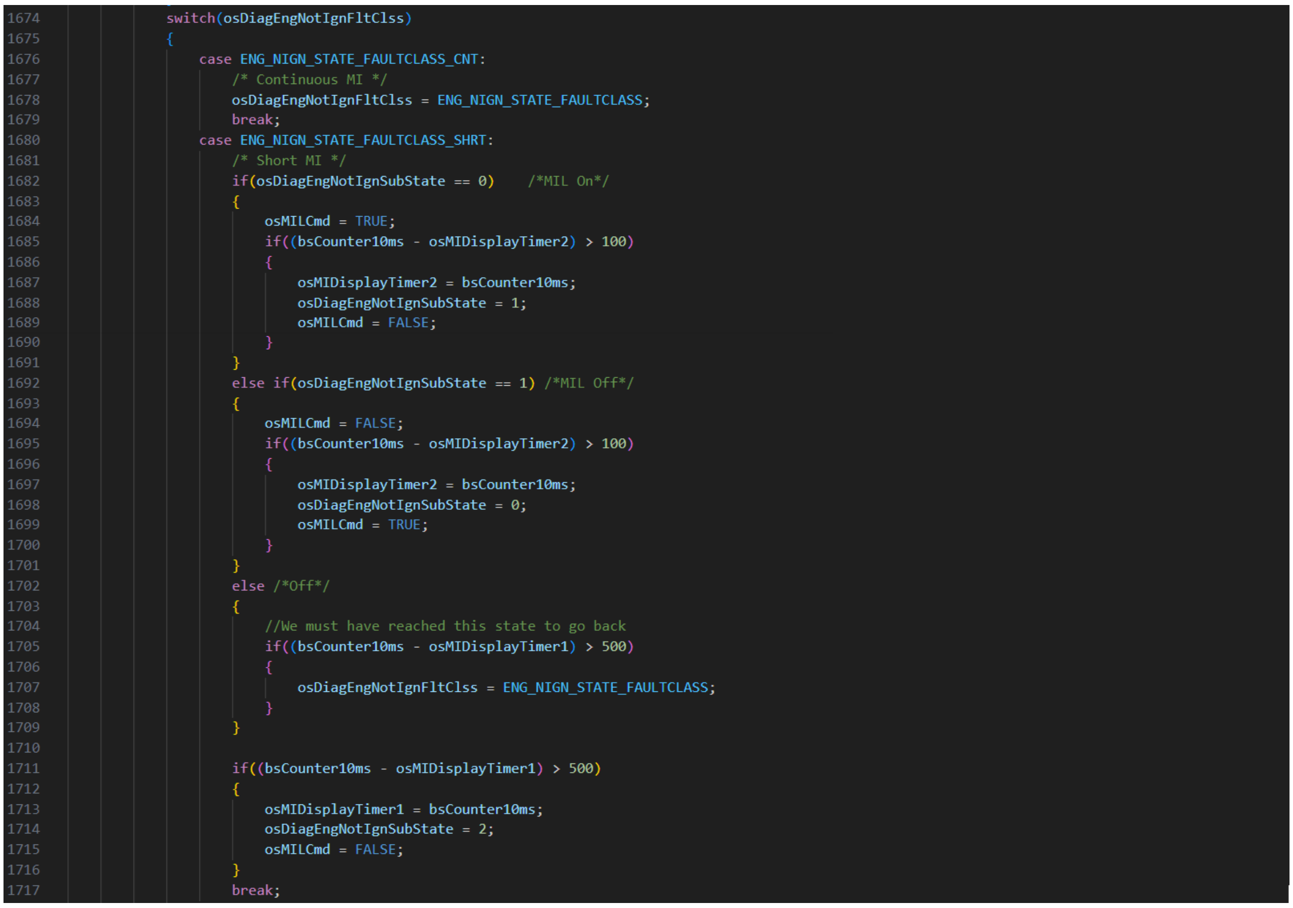
```c
        case ENG_NIGN_STATE_FAULTCLASS:

            //Checks in which state to enter if none is currently active, and initializes the variables
            if(osDiagEngNotIgnFltClss == ENG_NIGN_STATE_FAULTCLASS)
            {
                if((bsCntFaultClassA>0) || ((bsCntFaultClassB1>0) && (bsB1MainCnt>200)))
                {
                    /* Continuous MI */
                    osMILCmd = TRUE;
                    API_DIAG_activateContinuousMI();
                    osMILActMode = 0x3;
                    osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS_CNT;
                }
                else if((bsAOpSeq < bsOP_SEQ_A_THR) || (bsB1B2OpSeq < bsOP_SEQ_B1B2_THR) || (bsCntFaultClassB1 > 0) || (bsCntFaultClassB2 > 0))
                {
                    /* Short MI */
                    osMIDisplayTimer1 = bsCounter10ms;
                    osMIDisplayTimer2 = bsCounter10ms;
                    osMILActMode = 0x2;
                    osContMIFIg = FALSE;
                    osDiagEngNotIgnSubState = 0;
                    osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS_SHRT;
                }
                else if(bsCntFaultClassC > 0)
                {
                    /* On Demand MI */
                    osMIDisplayTimer1 = bsCounter10ms;
                    osMIDisplayTimer2 = bsCounter10ms;
                    osMILActMode = 0x1;
                    osContMIFIg = FALSE;
                    osDiagEngNotIgnSubState = 0;
                    osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS_DMND;
                }
                else
                {
                    /* No Dtc MI */
                    osMIDisplayTimer1 = bsCounter10ms;
                    osMILActMode = 0x0;
                    osContMIFIg = FALSE;
                    osDiagEngNotIgnSubState = 0;
                    osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS_NODTC;
                }
            }
```

Figure 5.6.12c - FaultClass section to determine the sub-FSM state in API_DIAG_MIL_EngineNotIgnitedFSM, in api_diagobd.c

The second part of the case is what actually implements the *FaultClass* FSM, using a 'switch-case' construct on the *osDiagEngNotIgnFltClss* state variable.

The first case of this switch is *ENG_NIGN_STATE_FAULTCLASS_CNT*, and it simply keeps the lamp on.

The second case defines the MIL behaviour for the *Short MI* activation mode, and id uses a ternary variable, **osDiagEngNotIgnSubState**, to determine whether blink the lamp or keep it off. For the blinking, the secondary time counter is used, while the main time counter, *osMIDisplayTimer1*, is used both to keep the light off for 5 seconds after blinking and to move back to a new state by returning to the first part of the *ENG_NIGN_STATE_FAULTCLASS* case.

