POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science in Mechatronic Engineering

IMPLEMENTATION OF AUTOSAR COMMUNICATION STACK MODULES

Academic Supervisor

Prof. Massimo Violante

Student

Benedetta De Bernardo

252050

October 2019

Acknowledgments

I would like to thank my Academic Tutor, Professor Massimo Violante, for the support and valuable advices. My Company Tutor, Roberto Pozzo, for patience and for guiding me in this thesis activity.

I would like to express my greatest gratitude to my mother who taught me not to be afraid; to my father, he approached me to the world of engineering from an early age in the small things.
They gave me the opportunity to try a new experience that would forever change my life.

Thanks to my sisters and my grandfather, they make me feel at home despite being far away.

Thanks to my boyfriend for helping me overcome my insecurities.

Finally, thanks to my friends of the Politecnico, their presence made the university less heavy.

I would not have achieved this goal without you all.

Thank you.

Abstract

The following paper is the result of a six-month project at the Belgian company Multronic s.r.l in Carmagnola (TO); it designs, manufactures and supplies diesel engine emission aftertreatment systems.

The objective of the thesis activity is to implement a communication network for automotive application with the main reference to the basic elements of AUTOSAR architecture and features.
The idea is to have a code independent of the hardware, easy to reuse and with a good level of abstraction.

This paper shows the implementation of a firmware that can meet the need to have a code as modular as possible, satisfying the functional requirements in a single platform.

The project was characterized by a training phase through the study of the most commonly used automotive communication protocols and by an analysis of the characteristics of the fundamental standards.
In particular, the focus is on development of the Serial network and CAN network.

The fundamental points of the project are followed gradually: *in primis* there is a brief introduction that anticipates the characteristics of the developed SW, the main aspects of the testing phase and the requirements in terms of quality.

Afterwards, there is a presentation of the technical features of the HW used, followed by chapters that show the topics studied to be able to address this thesis activity.
The Chapters 5 and 6 represent the core of the project: the SW is described with its main characteristics and test activity.

The paper ends with a conclusive analysis of the activity proposing possible future developments.

Contents

List of figures

List of tables

List of abbreviations

µC – Microcontroller

ADC – Analog-to-Digital Converter

ASIL – Automotive Safety Integrity Level

AUTOSAR – AUTomotive Open System ARchitecture

BSW – Basic Software

BSWL – Basic Software Layer

CAN – Controller Area Network

CD – Collision Detection

CM – SPN Conversion Method

CMD – Command

CSMA – Carrier-Sense Multiple Access

DM – Diagnostic Message

DPF – Diesel Particulate Filter

EAL – ECU Abstraction Layer

ECT – Enhanced capture Timer

ECU – Electronic Control Unit

E/E – Electric/Electronic

EGR – Exhaust Gas Recirculation

FBC – Fuel Borne Catalyst

FMI – Failure Mode Indictor

FW – Firmware

HIS – Hardware Software Interaction

HW – Hardware

IDE – Integrated Development Environment

IIC – Inter-Integrated Circuit

ISO – International Organization for Standardization

LIN – Local Interconnected Network

MCAL – Microcontroller Abstraction Layer

MISO – Master-In/Slave-Out

MOSI – Master-Out/Slave-In

NOx – Nitrogen Oxides

NRZ – Non- Return to Zero

OC – Occurrence Count

OEM – Original equipment manufacturer

OS – Operating System

OSI – Open System Interconnection

P2P – Peer-to-Peer

PIT – Period Interrupt Timer

PWM – Pulse Width Modulation

RAM – Random-Access Memory

RX – Receive

SAE – Society of Automotive Engineers

SCI – Serial Communications Interface

SCK – Serial Clock

SCMD – Subcommand

SCR – Selective Catalytic Reduction

SDLC – Software Development Life Cycle

SOF – Start Of Frame

SPN – Suspect Parameter Number

SW – Software

SW-C – Software Component

TX – Transmit

VFB – Virtual Functional Bus

CHAPTER 1
INTRODUCTION

The modern era market poses new challenges in the automotive industry every day.
In particular, manufacturers are investing in obtaining integrated systems in vehicles that can be reused and standardized, trying to obtain new platforms that can follow the needs of the OEMs.
Companies require new, easily scalable features at low cost in a short time: naturally, this means increasing the complexity of the code.
Software is often not adaptable to any hardware, for this reason SW developers need to modify the code to overcome the dependency on OEMs and suppliers.

In this scenario, a new generation of software is born thanks to the spread of AUTOSAR standard.

Figure 1.1
Role of AUTOSAR

1.1 Software

In the following chapters, the AUTOSAR architecture will be discussed in more detail, but from the following figure it is already possible to notice the layered structure to which reference has been made to have an independent SW with a good level of abstraction.

We have:

- Basic Software (BSW)
- Runtime Environment Layer (RTE)
- Application Layer

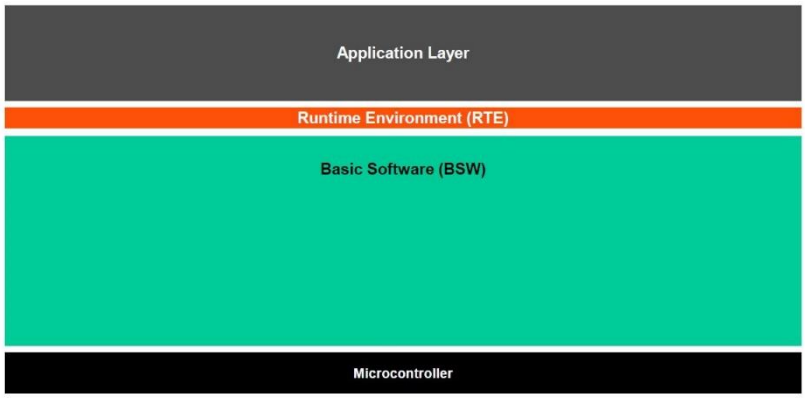


Figure 1.2
SW Layers Overview

In order to provide a uniform interface for the development of communication protocols, it is necessary to start from the levels closer to the microcontroller, and then to continue with those higher ones.

As will be more evident later on, this approach allows the implementation of an easily scalable FW and it gives the possibility to integrate different modules. In particular, this also guarantees a significant simplification in terms of maintenance.

1.2 Quality

The evolution of embedded systems imposes challenges in terms of security: when new features are added to meet the demands of the automotive market, software security must not be neglected. If an update or modification of the code is required, it is necessary to ask what kind of impact it has on the security mechanisms.
Understanding if there are improvements or not in these mechanisms guarantee good code quality.

To guarantee the SW quality, it is fundamental to have a measures against faults in order to have a program that is robust (and it does not break) both in working condition and not.
There are several possible countermeasures, i.e. Unit Test, Integration Test, System Test…
In general, SW faults must be taking into account during the SW development process.

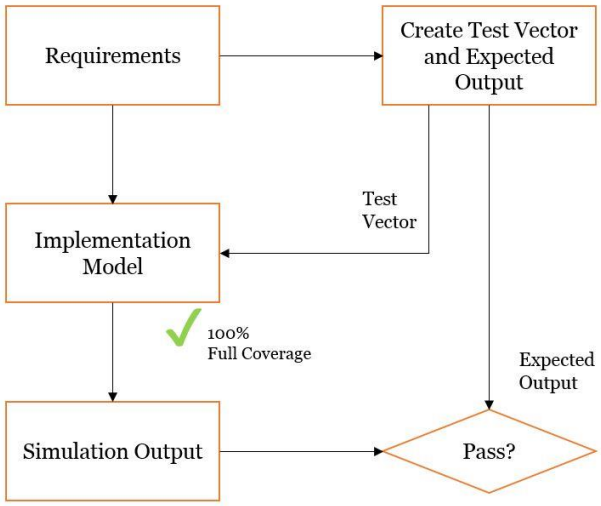


Figure 1.3
Testing

A fundamental standard reference is the ISO 26262, which gives guidelines to measure if it is done a sufficient testing on a single unit.

1.3 Test

The last step in the development of the SW involves a test phase.
Testing makes it possible to evaluate the correct functioning of the code and compliance with the specifications; in the case of this thesis, it allows to validate the interaction of levels within a network.
Nowadays tests are carried out following automated procedures, with the use of specific tools that allow to speed up the validation process.

In the following chapters, the tests performed will be shown on the developed code and the relative results.

CHAPTER 2
TECHNICAL SPECIFICATIONS

The scope of this chapter is to propose a functional description and technical specifications of the hardware used during the thesis activity: it is presented the ECU, its features and the main characteristics of the Microcontroller MC9S12XET256.
In the end, there is an overview of the network.

2.1 DE-TRONIC V3 ECU

The DE-TRONIC V3 ECU is designed for both 12V and 24V automotive applications and it is used in combination with Selective Catalytic Reduction (SCR), Diesel Particulate Filter (DPF), Exhaust gas recirculation (EGR) and Fuel Borne Catalyst (FBC) system.

Figure 2.1
DE-TRONIC V3 ECU External Box

It is able to drive all the loads connected to these modules with a maximum temperature of 85°C.

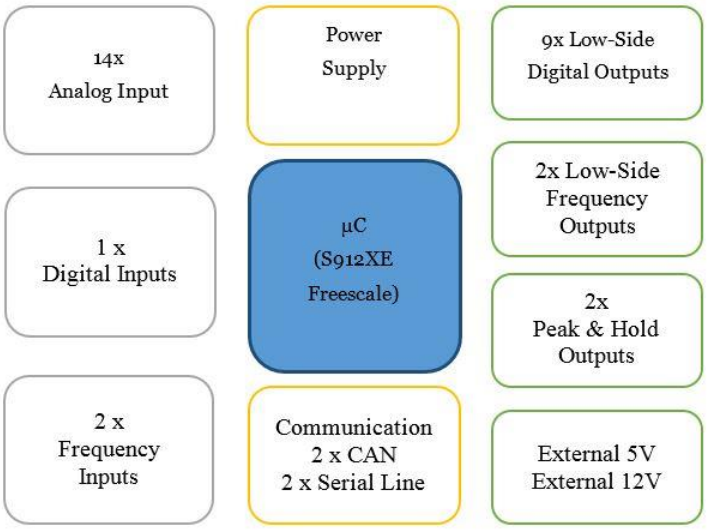


Figure 2.2
DE-TRONIC V3 Block Diagram

In the Figure 2.2 is shown the internal block diagram, the main section are the following:

- Microcontroller MC9S12XET256
- External connector
- Power Supply
- Analog Inputs
- Digital Inputs
- Frequency Inputs
- Low-Side Digital Outputs
- Low-Side Frequency Outputs
- Peak & Hold Outputs
- Communications
- External power supply (sensor, loads)

The general functions with their block diagram are listed below:

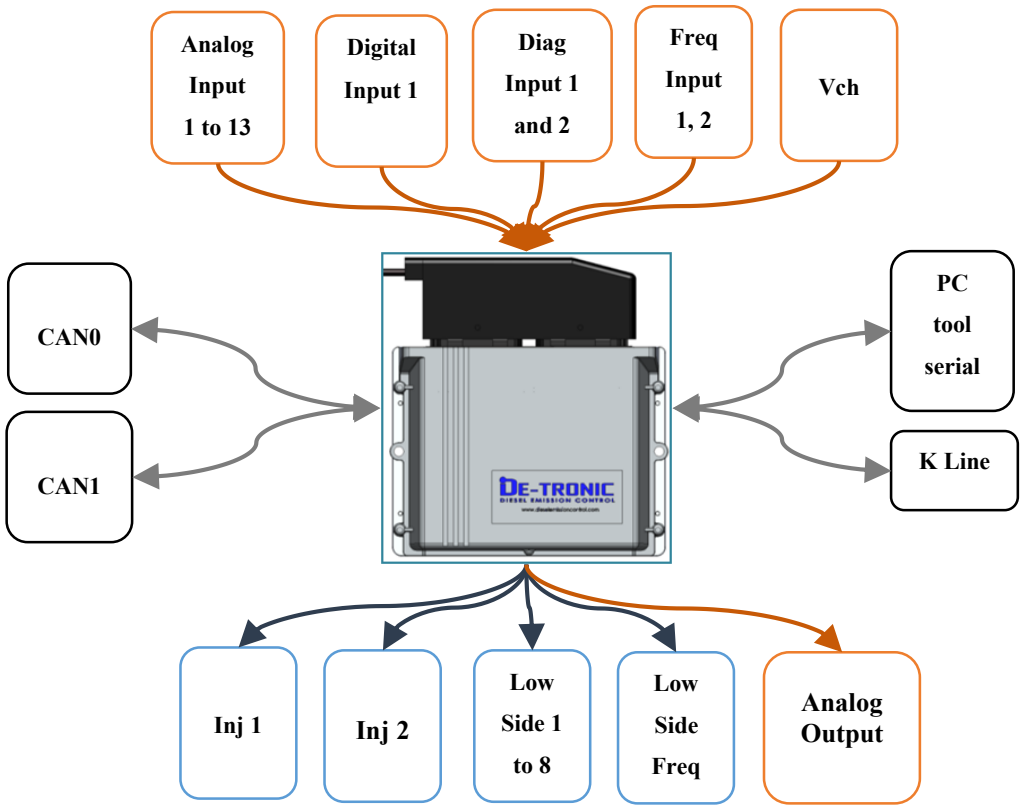


Figure 2.3
DE-TRONIC V3 Functionality

- Functional connections (Power Supply, GND)
- External power supply (5V for sensors)
- Analog Inputs
- Diagnostic analog reading
- Vch (Key contact input)
- Digital Input
- Frequency Input
- Actuator Low Side (Urea Heating, Urea Pressure Pump…)
- Peak and Hold Output (Urea injector, Diesel Injector)
- External Communications (CAN0,CAN1…)
- DAC output
- Internal clock with backup battery power supply
- Internal 125 Mbit flash memory

As regards the communication section, the ECU is equipped with a CAN0 line, CAN1 line, PC tool serial line, serial line and K line.

2.1.1 MC9S12XET256 µC

The microcontroller used for this thesis activity is the MC9S12XET256 produced by Freescale Semiconductor Inc.
It belongs to the MC9S12XE-Family of microcontrollers that are characterized by standard on-chip peripherals[1], including:

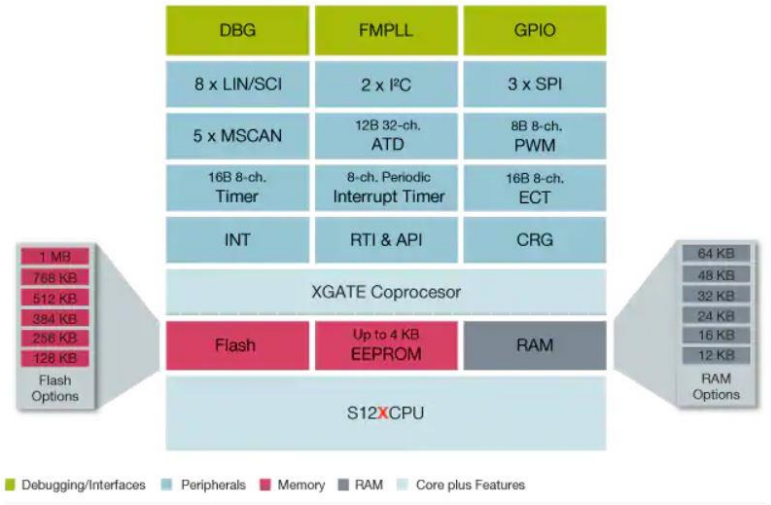


Figure 2.4
S12XE Microcontroller Block Diagram

- up to 64Kbytes of RAM
- 8 asynchronous SCIs
- 3 SPI
- 8-channel IC/OC ECT
- Two 16-channels
- 12-bit ADC
- 8 Channel PWM
- 5 CAN 2.0A

[1] MC9S12XEP100RMV1 – NXP , Datasheet

- software compatible modules (MSCAN12)
- 2 inter-IC bus blocks (IIC)
- 8-channel 24-bit periodic interrupt timer (PIT)
- 8-channel 16-bit standard timer module (TIM)

2.2 Network and IDE

The ECU described above is powered at 12V through a Programmable DC Laboratory Power Supply. To enable the communication between the PC and the device is used a PEMicro USB Multilink Debugger.

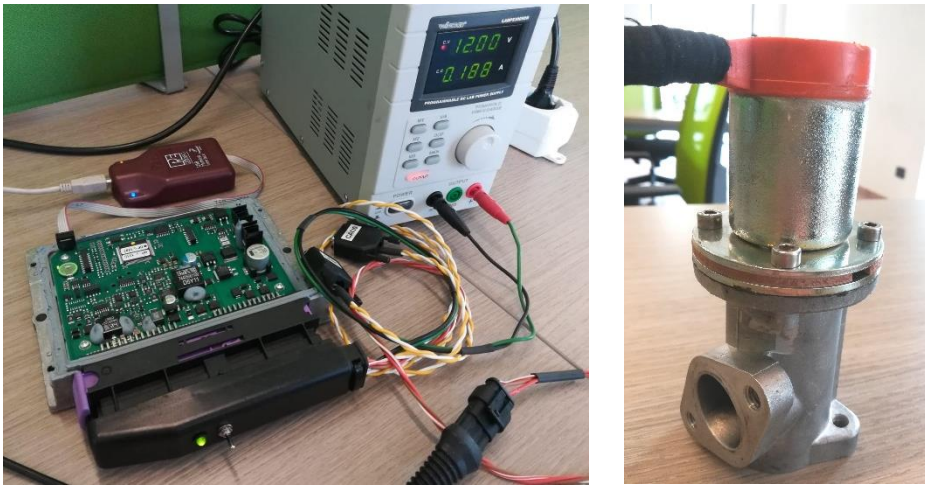


Figure 2.5
ECU, PEMicro Probe, Instrumentation (on the left),
EGR valve (on the right)

In addition, an EGR valve and a PowerView 101 display have been associated to verify the correct operation of the CAN communication networks.
This type of display offers the possibility to check engine parameters and more than 50 SAE J1939 parameters, giving a text description of fault condition.

The SW has been developed in C using CodeWarrior® IDE by NXP Semiconductors for editing, compiling and debugging.
In order to proceed with the test of the serial communication, it was used Hercules

Setup Utility (to send messages and verify the correctness of the reply); in the CAN case, P-CAN view allowed to view, transmit and record CAN data traffic.

CHAPTER 3
BACKGROUND

This section of the paper is dedicated to the description of the fundamental concepts on which the thesis activity is based.
The description of the AUTOSAR standard, the features of the ISO26262 standard are shown.

As it can be noticed, in the following paragraph is about "AUTOSAR light": during the development of the thesis activity, the goal was to obtain a code as independent as possible from the hardware, therefore close to the idea of the standard without going into its details.
The fundamental concepts studied during the in-company training phase will be presented.

3.1 AUTOSAR light

The AUTomotive Open System ARchitecture (AUTOSAR) consortium was founded in 2003 by an agreement among the largest protagonists of the automotive scenario, i.e. Bosch, BMW, Continental, Chrysler, Daimler, Siemens VDO and Volkswagen that represent the “Core Partners”.

Over time, the consortium has more than 100 members contributing to the growth of a new automotive programming paradigm.

The proposed architecture is one of the milestones of modern automotive programming. The idea with which AUTOSAR was born is to have a standardized methodology of development that allows to have software modules independent from the hardware. This is possible by having unified interfaces, which allow the integration of new software components over the lifetime of the vehicle with great simplicity (as long as they conform to the standard).

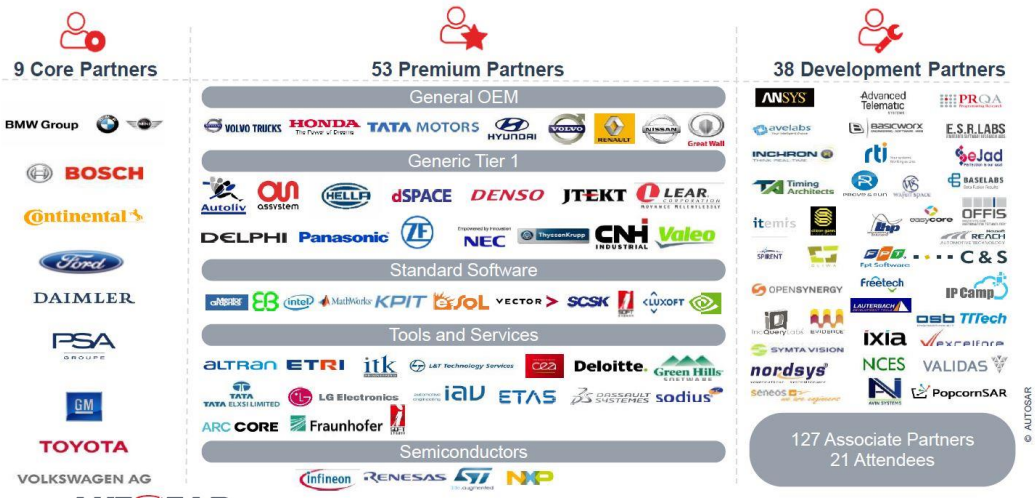


Figure 3.1
Core Partners and Partners

The advent of AUTOSAR has produced a great improvement in terms of software quality, costs and time.

