POLITECNICO DI TORINO

Master of Science in Mechatronic Engineering

Master's Degree Thesis

Design and Realization of an Open-Loop Simulator for ICE Control Units developing the Crankshaft and Camshaft Sensors Simulation

Academic Supervisor:

Prof. Massimo VIOLANTE

Kineton srl Supervisor:

Ing. Giuseppe SELVA

Candidate:

Claudio GAGLIO

Academic Year 2020/2021

Abstract

Automotive industry focuses its efforts to increase human safety and to reduce fuel consumption and pollutant emissions, in order to be in compliance with the more and more stringent environmental requirements imposed by governments. A large number of new electronic systems have been being developed over the years and Electronic Control Units (ECUs) represent the brain of the vehicles. Also internal combustion engine is mainly governed by embedded systems: the Engine Control Module manages the powertrain system, by controlling air-fuel ratio, fuel injection and ignition. The high complexity of the electronic architecture of a vehicle requires a time consuming and costly phase of simulation for verification and validation of the system. In this scenario, Hardware-in-the-Loop (HIL) simulation is performed to validate the real-time software running in ECUs, but some automotive companies do not need a whole HIL simulator to perform simple tests. Hence, an open loop crankshaft and camshaft sensors simulator can be used in order to simulate the four-stroke engine behaviour and the purpose of this master thesis is its design and realization. Firstly, a review of the working principle of a diesel internal combustion engine is presented. Furthermore, details about embedded systems, ISO-26262 and its V-model for software development are given, in order to see how it is possible to design and validate safety-related systems for automotive electrical systems. The first part of this project deals with the study of the main characteristics of a HIL simulator for powertrain, in order to understand how it generates the crankshaft and camshaft signals useful for the ECM. Starting from this study, the design and realization of the open-loop simulator is performed: the generated signals are digital, since they represent the output of Hall-effect active sensors. The implemented simulator is made up of some electronic devices: a high-performance microcontroller, a rotary encoder for setting the desired engine speed value for the simulation and a touch display used as graphical user interface. The application code is written in C-language: it contains the look-up table describing the crankshaft and camshaft pattern and all the I/O peripherals configurations (modules used to control the quadrature encoder, real-time interrupt, serial communication protocol and general input-output channels). As result, the digital signals are generated according to the real-time user settings and they are verified using an oscilloscope.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables

List of Figures

Acronyms

APU

Angular Processing Unit

ASIL

Automotive Safety Integrity Level

BDC

Bottom Dead Center

CAN

Controller Area Network

CCS

Code Composer Studio

CI

Compression Ignition

CPU

Central Processing Unit

CSV

Comma-Separated Values

E/E

Electronic and Electrical system

ECM

Engine Control Module

ECU

Electronic Control Unit

EQEP

Enhanced Quadrature Encoder Pulse

GIO

General-purpose Input/Output

HIL

Hardware-in-the-Loop

HMI

Human-Machine Interface

I/O

Input and Output

ICE

Internal Combustion Engine

IDE

Integrated Development Environment

ISR

Interrupt Service Routine

MIL

Model-in-the-Loop

NRZ

Nonreturn to zero

OS

Operating System

PIL

Process-in-the-Loop

PIT

Periodic Interrupt Timer

PFM

Pulse-Frequency Modulated

PWM

Pulse-Width Modulated

QM

Quality Measure

RISC

Reduced Instruction Set Computer

RTI

Real Time Interrupt

SCI

Serial Communication Interface

SI

Spark Ignited

SIL

Software-in-the-Loop

TDC

Top Dead Center

UART

Universal Asynchronous Receiver Transmitter

Chapter 1

Introduction

Nowadays, we are likely to use products whose behaviour is managed by embedded systems. Also, today's cars are governed by several computers.
Over the last 15 years, the number of embedded systems in a vehicle is dramatically increased, providing significant improvements in functionalities, performance, comfort, and safety. The sector of embedded electronics, and more precisely embedded software, has been increasing at an annual rate of 10% [1].
A whole range of electronic functions, such as cruise control, traction control, and stabilization control, are implemented in today's vehicles. All these functions need to exchange information with each other, sometimes with stringent time constraints: today, more than 2500 signals are exchanged through up to 70 ECUs on different types of networks.
The increasing complexity of electronic architectures embedded in a vehicle has led the automotive industry to adopt a distributed approach for implementing the set of functions and for communicating with a large number of sensors and actuators. Networks and protocols are the solutions for integrating functions, reducing the cost and complexity of wiring: CAN is at present the network that is the most widely implemented in vehicles.
Furthermore, state regulations (such as controlling exhaust emissions or mandatory active safety equipment) impose embedding complex control laws that can only be achieved with computer-based systems.
Even though the automotive software complexity leads to fundamental functionalities and system flexibility, it may introduce defects leading to system failures and safety problems. Hence, to address hazards due to electronic and electrical systems (E/E) malfunctions, an international standard has been defined: the ISO-26262. It is applied to functional safety, whose purpose is to minimize the danger that a possibly faulty system could cause. This standard provides requirements for the whole life cycle of E/E systems, including both hardware and software.

This Master thesis was carried out in "Kineton srl" company in Turin, specialized

in verification and validation of automotive power-train control systems. It is highlighted how the crankshaft and camshaft signals coming from crank and cam position sensors can be simulated. In order to understand the behavior of the signals, a preliminary study of the internal combustion engine, particularly the Diesel one type, is performed.

1.1 Diesel Internal Combustion Engine

1.1.1 Four Stroke Cycle Diesel Engine

The Diesel Engine carries out a full power cycle when a piston travels up and down twice, by turning the **crankshaft** through 720 degrees. This is verified since the crankshaft is connected to all the pistons by the connecting rods: in this way, the piston's reciprocating motion is converted into rotational movement of the crankshaft.

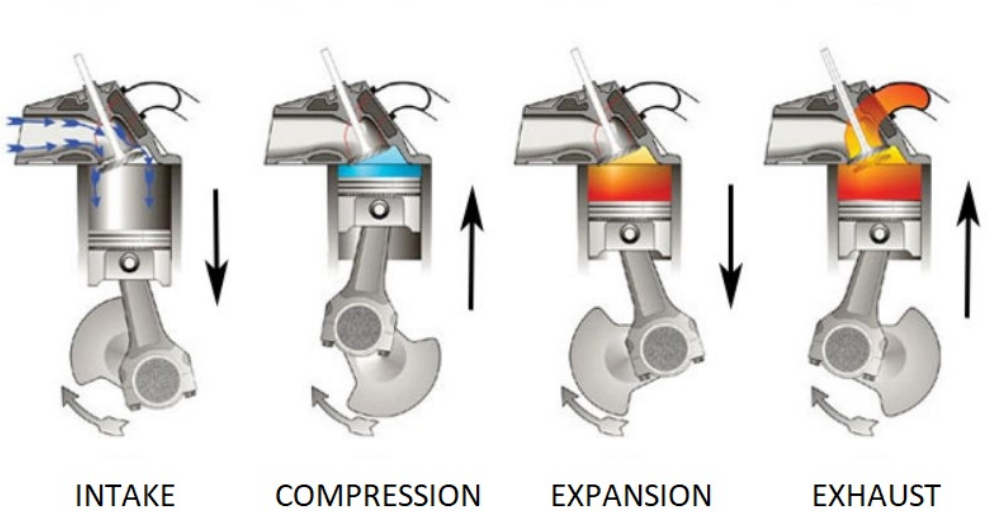

Figure 1.1: Four stroke cycle diesel engine

Stroke is the up or down piston movement inside the cylinder. A four-stroke Diesel cycle is composed of the sequence of the following types of strokes:

- **Intake stroke**. At the starting point, the piston is at the top dead center (TDC), and the intake valve opens as the piston moves down. This movement draws the air compressed by the turbocharger into the combustion chamber. Intake stroke ends at the bottom dead center (BDC).

