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 4](#_Toc171774582)

[1.1. Company Overview 4](#_Toc171774583)

[1.2. Thesis Goals 5](#_Toc171774584)

[1.3. Working Environment 7](#_Toc171774585)

[1.3.1. HDS9 7](#_Toc171774586)

[1.3.2. MATLAB & Simulink 7](#_Toc171774587)

[1.3.3. LabVIEW 8](#_Toc171774588)

[1.3.4. CANape 9](#_Toc171774589)

[1.4. Model Based Design 11](#_Toc171774590)

[1.4.1. General Overview 11](#_Toc171774591)

[1.4.2. MBD Flow V-Diagram 12](#_Toc171774592)

[2. Hardware Architecture 18](#_Toc171774593)

[2.1. Inputs 19](#_Toc171774594)

[2.2. Outputs 19](#_Toc171774595)

[2.3. Microcontroller 19](#_Toc171774596)

[2.4. Communication 20](#_Toc171774597)

[3. Software Architecture 26](#_Toc171774598)

[3.1. Architecture Levels 26](#_Toc171774599)

[3.2. API Design 26](#_Toc171774600)

[4. Part 1 - Memory Management (NVRAM) 26](#_Toc171774601)

[4.1. … 26](#_Toc171774602)

[5. Part 2 – Diagnostics (OBD) 26](#_Toc171774603)

[5.1. … 26](#_Toc171774604)

[5.2. Standards 26](#_Toc171774605)

[5.3. … 26](#_Toc171774606)

# 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 FIAT Auto’s GPL systems. Moreover, since 2010, China has been the main market for Metatron’s pressure regulator, for its production of heavy-duty engines. 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 amount 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 dragging behind parts that aren't needed anymore. 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 process to replace the existing one, overcomplicated by years of evolving standards and requirements, in anticipation of future implementations and porting of the OBD system on different boards.

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 and implement a strategy to handle fault detection and management. 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:

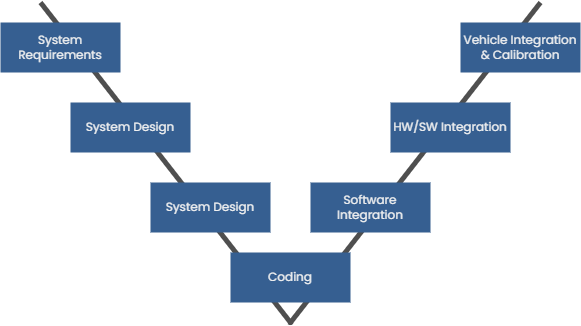


Figure 1.4.2 - V Diagram

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).

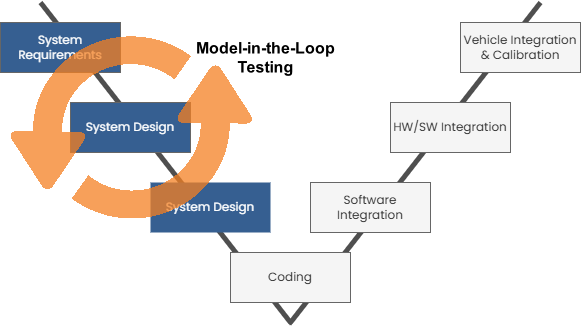
****

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.

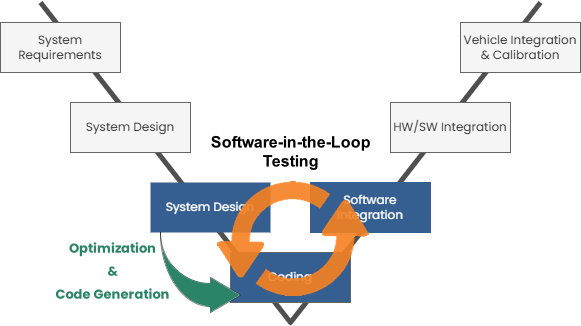


Figure 1.4.4 - SIL Testing

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.

Immagine che contiene testo, schermata, Carattere, Elementi grafici

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, Carattere, grafica

Descrizione generata automaticamente

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

As CAN communication has been fundamental to this thesis’ work, here is a short summary of the protocol:

1. **Working Principle**
2. CAN is a broadcast protocol that transmits data via a two-wired bus. The wires, called CAN Low and CAN High, operate on different voltage ranges and are both terminated by a 120Ω resistor. CAN High ranges from 2.5V up to 3.75V, while the other wire stays between 1.25V and 2.5V. A logical 1 is obtained when both wires’ voltage equals 2.5V, in which case the signal is called “Recessive”. When the two voltages are at the opposite ends of the ranges, respectively, 3.75V and 1.25V, the signal is said to be “Dominant”, and we obtain a logical 0.
3. The standard CAN bus, High-Speed CAN, operates with a baud rate of up to 1 Mbit/s, but some variants, such as the CAN Flexible Data-Rate (CAN FD) and CAN XL can reach baud rates multiple times higher, allowing for faster management of larger data sizes.
4. Immagine che contiene testo, diagramma, schermata, linea

   Descrizione generata automaticamente

Figure 2.4.1 - CAN Signal voltage levels

1. **CAN Frame**

A CAN network can be configured to use two message (or frame) formats: the standard or **base frame format** (defined in CAN 2.0 A and CAN 2.0 B), and the **extended frame format** (specified only in CAN 2.0 B).

