Politecnico di Torino

Master’s Degree in Computer Engineering

AY 2023/2024

???Sessione di Laurea Mese Anno

**Library definition for an automotive ECU API layer**

**using Model Based approach**

Memory Management & OBD

|  |  |
| --- | --- |
| Advisor: | Candidate: |
| Prof. Violante Massimo | Oliva Mattia |

|  |
| --- |
| Internship Tutor: |
| Ing. Bertrand Emilio |

Summary

[1. Introduction 5](#_Toc173774068)

[1.1. Company Overview 5](#_Toc173774069)

[1.2. Thesis Goals 6](#_Toc173774070)

[1.3. Working Environment 8](#_Toc173774071)

[1.3.1. HDS9 8](#_Toc173774072)

[1.3.2. MATLAB & Simulink 8](#_Toc173774073)

[1.3.3. LabVIEW 9](#_Toc173774074)

[1.3.4. CANape 10](#_Toc173774075)

[1.4. Model Based Design 12](#_Toc173774076)

[1.4.1. General Overview 12](#_Toc173774077)

[1.4.2. MBD Flow V-Diagram 13](#_Toc173774078)

[2. Hardware Architecture 18](#_Toc173774079)

[2.1. Inputs 19](#_Toc173774080)

[2.2. Outputs 19](#_Toc173774081)

[2.3. Microcontroller 19](#_Toc173774082)

[2.4. Communication 20](#_Toc173774083)

[3. Software Architecture 22](#_Toc173774084)

[3.1. AUTOSAR Principles 22](#_Toc173774085)

[3.2. Architecture Levels 23](#_Toc173774086)

[3.2.1. Application Layer (MBSL) 24](#_Toc173774087)

[3.2.2. Intermediate Layer (API, ASWL, DSWL) 25](#_Toc173774088)

[3.2.3. Basic Software (BSWL) 26](#_Toc173774089)

[3.3. API Design 27](#_Toc173774090)

[4. Part 1 - Memory Management (NVRAM) 29](#_Toc173774091)

[4.1. Strategy 29](#_Toc173774092)

[4.2. Implementation 32](#_Toc173774093)

[4.2.1. Memory Structure 32](#_Toc173774094)

[4.2.2. Read & Write Operations 34](#_Toc173774095)

[4.2.3. Startup 35](#_Toc173774096)

[4.2.4. Shutdown 35](#_Toc173774097)

[4.3. LabVIEW 35](#_Toc173774098)

[4.4. Test #1 35](#_Toc173774099)

[4.5. Test #2 35](#_Toc173774100)

[4.6. Results 35](#_Toc173774101)

[5. Part 2 – Diagnostics (OBD) 35](#_Toc173774102)

[5.1. … 36](#_Toc173774103)

[5.2. Standards 36](#_Toc173774104)

[5.3. … 36](#_Toc173774105)

# 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.

## 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.



Figure 1.1 - Metatron Offices

## 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.

## 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.

### 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.

Immagine che contiene elettronica, Componente elettrico, Componente di circuito, Componente di circuito passivo

Descrizione generata automaticamente

Figure 1.3.1 - HDS9

Thanks to its hardware specifications (described in more details in the “[Hardware Architecture](#_Hardware_Architecture)” chapter) and performances, this ECU has been an ideal platform to test and validate this thesis’ work.

### 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.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

Figure 1.3.2 - MATLAB and Simulink sample screen

### 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.

Immagine che contiene testo, software, Software multimediale, Icona del computer

Descrizione generata automaticamente

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

### 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.

Immagine che contiene testo, schermata, Software per la grafica, Software multimediale

Descrizione generata automaticamente

Figure 1.3.4 - ECU calibration with CANape

## Model Based Design

### 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:

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

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.

### 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:

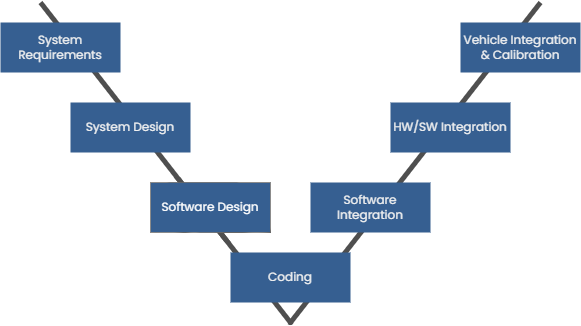


Figure 1.4.2 - V Diagram

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.

1. **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.

1. **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).

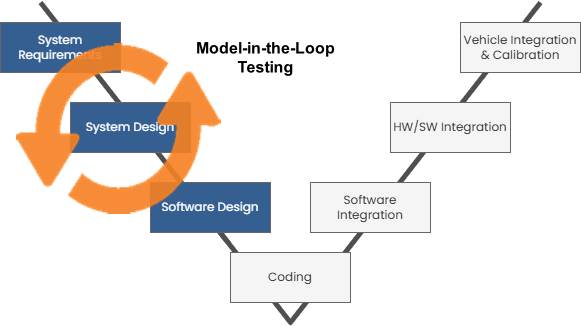


Figure 1.4.3 - MIL Testing

1. **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.