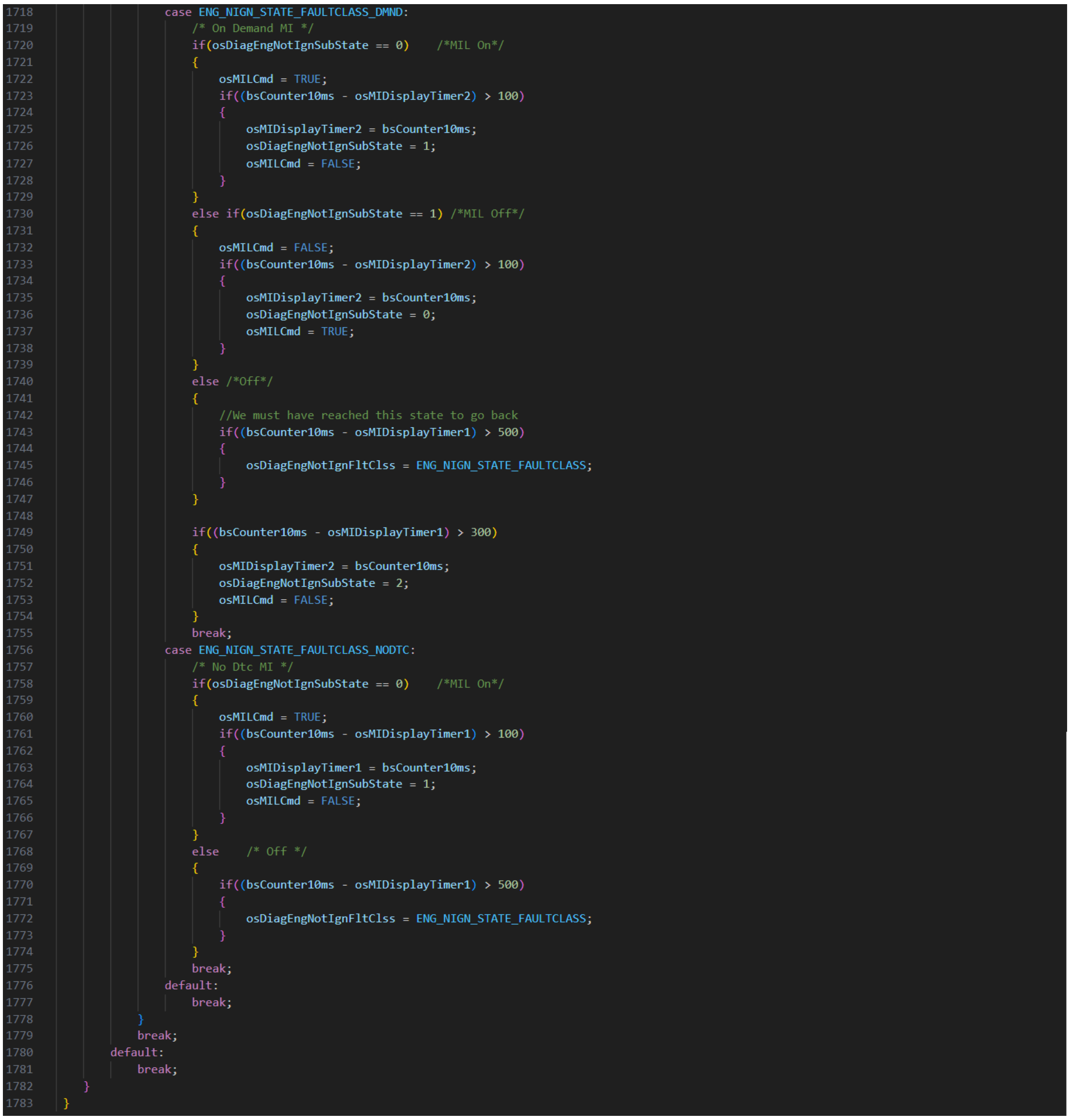


```c
switch(osDiagEngNotIgnFltClss)
{
    case ENG_NIGN_STATE_FAULTCLASS_CNT:
        /* Continuous MI */
        osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS;
        break;
    case ENG_NIGN_STATE_FAULTCLASS_SHRT:
        /* Short MI */
        if(osDiagEngNotIgnSubState == 0)    /*MIL On*/
        {
            osMILCmd = TRUE;
            if((bsCounter10ms - osMIDisplayTimer2) > 100)
            {
                osMIDisplayTimer2 = bsCounter10ms;
                osDiagEngNotIgnSubState = 1;
                osMILCmd = FALSE;
            }
        }
        else if(osDiagEngNotIgnSubState == 1) /*MIL Off*/
        {
            osMILCmd = FALSE;
            if((bsCounter10ms - osMIDisplayTimer2) > 100)
            {
                osMIDisplayTimer2 = bsCounter10ms;
                osDiagEngNotIgnSubState = 0;
                osMILCmd = TRUE;
            }
        }
        else /*Off*/
        {
            //We must have reached this state to go back
            if((bsCounter10ms - osMIDisplayTimer1) > 500)
            {
                osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS;
            }
        }

        if((bsCounter10ms - osMIDisplayTimer1) > 500)
        {
            osMIDisplayTimer1 = bsCounter10ms;
            osDiagEngNotIgnSubState = 2;
            osMILCmd = FALSE;
        }
        break;
```

Figure 5.6.12d - The first two states of the implemented FaultClass FSM in API_DIAG_MIL_EngineNotIgnitedFSM, in api_diagobd.c

The third switch case, i.e., when the *osDiagEngNotIgnFltClss* variable assumes the value **ENG_NIGN_STATE_FAULTCLASS_DMND**, is used to implement the *OnDemand MI* activation mode. It exploits the same ternary variable of the previous case and works in the same way, but with slightly different timings (as the MI must blink twice rather than thrice per cycle).

The last case, *ENG_NIGN_STATE_FAULTCLASS_NODTC*, still uses the same ternary variable *osDiagEngNotIgnSubState* but only needs two values, as the lamp must blink only once. For this reason, unlike the two preceding states, it only uses the *osMIDisplayTimer1* counter for both the MIL and to move to the other states (by returning to the first part of the *ENG_NIGN_STATE_FAULTCLASS* case).



```
                case ENG_NIGN_STATE_FAULTCLASS_DMND:
                    /* On Demand MI */
                    if(osDiagEngNotIgnSubState == 0)    /*MIL On*/
                    {
                        osMILCmd = TRUE;
                        if((bsCounter10ms - osMIDisplayTimer2) > 100)
                        {
                            osMIDisplayTimer2 = bsCounter10ms;
                            osDiagEngNotIgnSubState = 1;
                            osMILCmd = FALSE;
                        }
                    }
                    else if(osDiagEngNotIgnSubState == 1) /*MIL Off*/
                    {
                        osMILCmd = FALSE;
                        if((bsCounter10ms - osMIDisplayTimer2) > 100)
                        {
                            osMIDisplayTimer2 = bsCounter10ms;
                            osDiagEngNotIgnSubState = 0;
                            osMILCmd = TRUE;
                        }
                    }
                    else /*Off*/
                    {
                        //We must have reached this state to go back
                        if((bsCounter10ms - osMIDisplayTimer1) > 500)
                        {
                            osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS;
                        }
                    }

                    if((bsCounter10ms - osMIDisplayTimer1) > 300)
                    {
                        osMIDisplayTimer2 = bsCounter10ms;
                        osDiagEngNotIgnSubState = 2;
                        osMILCmd = FALSE;
                    }
                    break;
                case ENG_NIGN_STATE_FAULTCLASS_NODTC:
                    /* No Dtc MI */
                    if(osDiagEngNotIgnSubState == 0)    /*MIL On*/
                    {
                        osMILCmd = TRUE;
                        if((bsCounter10ms - osMIDisplayTimer1) > 100)
                        {
                            osMIDisplayTimer1 = bsCounter10ms;
                            osDiagEngNotIgnSubState = 1;
                            osMILCmd = FALSE;
                        }
                    }
                    else    /* Off */
                    {
                        if((bsCounter10ms - osMIDisplayTimer1) > 500)
                        {
                            osDiagEngNotIgnFltClss = ENG_NIGN_STATE_FAULTCLASS;
                        }
                    }
                    break;
                default:
                    break;
            }
            break;
        default:
            break;
    }
}
```

Figure 5.6.12e - The last two states of the FaultClass FSM and the last lines of the 'engine not ignited' FSM, in api_diagobd.c

The FSM for the *engine ignited* is implemented in the **API_DIAG_MIL_EngineIgnitedFSM** function. As expected from the FSM graphs shown in the 'Strategy' section, this function's structure is far simpler than the *engine not ignited* one. It does not require a personal state variable anymore because it is sufficient to keep track of the combination of the activation mode and the *osMIDisplayTimer1* to produce the required MIL behaviour.
The implementation is similar to the first part of the *FaultClass* section of the *engine not ignited* function, with a set of mutually exclusive conditions checked with an 'if-elsif' construct.

The first condition is the one used for the *Continuous MI* activation on the class A faults and the class B1 with more than 200 hours. The verification of the condition sets the activation mode variable to 0x3, turns on the light, and calls *API_DIAG_activateContinuousMI*.