The top-goal of the consortium are:

- Improve portability
- Standardize of basic SW features of ECUs
- Keeping an open architecture
- Composability

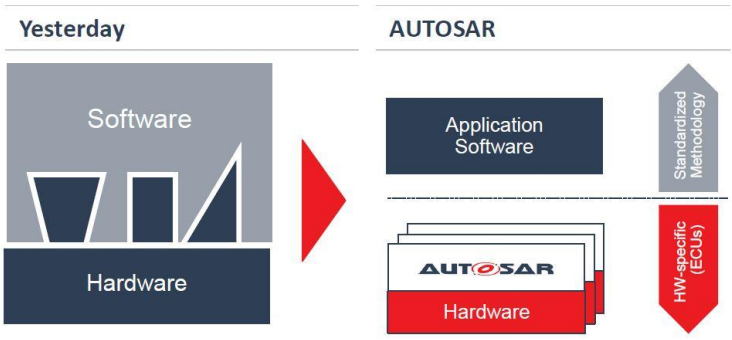


Figure 3.2

SW comparison yesterday and today

3.1.1 Layered Software Architecture

What characterizes AUTOSAR is its modular and layered structure, which involves the use of standardized interfaces in such a way as to allow communication between the various components (the Figure 2.3 gives a coarse view of the layered architecture). As anticipated in the introduction, the main layers are Application Layer, Runtime Environment (RTE), Basic Software (BSW) and then the Microcontroller.

The BSW is in turn divided in

- Services Layer
- ECU Abstraction Layer
- Complex Drivers
- Microcontroller Abstraction Layer

The Microcontroller Abstraction Layer is identified as MCAL and represents the lowest level with the task of mapping the ECU peripherals; it contains all the services and modules with direct access to the µC, i.e. Microcontroller Drivers(e.g. MCU), Memory Drivers(e.g. RAM Test), Communication Drivers (e.g. CAN Driver) and I/O Drivers.

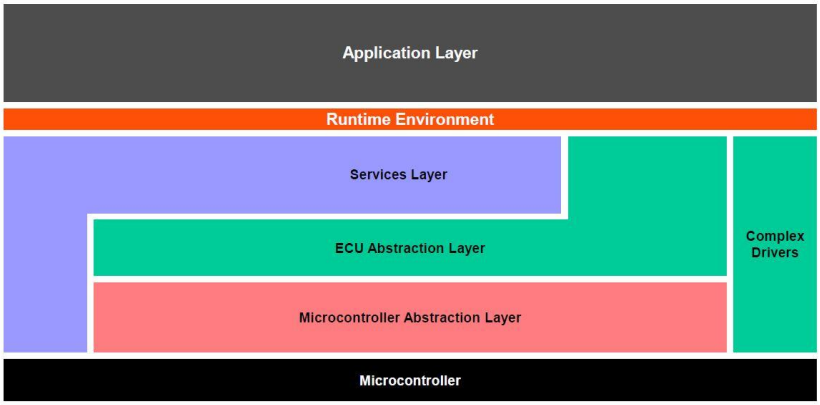


Figure 3.3
Overview of Layered Software Architecture

The ECU Abstraction Layer is identified as EAL and it gives independence to the upper layers of ECU hardware structure; it contains all the services and interfaces for the external devices, i.e. On-board Device Abstraction(e.g. External Watchdog Driver), Memory Hardware Abstraction (e.g. Memory Interface), Communication Hardware Abstraction (e.g. CAN Interface) and I/O Hardware Abstraction.

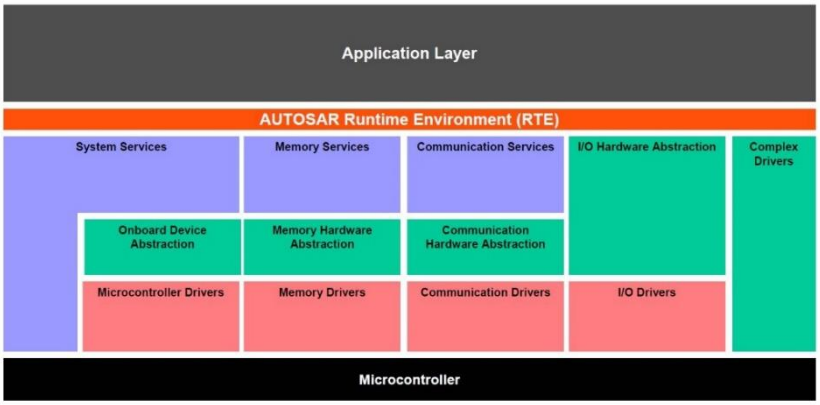


Figure 3.4
Detailed overview of Layered Software Architecture

The highest level is the Services Layer; it gives a direct access to the OS, diagnostic functionalities and the management of vehicle network communication.

The Complex Drivers covers all the layers, from the HW to the RTE; it gives the possibility to include special drivers for devices, for timing constrains or because they are not defined in AUTOSAR architecture.

The RTE works like an interface and it is a Middleware Layer: it allows the communication between applications and connect them to OS and HW. The Runtime Environment isolates the application layer and so Software Components can be independent from the ECU layout and the environment.

3.1.2 Software Component

The system functionalities of an application are developed in Software Components (SW-Cs) than can include a large set of functions; it needs to have a well-defined and standardized interface for a successful interaction over the ECU (or several ECUs is the system is complex).
It is important to underline that SW-Cs are atomic (they are distributed over one ECU) and AUTOSAR does not give information about their implementation, it provides specification for a successful interaction between them.

All the data, resources and interfaces needed by the SW-C (e.g. client, server, ports…) are contained in the SW-C Description, which also give information about their specific implementation.

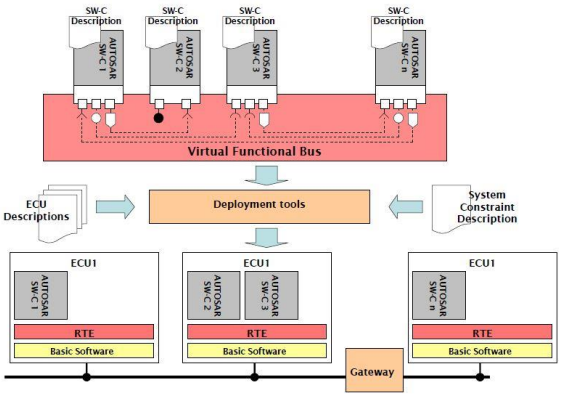


Figure 3.5
Software Components and Communication

3.1.3 SW-C Communication

SW-C Communication is possible only with a specific standardized interface that is identified as Port Interface: it consists on ports that provide or require data according to a certain agreement.
The port must provide both Client and Server implementation; the client is the one that always starts the communication to request a service by the server.
If the port

- provides the element is identified as PPort
- requires the element is identified as RPort.

The communication mechanism and interface between SW-Cs is provided by the RTE; the sum of RTEs on different ECUs implements the Virtual Functional Bus (VFB) that gives the possibility to different software components to establish an exchange of data, without caring about which ECU is running: it implements all the necessary for the communication. Moreover, it allows the relocation on other ECUs of the software components.
The set of instructions that can be executed by the RTE is identified as Runnable, that can be seen as a task running on an ECU.

3.2 ISO 26262

Safety is an essential concept for automotive industry, carmakers aim to security as a key selling point to take advantage of the competition. It is important to have an absence of risk and good measures to manage them.
Comes into play here the ISO 26262, an international standard for E/E systems that defines the safety-related requirements covering the entire vehicle life cycle process:

- Requirements specification
- Design
- Implementation

- Integration
- Verification
- Validation
- Configuration

The standard indicates all the steps to follow in every phase in order to ensure the avoid of control systematic failures.
The most important aspect of the standard is the concept of Functional Safety that is defined as "the part of the overall safety of a system that depends on the system operating correctly in response to its inputs, including the safe management of likely operator errors, hardware failures and environmental changes"[2].
It should be emphasized that functional safety does not implies the total absence of risks of incorrect operation, but it implies the absence of risks that cannot be accepted due to the malfunctioning of E/E systems.

3.2.1 ISO 26262 Structure

The standard ISO 26262 is divided in 10 parts:

1. Vocabulary
2. Management of functional safety
3. Concept phase
4. Product development at the system level
5. Product development, hardware level
6. Product development, software level
7. Production and operation
8. Supporting processes
9. ASIL-oriented and safety-oriented analysis
10. Guideline on the safety standard

[2] ISO, Road vehicles - Functional Safety - 26262-6. ISO, 2011.

The thesis activity is focused on SW development, addressed in section 6 of the standard.
In fact, the part 6 is related to the production development in terms of software, it describes the phases and the methods in order to have compliance with the standard.

The following steps compose this section of the standard:

- *Initiation:*
  Establishment of guidelines, plan of functional safety activities.

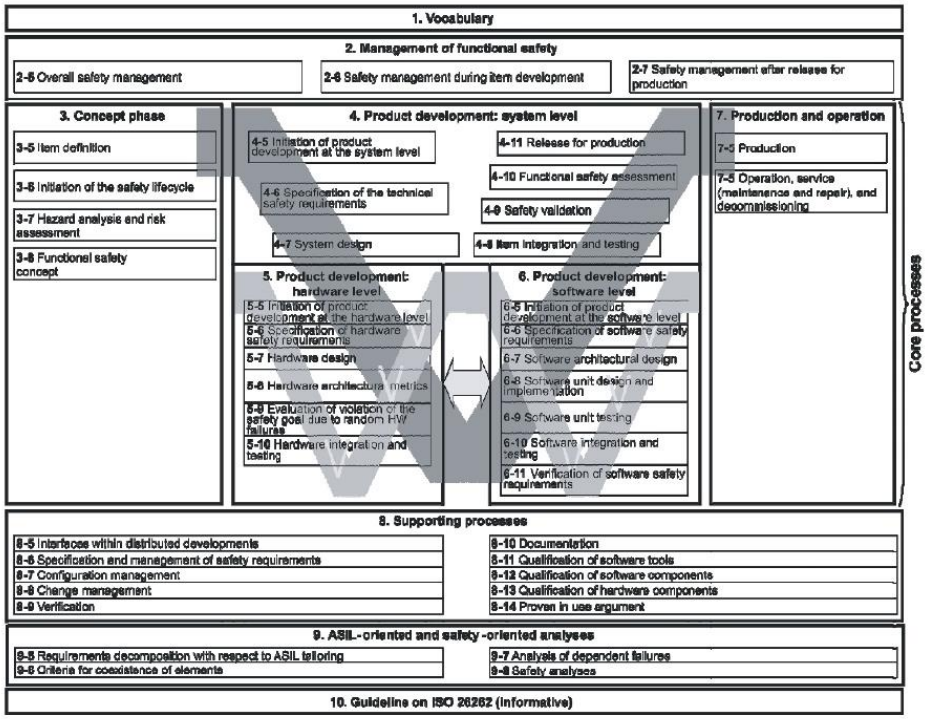


Figure 3.6
ISO 26262 Structure

- *Specification of software safety requirements:*
  Specification of safety requirements and HIS. Here there is the addressing of function module able to detect and handle faults, in order to maintain a safe state.

- *Software architectural design:*
  Design of SW architecture, representation of SW components in terms of Static Aspects (interfaces and data path) and Dynamic Aspects (timing). Feasibility and testability are taken into account, in particular it is checked that SW safety requirements are respected.

- *Software unit design and implementation:*
  Code development according to coding guidelines and design specification.

- *Software unit testing:*
  Test phase, here it is checked that the single module works correctly without undesired functionality.

- *Software integration and testing*
  All the SW entities are integrated. The SW must be compatible with SW architectural design; the test phase shows that the SW is robust through interface test, fault-injection test, back-to-back comparison (if possible)…

- *Verification of software safety requirement:*
  Demonstration that the system reflects the expected results and works correctly.

The standard gives provision about the methods and measures for each phase with specific table; in this way, the methods can be applied according to ASIL level in relation to specific the safety goal.
The ASIL represents the measure of a risk of failure in a system component. There are four level of risk A – B – C – D, from the least to the most important; when there is no safety requirement there is the option QM (quality management).

It is possible to define the ASIL level for each hazardous event by

- *Severity:*
  Measures the severity of the damage in case of a system failure (damage in

terms of people and property).
Classes: S0 (No injuries) – S1 – S2 – S3 (Fatal injuries)

- *Exposure:*
  Probability that a fault could be a safety hazard.
  Classes: E0 (Low probability) – E1 – E2 – E3 (High probability)

- *Controllability:*
  It measures the probability that a dangerous situation can be avoided.
  The danger may be due to the driver or external factors.
  Classes: C0 (Easily Controllable) – C1 – C2 – C3 (Uncontrollable)

Combining Severity, Exposure and Controllability it is possible to determinate the ASIL with the help of the table proposed by ISO 26262 Part3 (Figure 2.7).

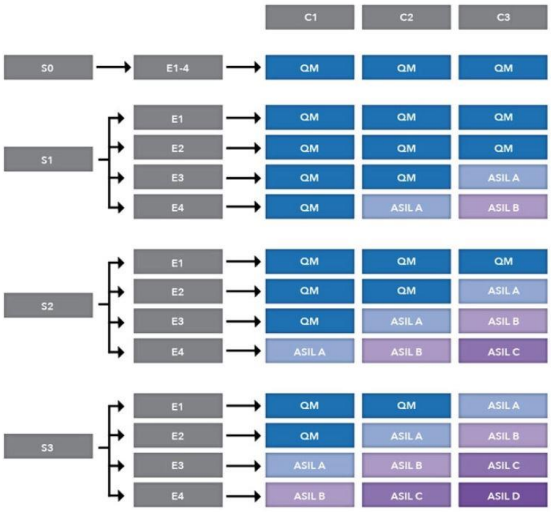


Figure 3.7
ASIL determination

3.2.2 V-Model

The automotive industry is constantly evolving and the complexity of the systems is increasing thanks to new technologies. For programmers it is therefore necessary

to follow a very precise coding and testing scheme, in order to have a correctly functioning product that meets market needs and user requirements.
It is possible to refer to different SDLCs (Waterfall Model, V-shaped Model, Iterative Model…); in the case of this thesis, it was considered the V-shaped model.

The V-Model, known as Verification and Validation Model, represent the development phases of a SW where the process are performed according to a V scheme (Figure 2.7).

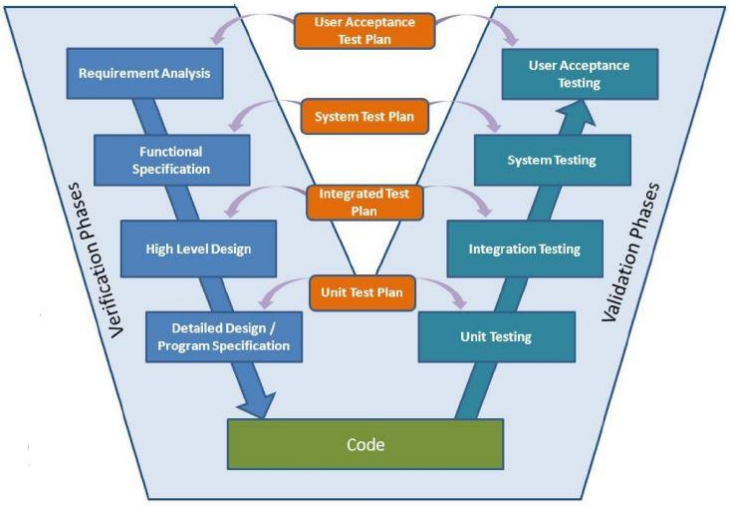


Figure 3.8

SW development flow,
V-model

The main characteristic of this model is that each step is associated to a corresponding testing activity.
From the Figure 2.7 is evident that the model is composed by a Verification and a Validation phase.

- Verification Phase

  Requirement Analysis:

  This first step is related to the requirement analysis, in particular it is generated a document with all the requirements and specification in terms of data, performance and functions. In this phase, it is define what is the expected behaviour and not the design of the SW.

Functional Specification:

Here there is the study of the requirement document and it is possible to figure out what could be a feasible implementation of the SW. This is phase is also called System Design.

High Level Design:

Also known as Architecture Design, here there is the SW architecture design that consists in the definition of interphases, architecture diagram and list of modules; it is a high-level design.

Detailed Design/Program Specification:

This phase can be identified as low-level design (LLD) step. Programmers split the system in modules and they start to work on a pseudocode (design of databases, API interfaces, error…).

Here and in the *Code* phase there is the real coding process development.

- Validation Phase

  Unit Testing:

  In this phase a program module is tested individually; in particular here it is checked that this entity works correctly regardless of the rest of the code. This test is useful to eliminate any bugs.

  Integration Testing:

  This test checks the interoperability and correct communication of the entities. The whole system can be tested.

  System Testing:

  At this stage of the model, it is checked that the functional and non-functional requirements are correctly met through stress and regression testing.

User Acceptance Testing:

This represents the last step and the business users perform it. It is checked that the systems respects the requirements using realistic data and it is possible to define if the system is ready for the real world or not.

The thesis activity was a good opportunity to better understand the phases of the V-model.

In particular, the work focused on the requirements analysis and SW development phase; the modules were individually tested in the debug phase, and were subsequently integrated to check the correct connection of the network during the Validation Phases.

CHAPTER 4

COMMUNICATION PROTOCOLS

An Embedded System is a microprocessor-based electronic system that integrates HW and SW, generally has a custom-designed HW platform and is identified as "special purpose" controller (it is designed for a specific use and it is not programmable by the user).
It receives inputs (for example from connected sensors) and produces outputs, so it is necessary that a communication take place between the devices: this is possible by resorting to communication systems that can be both HW and SW.

This section briefly shows the characteristics of communication protocols in embedded systems with particular attention to serial and CAN communication, developed and studied during the thesis activity.

4.1 Protocols in Embedded Systems

The set of rules and characteristics of the communication are contained in the Protocol, which defines the guidelines on how the exchange of information must occur among devices.
Protocols can be classifies as

- Inter-System
- Intra-System

In the first case, the communication is between two different devices and takes place via a bus system (Figure 4.1); an example could be the case of a PC-

development board communication.

They can be classified in:

- USB
- UART
- USART

Figure 4.1
Inter-System Protocol

In the second case, the communication is between two components that belongs to the same circuit (Figure 4.2); an example could be an accelerometer connected to the controller.

Figure 4.2
Intra-System Protocol

They can be classified in:

- I2C
- SPI
- CAN

In this thesis, the attention is focused on the SCI protocol and CAN protocol; in the following subparagraphs there is a brief description of both.

4.2 Serial Communication Interface (SCI)

The Serial Communication Interface (SCI), also known as Universal Asynchronous Receiver / Transmitter (UART), allows bit-to-bit communication that enables data exchange between a µC and a peripheral device, or between µCs. The SCI can operate in half-duplex mode (one TX, one RX) or in full-duplex mode (TX and RX is simultaneous).

It also has these following features[3]:

- Standard mark/space non-return-to-zero (NRZ) format
- 13-bit baud rate selection
- Programmable 8-bit or 9-bit data format
- Separately enabled transmitter and receiver
- Programmable polarity for transmitter and receiver
- Programmable transmitter output parity
- Interrupt-driven operation with eight flags (Noise error, Parity error…)
- Receiver framing error detection
- Hardware parity checking
- 1/16 bit-time noise detection

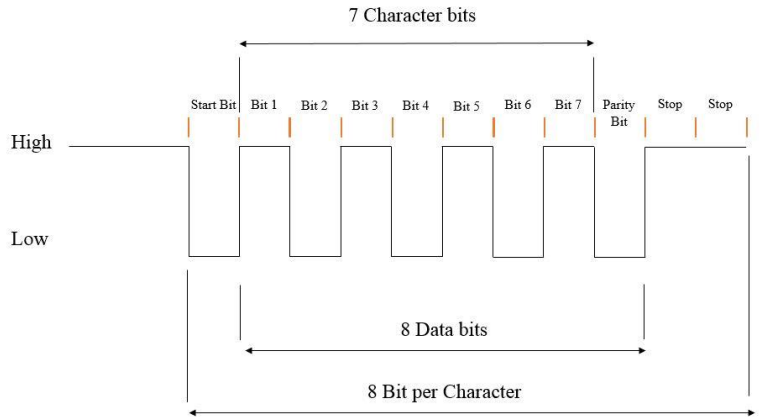


Figure 4.3
SCI, Frame

[3] MC9S12XEP100RMV1 – NXP Datasheet, Chapter.20

It is important to underline that the term “asynchronous” means that the communication is not managed by a clock signal, but the transmission is regulated by Start and Stop bits. TX and RX must work under the same baud rate.

The communication mechanism is the following: it is enabled the TX through the Transmit Enable, the data is loaded in a data register, when it is full the data is transferred, by the HW, to a shift register and then outputted to the RX. When the Receive Enable is set, the RX receive the data and it reads it at a specific baud rate.

4.3 Controller Area Network (CAN)

The Controller Area Network (CAN) is a communication protocol designed by Robert Bosch GmbH in the early 1980s.
The birth of this protocol derives from the need to have a communication between electronic components that became more and more complex, in order to improve automotive performance.
Given the difficulty in creating robust communication between these new devices, the CAN network was proposed by offering a flexible solution with a single cable that connects all electronic devices.

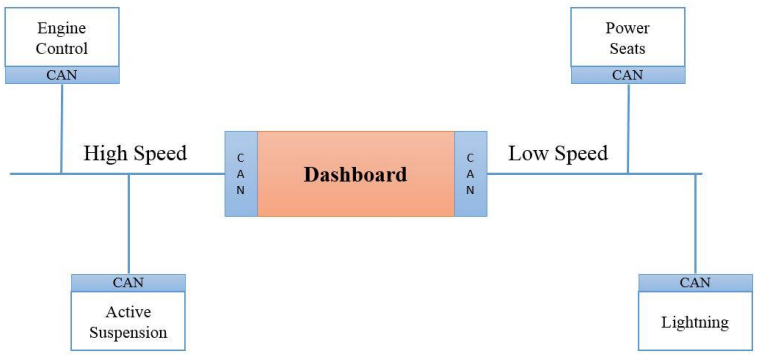


Figure 4.4
Simple example of connection
of devices through CAN Protocol

Before the introduction of the CAN Network, the connection was point-to-point and it was fine just in case of limited functions, but linking the ECU with all the many electronic devices ensuring a real time exchange of data requires a more complete and complex solution like the CAN.