- **Compression stroke**. The piston raises from the BDC while the intake and exhaust valves are closed. The upward movement makes the piston compress the in-cylinder charge, heating it up. Fuel is sprayed into the cylinder at the end of the compression stroke. In Diesel engines, the quantity of fuel injected determines the power and the torque provided to the crankshaft, but also the pollutant emissions.

- **Combustion stroke**. Fuel self-inflames on contact with the hot in-cylinder charge; indeed, diesel engines are also called Compression Ignition (CI) engines. The resulting combustion increases the internal energy and creates an expansion, which drives the piston down to the BDC thanks to the converted kinetic energy.

- **Exhaust stroke**. It is the last stroke before the cycle begins again with the intake stroke. With its upward movement and the contemporary open condition of the outlet valve, the exhaust gases are pushed out from the combustion chamber. The final position of the exhaust stroke is the TDC.

Valves timing is controlled by the **camshaft** that is connected to the crankshaft through a toothed belt. The camshaft consists of a cylindrical rod having a number of cams along its length, one for each valve. A cam lobe forces a valve open by pressing on the normally closed valve as it rotates: the valve returns to be closed when cam lobe does not press it anymore. One complete four-stroke engine cycle takes only one revolution of the camshaft, so its rotational velocity is half of the crankshaft one. Hence, the camshaft position is fundamental to derive the current engine stroke, and this is very important for the starting of a four-stroke engine.

Ideal Diesel Cycle

It is possible to make some considerations for the thermodynamic processes of an ideal four-stroke diesel engine. The combustion is controlled by the injection of fuel to maintain constant pressure; therefore, it takes place in an isobaric state change during the downward movement of the piston. The injection ratio or load is defined as the volume ratio between steps 2 and 3 in the pV diagram shown in Figure 1.2: $\rho = \frac{V_3}{V_2}$

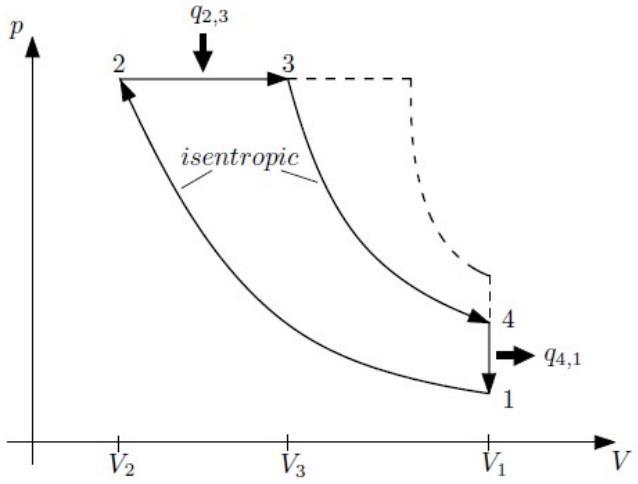

Figure 1.2: pV-diagram for Diesel Engine

The more fuel is injected, the larger the injection ratio ρ and the longer the distance between steps 2 and 3. The different steps of the cycle are the following:
$1 \rightarrow 2$: Isentropic compression
$2 \rightarrow 3$: Isobaric gain of thermal energy $q_{2,3}$
$3 \rightarrow 4$: Isentropic expansion
$4 \rightarrow 1$: Isochoric heat loss ($q_{4,1}$ is negative).
The thermodynamic efficiency of the ideal cycle can be calculated through some transformations, and it depends on the injection ratio:

$$\eta_{th} = 1 - \frac{1}{\epsilon^{\kappa-1}} \frac{1}{\kappa} \frac{\rho^{\kappa} - 1}{\rho - 1} \tag{1.1}$$

where $\kappa = \frac{c_p}{c_v}$ is the ratio of specific heats and $\epsilon = \frac{V_1}{V_2}$ is the compression ratio.
It can be noticed that the efficiency η_{th} decreases as the injection ratio ρ increases. The Diesel compression ratio ϵ is much higher than that of SI engines and compensates for the lower efficiency due to the high injection ratio.

1.1.2 Overview of the Engine Control Module

In today's engines, electronic control is fundamental to fulfill the more and more stringent requirements about fuel consumption, pollutant emissions, and engine efficiency improvement. Carmakers distinguish five domains for embedded electronics in a car (but sometimes the membership of only one domain for a given compartment is not easy to justify):

1. The power train domain that refers to the systems participating in the longitudinal propulsion of the vehicle, including engine, transmission, and all subsidiary components;
2. The chassis domain, related to the relative position and movement of the wheels (steering and braking);
3. The body domain that manages equipment for car's user such as airbag, wiper, lighting, air conditioning, window lifter, seat equipment;
4. The HMI domain that includes components allowing information exchange between electronic systems and the driver;
5. The telematic domain that allows information exchange between the vehicle and the outside world.

The Engine Control Module (ECM) is the embedded system that governs the engine control system in the power train domain: it takes the input signals coming from different sensors and computes the output values required by the actuators in order to perform the desired behavior.

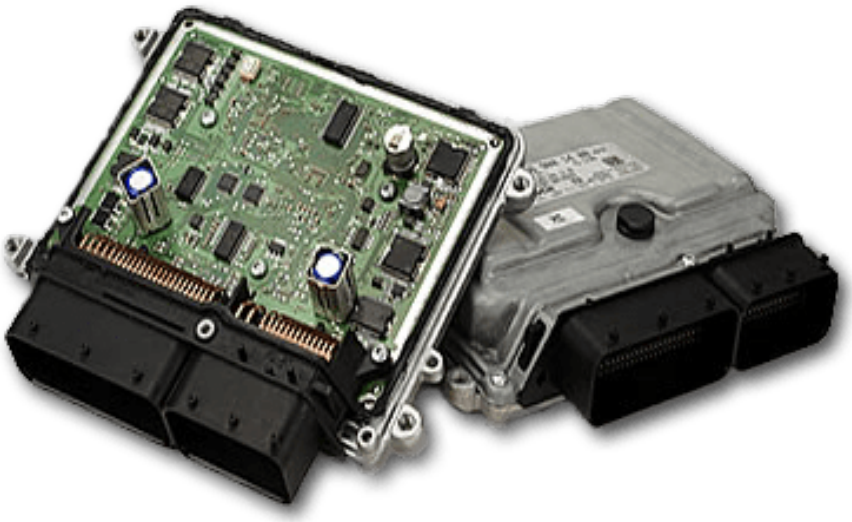

Figure 1.3: An ECM with and without cover

The ECM communicates with all the other ECUs of the vehicle through a CAN/LIN network, which allows the control of basic engine operations and the implementation of several vehicle functionalities that require the sharing action among several vehicle subsystems (such as Stop&Start, Cruise Control, and speed limiter).