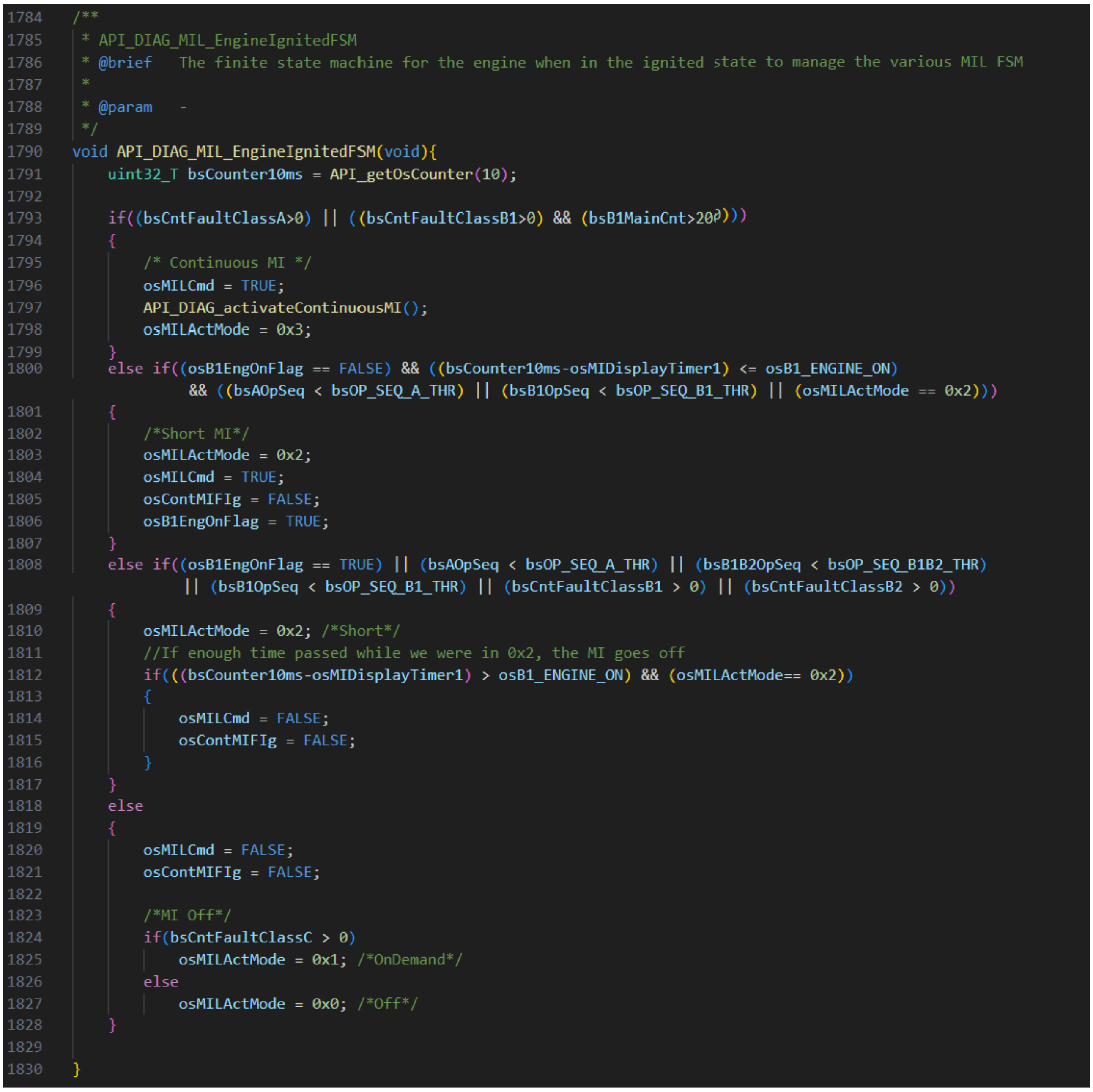
The second condition is used for the part of the Short MI mode during which the lamp is kept on. The function checks that the *osB1EngOnFlag* is still off and that the first 15 seconds (**osB1_ENGINE_ON**) since the system's startup haven't already passed. It also checks that either the activation mode is already 0x2 or that it still hasn't passed a guidelines-defined number of consecutive operating sequences without class A or B1 faults.
If the conditions are respected, the lamp is turned on, the mode is set to 0x2, the *osB1EngOnFlag* that indicates the lasting presence of class B1 faults is set to *true*, while the flag for *Continuous MI*, *osContMIFlg*, is set to *false.*

The third set of conditions is the one that determines the behaviour of the lamp in *Short MI* mode after the first 15 seconds of operation. The condition is verified if either one of the following checks is true:

- The flag *osB1EngOnFlag* is set.
- The number of consecutive operating sequences without continuous MI activation, *bsAOpSeq*, is lower than the value defined by the regulations, **bsOP_SEQ_A_THR** (i.e., 3).
- The number of consecutive of consecutive operating sequences without active B1 or B2 faults is lower than their respective thresholds (i.e., 3).
- The number of active faults of class B1 or B2 is greater than zero.

If the conditions are cleared, the function sets the activation mode to 0x2, and then, if the required amount of time (*osB1_ENGINE_ON*) has already passed, the MIL is turned off, and so is the *osContMIFlg*.

The final 'else' of the construct contains the code for all the sub-states if the *MIOff* state of the original *engine ignited* FSM. Just like the previous case, it sets the *osMILCmd* and *osContMIFlg* flags to *false,* then sets the activation mode to 0x1 or 0x0 depending on whether there are active faults of class C or not.



```c
/**
 * API_DIAG_MIL_EngineIgnitedFSM
 * @brief   The finite state machine for the engine when in the ignited state to manage the various MIL FSM
 *
 * @param   -
 */
void API_DIAG_MIL_EngineIgnitedFSM(void){
    uint32_T bsCounter10ms = API_getOsCounter(10);

    if((bsCntFaultClassA>0) || ((bsCntFaultClassB1>0) && (bsB1MainCnt>200)))
    {
        /* Continuous MI */
        osMILCmd = TRUE;
        API_DIAG_activateContinuousMI();
        osMILActMode = 0x3;
    }
    else if((osB1EngOnFlag == FALSE) && ((bsCounter10ms-osMIDisplayTimer1) <= osB1_ENGINE_ON)
             && ((bsAOpSeq < bsOP_SEQ_A_THR) || (bsB1OpSeq < bsOP_SEQ_B1_THR) || (osMILActMode == 0x2)))
    {
        /*Short MI*/
        osMILActMode = 0x2;
        osMILCmd = TRUE;
        osContMIFIg = FALSE;
        osB1EngOnFlag = TRUE;
    }
    else if((osB1EngOnFlag == TRUE) || (bsAOpSeq < bsOP_SEQ_A_THR) || (bsB1B2OpSeq < bsOP_SEQ_B1B2_THR)
            || (bsB1OpSeq < bsOP_SEQ_B1_THR) || (bsCntFaultClassB1 > 0) || (bsCntFaultClassB2 > 0))
    {
        osMILActMode = 0x2; /*Short*/
        //If enough time passed while we were in 0x2, the MI goes off
        if(((bsCounter10ms-osMIDisplayTimer1) > osB1_ENGINE_ON) && (osMILActMode== 0x2))
        {
            osMILCmd = FALSE;
            osContMIFIg = FALSE;
        }
    }
    else
    {
        osMILCmd = FALSE;
        osContMIFIg = FALSE;

        /*MI Off*/
        if(bsCntFaultClassC > 0)
            osMILActMode = 0x1; /*OnDemand*/
        else
            osMILActMode = 0x0; /*Off*/
    }

}
```

Figure 5.6.13 - API_DIAG_MIL_EngineIgnitedFSM, in api_diagobd.c

The functions seen until now are used to manage the status of the MIL, modifying its parameters, namely the activation mode and the on/off status, acting on system variables. To allow these variables to pilot the actual lamp, two APIs have been programmed: **API_DIAG_MIL_getMILCmd** and **API_DIAG_MIL_getMILActMode**.

In addition, to satisfy the requirements on the information that the OBD system must be able to provide, APIs to read the resettable continuous MI hour counter and the cumulative one have also been introduced: **API_DIAG_MIL_getContMICntr** and **API_DIAG_MIL_getCumulContMICntr**.