The main difference between the two formats is that the CAN base frame provides an identifier length of 11 bits, whilst the CAN extended frame supports an identifier length of 29 bits, consisting of an 11-bit identifier (base identifier) plus an 18-bit extension (identifier extension).

The selected format can be distinguished via the IDE bit, transmitted as dominant (logic 0) in case of base format, and recessive when the frame is using the extended format.

All CAN controllers supporting the extended format must be able to send and receive in base format.

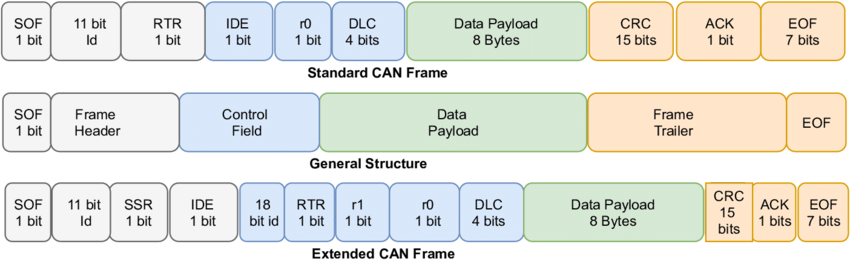
1. ****

Figure 2.4.2 - CAN message frame

The main fields of the CAN frame are:

* **SOF** (start of frame): marks the start of the frame and is a 'dominant' 0.
* **ID**: The message identifier, with 11 bits for standard frames and 29 bits for extended frames.
* **RTR** (remote transmission request): indicates whether a node is requesting a frame from other nodes or sending new data.
* **Control**: includes the IDE (Identifier Extension Bit), which is '0' for the standard frame, and the Data Length Code (DLC), which specifies the length of the data in the message (up to 8 bytes).
* **Data**: the payload whose length is stated in the DLC.
* **CRC** (cyclic redundancy check): used for error detection.
* **ACK** (acknowledgement): signals that the node correctly received the data.
* **EOF** (end of frame): marks the end of the frame.

Aside from the format, CAN frames are also differentiated by type, which can be one out of four:

* Data frame, containing the actual data for transmission.
* Remote frame, used by a destination node to request data from the source.
* Error frame, transmitted by a node that has detected an error.
* Overload frame, used for synchronisation purposes in order to let slow consumers catch up with fast producers by introducing a delay between data or remote frames.

To help with the interpretation and writing of CAN frames, a specific type of file, called Database CAN (DBC), can be used. DBC files are text files containing information to decode raw CAN bus data into human-readable values, acting as a sort of “translation file”.

A DBC file structure is made up by different parts:

* **Version**, which specifies the version of the DBC file.
* **New Symbols**, where new symbols used in the file are defined.
* **Bit Timing**, which contains bit timing information.
* A **Nodes** part, listing the Electronic Control Units (ECUs) or nodes.
* A “**messages**” section, in which the CAN messages and their related information are defined, including ID, name, data length, and producer ECU.
* A “**signals**” section, defining the individual data points within a message, including the signal name, start bit, bit length, byte order, data type, conversion factor, and unit.

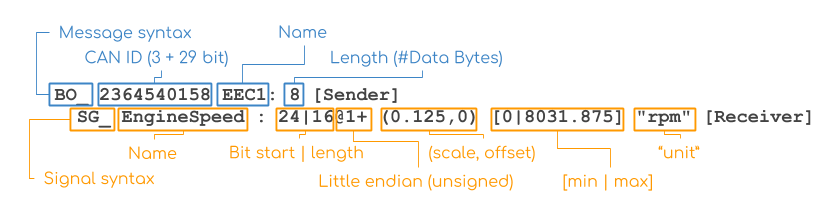
1. ****

Figure 2.4.3 - CAN message definition in a DBC file

DBC files standardise the way CAN data is communicated and interpreted across different devices and manufacturers, functioning as a library of signals and detailing how each piece of data should be interpreted, allowing software tools to convert this data into physical values.

**Advantages**

The use of the CAN protocol brings along many advantages, in particular:

1. The broadcast paradigm followed by the protocol makes it possible to greatly reduce the number of wires required for the communication of the various devices, which at the same time means a **reduction in** both weight and **costs**.
2. Thanks to the **single access point** for the data exchange between all the devices, CAN simplifies diagnostic, data logging, and configuration.
3. The physical communication strategy illustrated before makes the signal **robust** against disturbances and interferences, making it an optimal choice for applications that require high levels of safety, like those of the automotive field.
4. Using the frame ID, CAN **messages can be prioritized**, making it possible to define a simplified form of “differentiate service”.

**HDS9 Implementation**

On the HDS9, there are four CAN channels available, each with 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. The XCP protocol works with a master-slave paradigm, where the measurement system (e.g., CANape) assumes the master role and the ECU is the one responding to the memory access requests. This access is address-oriented, 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 2.4.5 - HDS9 CAN Connectors

# 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.

## 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 3.1.1 - 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 picture below:

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamente

Figure 3.1.2 - HDS9 SW architecture layers

### Application Layer (MSBL)

???

### Hand Coded Support Functions (ASWL)

???

### Application Programming Interface (API)

???

### Basic Software (BSWL)

???

Immagine che contiene testo, schermata, Carattere, Rettangolo

Descrizione generata automaticamente

Figure 4 - HDS9 BSWL internal modules

## 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 amount 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.

# Part 1 - Memory Management (NVRAM)

## …

# Part 2 – Diagnostics (OBD)

## …

## Standards

## …