The ECM collects a wide range of signals from several sensors like crankshaft rotational speed, fuel-rail pressure, intake-air temperature and mass, accelerator pedal position. It then interprets the input data by comparing them with the multidimensional performance maps and computes the output signals for the actuators in order to perform the required actions. The ECM is also the ECU responsible for serving some of the most important engine actions: it controls the injection timing and the fuel quantity, which are fundamental to optimize engine performance and reducing emissions. Indeed, the lambda sensor is used to detect whether the combustion inside the cylinder has been done with a rich or a lean mixture, described by the air-fuel ratio. If the provided mass air completely burns all of the fuel, the ratio is known as the stoichiometric mixture. Ratios lower than stoichiometric determine rich mixtures, which are less efficient but may produce more power. Ratios higher than stoichiometric determine lean mixtures, which are more efficient but may cause higher temperatures and lead to nitrogen oxides NOx. Depending on some input variables, such as the engine temperature, the air-fuel ratio, the mass of airflow and accelerator pedal position, the ECM can control the amount of fuel to inject into the engine cylinders.
Furthermore, engine timing functions for fuel injection and valve timing are controlled by the ECM thanks to the crankshaft and camshaft position sensors.
Another essential function performed by the ECM is the exhaust-gas control: a combination of exhaust-gas systems are controlled in order to reduce NOx, CO, HC and particulate emissions.
The ECM, as well as the other ECUs inside a vehicle, are developed following the V-model described in the international standard ISO 26262. The design is an iterative process linked with the validation and testing performed to catch failure modes that can lead to unsafe conditions or uncomfortable drive experience.

1.2 Embedded systems

An embedded system can be defined as a microprocessor-based electronic system designed to perform a specific function or a set of tasks. Its development is intended for best meeting the well-known prior requirements. Therefore, it consists of a customized hardware platform that is not re-programmable by the user. Indeed, the structure of an embedded system is composed of two major, tightly coupled sets of components:

- a set of **hardware components** that include a CPU, typically in the form of a microcontroller;
- **basic software and application**, typically included as firmware, that gives functionality to the hardware.

As briefly explained before, typical inputs in an embedded system are process variables and parameters that arrive via sensors and input/output (I/O) ports. The outputs are processed information for the user or can be in the form of control actions [2]. The exchange of input-output information can also occur with users via buttons, sensors, LEDs, displays, and other types of devices.

1.2.1 Hardware

The basic components of an embedded system architecture are the following:

- **RAM memory**, volatile memory storing data and instructions to be executed after power-up is completed;
- **CPU**, processor running software;
- **I/O**, peripherals to get inputs and to provide outputs;
- **Flash**, non-volatile memory for persistent storage.

The most used implementations of the reference hardware are:

- **Microcontroller-based implementation**, where most of the components are connected in a single device.

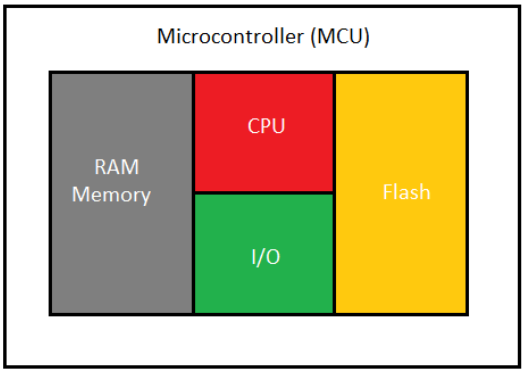

Figure 1.4: Microcontroller architecture

The systems in this implementation consist of a single monolithic component running at relatively low frequencies. This solution is cheap and occupies a small space, therefore it is suitable for simple real-time applications.

- **System-on-Chip implementation**, where the components are discrete, but the CPU is integrating some of them.

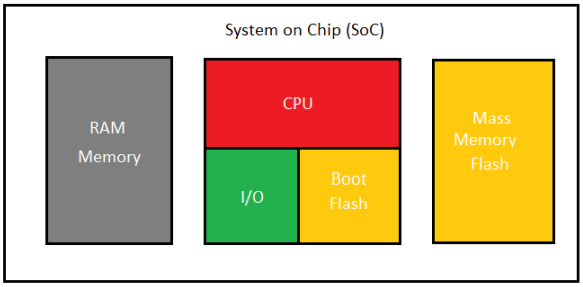

Figure 1.5: System-on-Chip architecture

This implementation is more complex than the previous one, and it requires additional networks and components for adding more functionality. The main advantage is the ability to support higher frequency and complex user-interfaces.

1.2.2 Basic Software

The structure of the basic software consists of two main layers: Operating System and System-call interface.

Operating System is the software that implements services using hardware: it is meant for abstracting hardware functionalities to simplify and optimize their usage [3]. The possible services are:

- *CPU manager*, which implements algorithms for handling the access to CPU resource and optimizing CPU time usage;
- *Memory manager*, which implements algorithms to satisfy the memory demands of application;
- *File manager*, which provides access to the persistent storage;
- *Device manager*, which implements services routines for the I/O peripherals;
- *Hardware-specific services*, which implements CPU-specific operations.

System-call interface is the part of the Basic Software which interfaces with the Application Software, giving an easy way to access the operating system services

to the application programmer.
The most used operating system architectures are: *Flat architecture*, *Layered architecture based on monolithic kernel* and *Microkernel architecture*.

Flat Architecture

There is not a strict separation between application and operating system. Therefore the services of the operating system are essentially functions that any application can invoke. It is intended to provide most of the functionalities in the least space, but malfunctions can freely propagate, corrupting the operating system. Even though it offers efficient communication, this architecture is not reliable.

Layered Architecture

This kind of architecture establishes the discrimination into layers of the different components of the operating system. Each layer uses services and functions provided by only lower layers, and it makes use of dedicated virtual address space. All the services are provided by an executable called **Monolithic Kernel**: memory is divided into user address space and Kernel address space.
Since there is a separation between application and basic software, any malfunction in user address space cannot affect the Operating System. Conversely, malfunctions in the OS components can corrupt the whole system; therefore, Layered architecture offers a *partial robustness*.

Microkernel Architecture

With this architecture, each component operates in a dedicated virtual address space. Microkernel is easier to extend, more reliable, and more secure. The total separation among layers address spaces leads to an *high robustness*, since malfunction in the user space cannot corrupt the whole system.

1.2.3 Real-Time Embedded Systems

The CPU scheduler is responsible for managing all the processes which the embedded system has to fulfill. Therefore, it needs to implement the *Scheduling Algorithms*: they are specific algorithms for identifying the proper criteria to pick up a process among the ones in ready-status and make it running. It is possible to find two kinds of systems, depending on which kind of Scheduling Algorithm the operating system implements:

- **Non real-time systems**, where the system must respond correctly to an external event whenever needed. The deadline for the process to be run is not specified, and it could also be infinite.

- **Real-time systems**, where the system must respond properly to an external event within a certain deadline. A deterministic behavior must be guaranteed in any possible operating condition. In automotive applications, all the functional safety purposes systems are real-time.

Real-time systems are mainly divided into three types:

- *Hard real-time*. If the deadline is missed, it may cause catastrophic consequences on the environment under control;
- *Firm real-time*. If the deadline is missed, no serious damages occur, but the results are useless.
- *Soft real-time*. Missing the deadline does not affect the system behavior, but it is desirable if it is met for computing purposes.

1.3 ISO-26262 and V-Shape Development Flow

With the trend of increasing technological complexity, more software contents and mechatronic implementations, there are increasing risks from systematic failures, hardware and/or software failures. **ISO-26262** is a specific international standard for the automotive industry. It provides methodologies useful for designing and validating safety-related systems that include one or more electrical and/or electronic (E/E) systems installed in series production road vehicles.
This standard addresses hazards due to malfunctions, providing requirements for the whole life cycle of these systems, depending on the risk for the customer.
ISO-26262 is focused on *functional safety*: it is the system safety that depends on the system operating correctly in response to its inputs. It avoids human damages when operator errors, environmental changes, and hardware failures happen. Functional safety is achieved when there is the absence of unreasonable risk due to hazards caused by malfunctioning of E/E systems.
ASIL (Automotive Safety Integrity Level) is a methodology used to quantify how a certain component is risky for the human user due to undesired effects. Hazard analysis and risk assessment is performed at the beginning of the safety life cycle and is classified according to a combination of three factors:

- **Severity (S)**: it determines the extent of harm to one or more individuals that can occur in a potentially hazardous event, from no injuries (S0) to life-threatening/fatal injuries (S3);
- **Exposure (E)**: it defines the probability of occurrence of the operational conditions in which the injury can happen. It goes from incredibly unlikely (E0) to high probability (E4);

- **Controllability (C)**: it defines the ability to avoid specified harm or damage through the timely reactions of the persons involved, possibly with support from external measures. It goes from controllable in general (C0) to difficult to control or uncontrollable (C4).