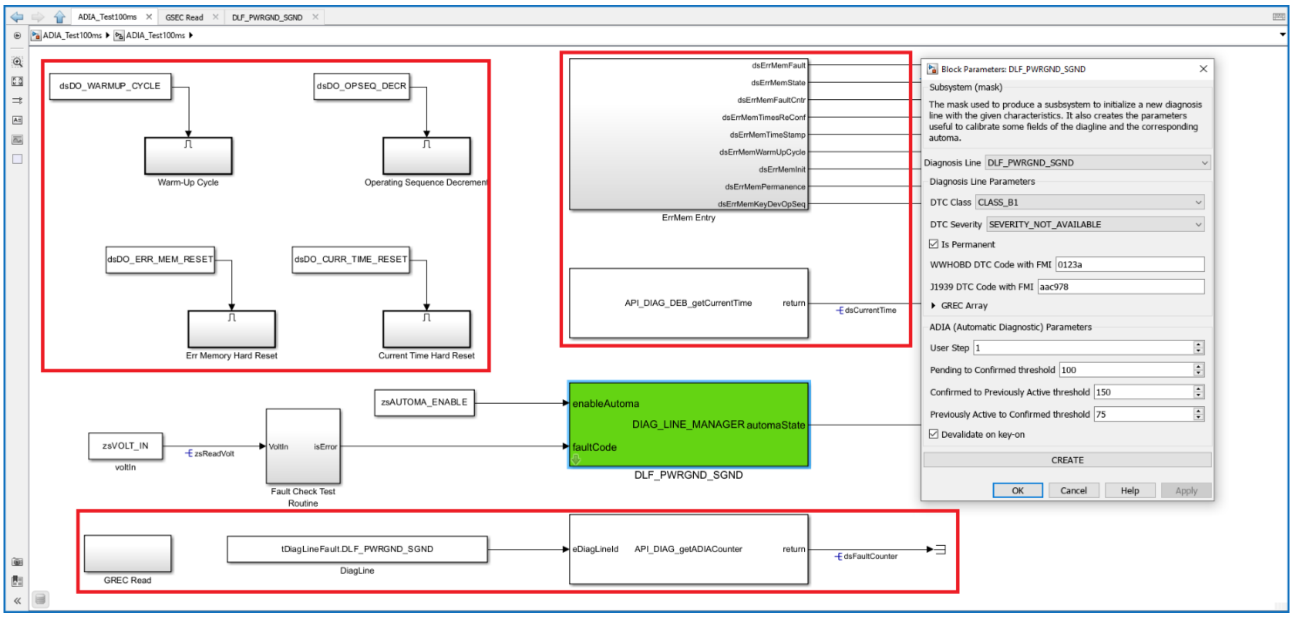
5.7. Example Applications

To ensure that the implemented solutions worked as intended, an example application has been developed in parallel with the implementation of each component of the diagnosis flow, and an additional 'Demo' application has been created once the work has been completed. This allowed to perform parallel testing, as new parts were implemented, as well as integration testing, ensuring that the pre-existing code's behaviour was not altered in the process.

5.7.1. Parallel Test Models

The designed application implements the totality of the blocks required for the diagnosis flow, from the detection of the fault to the activation of the recovery lines. The models also simulate the work of external tools to read the values contained in the error memory.

To facilitate the testing, temporary APIs (enclosed in red squares in the following images) have been created to allow, through the use of CANape, the visualisation of certain data that wouldn't normally be accessible during the system operation. As these functions were developed alongside the solution, some of those debug-only APIs covers the same scope of certain APIs provided for the external tools.



Block Parameters: DLF_PWRGND_SGND

Subsystem (mask)

The mask used to produce a susbsystem to initialize a new diagnosis line with the given characteristics. It also creates the parameters useful to calibrate some fields of the diagline and the corresponding automa.

Diagnosis Line: DLF_PWRGND_SGND

Diagnosis Line Parameters

DTC Class: CLASS_B1

DTC Severity: SEVERITY_NOT_AVAILABLE

☑ Is Permanent

WWHOBD DTC Code with FMI: 0123a

J1939 DTC Code with FMI: aac978

▸ GREC Array

ADIA (Automatic Diagnostic) Parameters

User Step: 1

Pending to Confirmed threshold: 100

Confirmed to Previously Active threshold: 150

Previously Active to Confirmed threshold: 75

☑ Devalidate on key-on

Figure 5.7.1 – The model for testing the fault management

The first model shown is called **ADIA_Test100ms**. It was used to test the ADIA implementation, the correct insertion in memory of a fault's own data, and the

consequent activation of the recovery lines. It was also used to test the mask block. The generated code was executed in the operating system routine *API_OS_Task100ms.*
The 'normal' model parts include:

- A constant block for the calibration **zsVOLT_IN**, used to manipulate the input value read from the sensors. In the final test, the calibration was replaced with a digital input in order to manually create an open circuit fault.
- The **Fault_Check_Test_Routine** subsystem, that takes the signal outputted from the constant block, **zsReadVolt**, compares it to a threshold, and returns an error code when the value is below the threshold.
- The diagnosis line and ADIA mask block, with the mask opened in *Figure 5.7.1*, with the selected fault ID being **DLF_PWRGND_SGND**. The block takes as input the enable signal, obtained from the calibration **zsAUTOMA_ENABLE**, and the result of the fault check routine, in the signal **dsDLF_PWRGND_SGNDPtErr**. The value returned from the ADIA manager is stored in the signal **dsDLF_PWRGND_SGNDErrSt**, not visible in the picture as it is inside the mask block.

The parts of the model used for testing purposes are:

- On the top-left of the picture, four enabled subsystems, each piloted by a different calibration, and each one containing a different API block:
    - Piloted by the calibration **dsDO_WARMUP_CYCLE**, calls the API for the completion of a warm-up cycle.
    - Piloted by **dsDO_OPSEQ_DECR**, enables the API block for the completion of an operating sequence.
    - Piloted by **dsDO_ERR_MEM_RESET**, uses a testing-only API to reset the whole error memory.
    - Piloted by **dsDO_CURR_TIME_RESET,** uses a testing-only API to reset the system timers.
- On the top-right, the **ErrMem_Entry** subsystem uses various testing-only functions to read the various fields of an entry of the error memory faults array given its index, and the associated signals. Just below it, the testing-only API_DIAG_DEB_getCurrentTime is used to read the current time of the system and output it to the **dsCurrentTime.**
- At the bottom of the picture, the **GREC_Read** subsystem uses the API block to read a recovery line given its index, **API_DIAG_GREC_getEntry**; although this block is available under normal circumstances, the read value of the lie should pilot a recovery function, not implemented here. Next to it is the debug-only function

block to read the current value of the fault counter of an ADIA, given the ID of the ADIA's fault.

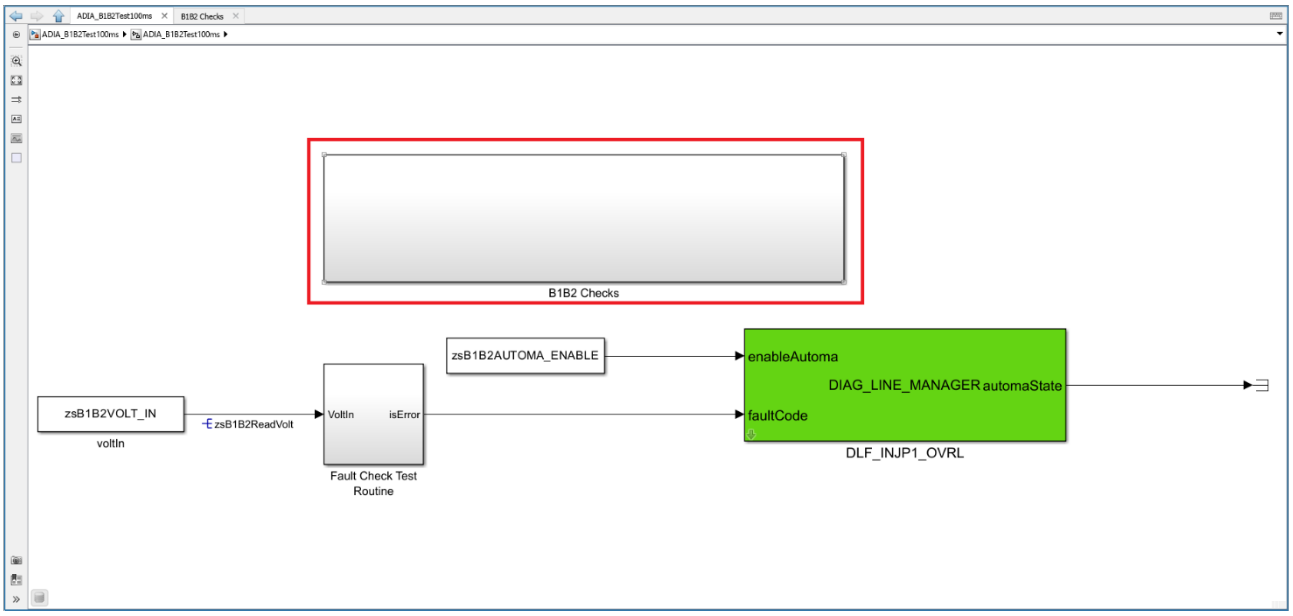


Figure 5.7.2 - The model to test the B1 and B2 counters, ADIA_B1B2Test100ms

The second model introduced is the one used to test the correctness of the implementation of the B1 and B2 faults' timers and counter, **ADIA_B1B2Test100ms**.