The CAN Protocol is used not only in the automotive application, but also in medical industries, aircraft and so on, because of its many positive aspects:

- It is low cost, in terms of price/performance ratio
- Reliability, good errors detection and handling
- Flexibility, no limitation in number of nodes
- Capability of broadcast communication
- Data rate up to 1 MBit/s for a bus of 40 m (see *Data-Link Layer*).

4.3.1 CAN Architecture

To transfer data, the CAN Protocol uses the ISO/OSI Reference model:

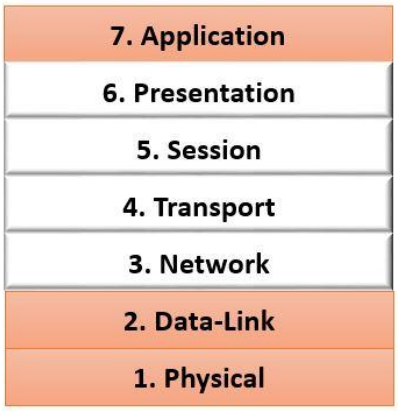


Figure 4.5
ISO/OSI Reference Model

This protocol uses only the Physical Layer, the Data-Link Layer and the Application Layer.

- *Physical Layer*

  The CAN Protocol is a 2-wire bus (CAN High, CAN Low) terminated by a resistance of 120Ω; having two wires means being immune to the noise because by means of Common Mode Rejection. Without the twisted pair cable the signal can be disturbed.

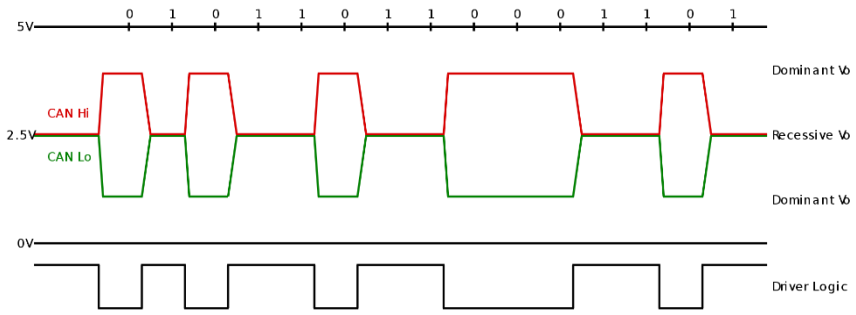


Figure 4.6
CAN Signals

The bus can have a dominant (0) or recessive value (1); the dominant bit always overwrite the recessive one.

In order to do not cause errors during the transmission of data, it is necessary to that all the nodes involved in the network must work at the same nominal bit rate (number of bits per seconds).

Synchronization is possible with the first transition recessive-dominant bit after a period SOF and a resynchronization is done every time there is this transition, in this way it is possible to reduce of noise and guarantee a correct arbitration. To be sure, about the synchronization, NRZ is used.

- *Data-Link Layer*

  This layer guarantees a reliable transmission, without errors: after sending a data, it waits for a confirmation ACK.

The CAN Protocol has four different type of frame:

- Data Frame: a node transmits a data to any other node of the network (Figure 4.7);
- Remote Frame: a node request a specific data to another node
- Error Frame: the node in TX or in RX detect an error, so it sends six dominant bit and an error flag delimiter of eight recessive bits.
- Overload Frame: the node is not ready for reception; to avoid overload.

In the following Figure is shown the Data Frame structure:

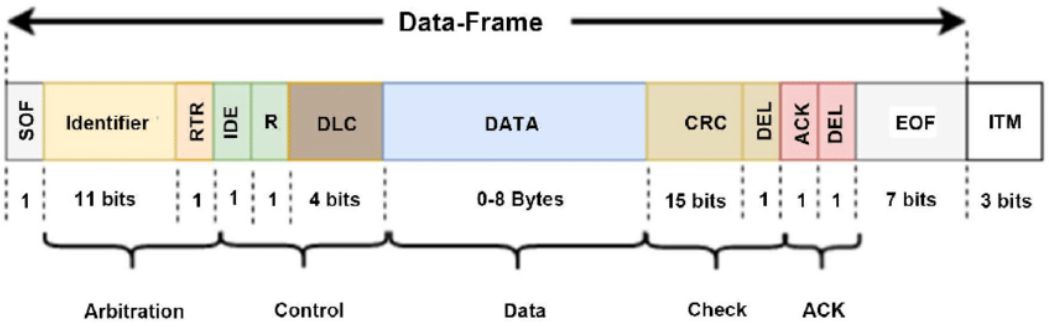


Figure 4.7
CAN Frame structure

A frame can be Standard or Extended:

- *Standard Frame:* 11 bits of ID, frame of 44 bits to 108 bits, data from 0 to 64 bits.
- *Extended Frame:* 29 bits of ID, frame of 62 bits to 129, data from 0 to 64 bits.

The CAN Protocol is characterized by a mechanism of arbitration to avoid conflicts when more than one node tries to transmit data over the network.

Every transmitter node compares the value that it sends with the value that is on the bus: if the bit is the same, the node can go on with the transmission, if not it must stop to transmit.
The arbitration is done by message priority considering the ID bits of the frame over the network: smaller is the value, higher is the priority.

CAN is CSMA/CD protocol, it verifies the absence of traffic on the bus before starting the transmission (regulated in relation to the frame priority): the node waits for a specific time then it start sending data. If more than one node start to transmit, they detect this collision (collision detection, CD) and stop the transmission.

- *Application Layer*
  It defines the CAN configuration, for example the ID format and the main function is the network management.

4.3.2 CAN Software Configuration

The CAN SW is structured three layers: HW, Basic Software and Application.
The Basic Software implements the:

- *Communication Services*
  In order to use driver services it define appropriate API.
- *Protocol Controller Driver*
  Related to the management of the operation of the protocol
- *Transceiver driver*
  Related to the physical aspects of the CAN Protocol

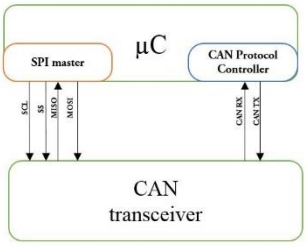


Figure 4.8
Example Transceiver

CHAPTER 5
IMPLEMENTATION

This chapter shows the details of the thesis, based on the knowledge presented in the previous paragraphs acquired in the first period of activity.

The first interface presented is the serial one: the structure, the main features and the common thread of this FW structure will be shown.
The same analysis scheme is used for the implementation of the CAN network.

5.1 Serial Interface

The task of the first software module is to provide a uniform interface to the Serial Interface used for custom connection between ECU and PC tool interface.

5.1.1 FW structure

The Figure 5.1 shows the structure of the FW relating to the serial interface. The approach is to start implementing the "lower" levels and then go up and forward with abstraction layers.

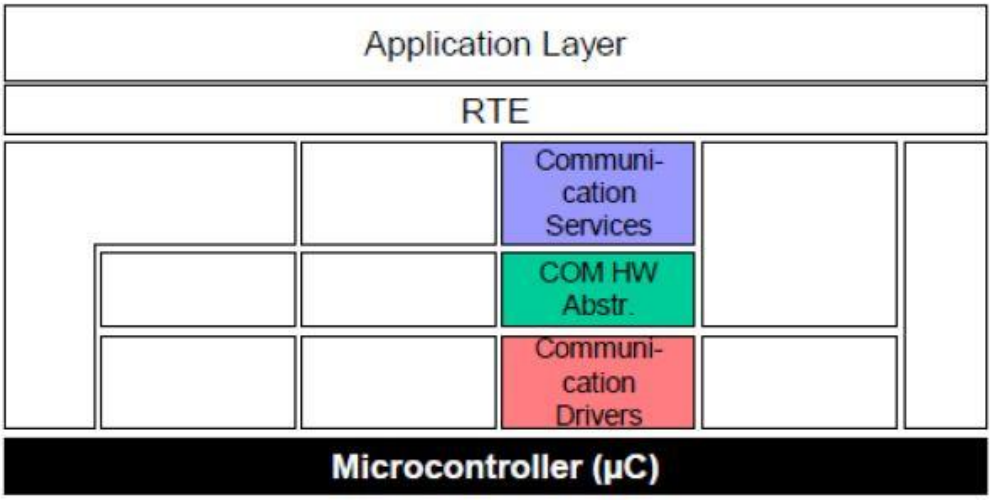


Figure 5.1
Serial Interface - Communication Stack

The Figure 5.1 shows the modules developed in the following order:

- *Communication Drivers*
  MCAL, in Pink.
  This is the module with a direct access on-chip, external devices are mapped.
  Dependent on µC.

- *Communication Hardware Abstraction*
  ECAL, in Green.
  It provides a mechanism of access to devices regardless if they are on-chip or on-board.
  Independent of µC, dependent on ECU HW.

- *Communication Services,*
  In Blue.
  It provides management of vehicle network, ECU state. Basic SW modules for applications.
  Independent of a µC and ECU.

5.1.2 Communication Drivers, SCI

This layer is structured in two macro-modules: MCAL_cfg and MCAL_S12XET256.

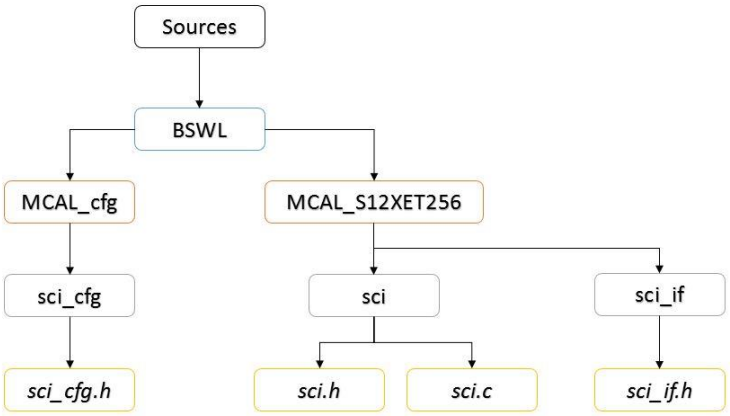


Figure 5.2
Communication Drivers Structure

sci_cfg.h

The first one is related to the SCI Layer Configuration, using the MCU datasheet it is possible to set the register with standard values at start-up.

In the Figure 5.2 is shown a portion of the file *sci_cfg.h*. The main characteristic of these definitions is that all the register are dependent on the micro so as to be easily adaptable if there were changes in the HW (and therefore different settings). This is possible simply using the *#define*: if there were any changes, it will not be necessary to modify the entire code, but it will be sufficient to replace the new values and consequently the SW will adapt to this change.
With this small detail, it is possible to have a leaner and faster setting.

In this header file SCI Control Register 1, SCI Control Register 2, SCI Baud Rate and SCI Parity bit are set according standard values.
Constant definitions such as BAUD_RATE_115200, are set in *sci_if.h* contained in the other module.

```
/*****************************************************************************
*  SCI Layer Configurations
*****************************************************************************/

/* SCI init */

#define STATUS_SCI1_RX_TX                       SCI1SR1 & 0x20
#define SCI1_DATA_RX_TX                         SCI1DRL
#define SCI1_CONTROL_REG2                       SCI1CR2
#define SCI1_ENABLE_RX                          0x24
#define SCI1_ENABLE_TX                          0x88


#define SCI_Control_Register_1 0x00

/* SCI1CR1 = 0x00;          SCI Control Register 1

                                        0b00000000
                                          |||||||_ PT (Parity Bit)
                                          ||||||__ PE (Parity Enable)
                                          |||||___ ILT
                                          ||||____ WAKE
                                          |||_____ M
                                          ||______ RSRC
                                          |_______ SCISWAI
                                          ________ LOOPS
*/


#define SCI_Control_Register_2 0x24

/*  SCI1CR2 = 0x24;          SCI Control Register 2 (start only with RX interrupt active)

                                        0b00100100
                                          |||||||_ SBK (Send Break bit)
                                          ||||||__ RWU (Receiver Wakeup bit)
                                          |||||___ RE (Receiver Enable bit)
                                          ||||____ TE (Transmitter Enable bit)
                                          |||_____ ILIE (Idle Line Interrupt Enable bit)
                                          ||______ RIE (Receiver Full Interrupt Enable bit)
                                          |_______ TCIE (Transmission Complete Interrupt Enable bit)
                                          ________ TIE (Transmission Interrupt Enable bit)
*/

/* Standard values at start-up uC */

#define SCI_StandardBaudRate  BAUD_RATE_115200

/* SCI1BD = (uint)SCI1BaudAtt;

                            SCI1BD = 0x0011;    115200
                            SCI1BD = 0x0023;    57600
                            SCI1BD = 0x0034;    38400
                            SCI1BD = 0x0068;    19200

                             SCI Baud Rate Register

                                          SCICLK        32 MHz
                             Baud Rate = ---------- = ---------- = 117674    (115200)
                                         16 * SBR      16 * 17
```

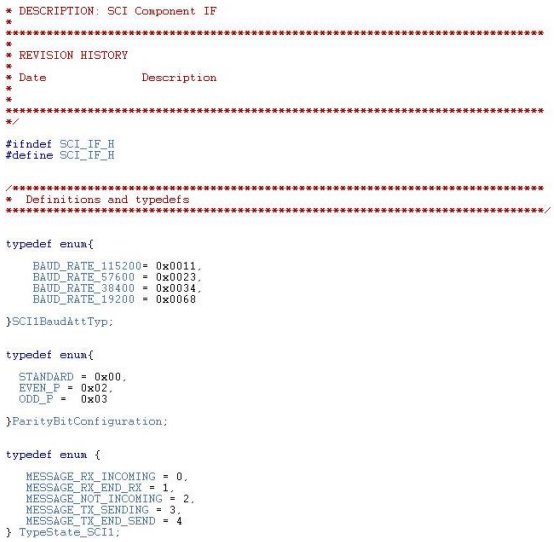
Figure 5.3

sci_cfg.h (MCAL_cfg)

MCAL_S12XET256

Inside MCAL_S12XET256, there are the *sci_if* (it contains *sci_if.h)* and *sci* (it contains *sci.c* and *sci.h*) modules.

Sci_if.c contains all the definition and typedefs (Figure 5.4), variables and constants, functions and stubs used in the *sci.c* file.



```
* DESCRIPTION: SCI Component IF
*
*******************************************************************************
*
* REVISION HISTORY
*
* Date              Description
*
*
*******************************************************************************
*/

#ifndef SCI_IF_H
#define SCI_IF_H


/*******************************************************************************
*  Definitions and typedefs
*******************************************************************************/


typedef enum{

    BAUD_RATE_115200= 0x0011,
    BAUD_RATE_57600 = 0x0023,
    BAUD_RATE_38400 = 0x0034,
    BAUD_RATE_19200 = 0x0068

}SCI1BaudAttTyp;


typedef enum{

  STANDARD = 0x00,
  EVEN_P = 0x02,
  ODD_P =  0x03

}ParityBitConfiguration;


typedef enum {

   MESSAGE_RX_INCOMING = 0,
   MESSAGE_RX_END_RX = 1,
   MESSAGE_NOT_INCOMING = 2,
   MESSAGE_TX_SENDING = 3,
   MESSAGE_TX_END_SEND = 4
} TypeState_SCI1;
```

Figure 5.4

sci_if.h (MCAL_S12XET256) definitions

The use of symbols in the *typedef enum* (such as STANDARD, EVEN_P, ODD_P ...) facilitates code modification and makes the debugging phase more intuitive and fast.

```
/**
*******************************************************************************
*
*  Function:    SCI_ChangeBaudSCI1
*  Description: baud management SCI
*
*******************************************************************************
*/

#pragma CODE_SEG SCI_PLACEMENT_CODE

void  SCI_ChangeBaudSCI1(SCI1BaudAttTyp BaudRate)
{
    SCI1BD = (uint)BaudRate;
}


/**
*******************************************************************************
*
*  Function:    SCI_ParityBitSCI
*  Description: parity bit SCI1
*
*******************************************************************************
*/

#pragma CODE_SEG SCI_PLACEMENT_CODE

void  SCI_ParityBitSCI1(ParityBitConfiguration ParityBit)
{
    SCI1CR1 = (byte)ParityBit;
}


/**
*******************************************************************************
*
*  Function:    SCI_Timer1ms
*  Description: SCI_Timer1ms
*
*******************************************************************************
*/

#pragma CODE_SEG SCI_PLACEMENT_CODE

void SCI_Timer1ms(void)
{

    if (u8SCI_ContSCI1TimeOut < SCI1_TIME_OUT)
    {
        u8SCI_ContSCI1TimeOut++;
    }
}
```

Figure 5.5

sci.c (MCAL_S12XET256) functions

As shown in the last figure, the *sci.c* file implements the “lower level functions” such as the setup of the parity bit, the timer or the choice of the baud rate value for the serial communication.

Important functions to pay attention to are those of “Init”.
The function *void SCI_InitSCI1(void)*is necessary in order to have a setup and initialisation of the SCI control registers .

```
#pragma CODE_SEG SCI_PLACEMENT_CODE

void  SCI_InitSCI1(void)
{
    SCI1BD = (uint)SCI_StandardBaudRate;

    SCI1CR1 = (byte)SCI_StandardParity;

    SCI1CR2 = SCI_Control_Register_2;
}
```

Figure 5.6
sci.c (MCAL_S12XET256) SCI_InitSCI1

```
/**
*****************************************************************************
*
*  Function:    SCI_IntSchedulerSCI1
*  Description: scheduler
*
*****************************************************************************
*/

#pragma CODE_SEG SCI_PLACEMENT_CODE

void  SCI_IntSchedulerSCI1(void)
{

    if(STATUS_SCI1_RX_TX)       /* RX active */
    {
        u8SCI_ContSCI1TimeOut = 0;

        AckDatRxSCI1 = SCI1_DATA_RX_TX;

        u8SCI_BuffRxSCI1[u8IndRxSCI1] = AckDatRxSCI1;

        SCI_StateSCI1 = MESSAGE_RX_INCOMING;

        u8IndRxSCI1++;

        //SCI_StateSCI1 = MESSAGE_RX_END_RX;
    }
    else
    {
        if (u8SCI_NumByteTxSCI1 > 0)
        {
            SCI1_DATA_RX_TX = u8SCI_BuffTxSCI1[u8SCI_IndTxSCI1++];

            u8SCI_NumByteTxSCI1 --;
        }
        else
        {
            SCI1_CONTROL_REG2 = SCI1_ENABLE_RX; /* enable RX */
            SCI_StateSCI1 = MESSAGE_TX_END_SEND;
        }
    }

}
```

Figure 5.7
sci.c (MCAL_S12XET256) SCI_InitSchedulerSCI1

The second Init function, *void SCI_IntSchedulerSCI1(void)*, is related to the scheduler. In particular, it enables the reception and checks the RX status. In the debug phase, it is easily possible to see that a reception is happening or is finished thanks to the symbolic messages "MESSAGE_RX_INCOMING", "MESSAGE_TX_END_SEND" and so on.

The functions implemented in this module are called in the *main.c*, together with the ones that enable reception and transmission as shown in the Figure 5.8 and Figure 5.9.

```
/**
*****************************************************************************
*
*  Function:    SCI_EndMessageRXSCI1
*  Description: Wait End Message RX
*
*****************************************************************************
*/

#pragma CODE_SEG SCI_PLACEMENT_CODE

void SCI_EndMessageRXSCI1(void)
{
    if ((u8SCI_ContSCI1TimeOut >= SCI1_TIME_OUT) && (SCI_StateSCI1 == MESSAGE_RX_INCOMING))
    {

      //  FineRxSCI1 = 0xFF;                          // Rx terminated, corret
        SCI_StateSCI1 = MESSAGE_RX_END_RX;
        SCI1CR2 = 0x00;                             // disable RX and keep Tx disable
        u8IndRxSCI1 = 0;
    }
}
```

Figure 5.8

Sci.c (MCAL_S12XET256) Disable RX

```
#pragma CODE_SEG SCI_PLACEMENT_CODE

void SCI_SendMessageSCI1(byte MessLength)
{
    u8SCI_NumByteTxSCI1 = MessLength;
    SCI_StateSCI1 = MESSAGE_TX_SENDING;
    u8SCI_IndTxSCI1 = 0;
    SCI1_CONTROL_REG2 = SCI1_ENABLE_TX;  /* Enable TX */
}
```

Figure 5.9

sci.c (MCAL_S12XET256) Enable TX

The use of *#pragma* directive is necessary for each function because it specifies where its segment is allocated. CODE_SEG ensures that all definitions and function declarations are in the same segment, in this case SCI_PLACEMENT_CODE (that is OTHER_ROM).

Before proceeding with the description of the code related to Communication Hardware Abstraction (section 5.1.4), the structure of the messages exchanged between PC and ECU is presented below.

5.1.3 Messages Structure

Each message has a precise structure, stand out in:

- Request from PC to ECU

STX	START byte message
LEN	Total message length
RX	Receiver Code
TX	Transmitter Code
CMD	Command Code
SCMD	Subcommand Code
DATA	N bytes of data
CHK	Checksum: Sum without reporting from LUNG to last DATA BYTE
ETX	Message STOP Byte

Table 5.1
Request from PC to ECU Structure

- Positive reply message from ECU to PC

STX	START byte message
LEN	Total message length
RX	Receiver Code
TX	Transmitter Code
ACK	Acknowledge Command Code (CMD received + 0x20)
SCMD	Subcommand Code
DATA	N bytes of data

CHK	Checksum: Sum without reporting from LUNG to last DATA BYTE
ETX	Message STOP Byte

Table 5.2
Positive reply message from ECU to PC Structure

- Negative reply message from ECU to PC

STX	START byte message
LEN	Total message length
RX	Receiver Code
TX	Transmitter Code
KO	KO (“k” in ASCII)
ERR	Error Code
DATA	N bytes of data (if any)
CHK	Checksum: Sum without reporting from LUNG to last DATA BYTE
ETX	Message STOP Byte

Table 5.3
Negative reply message from ECU to PC

Each byte of the message takes on a very precise value and refers to the Multronic S.r.l. DOCUMENT (updated during the thesis activity) N°: 001-19 ELECTRONIC CODIFICATION of the ECU.