The combination of these factors gives the ASIL of a system: ASIL A represents the least stringent and D the most stringent level, which requires a significant effort in the development and testing of the system in order to ensure safety. The level QM indicates that no safety requirement is needed.

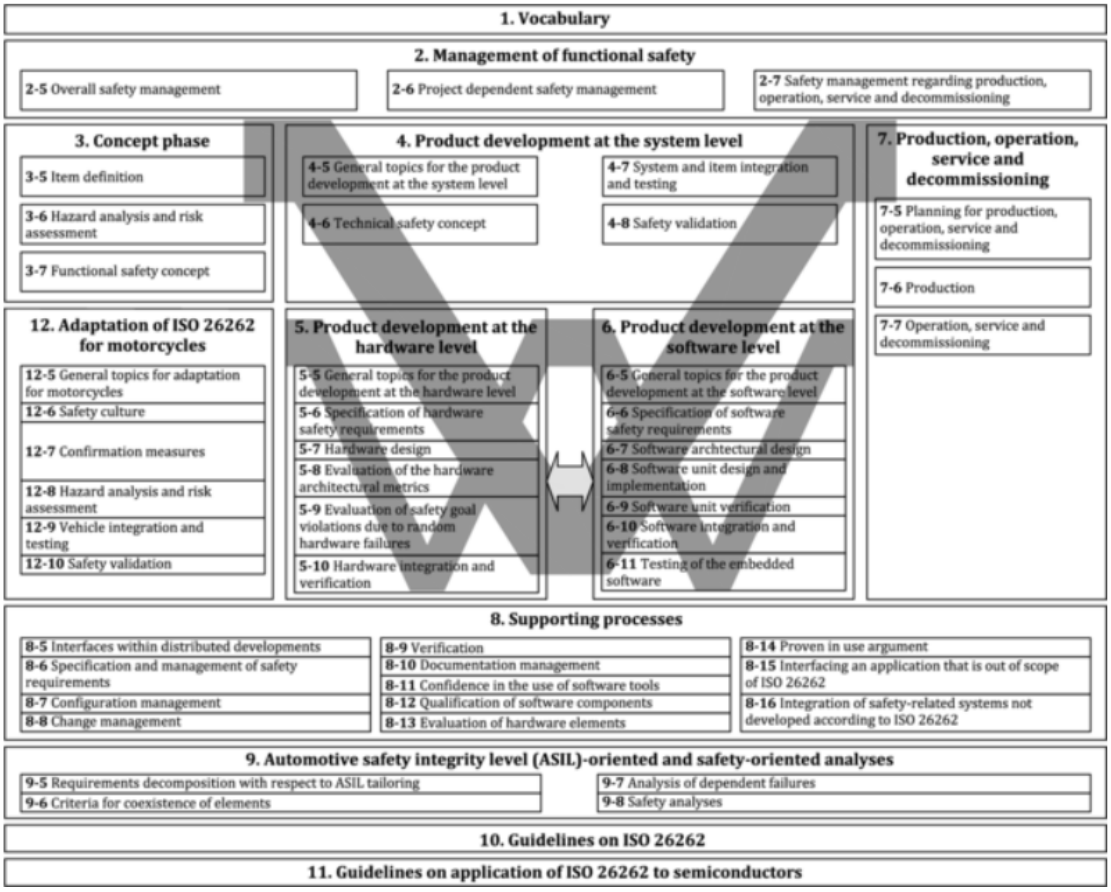

Figure 1.6: Overview of the ISO-26262 series of standards

The 2018 version of ISO 26262 consists of twelve parts, ten normative parts (parts 1 to 9 and 12), and two guidelines (parts 10 and 11) [4]:

1. Vocabulary
2. Management of functional safety

3. Concept phase
4. Product development at the system level
5. Product development at the hardware level
6. Product development at the software level
7. Production, operation, service and decommissioning
8. Supporting processes
9. Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis
10. Guidelines on ISO 26262
11. Guidelines on application of ISO 26262 to semiconductors
12. Adaptation of ISO 26262 for motorcycles

As shown in Figure 1.6, the ISO-26262 series of standards is based upon a V-model as a reference process model for the different phases of product development. The shaded "V"s represent the interconnection among ISO 26262-3, ISO 26262-4, ISO 26262-5, ISO 26262-6, and ISO 26262-7. It can be summarized in the V-shape development flow shown in Figure 1.7, composed by a descending branch followed by an ascending branch.

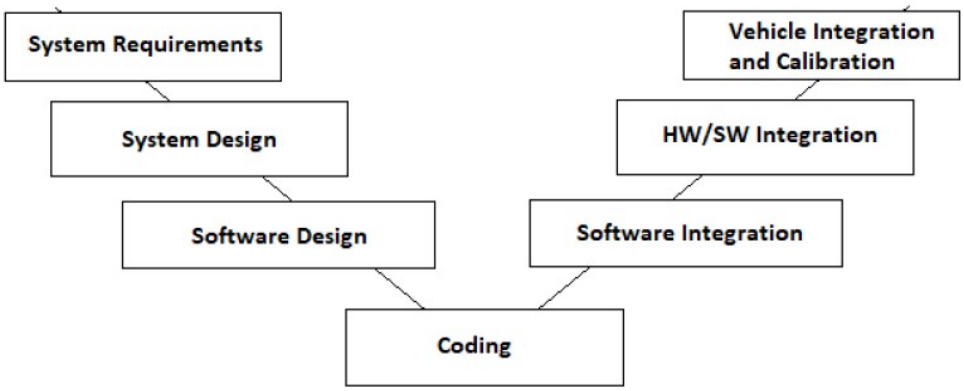

Figure 1.7: V-shape development flow

The different phases are the following:

- **System Requirements**: in this step, the item is defined and described with its dependencies on and interaction with the environment and other items;

- **System Design**: the item is designed with its functionalities, which are divided into submodules;
- **Software Design**: development of the functions to be used for implementing the subsystems functionalities;
- **Coding**: code is generated from the functions previously designed. Automatic code generation is very efficient even if the code needs to be optimized and it avoids unintended coding and requirements faults;
- **Software Integration**: the implemented software is merged;
- **Hardware and Software Integration**: the whole software is integrated into the embedded hardware, and the behavior is tested;
- **Vehicle Integration and Calibration**: the item is tested into the vehicle, and its functionalities are checked with the validation process.

All the phases are correlated with the opposite phases: whether an error occurs in a step after the coding, it is required to go back to the corresponding phase of the descending branch and modify it.
The standard establishes two implementation phases of the product development, which are made in parallel: *hardware level* and *software level*. Specific prescriptions are given about methods to be used and measures to be collected for each implementation.

1.3.1 Product Development: Software level

Software development is described in part 6 of ISO-26262, and it represents one of the most critical steps for the implementation of product development. It is composed of the following phases[5]:

- *Initiation*: planning of the functional safety activities during sub-phases of software development. In this phase, suitable methods, guidelines, and tools are chosen, considering qualified software tools, design, and coding guidelines.
- *Specification of software safety requirements*: design and technical safety concept must be consistent with the software safety requirements and the Hardware-Software Interaction.
- *Software unit design and implementation*: design of the software units and their verification through the generated source code and the object code.
- *Software unit testing*: the demonstration that the software units fulfill the software unit design specifications and do not contain undesired functionality. A test procedure is established and then performed.