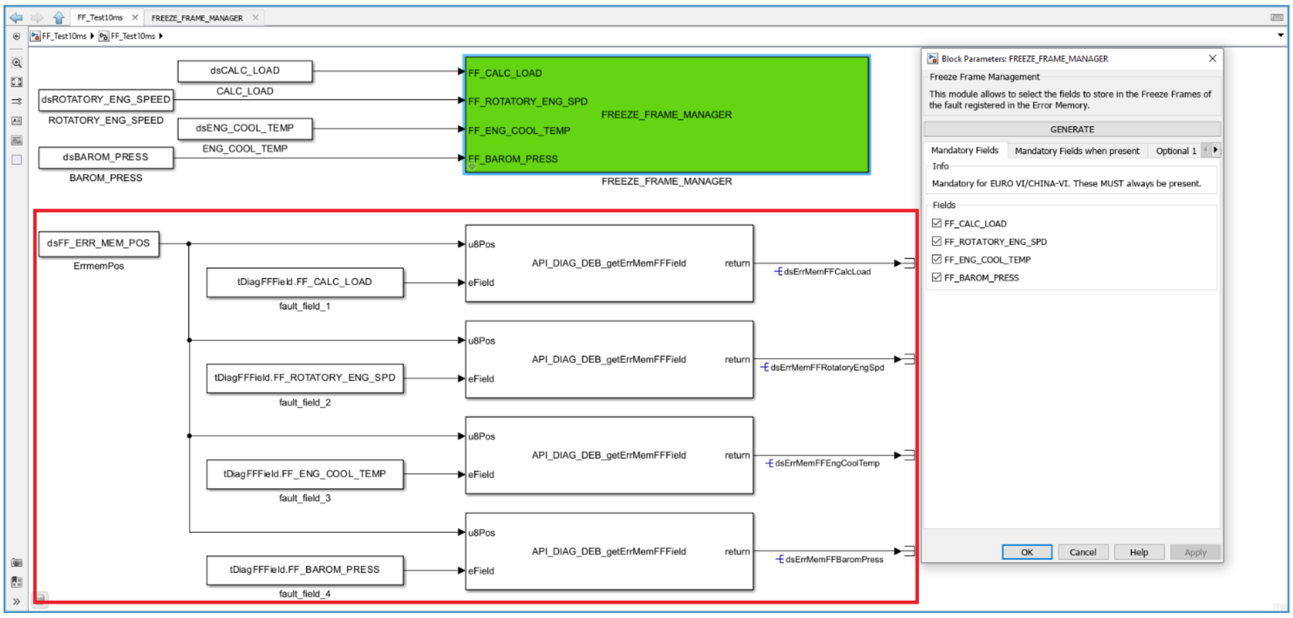
The non-testing part of the model is the same as the previous image, with the set of blocks used to define a diagnosis line and feed to its ADIA the result of a fault check routine. Having both this model and the previous one allowed to validate the solution with multiple faults of different classes present at the same time.

The testing component for this model, **B1B2_Checks**, is a subsystem that contains various testing-only API blocks that allow to read the following information:

- The number of active faults of classes B1 and B2.
- The main and secondary counters for the time passed with an active B1 fault.
- The number of continuous operating sequences without active B1 or B2 faults.
- The flags used to check if a fault of one of those two classes went to the *Confirmed and active* state during the current operating sequence.

As one might notice, some of the testing APIs provide information that should be obtainable through the functions written for the external tools. This is because, while these test models were created hand in hand with the solutions they tested, the tools were one of the last topics covered.

This had the beneficial effect of having already a solid base to start from when implementing the tool's API: the test functions.



Block Parameters: FREEZE_FRAME_MANAGER

Freeze Frame Management

This module allows to select the fields to store in the Freeze Frames of the fault registered in the Error Memory.

GENERATE

Mandatory Fields | Mandatory Fields when present | Optional 1

Info

Mandatory for EURO VI/CHINA-VI. These MUST always be present.

Fields

- ✓ FF_CALC_LOAD
- ✓ FF_ROTATORY_ENG_SPD
- ✓ FF_ENG_COOL_TEMP
- ✓ FF_BAROM_PRESS

Figure 5.7.3 - The model for testing the Freeze Frame setup mask, FF_Test10ms

The third model, FF_Test10ms, was created to put to the test the freeze frame system and the mask to setup the global freeze frame. The freeze frame management, although designed with the other parts of the error memory, was actually implemented later than the rest; here's why it was only tested with the third Simulink model.

The actual system part is only the mask block for the global freeze frame (with the open GUI in *Figure 5.7.3*), with just the four mandatory fields selected and with the corresponding ports piloted by four calibrations: **dsCALC_LOAD** for the engine torque, **dsROTATORY_ENG_SPEED** for the rotatory speed of the engine, **dsENG_COOL_TEMP** for the coolant temperature, and **dsBAROM_PRESS** for the estimated barometric pressure. Using calibrations simplified the tests and allowed us to check if the freeze frames copied in memory changed along the global one.

The test-only part was a set of blocks used to read the value of a freeze frame field, given its enumerative label, of a fault in a certain position in the error memory, where the position was directly passed with the **dsFF_ERR_MEM_POS** calibration. There was one of these blocks for each of the mandatory fields, and each had its own output signal to visualise it on CANape.

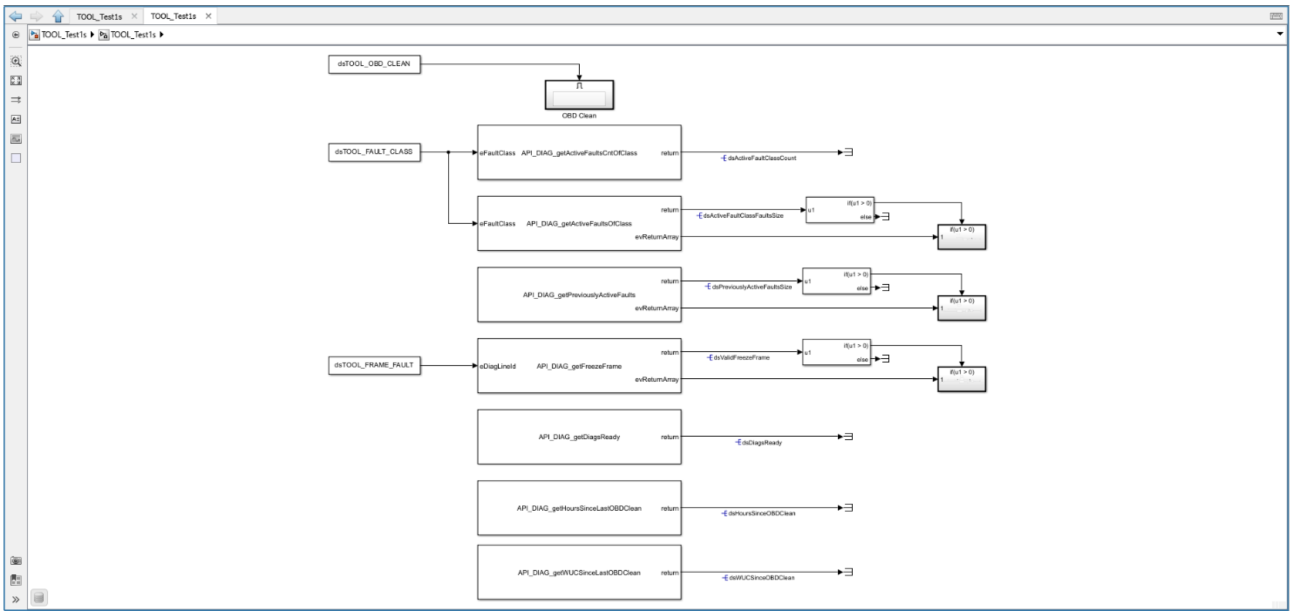


Figure 5.7.4 - The model to test the APIs available for the external tools, TOOL_Test1s

As previously stated, the majority of the APIs for the external diagnostic tools were the last to be developed, and many used the previously mentioned test-only functions as a base. The model designed to test these APIs is **TOOL_Test1s**, partially visible in *Figure 5.7.4*.