For privacy reasons, these values cannot be shown explicitly.

The requests from PC to ECU are to characterize a command byte (CMD) which can assume 8 different values in hexadecimal depending on the type of request. Each type of CMD, in turn, has a set of subcommands SCMD (characterized by a certain value as well).

The possible CMDs and some examples of possible SCMDs are shown in the following set of requests:

- CMD: Read request
  SCMD: Reading of N-byte from RAM, 2-byte address, reading of N data from ECU….

- CMD: Write request
  SCMD: Writing of N-byte on RAM, 2-byte address, writing of data and time of ECU internal clock…

- CMD: Start - Stop writing, programming and downloading
  SCMD: Start programming file s19 ECU, Stop programming file s19 ECU.

- CMD: Lock-unlock flash
  SCMD: Unlock Internal Flash, Lock Internal Flash.

- CMD: Hook and Test present
  SCMD: Test Present ECU

- CMD: Memory cleaning
  SCMD: Clear n-th External Flash…

- CMD: General Messages
  SCMD: ECU Reset, Subcommand saving EEPROM Memory…

- CMD: Testing
  SCMD: External Flash Test, Request activation digital output...

5.1.4 Communication Hardware Abstraction, SCI

At this point, it is possible to level up and start a first level of abstraction.

In order to describe the structure of this macro module, it is useful to refer to the following figure:

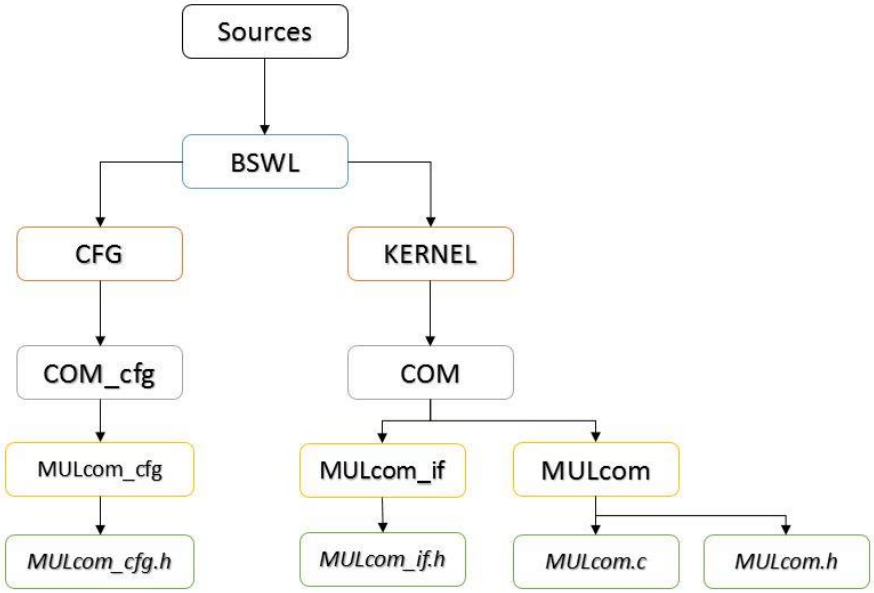


Figure 5.10
Communication HW Abstraction Structure, SCI

The prefix "MUL" is used in order to resume the company name (MULtronic).

MULcom_cfg.h

This branch of the software structure is related to the COM Component Configuration, are defined:

- *ECU Addresses*
- *Read/Write EEPROM*
- *Answer Read*
- *Answer Write*
- *Answer Lock/Unlock*
- *Answer Hook Test*
- *Answer Memory*

- *Answer General Message*
- *Answer Testing*
- *Position of byte in Message Frame*

The MULcom layer configuration is done through the use *#define*. In the case of the *ECU Addresses,* all the registers in hexadecimal are defined.

```
/* MULcom Position in Message Frame */
#define MULcom_START_MESSAGE_POSIT                0
#define MULcom_LENGTH_POSIT                       1
#define MULcom_RECEIVER_CODE_POSIT                2
#define MULcom_TRANSMITTER_CODE_POSIT             3
#define MULcom_COMMAND_POSIT                      4
#define MULcom_SUBCOMMAND_POSIT                   5
```

Figure 5.11

Element position in a message frame

The other set of definition have the structure shown in the figure below:

```
/*MULcom presence MULcom_AnswerWrite*/
#define PRESENCE_SCMD_RAM_NBYTE_WRITING_2BYTE
#define PRESENCE_SCMD_RAM_NBYTE_WRITING_3BYTE
#define PRESENCE_SCMD_INTERNAL_FLASH_NBYTE_WRITING
#define PRESENCE_SCMD_INTERNAL_EEPROM_NBYTE_WRITING_2BYTE
```

Figure 5.12

Example of #define in *MULcom_cfg.h*

For each type of SCMD of all possible CMD, there is a corresponding *#define PRESENCE_SCMD_...* In following pages, it will be shown how these will be necessary for handling messages in the *MULcom.c* file.

MULcom.h

The file related to the MULcom Component Header, is dedicated to the definition and declaration of variables, arrays and so on.

```
extern uint BaudRateSCI1change;
extern uint BaudRateNewSCI1;
extern uint StatoFE;
extern uint StartMemFlash;

byte    FWVerUDM[8]     = {0,0,0,0,0,0,0,0};
byte    BootVerUDM[8]   = {0,0,0,0,0,0,0,0};
byte    CodeUDM[8]      = {0,0,0,0,0,0,0,0};
byte    CodRevUDM[8]    = {0,0,0,0,0,0,0,0};
byte    BatchUDM[8]     = {0,0,0,0,0,0,0,0};
byte    SerialNumUDM[8] = {0,0,0,0,0,0,0,0};
byte    DateUDM[8]      = {0,0,0,0,0,0,0,0};
```

Figure 5.13

Example of variables in *MULcom.h*

To be consistent in the names used, a name-change is done in the same file for each element in this way:

```
/*****************************************************************************
*  SWC Inputs definition
*****************************************************************************/

#define MULcom_GetSCI1( value )                 u8SCI_BuffRxSCI1[ value ]
#define MULcom_GetSCI1State()                   SCI_StateSCI1
#define MULcom_SendMessageSCI1( value )         SCI_SendMessageSCI1( value )
#define MULcom_ResetCOP()                       ResetCOP()
#define MULcom_EEXT25C080_SaveVarEEPROMExt()    SaveVarEEPROMExt()
#define MULcom_WriteMappaturaEEE( value1,value2 )  WriteMappaturaEEE( value1,value2 )

#define MULcom_AddH                             AddH
#define MULcom_AddM                             AddM
#define MULcom_AddL                             AddL
#define MULcom_NByte                            NByte
#define MULcom_StSlaveSCRInfo                   StSlaveSCRInfo
#define MULcom_StatoOrol                        StatoOrol
```

Figure 5.14

Change-name in *MULcom.h*

In MULcom.h, there are SWC Outputs Assignments such as

```
/*****************************************************************************
  *  SWC Outputs assignement
  *****************************************************************************/

 #define MULcom_PutSCI1( value, value1 )        u8SCI_BuffTxSCI1[ value ] = value1
 #define MULcom_ReadPutSCI1( value )            u8SCI_BuffTxSCI1[ value ]
```

Figure 5.15

SWC Outputs Assignment, *MULcom.h*

This is done to avoid problems in terms of interoperability between SW modules. These two element shown in the Figure 5.12, together with *MULcom_GetSCI1(value)* and *MULcom_GetSCI1State()*, are used in the file

MULcom.c to manage the answer (see next section *MULcom.c*).

MULcom.c

This file contains the core of the management of request messages from PC to ECU and transmits the corresponding response or an error message if something goes wrong.
The code has a "waterfall" structure, i.e. the execution of a function triggers the call to another one and so on. To better understand the dynamics of the SW it is useful to refer to the graphs shown in this section.

The first function performed (called in the Main Loop in *main.c*) is *MULcom_Answer ()*.

The scans the message that is received and checks step by step if the data is actually a valid request:

1. Check if a message has been received;
2. Checksum Check (comparison of sent and calculated checksum value);
3. Start Byte Check ;
4. Length and end of the frame Check;
5. RX Code Check;
6. TX Code Check.

If the message does not comply with the expected format, an error frame corresponding to the anomaly detected is sent.

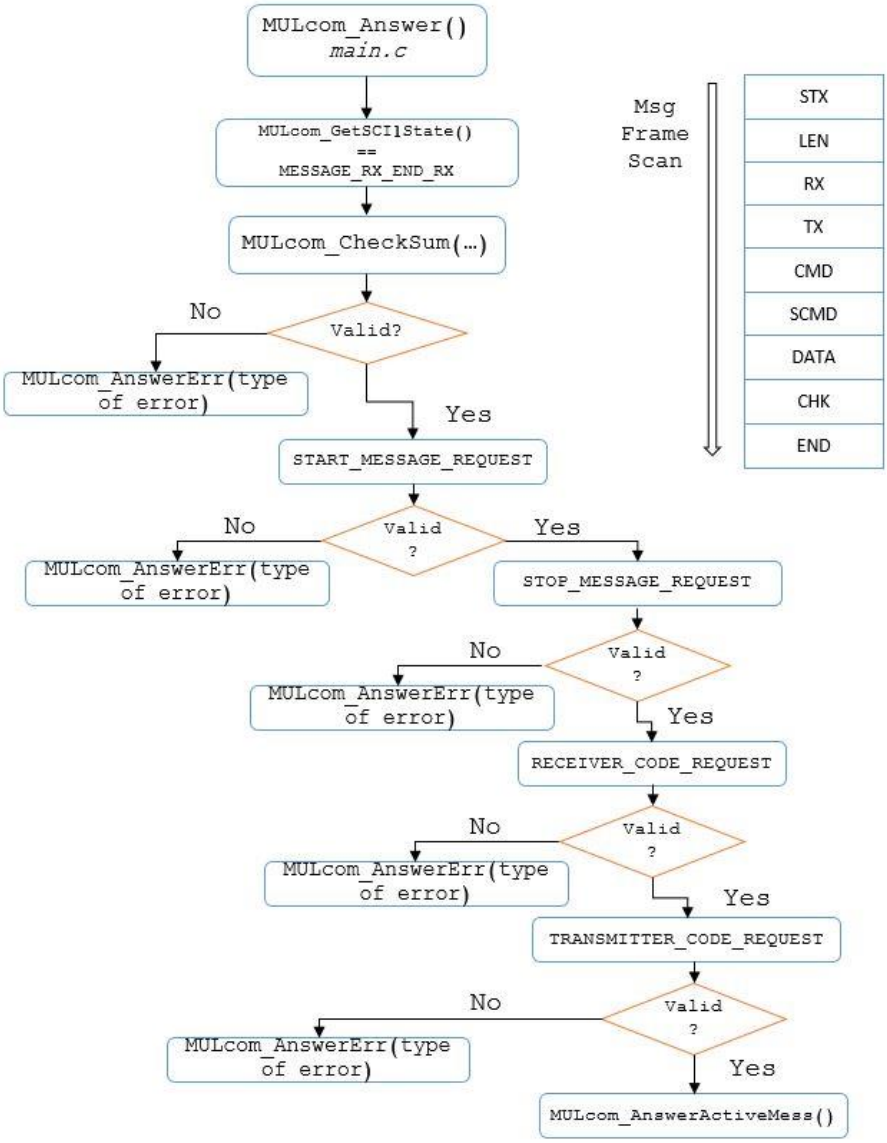


Figure 5.16

Message Frame Scan

```
/**
*****************************************************************************
*
*  Function:    MULcom_Answer
*  Description: Answer to a request
*
*****************************************************************************
*/

#pragma CODE_SEG MULcom_PLACEMENT_CODE

void  MULcom_Answer(void)
{

    if (MULcom_GetSCI1State() == MESSAGE_RX_END_RX)
    {

        if ( MULcom_CheckSum((byte *far)&MULcom_GetSCI1( MULcom_LENGTH_POSIT ), MULcom_GetSCI1(MULcom_LENGTH_POSIT) - 3) == MULcom_GetSCI1(MULcom_GetSCI1(MULcom_LENGTH_POSIT) - 2))
        {
            if (MULcom_GetSCI1(MULcom_START_MESSAGE_POSIT) == MULcom_START_MESSAGE_REQUEST)
            {

                if (MULcom_GetSCI1(MULcom_GetSCI1(MULcom_LENGTH_POSIT) - 1) == MULcom_END_MESSAGE_REQUEST)
                {

                    if(MULcom_GetSCI1(MULcom_RECEIVER_CODE_POSIT) == MULcom_RECEIVER_CODE_REQUEST)
                    {

                        if(MULcom_GetSCI1(MULcom_TRANSMITTER_CODE_POSIT) == MULcom_TRANSMITTER_CODE_REQUEST)
                        {
                               MULcom_AnswerActiveMess();
                        }
                        else
                        {
                            /* invalid TRANSMITTER_CODE_REQUEST */
                            MULcom_AnswerErr(MULcom_WRONG_TX_IDENTIFIER_ERROR);
                        }
                    }
                    else
                    {
                         /* invalid RECEIVER_CODE_REQUEST */
                         MULcom_AnswerErr(MULcom_WRONG_MESSAGE_RECEIVED_ERROR);
                    }
                }
                else
                {
                    /* invalid STOP_MESSAGE_REQUEST */
                    MULcom_AnswerErr(MULcom_INCORRECT_MESSAGE_SEQUENCE_ERROR);
                }
            }
            else
            {
                 /* invalid START_MESSAGE_REQUEST */
                 MULcom_AnswerErr(MULcom_INCORRECT_MESSAGE_SEQUENCE_ERROR);
            }
        }
        else
        {
            /* invalid CHECK_SUM */
            MULcom_AnswerErr(MULcom_CHECKSUM_ERROR);
        }
    }

}
```

Figure 5.17

Mulcom_Answer(void)

If everything is correct, the *void MULcom_AnswerActiveMess(void)* function is executed:

```
void MULcom_AnswerActiveMess(void)
{
    switch (MULcom_GetSCI1(MULcom_COMMAND_POSIT))                  /*  Command Position:4                    */
    {
        case MULcom_CMD_READING:                                   /*  CMD READING                           */
        {
              MULcom_AnswerRead();
              break;

        }

        case MULcom_CMD_WRITING:
        {                                                           /*  CMD WRITING                           */
             MULcom_AnswerWrite();
             break;

         }
                                                                   /*  CMD START/STOP/PROGRAMMING/DOWNLOAD */
        case MULcom_CMD_START_STOP_PROGR_DOWNLOAD:
        {
             MULcom_AnswerStartStop();
             break;
         }

        case MULcom_CMD_LOCK_UNLOCK_INTERNAL_FLASH:                /*  CMD LOCK UNLOCK INTERNAL FLASH      */
        {
             MULcom_AnswerLockUnlock();
             break;
        }
```

```
        case MULcom_CMD_HOOK_TEST:                                  /* CMD HOOK AND TEST MESSAGE          */
        {
             MULcom_AnswerHookTest();
             break;
        }

        case MULcom_CMD_MEMORY_COMMAND:                             /* CMD CLEANING MEMORY                */
        {
            MULcom_AnswerMemory();
            break;

        }

        case MULcom_CMD_GENERAL_MESSAGE:                            /* CMD GENERAL MESSAGE                */
        {
            MULcom_AnswerGeneralMessage();
            break;
        }

        case MULcom_CMD_TESTING_MESSAGE:                            /* CMD TESTING MESSAGE                */
        {
            MULcom_AnswerTesting();
            break;
        }


      default:
      {
          MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);        /* CMD ERROR                          */
          break;
      }
     }
}
```

Figure 5.18

Mulcom_AnswerActiveMess()

At this point in the execution flow the type of command is checked, depending on the CMD the corresponding function is called.

Whenever there is any problem, an error function is launched (this applies to all steps):

```
void  MULcom_AnswerErr(byte errorCode)
{

    MULcom_PutSCI1(MULcom_START_MESSAGE_POSIT, MULcom_START_MESSAGE_POS_RESPONSE);
    MULcom_PutSCI1(MULcom_LENGTH_POSIT, MULcom_ERROR_MESSAGE_LENGHT);
    MULcom_PutSCI1(MULcom_RECEIVER_CODE_POSIT, MULcom_GetSCI1(MULcom_TRANSMITTER_CODE_POSIT));
    MULcom_PutSCI1(MULcom_TRANSMITTER_CODE_POSIT, MULcom_RECEIVER_CODE_REQUEST);
    MULcom_PutSCI1(4, MULcom_KO_CODE_RESPONSE);
    MULcom_PutSCI1(5, errorCode);
    MULcom_PutSCI1(6, MULcom_CheckSum((byte *far)&MULcom_GetSCI1( MULcom_LENGTH_POSIT ), MULcom_GetSCI1(MULcom_LENGTH_POSIT) - 3) );
    MULcom_PutSCI1(7, MULcom_END_MESSAGE_POS_RESPONSE);

    MULcom_SendMessageSCI1(MULcom_ERROR_MESSAGE_LENGHT);
}
```

Figure 5.19

Mulcom_AnswerErr(byte errorCode)

Once the type of CMD is identified, there is the check of the SCMD byte to identify the type of subcommand and finally recalls the function that generates the correct answer.

For example in Figure 5.19, there is the function relative to the CMD of reading and some of its calls to the SCMD functions.

```
void MULcom_AnswerRead(void)
{
    switch(MULcom_GetSCI1(MULcom_SUBCOMMAND_POSIT))       /* Subcommand Position: 5  */
    {
        case MULcom_SCMD_RAM_NBYTE_READING_2BYTE:
        {
            #if defined (PRESENCE_SCMD_RAM_NBYTE_READING_2BYTE)
            MULcom_ReadingRam_2byte();
            #else
            MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);
            #endif
            break;
        }
        case MULcom_SCMD_RAM_NBYTE_READING_3BYTE:
        {
            #if defined (PRESENCE_SCMD_RAM_NBYTE_READING_3BYTE)
            MULcom_ReadingRam_3byte();
            #else
            MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);
            #endif
            break;
        }
        case MULcom_SCMD_INTERNAL_FLASH_NBYTE_READING:
        {
            #if defined (PRESENCE_SCMD_INTERNAL_FLASH_NBYTE_READING)
            MULcom_Reading_Internal_Flash();
            #else
            MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);
            #endif
            break;
        }
        case MULcom_SCMD_INTERNAL_EEPROM_NBYTE_READING_2BYTE:
        {
            #if defined (PRESENCE_SCMD_INTERNAL_EEPROM_NBYTE_READING_2BYTE)
            MULcom_Reading_Internal_EEPROM2();
            #else
            MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);
            #endif
            break;
        }
        case MULcom_SCMD_INTERNAL_EEPROM_NBYTE_READING_3BYTE:
        {
            #if defined (PRESENCE_SCMD_COMMAND_NOT_ALLOWED_ERROR)
            MULcom_Reading_Internal_EEPROM3();
            #else
            MULcom_AnswerErr(MULcom_COMMAND_NOT_ALLOWED_ERROR);
            #endif
            break;
        }
```

Figure 5.20

Mulcom_AnswerRead(void)

As an example, the internal flash reading function is chosen to show how the frame is structured.

```
void MULcom_Reading_Internal_Flash(void)
{

    ulong AppAddr3B = 0;
    byte i;


    AppAddr3B = *(ulong *far)&MULcom_GetSCI1(5);
    AppAddr3B &= 0x00FFFFFF;

    if (((AppAddr3B >= MULcom_INIZIO_FI_R_PC) && ((AppAddr3B + MULcom_GetSCI1(9) - 1) <= MULcom_FINE_FI_PC)) ||
       ((AppAddr3B >= MULcom_INIZIO_FI_R_PC2) && ((AppAddr3B + MULcom_GetSCI1(9) - 1) <= MULcom_FINE_FI_PC2)))
    {

        MULcom_PutSCI1(MULcom_START_MESSAGE_POSIT, MULcom_START_MESSAGE_POS_RESPONSE);                    /* Start  */
        MULcom_PutSCI1(MULcom_LENGTH_POSIT,RESPONSE_LENGHT);                                              /* Lenght */
        MULcom_PutSCI1(MULcom_RECEIVER_CODE_POSIT, MULcom_GetSCI1(MULcom_TRANSMITTER_CODE_POSIT));        /* RX     */
        MULcom_PutSCI1(MULcom_TRANSMITTER_CODE_POSIT,MULcom_TRANSMITTER_CODE_POS_RESPONSE );              /* TX     */
        MULcom_PutSCI1(4,MULcom_ACK);                                                                     /* ACK    */
        MULcom_PutSCI1(5,MULcom_SCMD_INTERNAL_FLASH_NBYTE_READING);                                       /* SCMD   */


            for (i = 1; i <= MULcom_GetSCI1(9); i++)
            {
             MULcom_PutSCI1(5 + i,*(byte *far)AppAddr3B++);
            }

        MULcom_PutSCI1(MULcom_GetSCI1(9)+6,MULcom_CheckSum((byte *far)&MULcom_ReadPutSCI1( MULcom_LENGTH_POSIT ), MULcom_ReadPutSCI1(MULcom_LENGTH_POSIT) - 3) );
        MULcom_PutSCI1(MULcom_GetSCI1(9)+7,MULcom_END_MESSAGE_POS_RESPONSE);
        MULcom_SendMessageSCI1(RESPONSE_LENGHT);
    }else
    {
       MULcom_AnswerErr(MULcom_WRONG_ADDRESS_READING_ERROR);
    }

}
```

Figure 5.21

Mulcom_Reading_Internal_Flash(void)

Through *MULcom_PutSCI1(... , ...)* is possible to structure each element of the response frame plus the checksum calculation, depending on the type of CMD and SCMD.