- *Software integration and testing*: the whole software elements are integrated into the embedded software, that must realize the software architecture design.

- *Verification of software safety requirements*: it must demonstrate that the embedded software satisfies its requirements in the target environment.

Software testing is the fundamental process that allows functional safety achievement in order to avoid hazardous events. Hardware faults and software faults can occur, but they must be fixed as soon as possible: the later the error is detected, the more expensive the designing phase is.
Countermeasures against hardware faults are implemented in the item software; countermeasures against software faults are realized on the development process, but they are detected during software testing. In Figure 1.8 the typical V-model for software development is represented.

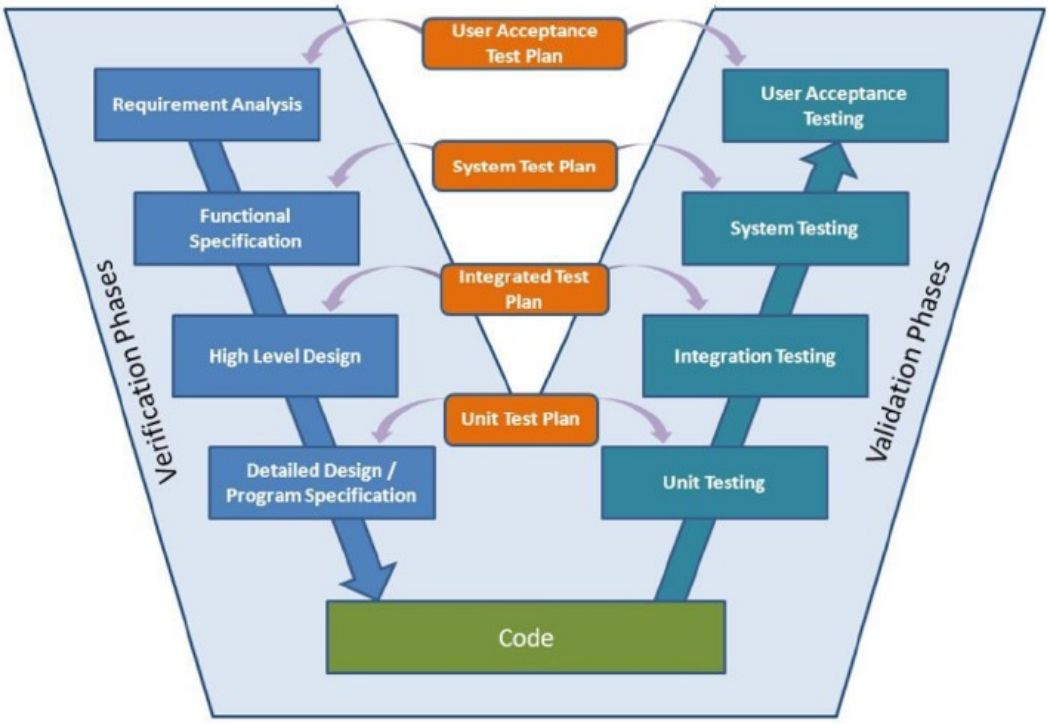

Figure 1.8: V-model for Software development

The software development process is split into two main phases: verification and validation, which are strictly interconnected. In the validation phase, it is possible to perform three kinds of testing:

- **Unit testing**: a test is applied to each module of the software as it is isolated from the rest of the system in order to confirm the correct behavior.

- **Integration testing**: a test is applied to units combined together in such a way to form a subsystem to test their interfaces. The aim is to confirm the interoperability of the units of the application.

- **System testing**: it exercises the functionality provided by the item in its entirety, and it must demonstrate that the embedded software satisfies its requirements in the target environment.

1.4 Validation of Embedded Control Algorithms

Hardware and Software faults avoidance, as previously seen, allows to reduce the timing of software production, hence reduce the cost, and improve safety conditions. The requirements can be met by performing some development check **simulations**. The main simulation methodologies are:

- Model-in-the-Loop

- Software-in-the-Loop

- Processor-in-the-Loop

- Hardware-in-the-Loop

Automotive software is normally implementing a reactive system composed by a *controller*, which will be running on the target system, and a *plant*, which will be connected to the target system providing status and receiving commands.

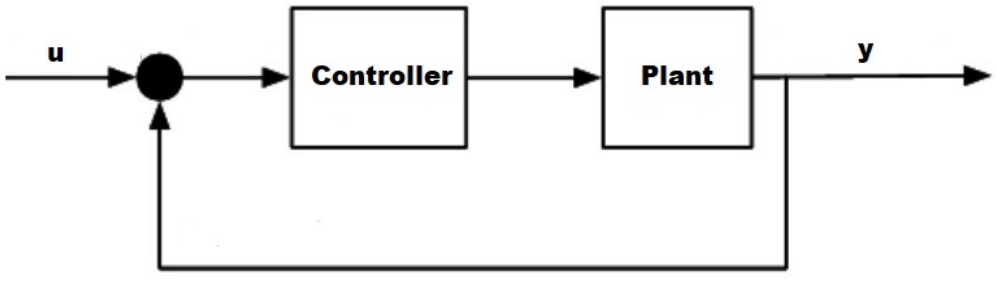

Figure 1.9: Behavioural model of the software

1.4.1 Model-in-the-Loop

Both the plant (systems to be modeled) and the controller (algorithm to control the plant) are modeled by following the requirements. After modeling, the simulation is essential since it helps in refining models and evaluates design alternatives. The purpose of Model-in-the-Loop testing is to simulate the *controller model* along with the *plant model* to validate the correct functionalities: both dynamical models are executed into a native simulation environment (i.e., Simulink) and run on a development host.
The simulation is purely functional since it neglects the time the controller will take for running on embedded hardware.
A reference signal is given as input to the feedback chain, and the MIL simulation verifies whether the output is compliant with the expected results. If the output is different, then the controller must be changed through an iterative process.

1.4.2 Software-in-the-Loop

The software must be related to a specific hardware platform; hence it must be adapted for that target through an optimization process: Software-in-the-Loop simulation.
SIL is the process that allows validating whether the generated code is working as it is expected with respect to its model. The implementation code is co-simulated on a development host with the plant model to test its correctness. The details added by the code generation tool must not affect the controller behavior, and the optimized code must be compliant with the model behavior.

1.4.3 Processor-in-the-Loop

The main purpose of Processor-in-the-Loop is to validate the correct behavior of the generated code once it is implemented into real embedded hardware (either evaluation board or ECU). The generated code is executed by the target hardware, while the plant model is executed on a development host.
Co-simulation is used to enable the connection of the controller software running on the target hardware with the plant model. Since the plant is still running in the native simulation tool of the development host (i.e., Simulink), the time spent can be faster or slower than the actual system: the control system is not working in real-time. Therefore, this simulation is very close to the real implementation, but only the functional aspect is considered, while the timing aspect is neglected. The design can only verify whether the platform-dependent code works properly in compliance with hardware requirements.

1.4.4 Hardware-in-the-Loop

Hardware-in-the-Loop's purpose is to validate the functional and timing behavior of real-time software. The controller software code, obtained through automatic code generation starting from a model, is executed into a target hardware (ECU) that communicates with dedicated hardware that emulates the real plant via a suitable harness cabling (real-time simulator).