The model simply uses the given APIs to check the information that an OBD system is required to provide as per regulations. The only additional components are the 'If Action' subsystems used for those APIs that return an array and the number of found elements, such as **API_DIAG_getActiveFaultsOfClass**, **API_DIAG_getPreviouslyActiveFaults**, and **API_DIAG_getFreezeFrame**. In those cases, the model used a conditional subsystem that, if the number of retrieved elements was greater than zero, a subsystem was executed to sift through the returned array looking for a specific entry. The entry to find was determined using a calibration; in the following image, **dsTOOL_FFPOS**.

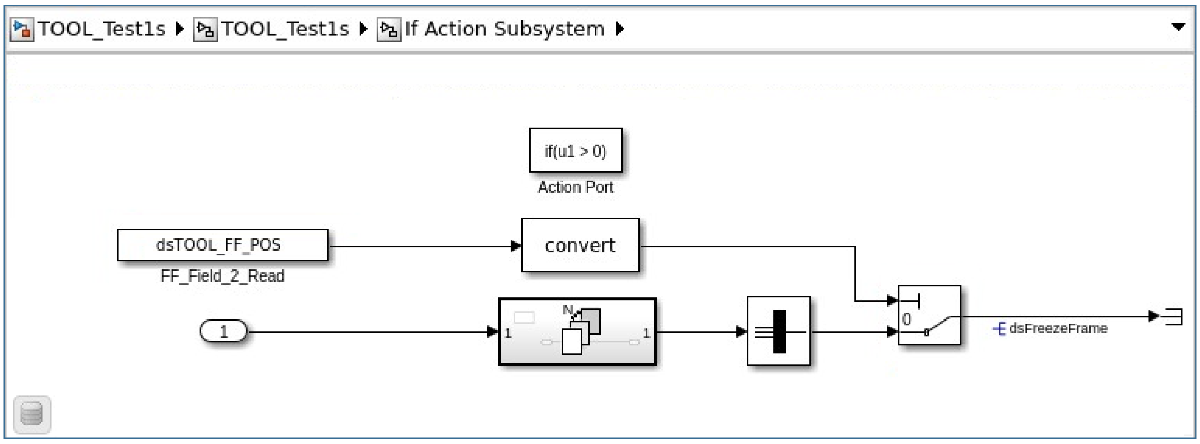


Figure 5.7.5 - Example of 'If action' subsystem in the TOOL model

5.7.2. Demo application

The Demo application was designed to analyse the process from a user's perspective, following the complete flow of the diagnostic functions, starting with the definition of a fault check routine and ending with the execution of the recovery function.

The application works by detecting the presence of an open circuit on a PWM output piloting a LED. The open load will be produced by the tester by disconnecting the controlled pin on the board's **Break-Out Box** (BOB).
Once the fault check routine detects the presence of the open circuit, it issues an error message that gets fed to the diagnosis line's mask block. Once this fault gets elaborated from its ADIA and becomes confirmed, its recovery lines will be activated. One of these recovery lines will pilot a recovery function. In the meantime, another LED will work as the MIL.

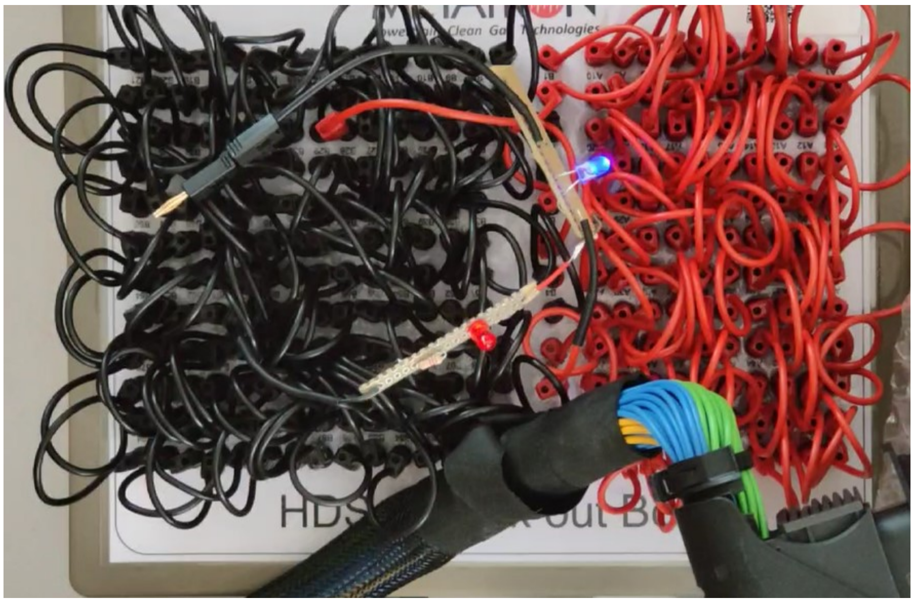


Figure 5.7.6 – The board's BOB with the two LEDs, red for the MIL and blue for the piloted pin

Aside from the model that contains the global freeze frame mask, the demo required two other models.

The first one was used to generate the output to pilot the LED on the observed pin, making use of the Metatron Library's **API_PWMOUT_setPeriodAndDuty** to determine the parameters for the PWM signal.

The second, and main, model implemented the various parts of the OBD diagnosis flow and was called **Demo_App100ms**.

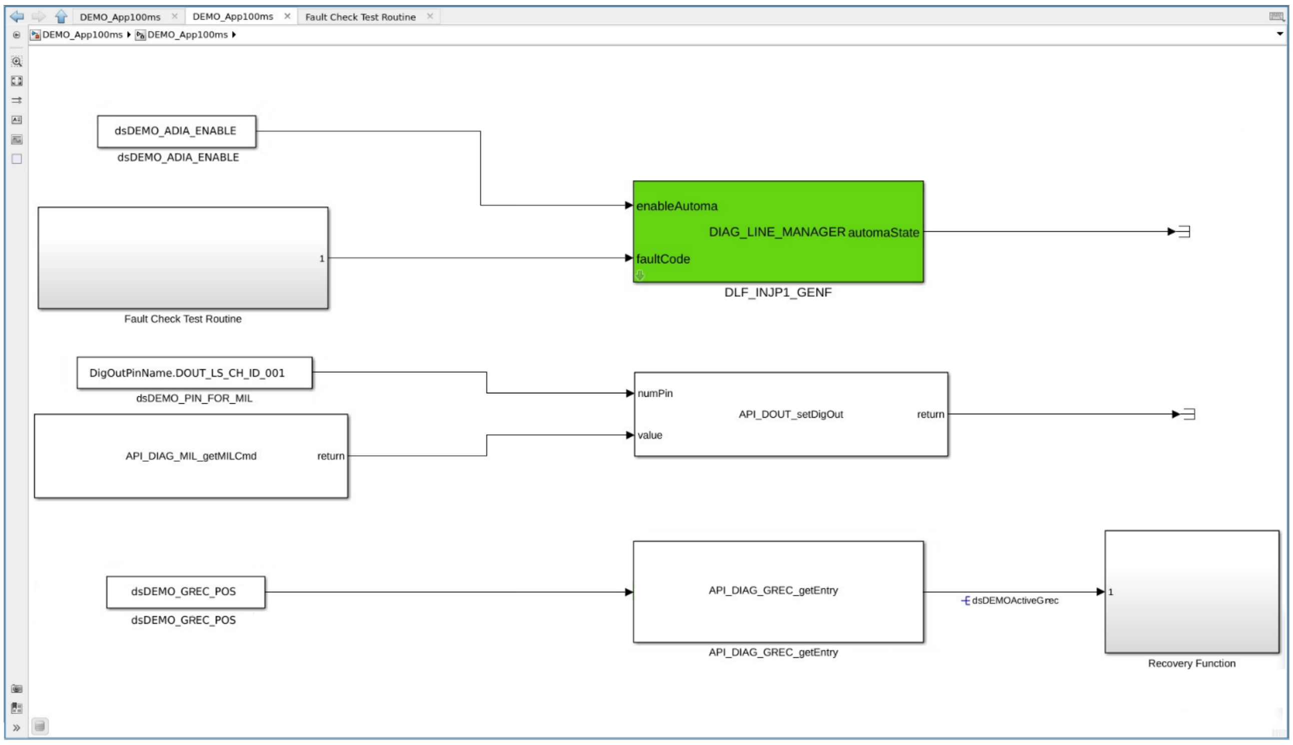


Figure 5.7.7 - DEMO application

The main model was divided into three parts, that can be distinguished in *Figure 5.7.7*, from top to bottom: a first part inherent to the fault, a middle part for the MIL, and the bottom part used for the recovery function.