1. **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.

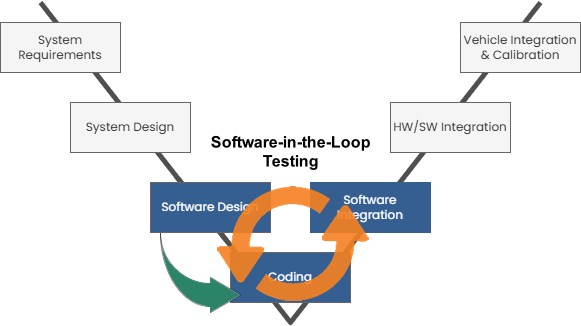


Figure 1.4.4 - SIL Testing

1. **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.

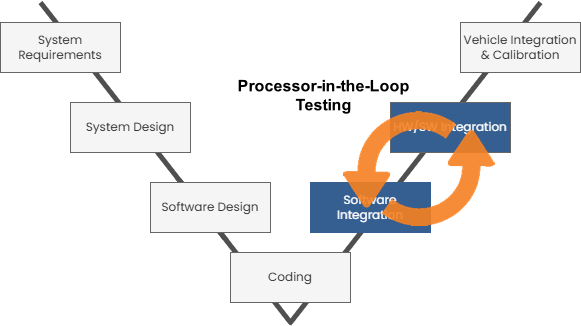


Figure 1.4.5 - PIL Testing

1. **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.

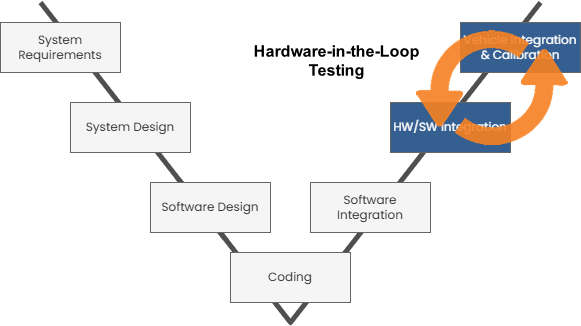


Figure 1.4.6 - HIL

# 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:

Immagine che contiene testo, schermata, Carattere, Parallelo

Descrizione generata automaticamente

Figure 2 - ECU block diagram

## 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.

## 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.

## 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

Immagine che contiene testo, schermata, diagramma, numero

Descrizione generata automaticamente

Figure 2.3 - Microcontroller schematics

## 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.

Immagine che contiene connettore, Alimentazione elettrica, cavo, interno

Descrizione generata automaticamente

Figure 2.4 - HDS9 CAN Connectors

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

# 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.

## 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.

Immagine che contiene testo, schermata, Carattere, Marchio

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

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.

## 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.

Immagine che contiene testo, schermata, grafica, diagramma

Descrizione generata automaticamente

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:

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamente

Figure 4.2.2 - HDS9 SW architecture layers

### 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.

### 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.

### 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, a sit 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.

Immagine che contiene testo, schermata, Carattere, Rettangolo

Descrizione generata automaticamente

Figure 3.2.3 - HDS9 BSWL internal modules and layers subdivision

## 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.

# 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 (NVRAM) 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 would have contributed to 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.

## 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.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

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.