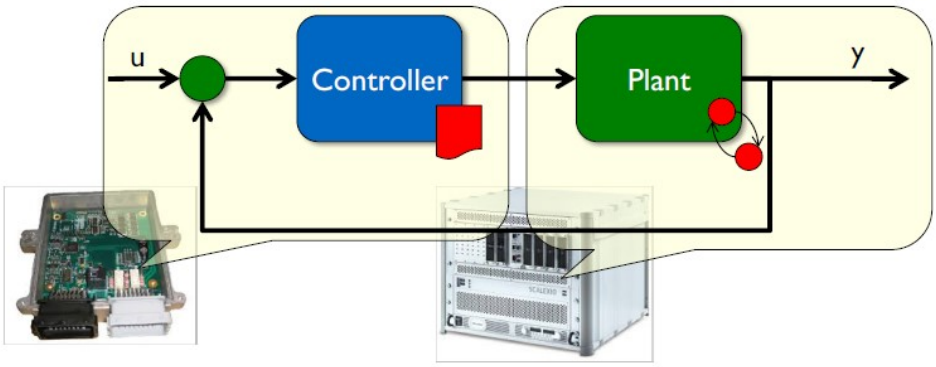

Figure 1.10: HIL behavioural model

The implementation code is co-simulated with the plant to test its correctness. The code runs on the target hardware, while the plant on rapid prototyping hardware. In this way, it is possible to test interactions with hardware and real-time performance. HIL testing is heavily used in the automotive industry for a complex real-time embedded control system, such as the **ECM**.
Even though *in-vehicle tests* are the most realistic scenarios, they have some limitations:

- dangerous: the driver can be in danger due to a test fault;
- limited: real plant can limit the feasibility of test cases;
- not deterministic behavior: repetition of a maneuver cannot always be the same;
- very expensive: a working prototype must be built.

These limitations are overcome by **HIL**, whose main advantages are:

- *reduced risk associated with failure*;

- *testing with several plant variations*: HIL allows to model and simulates different plants in a virtual environment in order to perform several tests with different variations in few time;
- *reduced time-to-market and costs*: HIL is usually performed in parallel with the plant development. This allows testing the controller before other components of the control system are ready. As a result, the time spent and costs are reduced;
- *testing faults mode*: faults can be introduced through software in safety conditions in order to test the behavior of the embedded system in dangerous situations. This testing mode is very difficult to perform in a real plant and may harm the driver.

In a typical HIL simulator, suitable for ECM testing, a real-time processing unit executes a mathematical model, which emulates the engine dynamics and the behavior of virtual sensors and actuators. I/O units provide all the possible channels needed to connect the ECM connectors correctly to the simulator. This enables to measure ECM outputs signals, use them in the real-time application computation, and to provide input signals to the control unit.
In addition, some vehicle sensors and actuators can be physically connected to the simulator as external components: this is done in order to get a more realistic behavior of components that are extremely non-linear and difficult to model precisely.
Furthermore, the ECM is not isolated from the rest of the vehicle but communicates with almost all the other vehicle ECUs through a CAN/LIN network; to simulate this, virtual ECUs are implemented inside the system model in order to reproduce the control logic behind their CAN messages read by the ECM.
At the end of the configuration, the ECM behaves like it is connected inside a real vehicle, and the simulator provides the required signals to validate its control application.

Chapter 2

dSpace SCALEXIO simulator

A typical HIL simulator for automotive applications is the SCALEXIO, realized by dSpace company.

2.1 Overview of SCALEXIO Systems

SCALEXIO systems are designed for testing ECUs in a Hardware-in-the-Loop simulation. Figure 2.1 shows an example of a SCALEXIO system that is connected to the ECU, external loads, a tester device, and the host PC [6]:

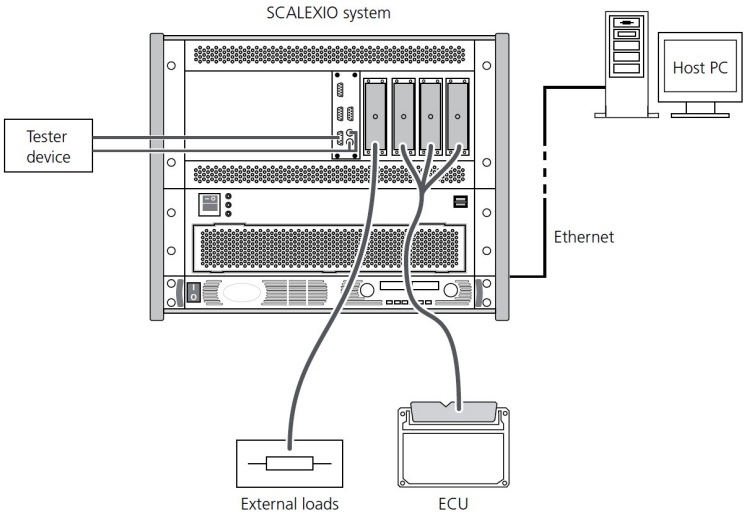

Figure 2.1: SCALEXIO system with main connections

A SCALEXIO system simulates the ECU-controlled system in order to perform tests. Considering the particular case of an ECM, the simulator behaves like a real vehicle. Indeed, the SCALEXIO system can:

- provide the battery voltage to the ECM;
- generate the sensor signals which are required by the ECM;
- measure the signals which the ECM generates for actuator control;
- connect the output signals of the ECM to loads to simulate the actuators;
- receive bus signals which are sent by the ECM and send bus signals to the ECM;
- simulate failures in the wiring of the ECM.

The behavior of the controlled system is specified by a real-time application that runs on the SCALEXIO Processing Unit. The real-time application is implemented on the host PC and downloaded to the SCALEXIO system via Ethernet.

2.1.1 Hardware

The real-time hardware of a SCALEXIO consists of several boards or unit types:

- **SCALEXIO Processing Unit**, which provides the calculation power to calculate the real-time applications;
- **IOCNET**, which connects the real-time processor with more than 100 I/O nodes to I/O units. it executes real-time performance plus time and angular clocks;
- **MultiCompact I/O units and boards**, which provide a greater number of I/O channels for specific applications. The channels have failure routing units in order to perform failure simulations. MultiCompact I/O units provide ECU/load connectors with fixed pinouts: the typical unit used is **DS2680 I/O Unit**, which is very important in automotive simulation, and it is explained in details later on. DS2690 is a digital I/O board providing 30 digital I/O channels for signal measurement and signal generation;
- **HighFlex I/O boards**, which are used for different I/O functions since they are versatile and finely scalable. The channels are selected and configured in *ConfigurationDesk*. Typical boards are DS2601 for signal measurement, DS2621 for signal generation, DS2642 for the power switch and failure simulation, DS2671 bus board;

- **SCALEXIO I/O boards**, which provide a large number of I/O channels with dedicated channel types and a focus on I/O functions without a current-related functionality;

- **Slot units**, which are used to install SCALEXIO boards in a SCALEXIO rack;

Hence, the SCALEXIO simulator can have several hardware configurations depending on the units and boards installed. Figure 2.3 shows the front view of a SCALEXIO system that includes a DS2703 6-Slot Unit and DS2680 I/O Unit, connectors for connecting external devices to the simulator.