In every function, it is always considered that there may be an anomaly. In particular, it is possible to find and report the following errors:

- Wrong checksum
- Message code received non-existent
- Incorrect TX identification code
- Message queue not received
- Writing failed
- Incorrect writing address
- So on…

5.2 CAN Interface

The task of the module is to provide a uniform interface to the CAN network.
The CAN Communication Stack supports classic CAN 2.0.

5.2.1 FW structure

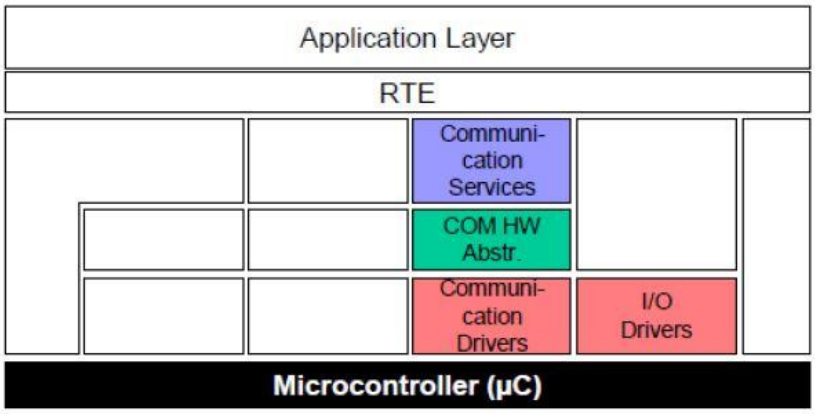


Figure 5.22
CAN – Communication Stack

For the CAN network the same approach was taken into account, in fact even the structure of the FW is very similar to that of serial communication.

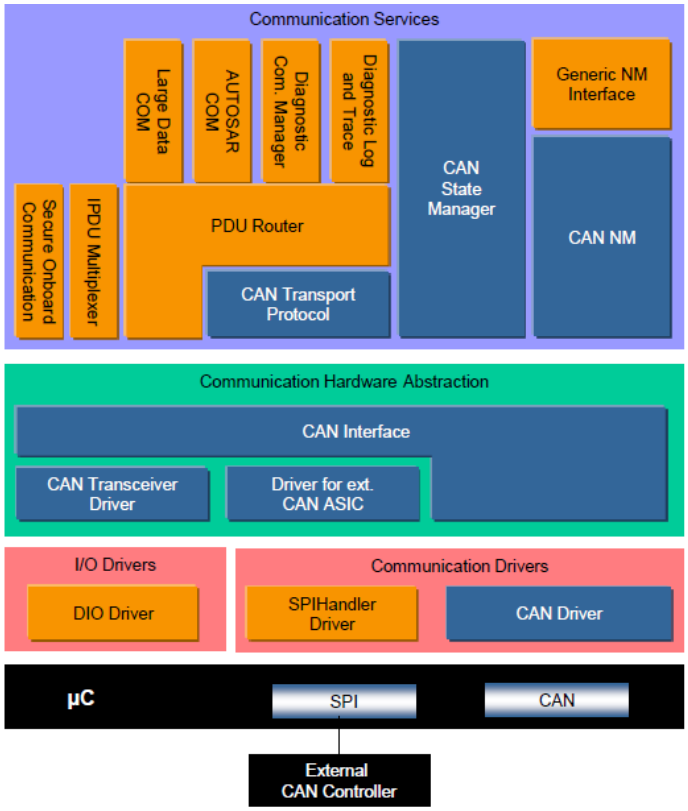


Figure 5.23

CAN – Layered Structure Example

Considering the Figure 5.22, it is possible to refer to the description of page 37. The difference is that in this case, at the communication drivers level, there are the drivers also for the I/O.

5.2.2 Communication Drivers, CAN

In the Figure 5.24 is shown a flowchart of the code related to the CAN communication drivers, again here, a first macro module related to the CAN configuration and another one to the Kernel composes the structure.

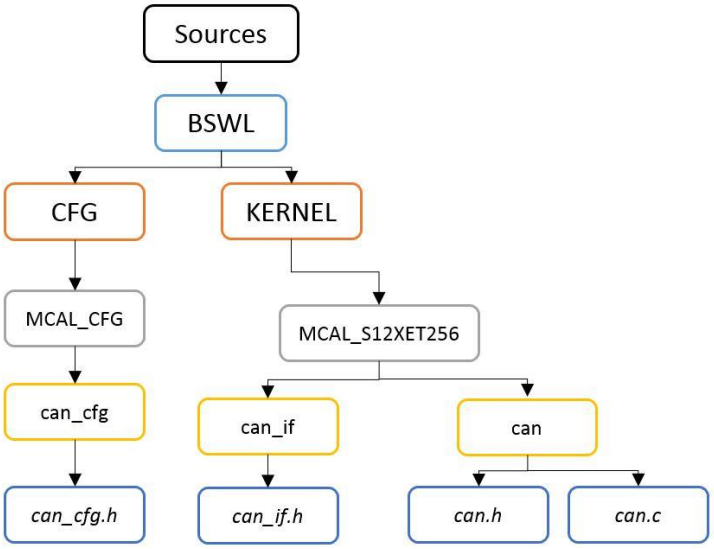


Figure 5.23
Communication Drivers, CAN

In the next sections the most important part of the code related to the CAN Network will be shown.
It must be underlined that the ECU provides two CAN connection (CAN0 and CAN1) and both of them have been developed and configured. The test was done on both lines as well.

can_cfg.h

This header file is related to the configuration of the CAN parameters such as Control Register 0 and 1, Bus Timing Register for CAN0 and CAN1.

Here there are several declarations related to the data rate and characteristics of the CAN message (info like standard/extended ID…).
Thanks to these *#defines* the setting of the parameters is facilitated. Here too, if there were variations in the HW, there would be no problem as the configuration file would be the only one to be modified: the rest of the code adapts to the different pinout.

```
#define DATA_FRAME          0       /* DATA frame                          */
#define REMOTE_FRAME        1       /* REMOTE frame                        */
#define EXTENDED_FRAME_ID   2047    /* max value of standard frame         */
#define STANDARD_FORMAT     0       /* ID standard frame (11 bit)          */
#define EXTENDED_FORMAT     1       /* ID extended frame (29 bit)          */


#define CAN0_Control_Register_0 0x01

/* CAN0CTL0 = 0x01;             // Inizialization Mode Request
                                //
                                // 0b00000001
                                //   ||||||||_ Enter Initialization Mode
                                //   |||||||__ Sleep Mode Request bit
                                //   ||||||___ Wake-Up disabled
                                //   |||||____ Time stamping disabled
                                //   ||||_____ Synchronized Status
                                //   |||______ CAN not affected by Wait
                                //   ||_______ Receiver Active Status bit
                                //   |________ Received Frame Flag bit
 */

 #define CAN0_Control_Register_1 0xC0

 /* CAN0CTL1 = 0xC0;                // Enable MSCAN Module
                                    //
                                    // 0b11000000
                                    //   ||||||||_ Initialization Mode Acknowledge
                                    //   |||||||__ Sleep Mode Acknowledge
                                    //   ||||||___ Wake-up low-pass filter disabled
                                    //   |||||____ BORM (Bus-Off Recovery Mode)
                                    //   ||||_____ Listen Only Mode disabled
                                    //   |||______ Loop Back Mode enabled
                                    //   ||_______ Bus clock as Clock Source
                                    //   |________ MSCAN Module enabled
 */

#define CAN0_Bus_Timing_Register_1 0x58

/* CAN0BTR1 = 0x58;             // 1 sample per bit, TSEG1 = 8 (9 Tq) e TSEG2 = 5 (6 Tq)
                                //
                                // 0b01011000
                                //   || ||  |_
                                //   || ||    |
                                //   || ||    |- TSEG1 = 8 -> 9 Tq
                                //   || |||___|
                                //   || |____
                                //   ||      |_ TSEG2 = 5 -> 6 Tq
                                //   ||_______|
                                //   |__________ 1 sample per bit

*/
#define CAN1_Control_Register_0 0x01
```

Figure 5.24

can_cfg.h (CFG)

can_if.c

The *can_if*.c file is used to define the structure of a CAN message in terms of Priority, Flag, Length, Data and ID, Pointer to the entry and exit of a circular buffer.

```
/* CAN0 */
typedef struct
{
    union
    {
        uchar B;
        struct
        {
            uchar priority  :3;     /* Priority CAN Message                          */
            uchar flag      :1;     /* Flag for the identification of the cell state
                                       1=full cell / 0=empty cell                    */
            uchar length    :4;     /* ID CAN Message                                */
        } b;
    } pfl;
    byte  message_p;                /* Real Priority on value between 0-255, in this case  */
    byte  data[8];                  /* CAN data message is not enought                     */
    ulong ID;                       /* ID CAN Message                                      */
} CAN0struct;


typedef union
{
    uchar B;
    struct
    {
        uchar pOut   :4;            /* Pointer to the exit of circular buffer       */
        uchar pIn    :4;            /* Pointer to the entry of circular buffer      */
    } b;
} CAN0ptr_TYPE;



/* CAN1 */

typedef struct
{
    union
    {
        uchar B;
        struct
        {
            uchar priority  :3;     /* Priority CAN Message                          */
            uchar flag      :1;     /* Flag for the identification of the cell state
                                       1=full cell / 0=empty cell                    */
            uchar length    :4;     /* CAN Message lenght                            */
        } b;
    } pfl;
    byte  message_p;                /* Real Priority on value between 0-255, in this case  */
    byte  data[8];                  /* CAN data message is not enought                     */
    ulong ID;                       /* ID CAN Message                                      */
} CAN1struct;
```

Figure 5.25

CAN structure (can_if)

After the definition of variables and constant as shown in the next Figure, the

```
/* CAN0 */

SCOPE CAN0struct     CAN_QueueCAN0[CAN0_QUEUE_LENGHT];
SCOPE CAN0ptr_TYPE   CAN_PointerCAN0;
SCOPE byte           u8CAN_TokenBufferCAN0;
SCOPE ulong          CAN0IDRx;


SCOPE uint   CAN0IDRxPGN;
SCOPE byte   CAN0IDRxPriority;
SCOPE byte   CAN0IDRxSend;
SCOPE byte   CAN0BuffRx[8];


/* CAN1 */

#pragma DATA_SEG __GPAGE_SEG CAN_PLACEMENT_RAM

SCOPE CAN1struct     CAN_QueueCAN1[CAN1_QUEUE_LENGHT];
SCOPE CAN1ptr_TYPE   CAN_PointerCAN1;
SCOPE byte           u8CAN_TokenBufferCAN1;
SCOPE ulong          CAN1IDRx;


SCOPE uint   CAN1IDRxPGN;
SCOPE byte   CAN1IDRxPriority;
SCOPE byte   CAN1IDRxSend;
SCOPE byte   CAN1BuffRx[8];
```

Figure 5.26a

Variables and constants (can_if)

function with their prototypes are defined here (Figure 5.26 b).

```
/* CONST */

/*****************************************************************************
*  Functions
*****************************************************************************/

/* CAN0 */
extern void CAN_InitCAN0(byte, byte, byte *far, byte *far);

extern void CAN_TransmissionCAN0(void);
extern void CAN_EnableInterruptCAN0(void);


extern void CAN_Intc_Int_TX_CAN0(void);
extern void CAN_Intc_Int_RX_CAN0(void);



#pragma CODE_SEG CAN_PLACEMENT_CODE

/* CAN1 */

extern void CAN_InitCAN1(byte, byte, byte *far, byte *far);

extern void CAN_TransmissionCAN1(void);
extern void CAN_EnableInterruptCAN1(void);

extern void CAN_Intc_Int_TX_CAN1(void);
extern void CAN_Intc_Int_RX_CAN1(void);
```

Figure 5.26b

Functions (can_if)

Each function must have the keyword *extern*, in this way it is available globally throughout the function execution.

The functions *CAN_Intc_Int_TX_CAN0, CAN_Intc_Int_RX_CAN0* (in this case CAN0 is taken as an example, but also applies to CAN1) are Interrupt Function related to the transmission and the reception of CAN messages.
They are defined in *can.c* file and they contain a call to another function defined in a higher level of abstraction (*MULcan.c)*, which handles the buffer initialisation, the pointer and so on.
Every time there is a transmission or reception of a CAN message, a proper Interrupt comes in play.
The Figure 5.27 shows a portion of code where it is done a change-name, it is necessary in order to use the function in orange (defined in *can.c*) inside the interrupt function in the *intc.h* file.

```
/*****************************************************************************
*  SWC Inputs definition
*****************************************************************************/

#define Intc_SCI_IntSchedulerSCI1()              SCI_IntSchedulerSCI1()

#define Intc_CAN_IntCAN0_TX()                    CAN_Intc_Int_TX_CAN0()
#define Intc_CAN_IntCAN0_RX()                    CAN_Intc_Int_RX_CAN0()
#define Intc_CAN_IntCAN0_ERR()                   Int_CAN_0_ERR()
#define Intc_CAN_IntCAN0_EGR_READ()              CAN_EGR_Read_Position_CAN0()


#define Intc_CAN_IntCAN1_TX()                    CAN_Intc_Int_TX_CAN1()
#define Intc_CAN_IntCAN1_RX()                    CAN_Intc_Int_RX_CAN1()
#define Intc_CAN_IntCAN1_ERR()                   Int_CAN_1_ERR()
```

Figure 5.27

CAN RX/ CAN TX Interrupt (*intc.h*)

```
#pragma CODE_SEG INTC_PLACEMENT_CODE

void interrupt Int_CAN_0_TX(void)
{
byte AppGPAGE;
    byte AppRPAGE;
    byte AppEPAGE;

    AppGPAGE = GPAGE;
    AppRPAGE = RPAGE;
    AppEPAGE = EPAGE;

    Intc_CAN_IntCAN0_TX();


    GPAGE = AppGPAGE;
    RPAGE = AppRPAGE;
    EPAGE = AppEPAGE;
}
```

Figure 5.28

Int_CAN_0_TX (void) (*intc.c*)

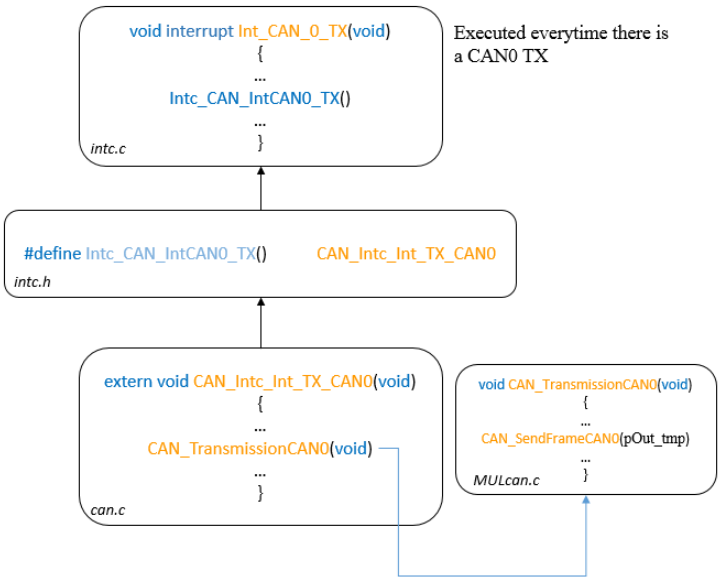


Figure 5.29

Cascade of calls for

the interrupt associated to CAN0 communication

can.c

The CAN Component kernel has the definition of all the variables buffers, used in the sub-modules related to the MCAL_S12XET256, necessary for the communication handled in the MULcan SW group.

```
ulong   CAN0IDTx         = 0;       /*   Identification Tx Message (no control bit)      */
ulong   CAN1IDTx         = 0;       /*   Identification Tx Message (no control bit)      */

byte    CAN0FrameTypeTx = 0;
byte    CAN1FrameTypeTx = 0;
                                   /*   Tx Type:
                                        0 - DATA_FRAME   -> Data Frame
                                        1 - REMOTE_FRAME -> Remote Frame
                                   */
byte    CAN0LengthTx     = 0;      /*   Transmitted data lenght (in byte)               */
byte    CAN1LengthTx     = 0;

byte    CAN0FrameTypeRx = 0;
byte    CAN1FrameTypeRx = 0;       /*   Rx Type:
                                        0 - DATA_FRAME   -> Data Frame
                                        1 - REMOTE_FRAME -> Remote Frame               */

byte    CAN1FrameFormRx = 0;
byte    CAN0FrameFormRx = 0;       /*   Rx Form:
                                        0 - STANDARD_FORMAT -> 11 bit ID
                                        1 - EXTENDED_FORMAT -> 29 bit ID               */

byte    CAN0BuffRx[8]   = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};     /* CAN0 buffer Rx */
byte    CAN0BuffRxMem[8] = {0,0,0,0,0,0,0,0};

byte    CAN1BuffRx[8]   = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};     /* CAN1 buffer Rx */
byte    CAN1BuffRxMem[8] = {0,0,0,0,0,0,0,0};

extern ulong CAN0IDRx   = 0;                                                    /* Rx Message Identification (no control bit) */
extern ulong CAN1IDRx   = 0;                                                    /* Rx Message Identification (no control bit) */

uint    CAN0IDRxPGN      = 0;                                                   /* PGN                                        */
byte    CAN0IDRxPriority = 0;                                                   /* Priority                                   */
byte    CAN0IDRxSend     = 0;                                                   /* Identify who is sending the message        */
```

Figure 5.30

Variables and buffer (*can.c*)

After that, the initialization function is defined for the CAN0 and CAN1 in a very similar way: the fundamental registers necessary for the activation mode request are set, together with the configuration of baud rate and, filters, bus timing register.

```
/*****************************************************************************
*
*  Function:    CAN_InitCAN0
*  Description: CAN_initCAN0
*
*****************************************************************************
*/

#pragma CODE_SEG OTHER_ROM

void CAN_InitCAN0(byte Baud, byte FilterType, byte *far FilterIDA, byte *far FilterIDM)
{
    CAN0CTL0 = CAN0_Control_Register_0;             /*    Initialization Mode Request                          */

    while (!(CAN0_INIZIALIZATION_MODE1))            /*    Waiting for ACK in Inizialization Mode               */
    {}

    CAN0CTL1 = CAN0_Control_Register_1;             /*    Enable MSCAN module                                  */

    //CAN0TXERR0ld = CAN0TXERR;                     /*    Memorize counter CANTXERR for testing                */
                                                    /*    during hook if there is a Tx error                   */

    CAN0BTR0 &= CAN_RESET_BAUD;                     /*    Reset Baud bit                                       */

    CAN0BTR0 |= Baud;

    CAN0BTR1 = CAN0_Bus_Timing_Register_1;          /*    1 sample per bit, TSEG1 = 8 (9 Tq) e TSEG2 = 5 (6 Tq) */

    CAN0IDAC &= CAN_RESET_FILTER;                   /*    Bit reset to set with FilterType                     */

    CAN0IDAC |= FilterType;


    CAN0IDAR0 = CAN0FiltriIDA[0];                       /*    ID Filter                                        */
    CAN0IDAR1 = CAN0FiltriIDA[1];
    CAN0IDAR2 = CAN0FiltriIDA[2];
    CAN0IDAR3 = CAN0FiltriIDA[3];

    CAN0IDMR0 = CAN0FiltriIDM[0];                       /*    ID Filter Mask (1 pass all, 0 filter activation) */
    CAN0IDMR1 = CAN0FiltriIDM[1];
    CAN0IDMR2 = CAN0FiltriIDM[2];
    CAN0IDMR3 = CAN0FiltriIDM[3];

    CAN0IDAR4 = CAN0FiltriIDA[4];                       /*    ID Filter                                        */
    CAN0IDAR5 = CAN0FiltriIDA[5];
    CAN0IDAR6 = CAN0FiltriIDA[6];
    CAN0IDAR7 = CAN0FiltriIDA[7];

    CAN0IDMR4 = CAN0FiltriIDM[4];                       /*    ID Filter Mask (1 pass all, 0 filter activation  */
    CAN0IDMR5 = CAN0FiltriIDM[5];
    CAN0IDMR6 = CAN0FiltriIDM[6];
    CAN0IDMR7 = CAN0FiltriIDM[7];



    CAN0CTL0 = CAN_END_REQUEST;                     /*    Initialization Mode End Request                      */

    while ((CAN0_INIZIALIZATION_MODE1) != 0)        /*    Waiting for ACK in Inizialization Mode               */
    {}

    CAN0RIER |= CAN0_ABILITAZION_STATUS_CHANGE_INT; /*    CAN Status Change Interrupt (CSCIE=1)                */
                                                    /*    Interrupt every status change (TSTATE1-0 = 11)       */
    CAN0RIER |= 0x01;                               /*    Enable Interrupt CAN RX                              */

}
```

Figure 5.31

CAN0 Init (*can.c*)

In the Figure 5.31 it is shown as an example the configuration of CAN0, in the case of CAN1 the structure is the same but the only difference is obviously related to the configuration of the communication channel.

When the setting is complete, it is time to manage the real communication: for both CAN lines, there are two function in order to manage the reception and transmission of data.

In the Figure 5.32 it is shown the function related to the frame reception both for Remote/Data and Standard/Extended frame.