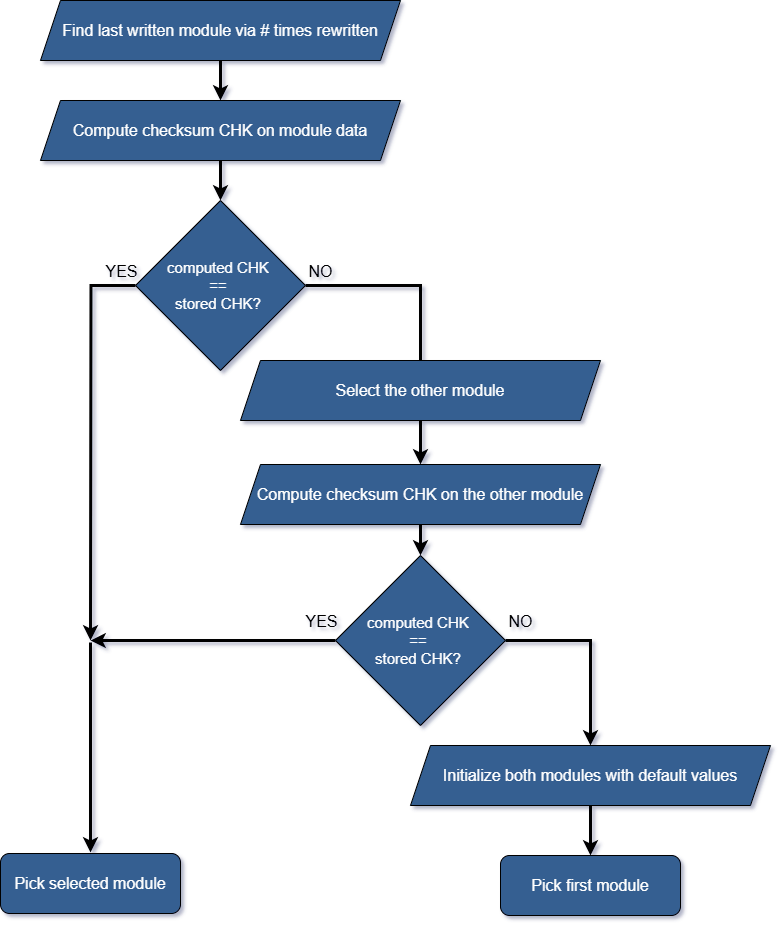


Figure 4.1.2 – NVRAM startup strategy

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 exploits 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, opening up both potential issues and opportunities.

## 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](#_Software_Architecture)” chapter of this thesis.

### 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.

Immagine che contiene schermata, testo

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

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.

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:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

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.

### 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 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.

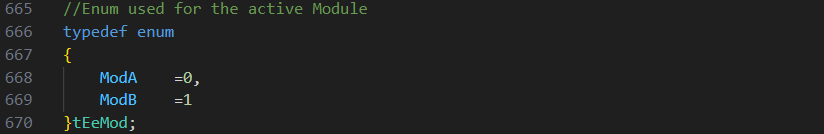


Figure 4.2.4 - NVRAM modules enumerative tEeMod



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**. As stated in the “strategy” section, 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 signal 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.

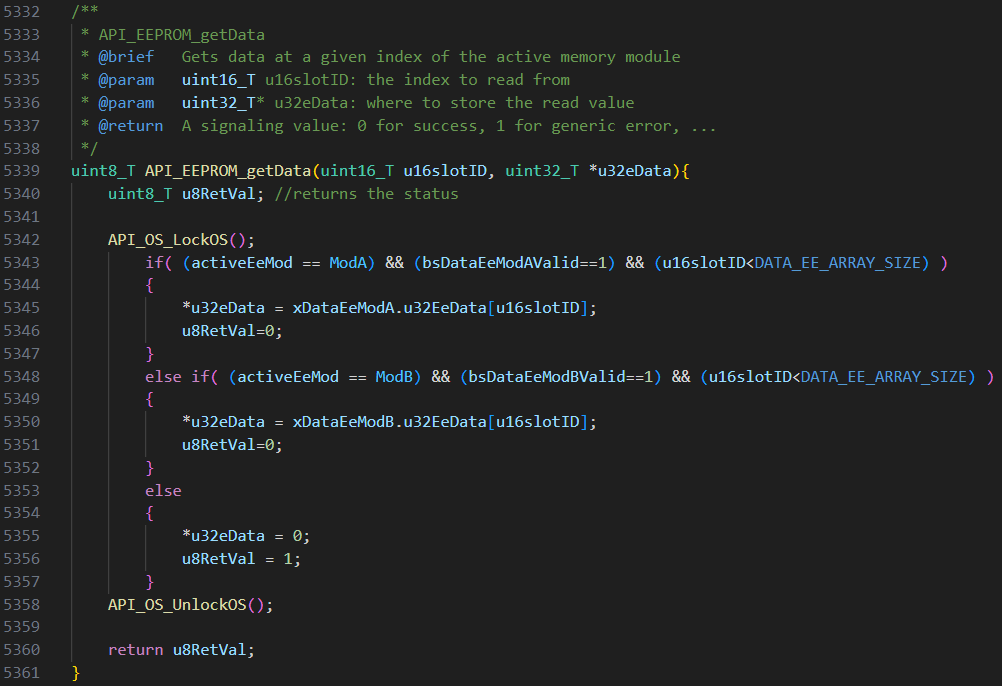


Figure 4.2.6 - API\_EEPROM\_getData

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.

Immagine che contiene testo, schermata, software

Descrizione generata automaticamente

Figure 4.2.7 - API\_EEPROM\_setData

As clearly shown by the snippets and what was reported, 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.

For ulterior improvements that have been adopted for the second part of this thesis, please refer to the “[Results](https://quillbot.com/grammar-check#_Results)” section of this chapter.

### Startup

???

* Show the default value for the struct
* MOSTRA IL CODICE CORRETTO con la copia d ai moduli in .eeprom alla memoria RAM

???

### Shutdown

???

* MOSTRA IL CODICE CORRETTO con la copia dalla memoria RAM ai moduli in .eeprom

???

## LabVIEW

## Test #1

## Test #2

## Results

??? migliorie: copia in RAM perl e modifiche delle entry in NVRAM, inidicizzazione tramite enumerativi 🡪 sono tutte cose che usiamo nella parte 2???

# Part 2 – Diagnostics (OBD)

## …

## Standards

## …