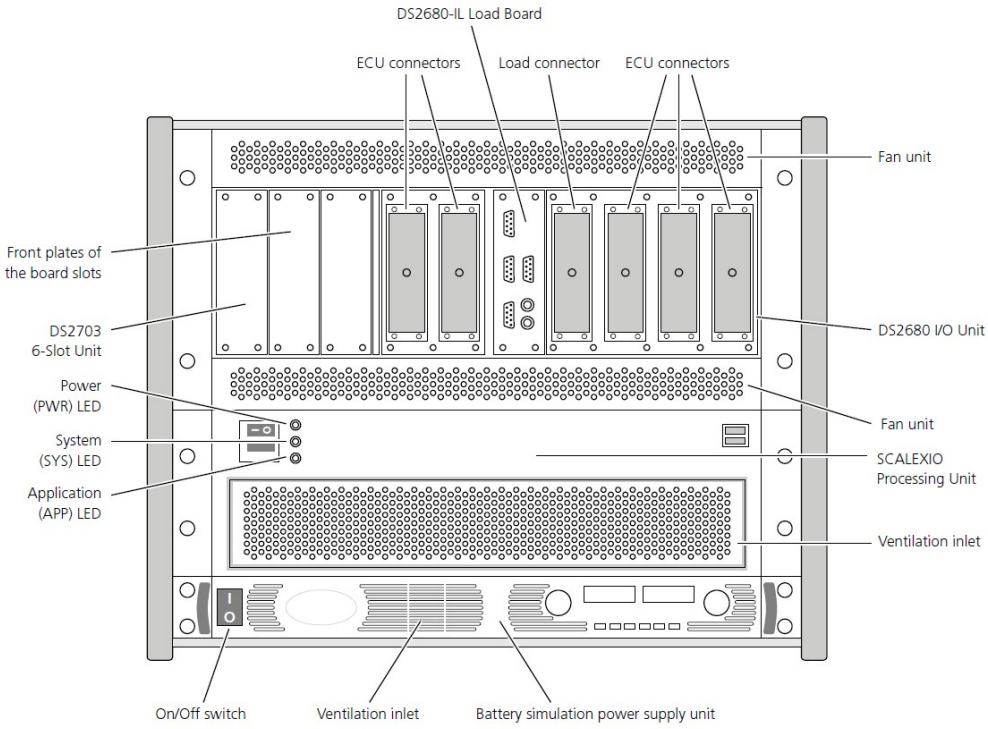

Figure 2.2: Front view of a SCALEXIO system

Two important units are present: **SCALEXIO Processing Unit** and **Battery simulation power supply unit**. A typical simulator front can have the following connectors:

- *ECU connectors* to connect the ECU to the simulator. The connectors contain all input and output signals for the ECU (sensor signals, actuator signals, bus signals, and power supply).

- *Load connector* to connect external loads to the simulator. Indeed the actuators or an equivalent external load must be connected to the ECU via the simulator, which can measure the signals for actuators.

Other specific connectors are available in the DS2680 I/O unit.

At the rear are connectors for connecting the simulator to the Ethernet and mains. A typical simulator rear can have the following connectors:

- *Mains connectors* to connect the simulator to the mains.
- *Ethernet connector* to communicate with the software tools running on the host PC.
- *Second Ethernet connector* to establish an independent Ethernet connection to a customer-specific device.
- *IOCNET connector* to connect the simulator to another simulator or I/O unit.
- *24V out connector* to provide a voltage of 24V for an external device.

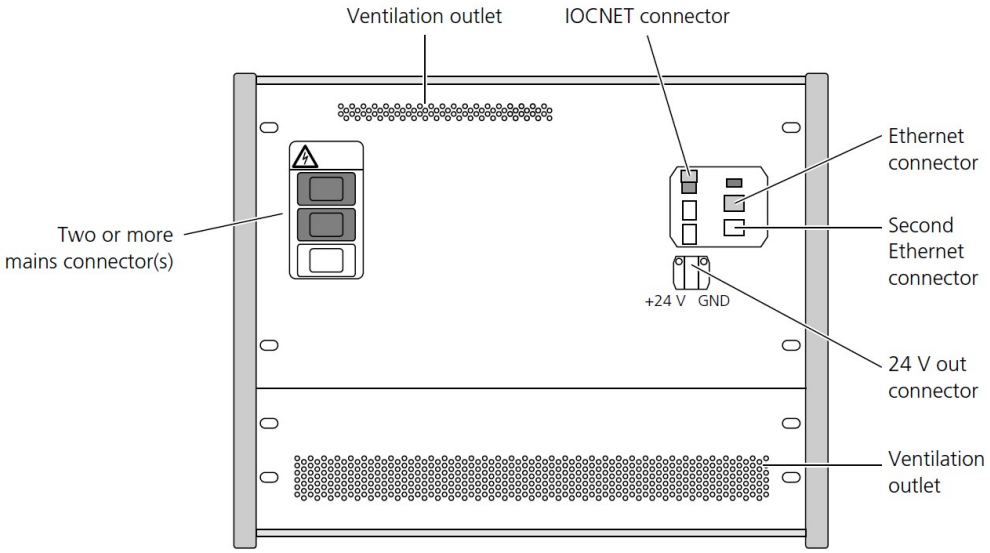

Figure 2.3: Rear view of a SCALEXIO system

SCALEXIO Processing Unit

The SCALEXIO Processing Unit provides the calculation power to calculate the real-time application.

Figure 2.4: A SCALEXIO Processing Unit

Its important components are:

- The *SCALEXIO Real-Time PC* consists of a high-standard industry ATX motherboard and a multi-core main processor. It has three status LEDs which are located on the simulator front: the *PWR* LED indicates whether the power supply is available; the *SYS* LED indicates the states of the boot and initialization process of the real-time PC; the *APP* LED indicates the states of a simulation run.

- The *DS2502 IOCNET Link Board* which provides the interface to communicate with SCALEXIO HighFlex boards and/or MultiCompact I/O units via IOCNET. This Board provides the following features:

  - IOCNET router with four or eight optical ports for IOCNET or Gigalink connection.

  - Master clock unit.

  - Six **APUs (Angular Processing Units)**, each with a resolution of 16 bit to simulate engine processing core functions. The supported applications are: injection/ignition capturing, crankshaft/camshaft signal generation, knock signal generation, 24 angle-based memories.

  - Support of up to eight CPU cores.

  - Active cooling.

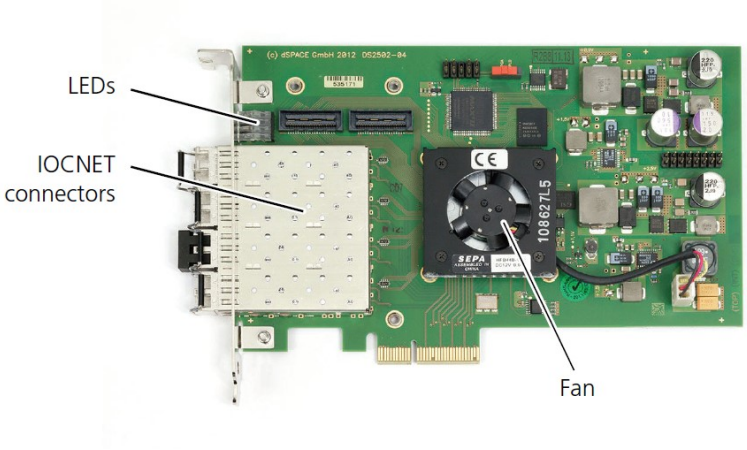

Figure 2.5: DS2502 IOCNET Link Board overview

Battery simulation power supply unit

The battery simulation power supply unit is used to generate voltages and currents used to simulate the vehicle battery. This unit is controlled via software by a battery simulation controller. A power switch is implemented for switching the outputs of the battery simulation power supply unit to pins on the simulator's front connectors or internal power supply rails.