It receives:

- Message IDRx

  Pointer to message received ID

- FrameTypeRx

  Pointer to the frame type: DATA Frame or REMOTE Frame

- LenghtRx

  Pointer to the data length in byte (REMOTE DATA Frame: 0)

- DataRx

  Pointer to the received buffer

```
#pragma CODE_SEG OTHER_ROM

byte CAN_ReceiveFrameCAN0(ulong *far MessageIDRx, byte *far FrameTypeRx, byte *far FrameFormRx, byte *far LengthRx, byte *far DataRx)
{
    byte i, ris;
    ulong ID;

    ID = *(ulong *far)CAN0RXIDR_ARR;                    /* read ID message                          */
    if ((*(ulong *far)CAN0RXIDR_ARR & 0x00080000))      /* head if standard frame or extended (IDE) */
    {
        // Extended Frame
        ID = ( ((ID >> 1) & 0x0003FFFF) | ((ID >> 3) & 0xFFFC0000) );              /* Prepare ID Rx         */
        *FrameTypeRx = (*(ulong *far)CAN0RXIDR_ARR & 1)? REMOTE_FRAME : DATA_FRAME;  /* Load frame type       */
        *FrameFormRx = EXTENDED_FORMAT;                                           /* Load Frame Format     */
    }
    else
    {   // Standard Frame
        ID >>= 21;                                                                          /* Prepare ID Rx preparo ID Rx */
        *FrameTypeRx = (*(ulong *far)CAN0RXIDR_ARR & 0x00100000)? REMOTE_FRAME : DATA_FRAME; /* Load type frame             */
        *FrameFormRx = STANDARD_FORMAT;                                                     /* Load format frame           */
    }

    *MessageIDRx = ID;                              /* Transfer ID RX message  */
    *LengthRx = CAN0RXDLR & 0x0F;                   /* Transfer lenght         */

    if (*FrameTypeRx == DATA_FRAME)
        for(i=0; i<*LengthRx; i++)
            *(DataRx +i) = *(CAN0RXDSR_ARR +i);     /* Change due to different definition of CAN0 register of the uC */
            //DataRx[i] = CAN0RXDSR0[i];            /* Transfer data (if frame)                                      */

    CAN0RFLG |= 0x01;                               /* ACK reception flag                                            */
    ris = CAN_RX_OK;                                /* Rx OK                                                         */
    return ris;
}
```

Figure 5.32

CAN_ReceiveFrameCAN0 (*can.c*)

The preparation of the message to be sent is done with a proper function in the same SW module; all the details and information can be set. If it is necessary it is possible to set the priority of the message as well.

```
#pragma CODE_SEG OTHER_ROM

void CAN_SendFrameCAN0(byte pointer)
{
    uchar i,txbuffer;                           /* Index i and variable for the number of selected tx buffer */
    ulong CAN0IDTx = CAN_QueueCAN0[pointer].ID;  /* Temporary variable for shift                              */

    CAN0TBSEL = CAN0TFLG;                       /* Check first empty tx buffer                                */
    txbuffer = CAN0TBSEL;                       /* Backup selected buffer                                     */

    if (CAN0IDTx <= EXTENDED_FRAME_ID)          /* Standard Frame or Extended                                 */
    {
        *(ulong *far)CAN0TXIDR_ARR = (CAN0IDTx<<21); /* Prepare ID standard frame                             */
    }
    else                                        /* Prepare ID extended frame                                  */
    {
        *(ulong *far)CAN0TXIDR_ARR = ( ((CAN0IDTx<<1) & 0x0007FFFF)|((CAN0IDTx<<3) & 0xFFE00000))| 0x00180000;
    }

    for(i=0;i<CAN_QueueCAN0[pointer].pfl.b.length;i++)
    {
        *(CAN0TXDSR_ARR + i)=CAN_QueueCAN0[pointer].data[i];
    }
                                                /* Load datas                                                 */
    CAN0TXDLR = CAN_QueueCAN0[pointer].pfl.b.length;  /* Load Tx data lenght                                  */
    //CAN0TXTBPR = CAN_QueueCAN0[pointer].pfl.b.priority;  /* Usually set at 0, max priority                 */

    CAN0TXTBPR = CAN_QueueCAN0[pointer].message_p;

    CAN0TFLG = txbuffer;                        /* Start Transmission                                         */

}
```

Figure 5.33

CAN_SendFrameCAN0 (*can.c*)

As mentioned in the page 59, the *can.c* file contains the definition of the function that enables the interrupt for the beginning of the communication (i.e. Figure 5.34).

```
void CAN_EnableInterruptCAN0(void)
{

    if (CAN0TXBusy == CAN_TX_OK)
    {
        CAN0TXBusy = CAN_BUSY;              /* Communication token CAN0 TX  */
        CAN0TIER |= 0x01;                   /* Enable interrupt             */
    }
}
```

Figure 5.34

CAN_EnableInterruptCAN0 (*can.c*)

CAN0TXBusy is a Semaphore to manage the transmission,

- 0 → Transmission ok
- 1 → Wait

The CAN0TIER register is held in the reset state when the initialization mode is active. Putting 0x01 in this register is an event that causes a interrupt request that starts TX.

5.2.3 Communication Hardware Abstraction, CAN

This section of the paragraph is reserved to the Communication Hardware Abstraction.

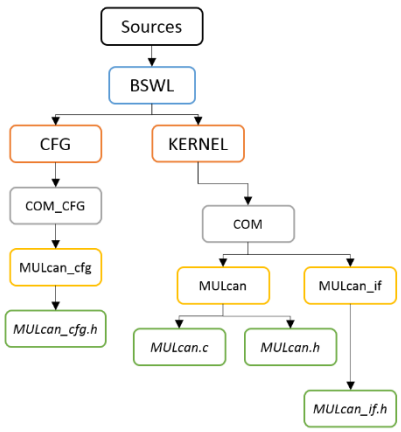


Figure 5.35

Communication Hardware Abstraction, CAN

MULcan.c

The load of data on the CAN network is carried out using a queue with circular buffer of length CAN0_QUEUE_LENGHT (in this case 8); the variable advancing in the array is identified with "b".

```
void MULcan_QueueLoadCAN0(ulong msg_ID, uchar *far b)
{
    u8CAN_TokenBufferCAN0 = 1;

    if ((CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].pfl.b.flag == 0))
    {
        /* Put token when buffer is loading so it's not possible to call interrupt from base timer   */

        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].ID = msg_ID;
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].pfl.b.length = 8;
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].pfl.b.priority = 0;
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].pfl.b.flag = 1;
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[0] = (uchar)(*b);
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[1] = (uchar)(*(b+1));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[2] = (uchar)(*(b+2));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[3] = (uchar)(*(b+3));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[4] = (uchar)(*(b+4));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[5] = (uchar)(*(b+5));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[6] = (uchar)(*(b+6));
        CAN_QueueCAN0[CAN_PointerCAN0.b.pIn].data[7] = (uchar)(*(b+7));

        if (CAN_PointerCAN0.b.pIn >= 7)      /* Eliminating messages, putting anomaly */
        {
            CAN_PointerCAN0.b.pIn=0;
        }
        else
        {
            CAN_PointerCAN0.b.pIn++;
        }
    }

    u8CAN_TokenBufferCAN0 = 0;

    /* Try to send message, otherwise there's the function at 1 ms tast      */

    CAN_EnableInterruptCAN0();
}
```

Figure 5.36

MULcan_QueueLoadCAN0 (*MULcan.c*)

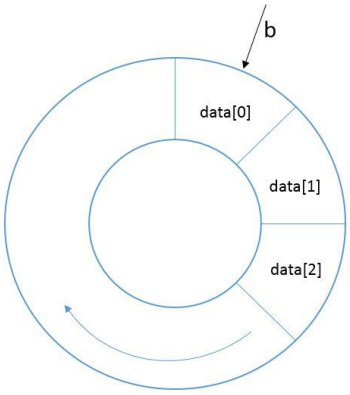


Figure 5.37

Circular Buffer for CAN Load

It is developed a version of the queue with the possibility to set the priority.

```
void CAN_TransmissionCAN0(void)
{
    byte i;
    byte pOut_tmp = 0;                      /* temporary variable to handle message transmission and  */
                                            /* messages with the same interrupt                       */

    pOut_tmp = CAN_PointerCAN0.b.pOut;      /* Current value of pOut pointer                          */


    if (CAN_QueueCAN0[pOut_tmp].pfl.b.flag == 1)
    {
        CAN_SendFrameCAN0(pOut_tmp);        /* CAN message transmission                               */

        CAN_QueueCAN0[pOut_tmp].pfl.b.flag = 0; /* No sent message                                    */

        CAN_QueueCAN0[pOut_tmp].message_p = 0;  /* Reset message priority not used                    */

        for (i=0;i<8;i++)                   /* Buffer initialisation                                  */
        {
            CAN_QueueCAN0[pOut_tmp].data[i] = 0xAA;
        }
        if (pOut_tmp >= 7)
        {
            pOut_tmp=0;                     /* Circual Buffer                                         */
        }
        else
        {
            pOut_tmp++;
        }
    }
    while ((CAN_QueueCAN0[pOut_tmp].pfl.b.flag == 1) && ((CAN0TFLG & 7) != 0))
    {
        /* If there are message to send and free trasmission */

        CAN_SendFrameCAN0(pOut_tmp);        /* CAN message transmission                               */

        CAN_QueueCAN0[pOut_tmp].pfl.b.flag = 0; /* No sent message                                    */

        CAN_QueueCAN0[pOut_tmp].message_p = 0;  /* Reset message priority not used                    */

        for (i=0;i<8;i++)                   /* Buffer initialization                                  */
        {
            CAN_QueueCAN0[pOut_tmp].data[i] = 0xAA;
        }
        if (pOut_tmp >= 7)                  /* Circual Buffer                                          */
        {
            pOut_tmp=0;
        }
        else
        {
            pOut_tmp++;
        }
    }

    CAN_PointerCAN0.b.pOut=pOut_tmp;        /* Reactivation of pOut after transmission of multiple
                                               messages with the same interrupt                       */
}
```

Figure 5.38

CAN_TrasmissionCAN0 (*MULcan.c*)

The transmission on CAN0/1 is handled by the function shown in the Figure 5.38.

5.2.4 Complex Device Driver

The Complex Device Driver accesses directly to the MCU (strong dependency) and reaches the RTE. It is used when there are functions that must be implemented

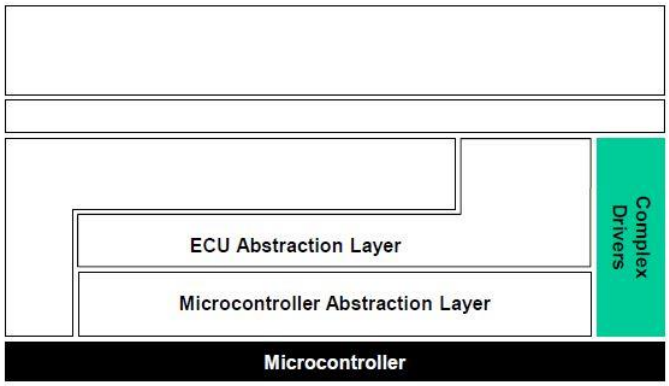


Figure 5.39

Complex Drivers Layer

and it is not possible to find them in other layers; in particular meets the need to operate with complex actuators and sensors.

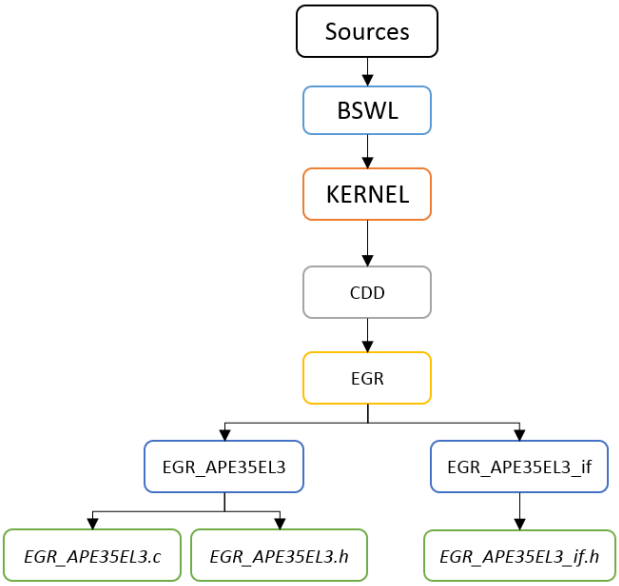


Figure 5.40
Complex Drivers Code Structure

This layer was used to develop the code for the EGR valve APE35EL2: this was used as an actuator to verify the correct operation of the CAN0 and CAN1 network. The valve opening percentage (the operational range is 0 to 100%) is sent on the CAN network, the reply message is interpreted to assess whether the correct value has been received.

5.2.5 EGR Valve

The valve used as an actuator for the network is a control system to manage the NOx emission generated during the combustion process in the engine cylinders.
The EGR reduces the combustion temperature bringing back part of the exhausted gas to the intake manifold, in this way the air is diluted.
Opening and closing is carried out via a computer in relation, generally, to the temperature detected by sensors.

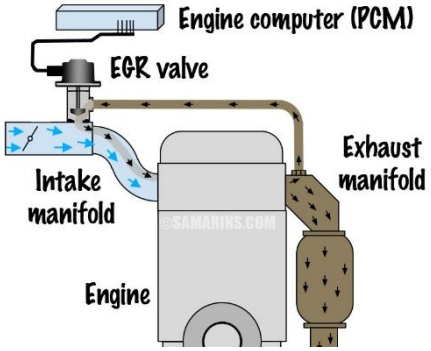


Figure 5.41

EGR Valve

The position control message is structured on 8 bytes and has a specific ID; the idea is therefore to replicate this frame.

ID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
18FDD500 h	**FF**	**FF**	**FF**	**FF**	**Valve Control**	**Valve Control**	**FF**	**FF**

Table 5.4

Control of EGR valve position

All the code is structured in the file *EGR_APE35EL3.c* with all the variables and the implementation of reading and sending function.

```
void CAN_EGR_PositionCAN0(byte EGR_Desired_Position)
{

    uint value;

    value= (uint)EGR_Desired_Position*400;

    EGR_CAN0IDTx= 0x18FDD500 ;

    EGR_CAN0BuffTx[0] = (byte)(0xFF);
    EGR_CAN0BuffTx[1] = (byte)(0xFF);
    EGR_CAN0BuffTx[2] = (byte)(0xFF);
    EGR_CAN0BuffTx[3] = (byte)(0xFF);
    EGR_CAN0BuffTx[4] = (byte)(value);
    EGR_CAN0BuffTx[5] = (byte)(value>>8);
    EGR_CAN0BuffTx[6] = (byte)(0xFF);
    EGR_CAN0BuffTx[7] = (byte)(0xFF);
    MULcan_QueueLoadCAN0(EGR_CAN0IDTx,EGR_CAN0BuffTx);
}
```

Figure 5.42

CAN_EGR_PositionCAN0 (*EGR_APE35EL3.c*)

In position 4 and 5 the hexadecimal (inverted) value of the valve opening percentage is entered.

The value is set in the debug phase and the variable is calculated as the multiplication of the entered value by 400.
This is done considering the examples on the actuator's datasheet: 100% open corresponds to the message with

- ID : 18FDD500 h
- Bytes:

FF	FF	FF	FF	**40**	**9C**	FF	FF

Hex: 9C40 → **Dec:** 40000

$$\frac{40000}{0.0025} = 100$$

Therefore, the desired value must be multiplied by 400 ($\frac{1}{0.0025}$).

The valve communication occurs through J1939 message; the details are presented in the following section.

```
void CAN_EGR_Read_Position_CAN0()
{


  if(CAN0IDRx == 0x18FD9422)

    {
       EGR_CAN0_Conversion =  CAN0BuffRx[1]<<8;
       EGR_CAN0_Conversion |= CAN0BuffRx[0];

       EGR_CAN0_Conversion = EGR_CAN0_Conversion/400;
    }


}
```

Figure 5.43

CAN_EGR_Read_Position_CAN0 (*EGR_APE35EL3.c*)

5.2.6 SAE J1939

The thesis path included the study of the SAE J1939 standard and the implementation of the relative SW module to be able to manage the EGR valve.

SAE J1939 is a standard used in the communication and diagnosis of components in commercial (heavy-duty) vehicles such as trucks, buses but it involves also agriculture and maritime domains.

Thanks to J1939 it is offered the possibility to have a standard way to communicate across ECUs giving a manufacturer interoperability, in contrast with passenger cars that are strictly related to specific protocols (different for every manufacturer).

The standard uses the CAN technology: if the CAN is the tool, J1939 is the "language".

It is characterized by:

- 29-bit extended ID (CAN 2.0B)
- Baud Rate typically of 250 kbits/s, but it supports 500 kbits/s
- **PGN**: Parameter Group Number, 18 of 29 bit of ID
- **SPN**: Suspect Parameter Number, 19 bit number used for diagnostic purpose
- Broadcast or P2P Messages

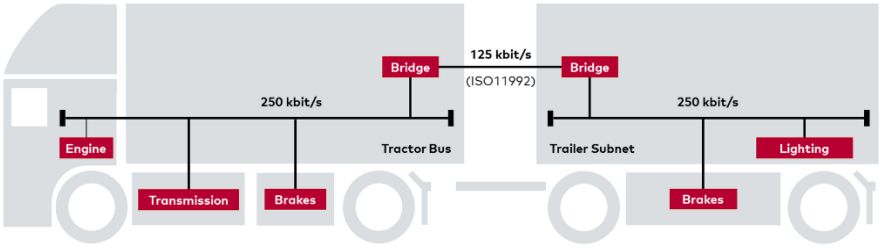


Figure 5.44
Typical J1939 Vehicle Network[4]

CAN Protocol provides to the standard the physical and data link layer of the ISO/OSI model, this means that on the bus there are small packets but a complex exchange of data is possible with a multi-packet messages, more than 8 bytes can be transferred.
The protocol specifies how to have a human-readable data by the conversion of starting data; to understand a J1939 message it is necessary to interpret its parameters.
The document J1939-71 gives all the information for the conversion of a large set of standardized J1939 message into readable data.

[4] https://www.vector.com/

PGN

Each frame it is characterized by a PGN that contains 8 bytes of data, divided into parameters (SPN).

The PGN is the ID of a J1939 message, it is unique and describes the function of the message and for looking up the SPN.

Consider the EGR valve and the PGN of the control message for the position as an example:

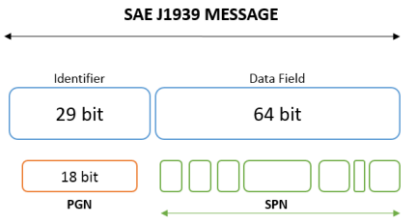


Figure 5.45

SAE J1939 Message

Control of EGR valve position (*PGN 64981 Electronic Engine Controller 5 EEC5)*

Transmission Repetition Rate: On request

Data Length: 8

Extended Data Page: 0

Data Page: 0

PDU Format: 253

PDU Specific: 213 PGN Supporting Information:

Default Priority: 6

Parameter Group Number: 64981 (0x00FDD5)

Start Position	Length	SPN	SPN Name
1-2	2 bytes	2789	Engine Turbocharger 1 Calculated Turbine Intake Temperature
3-4	2 bytes	2790	Engine Turbocharger 1 Calculated Turbine Outlet Temperature

5-6	2 bytes	2791	Engine Exhaust Gas Recirculation 1 Valve 1 Control 1
7.1	2 bytes	2792	Engine Variable Geometry Turbocharger (VGT) Air Control Shutoff Valve
7.3	2 bytes	5323	Engine Fuel Control Mode
7.5	2 bytes	5457	Engine Variable Geometry Turbocharger 1 Control Mode
8	1 bytes	2795	Engine Variable Geometry Turbocharger (VGT) 1 Engine Variable Geometry Turbocharger (VGT) 1 Actuator Position

Table 5.5

PGN 64981 Electronic Engine Controller 5 EEC5

The ID is 0x00FDD5, the PGN starts bit 9 with length 18, so in this case is 0x0FDD5, 64981 in decimal; looking for this value on the document SAE J1939-71 there will be *Electronic Engine Controller 5*. The documentation shows that there are seven SPNs related to this message.

Consider so that the PGN (64981, ID: 0x00FDD5) is identified; then the SPN 2791 is taken into account:

SPN 2791 - Engine Exhaust Gas Recirculation 1 Valve 1 Control 1

Desired percentage of maximum Exhaust Gas Recirculation (EGR) valve 1 opening. 0% means valve is closed. 100% means maximum valve opening (full gas flow).

Data Length: 8

Data Range: 0 to 100%

Data Page: 0

Type: Status

SPN: 2791

Parameter Group Number: 64981

The message received shown in the next Figure is used as an example of interpretation.
The document tells that the relevant data is in byte 5 (0x40) and 6 (0x1F):

Hex: 0x1F40 → **Dec:** 8000
In percentage, $\frac{8000}{400}$= 20% opening.

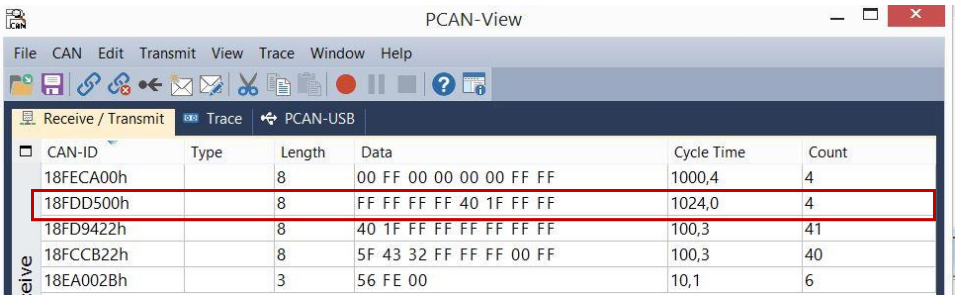


Figure 5.46
Example CAN Message

The standard J1939 allows messages longer than 8 bytes even if the CAN can support only eight byte of data transfers. A solution is to have a multiple packet communication: the J1939-21 defines all the guidelines to send and assemble the messages.
There are two types of possible transport:

- *Broadcast Announce Message (BAM)*
  It is a broadcast communication, the message by a sender is sent to all the other nodes. If something goes wrong, the receiver cannot reply or signal it.
  The sender is the only one that manage the data flow.
- *Peer-to-Peer*
  This protocol establish a sender-receiver communication. Here the receiver has the opportunity to influence the data flow and have the control on its data packets.