The first part included both the fault check routine and the diagnosis line block. The routine was simply an API that listened to the PWM pin and returned either a *DIAG_STATUS_OK* message or the detected fault.

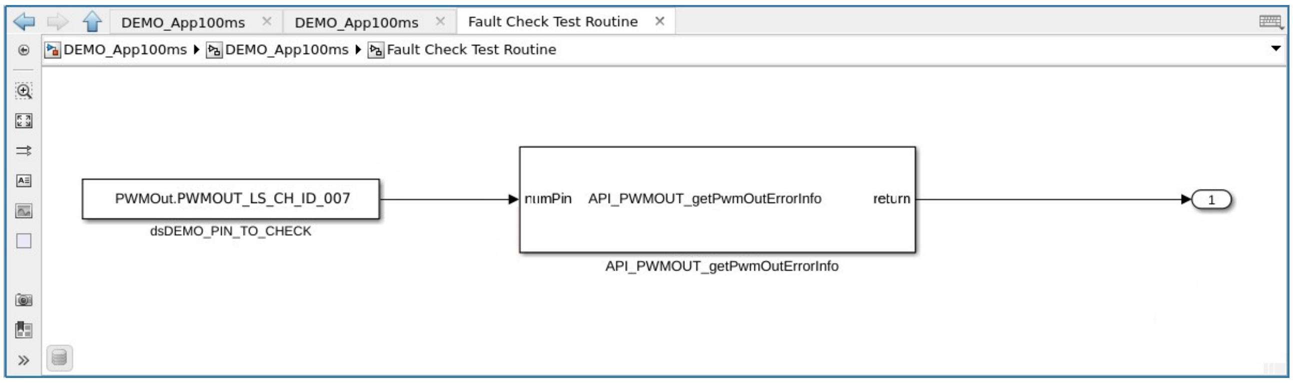

DEMO_App100ms
DEMO_App100ms
Fault Check Test Routine
DEMO_App100ms ▸ DEMO_App100ms ▸ Fault Check Test Routine
PWMOut.PWMOUT_LS_CH_ID_007
dsDEMO_PIN_TO_CHECK
numPin
API_PWMOUT_getPwmOutErrorInfo
return
API_PWMOUT_getPwmOutErrorInfo
1

Figure 5.7.8 - Fault check routine for the DEMO application

The middle section made use of the MIL APIs to read the status of the system's variable associated with the lamp being on or off. The read value was then fed to Metatron's **API_DOUT_setDigOut** block, to pilot a digital output pin connected to a LED. This way, the system simulated an actual indicator lamp, giving visual feedback to the tester.

The last part operated on the recovery lines; once the open load fault reached the *Confirmed and active* status, its associated recovery lines would activate. The recovery line in the position defined by the calibration **dsDEMO_GREC_POS** was used to pilot the 'Recovery Function' subsystem. Inside of it, the recovery line's value would be used to pilot another digital output, and the value could be checked either on CANape by reading the associated signal or by connecting to the digital output the LED that had been disconnected from the PWM pin to create the fault.

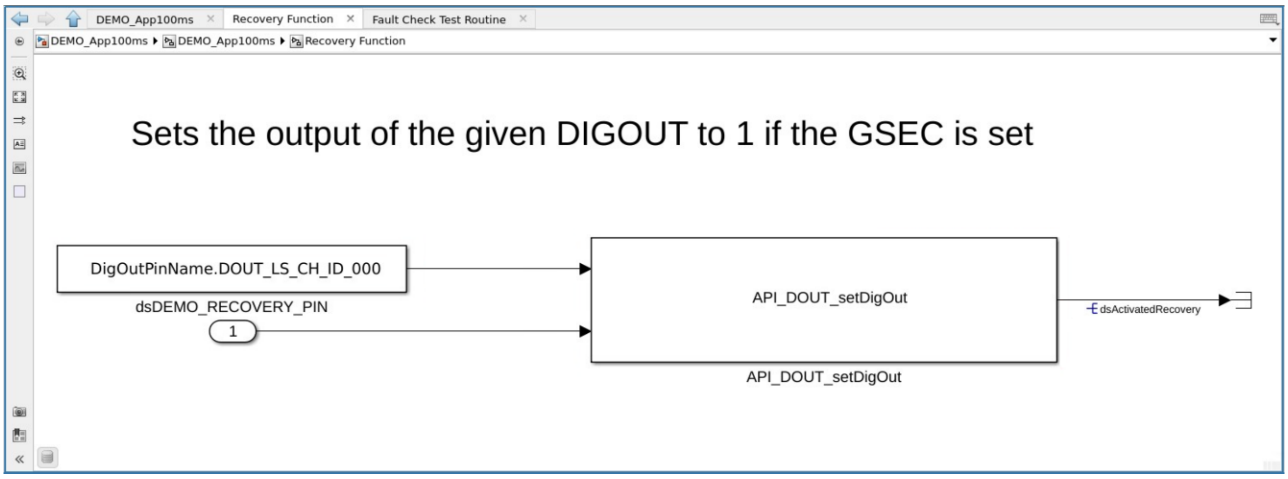

DEMO_App100ms
Recovery Function
Fault Check Test Routine
DEMO_App100ms ▸ DEMO_App100ms ▸ Recovery Function
Sets the output of the given DIGOUT to 1 if the GSEC is set
DigOutPinName.DOUT_LS_CH_ID_000
dsDEMO_RECOVERY_PIN
1
API_DOUT_setDigOut
API_DOUT_setDigOut
dsActivatedRecovery