To send a multipacket data the sender first transmits a BAM, it is a sort of warning message for the entire network. The BAM contains the PGN, the size, number of packets and information to allocate the resources necessary for a correct message reassemble.

The SAE J1939 has specific parameters intended for diagnostic messages called DM. They give information about the state of the system, in terms of health, and about malfunctions that have occurred in the automotive system.

There are several DM in J1939, but during the thesis activity the attention was focused on DM1, DM2 and DM3.

- *DM1- Active Diagnostic Trouble Codes*
  Provides lamp status and DTC information, together report diagnostic condition of the electronic component.
  DM1 is transmitted, usually, every second and on change of the state.
  Given a = lamp status
  b = SPN
  c = FMI
  d = CM and OC
  The message format is : a,**b,c,d**,b,c,d,**b,c,d**,b,c,d…

- *DM2- Previously Active Diagnostic Trouble Codes*
  Provides a list of previously active diagnostic codes.

- *DM3 – Diagnostic Data Clear/Reset of Previously Active DTCs*
  When it is supported, it indicates that all the diagnostic information related to the previously active trouble codes should be cleared or reset in the case of non-active trouble codes.

5.2.7 Implementation

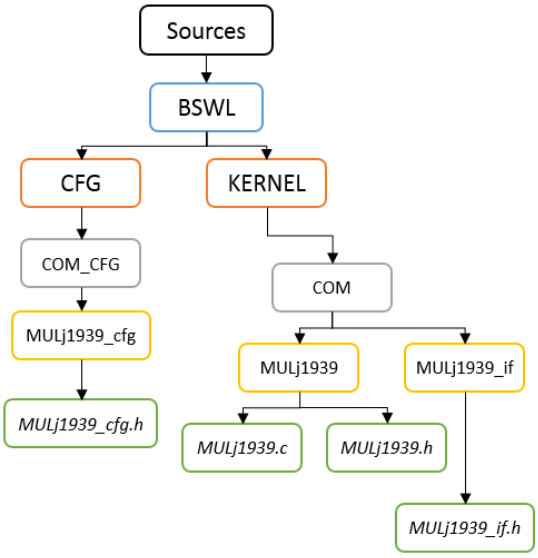


Figure 5.47

J1939 Code Structure

For the implementation of J1939 messages, it has been used the same code scheme as the previous modules: one part is linked to the configuration and one to the BSWL.

MULj1939_if.h

This module is dedicated to the definitions and typedef. In particular, the structs necessary for the lamp status definition (DM1) is shown below.

```
typedef struct
{
    byte IndicatorLamp  :1;
    byte RedLamp        :1;
    byte WarningLamp    :1;
    byte ProtectLamp    :1;
    byte bit4           :1;
    byte bit5           :1;
    byte bit6           :1;
    byte bit7           :1;
} StructLamp;


typedef struct
{
    uint        SPN;
    byte        FMI;
    StructLamp  Lamp;
} AnoDM1;
```

Figure 5.48

J1939 Lamp Struct

The components of lamp status are

- *Malfunction Indicator Lamp*
  Used for emission-related information
- *Red Stop Lamp*
  Used in a problematic scenario sever enough as to have the vehicle stop
- *Amber Warning Lamp*
  Used to report a vehicle problem but does not require a stop
- *Protect Lamp*
  Used to report a problem most probably not related to an electronic system.

Their role is to reflect the current state of the electronic component that is in TX. The second struct deals with the second part of the message form of DM1, SPN and FMI.

MULj1939.c

This file is reserved for the implementation of the DM1/2/3 messages with their relative timers and request functions, i.e.:

- void CAN0DM1(void);
- void MULj1939_MessaggioDM1NoFault(void);
- void MULj1939_MessaggioDM1Fault(byte);
- void MULj1939_MessaggioDM1BAM(void);
- void MULj1939_TrovaDM1Error(void);
- byte MULj1939_SPNFMI(byte);
- void MULj1939_Fault_Test(void);
- void J1939_Timer1ms(void);
- void J1939_Multi_Timer1ms(void);
- void CAN0DM2(void);
- void J1939_Multi_Timer1ms_DM2(void);

- void MULj1939_MessaggioDM2NoFault(void);
- void MULj1939_MessaggioDM2Fault(byte);

- void MULj1939_MessaggioDM2BAM(void);
- void MULj1939_TrovaDM2Error(void);

- void CAN0DM3(void);
- void DM3PCReset(void);

- void MULj1939_Request_CAN0(void);
- void MULj1939_Request_CAN1(void);

In the next figure is shown the array related to DM1 for every warning or fault, for the states in which one of them is not available the SPN is 65535 (0xFFFF).

```
//                                SPN    FMI  IndL RedL WarL ProL bit4, bit5, bit6, bit7
AnoDM1 MULj1939_TabAnoDM1[192]  =  {  0,      0,    0,   0,   0,   0,   0,   0,   0,   0,      // 0   - OK
                                      1350,   17,   1,   0,   1,   0,   0,   0,   0,   0,      // 1   - maintenance warning
                                      1350,    1,   1,   1,   0,   0,   0,   0,   0,   0,      // 2   - maintenance alarm
                                      81,      0,   1,   1,   0,   0,   0,   0,   0,   0,      // 3   - bypass mode
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 4   - backpressure latching
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 5   - P1 high warning
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 6   - P1 high alarm
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 7   - free
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 8   - free
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 9   - free
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 10  - SLI60 low
                                      4783,    1,   1,   0,   1,   0,   0,   0,   0,   0,      // 11  - SLI300 low
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 12  - SLI60 high
                                      4783,   16,   1,   1,   0,   0,   0,   0,   0,   0,      // 13  - SLI300 high
                                      65535, 255,   1,   1,   1,   1,   1,   1,   1,   1,      // 14  - SLI60 clog up level
                                      4783,    0,   1,   1,   0,   0,   0,   0,   0,   0,      // 15  - SLI300 clog up level
                                      3699,    1,   0,   0,   0,   1,   0,   0,   0,   0,      // 16  - TC low
```

Figure 5.49

Part of DM1 Table

The function related to the DM1 implementation (*void CAN0DM1(void)*), first checks the variables that depend on the situation in which the vehicle is located, to then read the anomalies taking care that *!((MULj1939_TabAnoDM1[i].SPN == 65535)* because not supported.

```
if (StatusDM1 == 0)                                 /* If I send a status message, everything is ok */
{
    MULj1939_MessaggioDM1NoFault();                 /* I compose the message   */

    ContTaskCAN0DM1 = 0;                            /* Relaunch the task timer */
}

else if (StatusDM1 == 1)                            /* If a single message     */
{
    for(i=1; (i<192); i++)                          /* It starts from 1 because with i = 0 it is all OK */
    {
        if (StatoDM1[i] == 1)                       /* Anomaly set  */
        {
            indice = i;
            i = 192;                                /* Anomaly found  */
        }
    }

    MULj1939_MessaggioDM1Fault(indice);             /* I compose the message   */

    ContTaskCAN0DM1 = 0;                            /* Relaunch the task timer */

}

else
{

    /* Calculate how many messages I have to send

       1 byte message number
       2 byte status lights
       2 bytes SPN
       1 FMI byte
       1 byte Occurrance

       common parts 2 bytes (lights state)
       parts per message 5 bytes (messenger number, SPN, FMI, Occurrance)

     */

    if (TxDM1 == 0)                                 /* First lap in the state machine for DM1 multi frame  */
    {
        NumMessaggi = (byte)((((uint)StatusDM1 * 4) + 2) / 7);

        if ((((StatusDM1 * 4) + 2) % 7) > 0)
        {
            NumMessaggi = NumMessaggi + 1;
        }

        ContTaskCAN0DM1 = 0;                        /* Relaunch the multi frame task timer   */

        StDM1 = 0;                                  /* Clear the state machine               */
        PosizStatoDM1 = 0;                          /* Reset error position on StatoDM1 []    */
        PosizMessDM1 = 1;

    }


TxDM1 = 1;                                      /* Active multi frame transmission           */
                                                /* I send an opening message                */
```

Figure 5.50

MULj1939.c, Part 1

Here the DM1 message is composed in relation to the value of the variable *StatusDM1*, based on its value the functions No-Fault or Fault are called.

```
void MULj1939_MessaggioDM1NoFault(void)
{

    CANIDTx = 0x18FECA00 | ((ulong)SourceAddress_E);
                                                    /* Load ID per message
                                                       Priority 6 (lower) 18
                                                       PGN      FECA
                                                       Source Address SA */

    CANBuffTx[0] = 0x00;                            /* Lamp status: all off */
    CANBuffTx[1] = 0xFF;
    CANBuffTx[2] = 0x00;
    CANBuffTx[3] = 0x00;
    CANBuffTx[4] = 0x00;
    CANBuffTx[5] = 0x00;
    CANBuffTx[6] = 0xFF;
    CANBuffTx[7] = 0xFF;

    if (FlagCANBroadOUT_E == 0)
    {
        MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
    }
    else if (FlagCANBroadOUT_E == 1)
    {
        MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
    }
}
```

Figure 5.51

MULj1939.c, NoFault

```
void MULj1939_MessaggioDM1Fault(byte indice)
{

    byte appoggio = 0;

    CANIDTx = 0x18FECA00 | ((ulong)SourceAddress_E);
                                            /* Load ID per message
                                               Priority 6 (lower) 18
                                               PGN    FECA
                                               Source Address SA */

    appoggio = (byte)(MULj1939_TabAnoDM1[indice].Lamp.IndicatorLamp) << 6;
    appoggio = appoggio  + ((byte)(MULj1939_TabAnoDM1[indice].Lamp.RedLamp) << 4);
    appoggio = appoggio  + ((byte)(MULj1939_TabAnoDM1[indice].Lamp.WarningLamp) << 2);
    appoggio = appoggio  + (byte)(MULj1939_TabAnoDM1[indice].Lamp.ProtectLamp);


    CANBuffTx[0] = appoggio;                /* Lamp status: all off */
    CANBuffTx[1] = 0xFF;

    appoggio = (byte)(MULj1939_TabAnoDM1[indice].SPN);

    CANBuffTx[2] = appoggio;                /* SPN low byte */

    appoggio = (byte)(MULj1939_TabAnoDM1[indice].SPN >> 8);

    CANBuffTx[3] = appoggio;                /* SPN high byte */
    //CANBuffTx[4] = TabAnoDM1[indice].FMI; /* FMI */
    CANBuffTx[4] = MULj1939_SPNFMI(indice);
    CANBuffTx[5] = 1;                       /* occurence value not present */
    CANBuffTx[6] = 0xFF;                    /* not used   */
    CANBuffTx[7] = 0xFF;                    /* not used   */

    if (FlagCANBroadOUT_E == 0)
    {
        MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
    }
    else if (FlagCANBroadOUT_E == 1)
    {
        MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
    }
}
```

Figure 5.52

MULj1939.c, Fault

At this point, a timer and switch-case (5 cases), that regulates the execution based on the value of StDM,1 manage the multiframe transmission.

The case 0 is for the BAM message, necessary to allow data exceeding 8 bytes in length to be sent.

```
void MULj1939_MessaggioDM1BAM(void)
{
    uint numByte = 0;

    numByte = StatusDM1 * 4 + 2;            /* Number of bytes transmitted */

    CANIDTx = 0x18ECFF00 | ((ulong)SourceAddress_E);
                                            /* Load ID per message
                                               Priority 6 (lower) 18
                                               PGN    FECA
                                               Source Address SA */

    CANBuffTx[0] = 0x20;                    // Control byte
    CANBuffTx[1] = (byte)numByte;           // Number of low part bytes transmitted (net of message number)
    CANBuffTx[2] = (byte)(numByte >> 8);    // Number of low part bytes transmitted (net of message number)
    CANBuffTx[3] = NumMessaggi;
    CANBuffTx[4] = 0xFF;
    CANBuffTx[5] = 0xCA;
    CANBuffTx[6] = 0xFE;
    CANBuffTx[7] = 0x00;

    if (FlagCANBroadOUT_E == 0)
    {
        MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
    }
    else if (FlagCANBroadOUT_E == 1)
    {
        MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
    }
}
```

Figure 5.53

MULj1939.c, BAM message

The case 1 relates to sending the first message and the lamp status calculation; the CAN TX buffer is loaded with the SPN, FMI and the values taken by the table of the Figure 5.48.

In relation to the flag *FlagCANBroadOUT_E*, the message is loaded on CAN0 or CAN1.

```
case 1:                                 /* First message */
{

CANIDTx = 0x18EBFF00 | ((ulong)SourceAddress_E);
                                        /*
                                           Load ID per message
                                           Priority 6 (lower) 18
                                           PGN EBFF
                                           Source Address SA
                                        */


CANBuffTx[0] = PosizMessDM1;            /* Message number */


    /* Lamp status calculation */

    for(j=1; (j<192); j++)              /*It starts from 1 because with i = 0 it is all OK  */
    {
        if (StatoDM1[j] == 1)           /* Anomaly set  */
        {
            if(MULj1939_TabAnoDM1[j].Lamp.IndicatorLamp == 1)
            {
                appoggio |= 0b01000000;
            }

            if(MULj1939_TabAnoDM1[j].Lamp.RedLamp == 1)
            {
                appoggio |= 0b00010000;
            }

            if(MULj1939_TabAnoDM1[j].Lamp.WarningLamp == 1)
            {
                appoggio |= 0b00000100;
            }

            if(MULj1939_TabAnoDM1[j].Lamp.ProtectLamp == 1)
            {
                appoggio |= 0b00000001;
            }
        }
    }

    CANBuffTx[1] = appoggio;
    CANBuffTx[2] = 0xFF;

    MULj1939_TrovaDM1Error();

    CANBuffTx[3] = (byte)MULj1939_TabAnoDM1[PosizStatoDM1].SPN;
    CANBuffTx[4] = (byte)(MULj1939_TabAnoDM1[PosizStatoDM1].SPN >> 8);
    //CANBuffTx[5] = MULj1939_TabAnoDM1[PosizStatoDM1].FMI;
    CANBuffTx[5] = MULj1939_SPNFMI(PosizStatoDM1);

    CANBuffTx[6] = 0x01;

    MULj1939_TrovaDM1Error();

    CANBuffTx[7] = (byte)MULj1939_TabAnoDM1[PosizStatoDM1].SPN;


    if (FlagCANBroadOUT_E == 0)
    {
        MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
    }
    else if (FlagCANBroadOUT_E == 1)
    {

    {
        MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
    }

    PosizStructDM1 = 2;                 /* Next package to write high byte of the SPN           */

    ContTaskCAN0DM1Multi = 0;           /* Relaunch the timer from the task for the next message */

    PosizMessDM1 = PosizMessDM1 + 1;

    StDM1 = 3;

    break;
}
```

Figure 5.54

MULj1939.c , Case 1

```
void MULj1939_TrovaDM1Error(void)
{

    byte stop = 0;

    while (stop == 0)
    {
        if (PosizStatoDM1 < 192)
        {
            PosizStatoDM1 = PosizStatoDM1 + 1;

            if (StatoDM1[PosizStatoDM1] == 1)
            {
                stop = 1;
            }
        }
        else
        {
            stop = 1;
        }
    }
}
```

Figure 5.55

MULj1939.c , TrovaDM1Errori function

```
case 2:                                 // PosizStructDM1 = 1
{

    CANIDTx = 0x18EBFF00 | ((ulong)SourceAddress_E);

                                        /* Load ID per message
                                           Priority 6 (lower) 18
                                           PGN EBFF
                                           Source Address SA */

    CANBuffTx[0] = PosizMessDM1;        /* Message number    */

    MULj1939_TrovaDM1Error();

    if (PosizStatoDM1 != 192)           /* No more errors, put 0xFF */
    {
        CANBuffTx[1] = (byte)MULj1939_TabAnoDM1[PosizStatoDM1].SPN;
        CANBuffTx[2] = (byte)(MULj1939_TabAnoDM1[PosizStatoDM1].SPN >> 8);
        //CANBuffTx[3] = TabAnoDM1[PosizStatoDM1].FMI;
        CANBuffTx[3] = MULj1939_SPNFMI(PosizStatoDM1);
        CANBuffTx[4] = 0x01;//provaDM1
    }
    else
    {
        CANBuffTx[1] = 0xFF;
        CANBuffTx[2] = 0xFF;
        CANBuffTx[3] = 0xFF;
        CANBuffTx[4] = 0xFF;
    }

    MULj1939_TrovaDM1Error();

    if (PosizStatoDM1 != 192)           /* No more errors, put 0xFF */
    {
        CANBuffTx[5] = (byte)MULj1939_TabAnoDM1[PosizStatoDM1].SPN;
        CANBuffTx[6] = (byte)(MULj1939_TabAnoDM1[PosizStatoDM1].SPN >> 8);
        //CANBuffTx[7] = MULj1939_TabAnoDM1[PosizStatoDM1].FMI;
        CANBuffTx[7] = MULj1939_SPNFMI(PosizStatoDM1);
    }
    else
    {
        CANBuffTx[5] = 0xFF;
        CANBuffTx[6] = 0xFF;
        CANBuffTx[7] = 0xFF;
    }

    if (FlagCANBroadOUT_E == 0)
    {
        MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
    }
    else if (FlagCANBroadOUT_E == 1)
    {
        MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
    }

    PosizStructDM1 = 4;

    ContTaskCAN0DM1Multi = 0;            /* Relaunch the timer from the task for the next message  */

    StDM1 = 5;

    PosizMessDM1 = PosizMessDM1 + 1;

    // change state

    if (PosizMessDM1 > NumMessaggi)     /* All messages sent */
    {
        TxDM1 = 0;                      /* Stop */
    }

    break;
}
```

Figure 5.56

MULj1939.c , Case 2

In the previous Figure the case 2 is shown as a further example.

The DM3 has the task of making a clear of the diagnostic data and possibly a reset of the previous DTCs, this can be easily done putting at 0 the buffer *FAnoDM_E[192]*.

```
void CAN0DM3(void)
{
    if (DM3Enable_E == 1)
    {
        if (RequestDM3 == 1)
        {

            byte i = 0;


            for (i = 0; i < 192; i++)
            {
                FAnoDM_E[i] = 0;
            }

             ACK positive response

            if (TypeDM3 == GLOBAL_REQ)
            {
                CANIDTx = 0x18E8FF3D;
                                                    /*  Load ID per message
                                                        Priority 6 (lower) 18
                                                        PGN         E8FF
                                                        Source Address SA  */

            }
            else
            {
                CANIDTx = ((0x18E80000) | ((ulong)SendDM3 << 8)) | ((ulong)SourceAddress_E);
                                                    /*  Load ID per message
                                                        Priority 6 (lower) 18
                                                        PGN         E8FF
                                                        Source Address SA
                                                    */
            }


            CANBuffTx[0] = 0;                           // control byte ACK positive
            CANBuffTx[1] = 0;
            CANBuffTx[2] = 0xFF;
            CANBuffTx[3] = 0xFF;
            CANBuffTx[4] = 0xFF;
            CANBuffTx[5] = 0xCC;
            CANBuffTx[6] = 0xFE;
            CANBuffTx[7] = 0x00;

            if (FlagCANBroadOUT_E == 0)
            {
                MULcan_QueueLoadCAN0(CANIDTx, CANBuffTx);
            }
            else if (FlagCANBroadOUT_E == 1)
            {
                MULcan_QueueLoadCAN1(CANIDTx, CANBuffTx);
            }

            RequestDM3 = 0;

        }
    }
}
```

Figure 5.57

MULj1939.c , DM3

5.2.8 Other units

In addition to the units presented in this section of the paper, the SW has a block of header files, inclusion files, libraries and, of course, a main file.

The module *Headers* contains

- *DefDefines.h*

  Definition of all the ECU address, mostly related to the EEPROM and the FLASH, and initialization of peripheral (PORT A, PORT B…)

- *DefPrototypes.h*
  Definition of function prototypes related to Init, peripheral, routines, interrupt functions.
- *VariablesEEPROM.h*
- *VariablesFLASH.h*
- *VariablesRAM.h*

The module *Includes* contains

- *MC9S12XET256.h*
  This header implements the mapping of I/O devices.

The module *Libs* contains

- *MC9S12XET256.c*

The interrupts are managed in the module *MCAL_S12XET256.*

CHAPTER 6
DEBUG AND TESTING

The last step of the code development process involves a testing phase. With regard to this paper, the focus was on verifying the correct functioning of the individual SW units and the correct interoperability between the modules.
The use of actuators has made it possible to verify the effective correctness of the communication network.

6.1 Testing

In section 3.2.2 of this paper, the different types of tests associated with the V-Model steps have been described.
In both networks an analysis was carried out in the debug phase, with direct control of the registers and verification of the execution flow; the goal is to obtain a SW that respects the project requirements and does not contain errors that could compromise the code.

As a first strategy it was used the unit test to validate the units of the code working properly, so it is checked the correct behaviour of function, methods, loops and so on.
Possible tests are:

- *Interface test*
  Verification that unit sends/receives data correctly

- *Local data structure test*
  Verification that local data structure are stored correctly.
- *Boundary condition test*
  Verification that boundary conditions are correctly handled.
- *Fault Injection Test*
  A fault is inserted into the system to analyse the consequent behaviour; it can be emulated by SW.