Figure 5.7.9 - Recovery function for the DEMO application

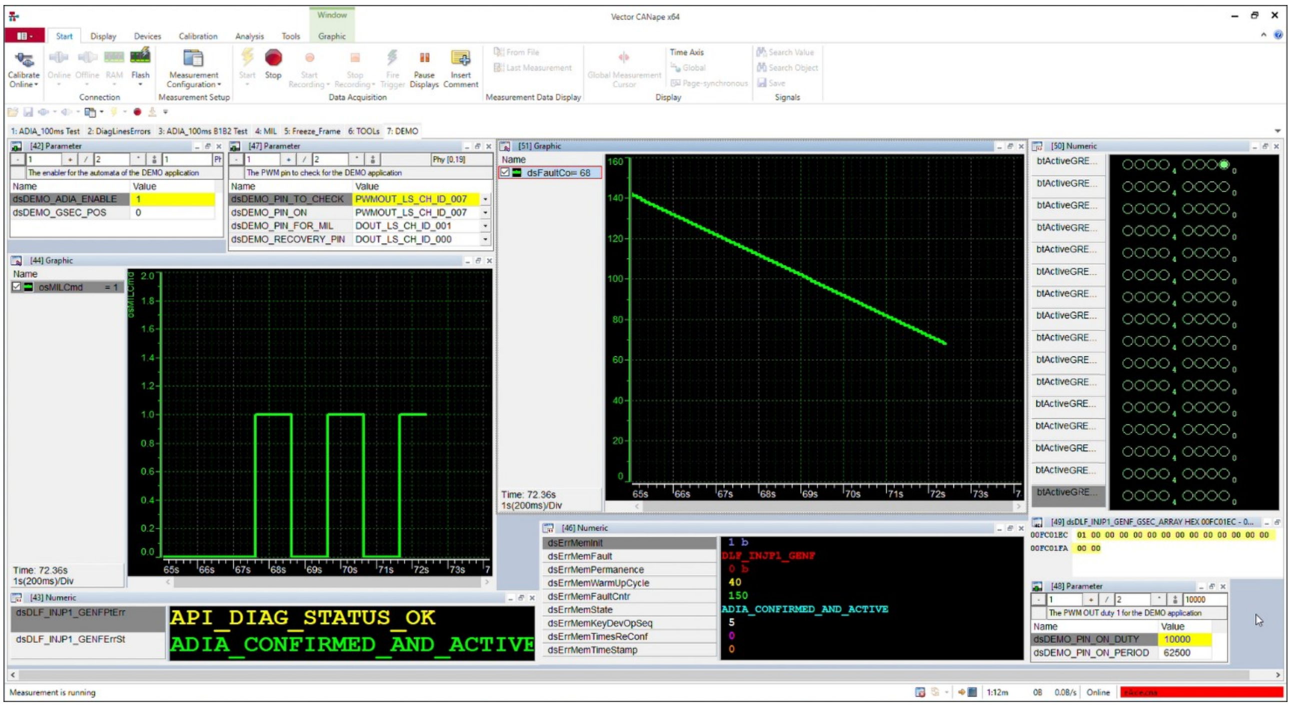
5.7.3. CANape visualisation and results

All the tests were conducted with the use of the CANape tool. This program not only allowed to watch the system variables values in real time, but thanks to the possibility of changing the calibrations at runtime, it also made it possible to efficiently trigger the various cases, e.g., it provided a way to modify the class of a fault without the need for rebuilding the code from the model, and it made it possible to shorten the required amount of hours to trigger certain events.

For each of the models/diagnosis flow parts, a CANape tab was created to put together all the calibrations and signal readings for a certain component of the solution in order to facilitate the study of the behaviour of each piece.

Thanks to the graphical visualisation of the variables and their history, CANape allowed for a better understanding of the system response and timings.

For example, it was particularly useful to visualise the MIL patterns; although the 'Demo' application included a LED that operated as the MIL, it was somewhat difficult to precisely discern the timings of the LED's blinking on the naked eye; having it reproduced on a timed graph on CANape facilitated the analysis. Moreover, the possibility provided by CANape to record the data and reproduce them later came in handy, i.e., to ensure that the required timings were being respected without having to focus solely on that during the tests.



[42] Parameter

The enabler for the automata of the DEMO application

Name	Value
dsDEMO_ADIA_ENABLE	1
dsDEMO_GSEC_POS	0

[47] Parameter

Phy [0,19]

The PWM pin to check for the DEMO application

Name	Value
dsDEMO_PIN_TO_CHECK	PWMOUT_LS_CH_ID_007
dsDEMO_PIN_ON	PWMOUT_LS_CH_ID_007
dsDEMO_PIN_FOR_MIL	DOUT_LS_CH_ID_001
dsDEMO_RECOVERY_PIN	DOUT_LS_CH_ID_000

Time: 72.36s
1s(200ms)/Div

[43] Numeric

dsDLF_INJP1_GENFPtErr	API_DIAG_STATUS_OK
dsDLF_INJP1_GENFErrSt	ADIA_CONFIRMED_AND_ACTIVE

[46] Numeric

dsErrMemInit	1 b
dsErrMemFault	DLF_INJP1_GENF
dsErrMemPermanence	0 b
dsErrMemWarmUpCycle	40
dsErrMemFaultCntr	150
dsErrMemState	ADIA_CONFIRMED_AND_ACTIVE
dsErrMemKeyDevOpSeq	5
dsErrMemTimesReConf	0
dsErrMemTimeStamp	0

[49] dsDLF_INJP1_GENF_GSEC_ARRAY HEX 00FC01EC - 0...

```
00FC01EC  01 00 00 00 00 00 00 00 00 00 00 00 00 00
00FC01FA  00 00
```

[48] Parameter

The PWM OUT duty 1 for the DEMO application

Name	Value
dsDEMO_PIN_ON_DUTY	10000
dsDEMO_PIN_ON_PERIOD	62500

Measurement is running   1:12m   0B   0.0B/s   Online

Figure 5.7.10 - CANape screen of an ongoing test, with the Demo tab open

Another instance in which CANape's graphical representation came in handy was to better visualise the active recovery lines; when the number of active faults with different associated lines grew, so did the difficulty to read the values. CANape's visual representation of the recovery table's rows in binary format (the rows of empty circles on the right in *Figure 5.7.10*) helped greatly at this juncture.

For what concerns the results obtained from these models, the designed strategy appears to be solid, and the implemented system respects the required behaviour imposed by the standards and the regulations.

The applications, although quite simple, allowed to check all the functionalities that had been implemented. Even so, the simple study of the system behaviour through these applications is far from the intensive testing that the found solution would require, like the one seen in the first part of this thesis.

To support the testing part, the ADIA and recovery lines management code has been integrated into a pre-existing project in the hope of verifying its functioning in a more work-intensive environment. So far, after some tweaking required to correctly integrate the code with a different version of the BSWL, the system seems to work as intended.

6. Conclusion

During this thesis work, several key milestones were achieved, laying a solid foundation for future projects.

First, a major accomplishment was the simplification of the system architecture and the fault's diagnosis flow. By refining these aspects, we ensured a more streamlined and efficient system that can be implemented with fewer complications, removing the need for external tools to perform the system's setup. This resulted in a reduction in the possible error points and overall setup times, which contributes to better performance and a more reliable platform.

Moreover, modularity was a core design objective. The pre-existing code has been almost completely redone from scratch, with the idea of improving the separation of concerns among the various components. By increasing the modularity of the system, we are not only catering to the current needs but also creating the possibility for future implementations across different ECUs, applications, and newer standards.

Another significant improvement was the introduction of more user-friendly interfaces through Simulink masks, offering users a simpler and more intuitive way to interact with the system. This eliminated the need for the tedious and error-prone process of manually editing external files, compiling them, and loading them into the system. This enhancement is expected to improve usability and reduce the complexity of system operation, with a consequent reduction of both error points and required time.

Of course, a key aspect of this work was ensuring that the system adheres to modern OBD standards. By guaranteeing conformity and compliance, we position the system for broader adoption across the heavy-duty automotive industry that relies on stringent adherence to the diagnostic standards for fault detection and reporting.

The final result has allowed the company to evaluate the impacts of a potential modification to its diagnostic set. There are, however, some areas requiring further development:

- The protocol integration with the Basic Software Level remains to be completed. This is a critical step toward ensuring seamless communication between different system layers and enhancing the system's overall functionality.

- Additionally, finding a suitable pilot project to fully test the system's capabilities in a real-world environment is a priority. Although part of the system has already been exported to a pre-existing project, a pilot project would provide invaluable insights, validate the system's performance, and uncover potential areas for further optimisation.

If integration is pursued, additional work will be required to incorporate these modules. This will also involve a thorough review to ensure regulatory compliance by removing configurable features that were originally added to simplify, automate, and speed up the testing process.

In summary, this thesis has successfully covered all the key points originally proposed to update and simplify the memory management and the OBD strategy of Metatron's system. Although some areas could still use some development, the work done thus far sets a strong base for future enhancements and implementations.

Acknowledgements

Ai miei genitori, che mi hanno sempre supportato.
Alla mia famiglia e i miei amici.
A Ettore, che mi è sempre stato ad ascoltare in questi anni di studio.
Grazie.

Bibliography

- *"Library definition for an automotive ECU API layer (using Model Based approach)"*, by L. Zannella
- "ECU design flow - Metatron experience", Metatron S.p.A.
- *https://www.metatron.it/it/prodotti/ch4-natural-gas-vehicle/ecus/hds*
- *https://mathworks.com/products/matlab.html*
- *https://mathworks.com/products/simulink.html*
- *https://www.ni.com/en.html*
- *https://www.vector.com/int/en/products/products-a-z/software/canape*
- *"Introduction to model-based software design"* by M. Violante
- *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://www.autopi.io/blog/what-is-wwh-obd/*
- *"ISO 27145-4: Road vehicles — Implementation of World-Wide Harmonized On-Board Diagnostics (WWH-OBD) - Communication requirements — Part 4"*, ISO
- *"J1939-21 Revised APR2015"*, SAE International
- *"GB 17691-2018 Annex F (English translation)"*
- *"Regulation 49 Revision 7 08/07/2015"*, UNECE