6.2 Serial Network Test

To verify the correctness of the serial network, a first debug was performed to correct any anomalies through the use of breakpoints and the check of the memory and registers value in the debug mode of the IDE.

The structure of the serial communication network is made in such a way that, having received a certain input message, the ECU must respond with a corresponding message.
The protocol specifies the structure of all the messages that the ECU can interpret, with relative positive or negative response (specific for each type of message).

The test was carried out by creating a set of messages, each corresponding to a request, sent via the Hercules SW.
In this way, it is possible to send data and see the relative answer, and also verify the correct operation with different baud rates and parity bits.
Voluntarily send incorrect messages or that communicate an anomaly it can be very useful to understand how the SW manages situations of malfunctioning and unforeseen events: the actual operation of the network is evaluated from the corresponding response.

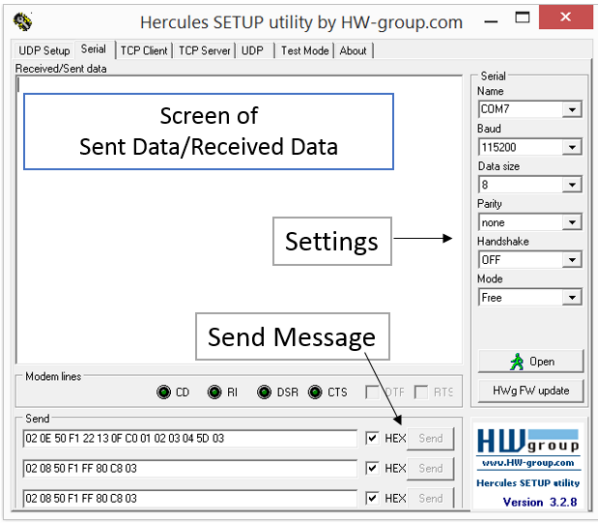


Figure 6.1

Hercules SETUP

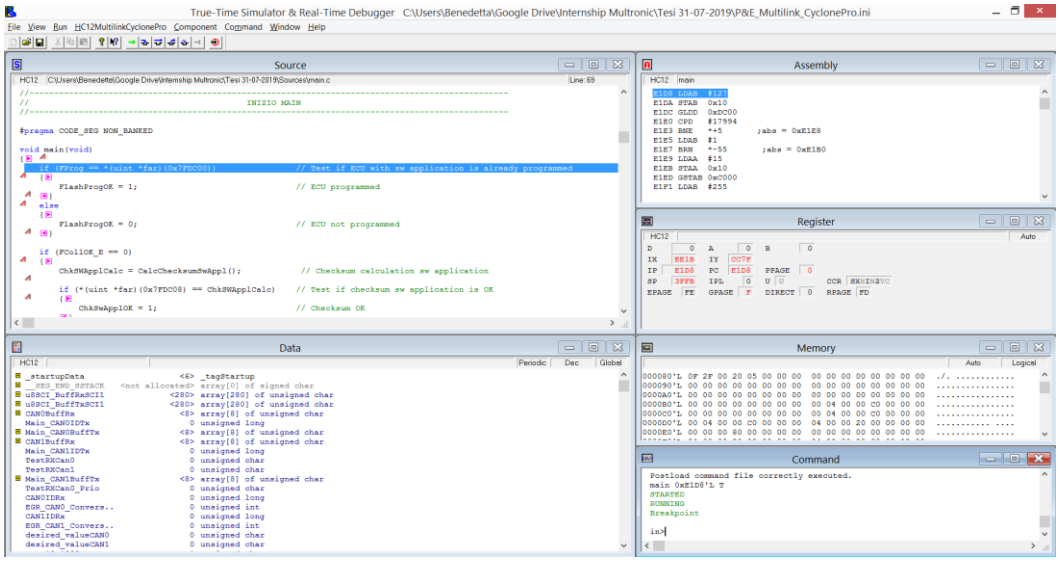


Figure 6.2

Debug Mode Interface

When a message is sent over serial, it can be checked what was actually received using the debug interface. Similarly, the verification can be done for the transmission using the registers *u8SCI_BuffRxSCI1* and *u8SCI_BuffTxSCI1*.

6.3 Can Network Test

As regards the CAN network, the test is carried out in two steps: the first one is done by checking the registers, verifying the messages sent/received and the correct behaviour of the timers via P-CAN View; the second one checks the correct operation of the actuator controlled on can.

Having CAN0 and CAN1 available, the tests are performed on both lines.

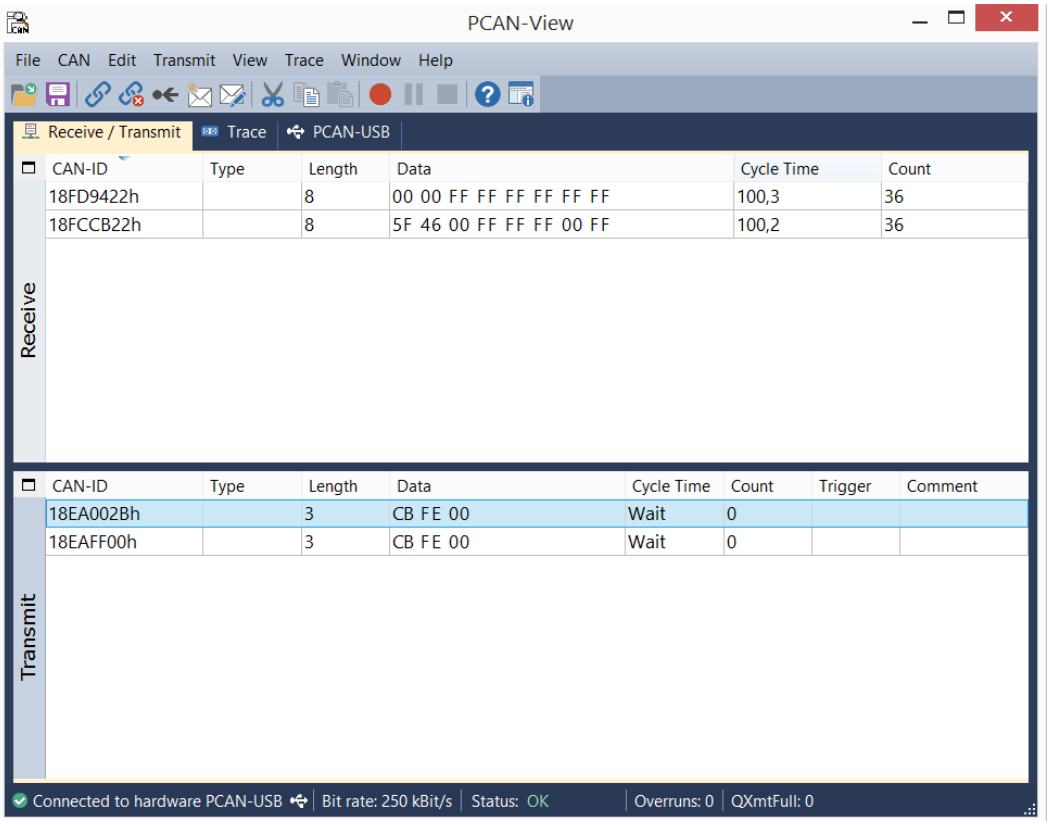


Figure 6.3

PCAN-View

The check made on the exchange of messages on the network follows the same principle as the serial network. To verify the correct transmission and reception, it is possible to take advantage of both the registers relating to the CAN-ID, date, length of the message, etc. and the tool PCAN-view, which gives the possibility to verify the correct functioning of the timers by checking the count section.

To speed up the test and verify the correct data exchange through the tool, in the mail there is a counter that updates the data sent on CAN. The counting is started

via the set of the *TestRXCAN0* variable.

```
/* CAN0 AND CAN1 WITHOUT PRIORITY */

if (TestRXCan0 == 1)
{
    if(ContTestCAN>=TimerTestCAN)
    {
        ContTestCAN=0;
        ContCANTest ++;

        Main_CAN0BuffTx[0] = (byte)ContCANTest;
        Main_CAN0BuffTx[1] = (byte)(ContCANTest >> 8);
        Main_CAN0BuffTx[2] = (byte)(ContCANTest >> 16);
        Main_CAN0BuffTx[3] = (byte)(ContCANTest >> 24);
        MULcan_QueueLoadCAN0(ContCANTest,Main_CAN0BuffTx);
    }
}


if (TestRXCan1 == 1)
{
    if(ContTestCAN>=TimerTestCAN)
    {
        ContTestCAN=0;
        ContCANTest ++;

        Main_CAN1BuffTx[0] = (byte)ContCANTest;
        Main_CAN1BuffTx[1] = (byte)(ContCANTest >> 8);
        Main_CAN1BuffTx[2] = (byte)(ContCANTest >> 16);
        Main_CAN1BuffTx[3] = (byte)(ContCANTest >> 24);
        MULcan_QueueLoadCAN1(ContCANTest,Main_CAN1BuffTx);
    }

}
```

Figure 6.4

Sending data on CAN

6.3.1 Test EGR Valve

The correct functioning of the CAN network is guaranteed by tests carried out on the EGR valve; the function that allows the set of the desired value is managed by

```
/*****************************************************************************/
/*                        TEST CAN ON EGR VALVE                              */
/*****************************************************************************/




if(ContTestCAN>=TimerTestCAN)
{
     ContTestCAN=0;
     CAN_EGR_PositionCAN0(desired_valueCAN0);


 }


 if(ContTestCAN>=TimerTestCAN)
{
     ContTestCAN=0;
     CAN_EGR_PositionCAN1(desired_valueCAN1);


 }
```

Figure 6.5

Set position EGR valve

a timer at 1.024 ms and is called in *main.c* as shown in Figure 6.5. The *desired_valueCANx* is set in the debug interface.

```
//------------------------------------------------------------------------------------------------------
//                                              TIMER BASE
//------------------------------------------------------------------------------------------------------

void interrupt Int_RTI(void)
{
    uint AppCalc = 0;
    byte AppGPAGE;
    byte AppRPAGE;
    byte AppEPAGE;

    AppGPAGE = GPAGE;        // IMPORTANT -> save GPAGE because the INT doesn't save it
    AppRPAGE = RPAGE;        // IMPORTANT -> save RPAGE because the INT doesn't save it
    AppEPAGE = EPAGE;        // IMPORTANT -> save EPAGE because the INT doesn't save it

    CRGFLG |= 0x80;                          // ACK interrupt

    // TIMER FONDAMENTALE 1.024 ms

    //^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    // Timer for message reception timeout on SCI1

    if (ContRxSCI1 < TimRxSCI1)
    {
        ContRxSCI1++;
    }

    if (ContTestSeriale < TimerTestSeriale)
    {
        ContTestSeriale++;
    }

    SCI_Timer1ms();


     if (ContTestCAN < TimerTestCAN)
    {
        ContTestCAN++;
    }


    J1939_Timer1ms();
    J1939_Multi_Timer1ms();
    J1939_Multi_Timer1ms_DM2();


    GPAGE = AppGPAGE;        // IMPORTANT -> restore GPAGE because the INT doesn't do it
    RPAGE = AppRPAGE;        // IMPORTANT -> restore RPAGE because the INT doesn't do it
    EPAGE = AppEPAGE;        // IMPORTANT -> restore EPAGE because the INT doesn't do it
}
```

Figure 6.6

Timers, *Interrupts.c*

6.3.2 Test J1939

As regards the J1993 messages, the control can be performed in a simple way using the PowerView 101 able to interpret the parameters of the data sent, showing the SPN and the IMF (Figure 6.7).

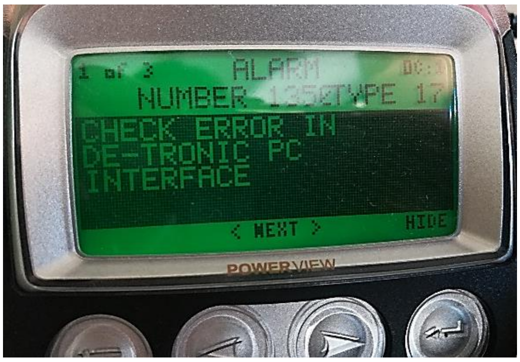


Figure 6.7

PowerView 101

The messages are sent using the debug interface, setting the FanoE array [192] in correspondence with the anomaly that one intends to test.

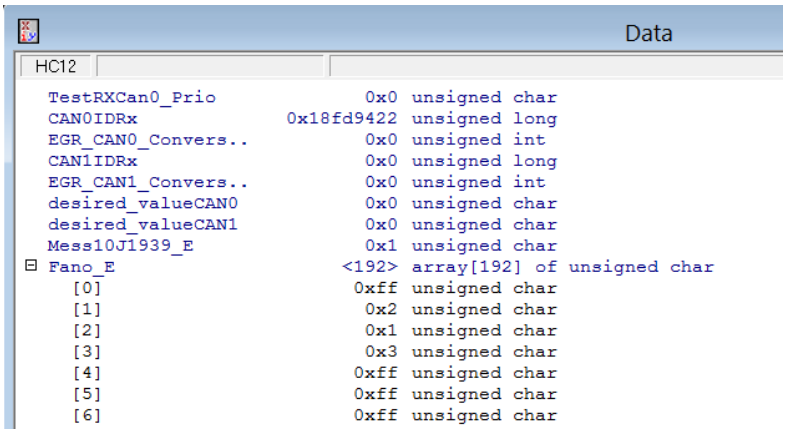


Figure 6.8

Fano_E[192]

For example in this case are set maintenance warning, maintenance alarm and bypass mode; the numbers indicate the priority of the message.

CHAPTER 7
CONCLUSION

This section of the paper briefly shows how it is possible to expand the network developed during the thesis activity; the goal is to have a more complete communication system for automotive application by implementing a LIN network.
Finally, the conclusions of this project are shown.

7.1 Future work, LIN interface

The Local Interconnected Network (LIN) is a serial communication protocol usually used for the components where a possible fault does not represent a critical scenario, in terms of safety (air conditioning, seat controls…).

It is composed by 16 nodes (1 master and up to 15 slaves), the communication is broadcast and is always initiated by the master that passes a token to the slaves: when one of them receives the token, they can send data over the network.

A LIN network is composed by a physical and data-link layer, in the next Figure it is shown a typical LIN communication:

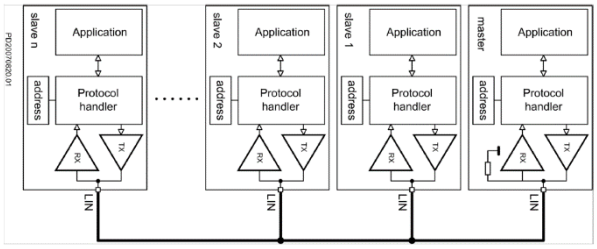


Figure 7.1

LIN Network – Physical Layer

A LIN message is characterized by a header (transmitted by the master) and a response (it can be transmitted by a slave or the master).

System Architecture

The LIN bus setup for the MC9S12XET256 µC requires a particular structure characterized by the interaction between the LIN Core and the application.

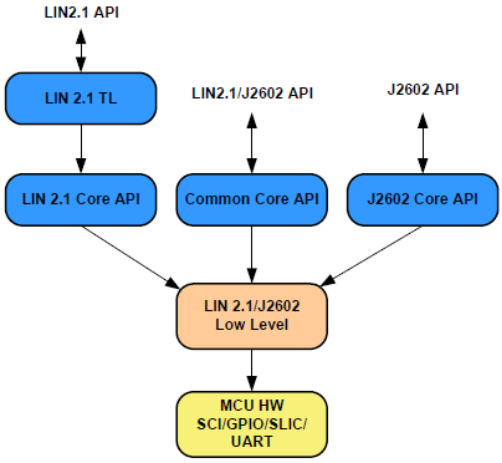


Figure 7.2

LIN Stack Structure

The LIN source code and the API must be integrated with the files inside the LIN Stack package: those files are the .h and the .c generated by a Node Configuration Tool on the basis of the Node Private Description File and LIN Configuration Description File.

- *Node Private Description File*

  It has the information about nodes (name, communication channel,

clock frequency…)

- *LIN Configuration Description File*
  LIN Cluster information.

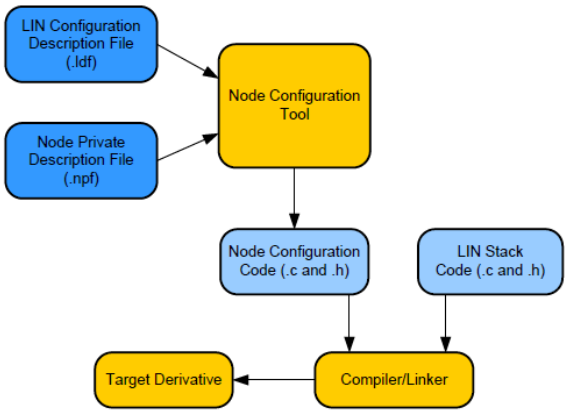


Figure 7.3
LIN Configuration

This package is provided by NXP Semiconductors and it is composed by six layers:

- BSP
- Low Level
- Core API
- Transport Layer
- Diagnostic services
- Application Layer

The LIN bus interface represent a very important low cost improvement for the network. Even if protocols such as automotive Ethernet or CAN FD are increasingly appreciated, LIN remains vital for the new and complex features required by the manufacturers.

The development of this type of interface represents a possibility to expand this thesis project by obtaining a communication system that is more ample and adaptable to different needs.

7.2 Conclusion

This paper shows the phases of the project developed as last step of the Master's Degree of the Politecnico di Torino in Mechatronic Engineering, from the documentation to the actual development of the code.
The thesis activity is based on the experience at the Belgian company Multronic s.r.l in Carmagnola (TO).

The idea is to implement communication interfaces based on the main features of the standard AUTOSAR, with the aim of creating a communication network between ECU of vehicles that is simple, easily adaptable and modifiable. The focus is on the Serial and CAN bus communication.

The first part of the thesis proposes a scenario of the work goal and briefly shows the studied topics necessary to have a clear vision on how to structure and implement the firmware: it is performed an overview of the fundamental concepts of AUTOSAR and the standard ISO 26262, in addition the V-shape model is presented.
A description of the network is also included, with the technical characteristics of the microcontroller.
Then, the main features of the communication protocols that are the object of the activity are included.

Going ahead with the paper, there is the core of the project that is the implementation of the communication protocols. The scheme of analysis is the same: first of all it is presented the FW structure, then the description start with the lowest layer of the stack module (the one closer to the HW), to get to communication drivers and communication hardware abstraction.

Within the CAN interface, the attention is also placed on the SAE J1939 and its implementation in order to be able to carry out the control of the EGR valve, used to verify the correct functioning of the network.

10935 lines of code have been developed, without considering the use of external libraries.

The tests carried out in the last period of activity have confirmed the validity of the product intended for the production of the control unit for the automotive market. In particular, network tests were performed using both SW and HW systems: in the case of the serial network, the correct operation of the information exchange was carried out by exploiting a serial communication via the PC-ECU. As far as the CAN network is concerned, the test was particularly interesting as it was possible to test using HW components in addition to the SW.

The FW obtained was implemented in relation to the time available, but has the potential to be expanded and made more effective in terms of tests: an example could be the automation of the process of sending messages, to have immediate results and a faster SW release.

References

AUTOSAR, Document number 664 - *Overview of Functional Safety Measures in AUTOSAR*, AUTOSAR CP Release 4.3.0

Briciu, Catalin-Virgil & Filip, Ioan & Heininger, Franz. (2013). *A new trend in automotive software: AUTOSAR concept*. 251-256.

CAN Specification 2.0 , Part A, Part B – Bosch, 1995

CAN Texas Instruments - *Application Report*, 2016

C. Virgil Briciu, I.Filip, F. Heininger. "*A new trend in automotive software: AUTOSAR concept.* Politehnica" University of Timisoara/ Faculty of Automation
and Computer Science, Timisoara, Romania. Continental Automotive Germany/Interior Body and Security, Regensburg, Germany. May 2013

Gade S, Kanase A, Shendge S, Uplane M. *SERIAL COMMUNICATION PROTOCOL FOR EMBEDDED APPLICATION*. International Journal of Information Technology and Knowledge Management
July-December 2010, Volume 2, No. 2, pp. 461-463

H.Martorell, J. Fabre, M. Lauer, M. Roy, R. Valentin. *Partial Updates of AUTOSAR Embedded Applications - To What Extent?*. 11th European Dependable Computing Conference (EDCC 2015), Sep 2015, Paris, France

ISO-11898: 2003

ISO, Road vehicles - *Functional Safety* - 26262-6. ISO, 2011.

M.Violante's Lecture – Model-Based Software Design Course a.y. 2017/2018, Network Technologies for Mechatronic Systems Course a.y. 2018/2019. Mechatronic Engineering, Politecnico di Torino.

MC9S12XEP100RMV1 – NXP, *Datasheet* (Rev 1.25)

National Instruments, J1939 Transport Protocol Reference Example, http://www.ni.com/example/31215/en/#toc4, March 2019

Niklas Amberntsson. *La progettazione di reti secondo Autosar*, https://www.elettronicanews.it/la-progettazione-di-reti-secondo-autosar/

Renesas - *AUTOSAR Layered Architecture*, https://www.renesas.com/us/en/solutions/automotive/technology/autosar/autosar-layered-architecture.html

Richard Bellairs. *What Is ISO 26262? An Overview,* SECURITY & COMPLIANCE, STATIC ANALYSIS, January 2019, https://www.perforce.com/blog/qac/what-iso-26262-overview

S12(X)Build Tools - *Reference Manual*

Samarins.com. *How Exhaust Gas Recirculation (EGR) system works*, https://www.samarins.com/glossary/egr-system.html, December 2018

Vector- *Transport Protocol* – https://www.vector.com/it/it/know-how/technologies /protocols/sae-j1939/#c26590