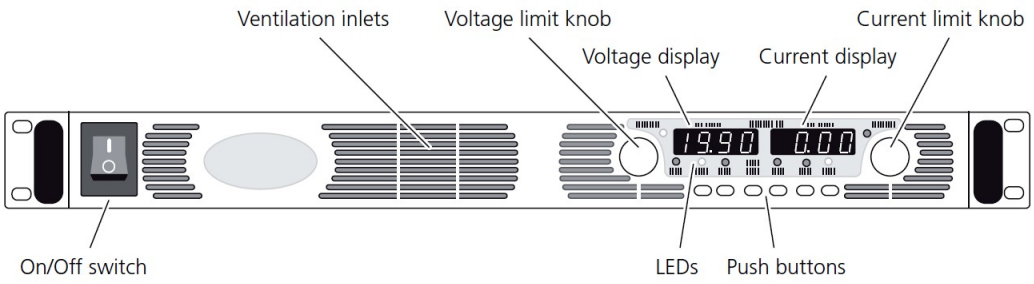

Figure 2.6: A battery simulation power supply unit of the Genesys^{TM} 1500W series

The unit's On/Off switch is used to switch the battery simulation power supply, the internal power supply of the simulator, and the SCALEXIO Real-Time PC. A voltage limit knob and a current limit knob allow adjusting settings via the front panel. The actual values simulated are visualized in a voltage display and a current display. Furthermore, some push buttons configure the battery simulation power unit manually, then the status and function information can be visualized

through LEDs. Hence, it is possible to specify an undervoltage limit or overvoltage protection via the front panel: they cannot be adjusted via software.

2.1.2 Software Tools

In order to execute real-time simulations, the SCALEXIO system must be implemented with a real-time application and must be controlled by a host PC. Software tools running on a host PC are used to work with the SCALEXIO system via Ethernet. The host PC not only creates the real-time application that models the controlled system but also implements and manages the application on the SCALEXIO simulator. Furthermore, the host PC can change parameter values and measure variables of the real-time application.
Failures in the ECU wiring are simulated by the host PC by controlling the failure simulation.
The software tools are grouped into three phases:

- **Implementing**: software tools and block sets for modelling and implementing the real-time application:
  - MATLAB/Simulink, which is used for modelling only the behaviour model of the controlled system;
  - ConfigurationDesk, which is used for implementing the I/O functionality on the simulator. This tool can build a real-time application containing the behaviour and I/O model;
  - Model Port Blockset, which contains Simulink blocks used to read or write signals of the I/O model;
  - RTI CAN MultiMessage Blockset, which contains Simulink blocks for implementing a CAN communication in the behaviour model;
  - RTI LIN MultiMessage Blockset, which contains Simulink blocks for implementing a LIN communication in the behaviour model;
  - FlexRay Configuration Package that is used for implementing FlexRay communication in the behaviour model;
  - Automotive Simulation Models, which are block sets useful for modelling the controlled system in Simulink;
  - RTI FPGA Programming Blockset.
- **Experimenting**: software tools for handling and controlling the real-time application and visualizing the simulation results:
  - ControlDesk Next Generation, which can be used for downloading the real-time application, calibrating parameters and measuring signals;

  - MotionDesk, which shows 3D animation during experimenting with a SCALEXIO system
  - ModelDesk used to change and download the parameters for Automotive Simulation Models to the real-time application.

- **Testing**: software tools for ECU test automation:
  - AutomationDesk, which can create and manage automation tasks;
  - Real-Time Testing, which can execute tests synchronously with the model.

2.1.3 DS2680 I/O Unit

The DS2680 I/O Unit is a SCALEXIO MultiCompact unit conceived for powertrain and vehicle dynamics projects, especially for HIL testing of ECUs. The unit is mounted in the SCALEXIO rack, and it is released with an integrated load board for internal loads and can be optionally equipped with a bus module.

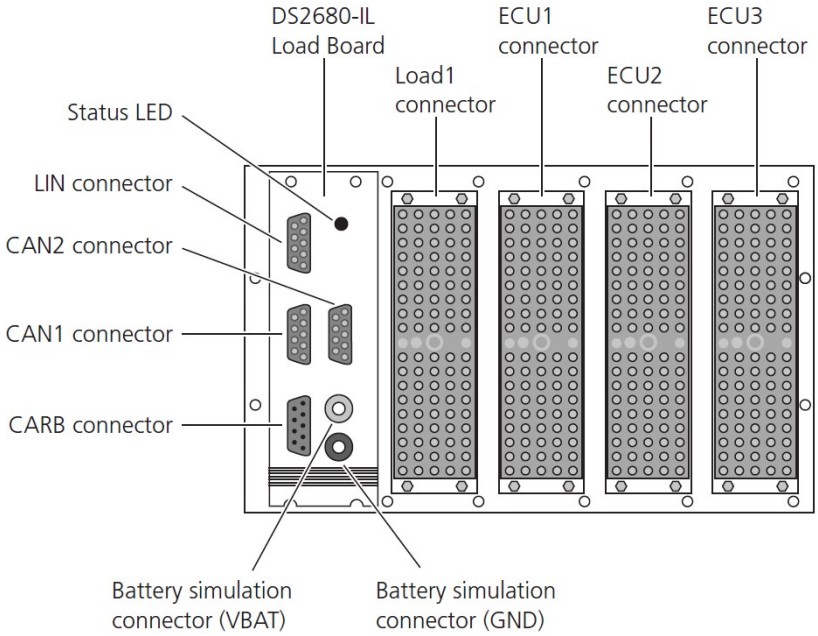

Figure 2.7: Front overview of the DS2680 I/O Unit

The unit's front has the following connectors and elements:

- *Battery simulation connectors* to connect devices to the simulated vehicle battery.

- *CAN1* and *CAN2* connectors to connect a device to a CAN bus if a DS2672 Bus Module is installed.
- *LIN* connector to connect a device to a LIN bus if a DS2672 Bus Module is installed.
- *CARB* connector to connect a tester device to the simulator.
- *ECU1*, *ECU2* and *ECU3* connectors to connect the ECU to the simulator. The connectors contain all input signals for the ECU (sensor signals, actuator signals, bus signals and power supply).
- *Load1* connector to connect external loads to the simulator.
- *Status LED* to indicate the state of the I/O unit.

The DS2680 provides a **Failure Insertion Unit (FIU)** consisting of a central FIU, fail rails and failure routing units (FRUs): they can be used for failure simulation with SCALEXIO HighFlex boards.
The **DS2680-IL Load Board** is a removable board of the SCALEXIO I/O Unit that can carry internal loads for each signal measurement channel and the special high-speed channels for lambda probes. It also provides some of the I/O unit's front connectors and elements.

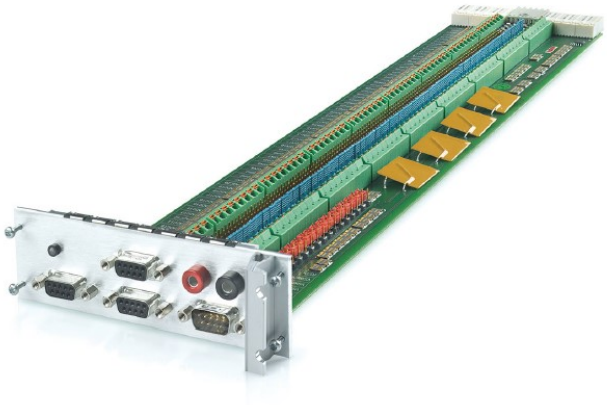

Figure 2.8: DS2680-IL Board

The DS2680 is connected to the SCALEXIO Processing Unit through an **Internal IOCNET router**, which provides also other ports for connecting DS2680 to external elements.
ConfigurationDesk is the software tool used to configure the different DS2680 I/O Unit's channels, that are classified in several types depending on the task performed:

- **Channels for battery simulation control and power switch**:
  - 1 *Power Control 1* to control the simulator's battery simulation power supply unit;
  - 6 *Power Switch 2* to switch the battery simulation supply voltage to connected external devices.
- **Channels for signal measurement**:
  - 20 *Analog In 1* for analog voltage measurement;
  - 30 *Digital In 1* for digital voltage measurement;
  - 18 *Flexible In 2* for digital voltage/current measurement and analog current measurement.
- **Channels for signal generation**:
  - 15 *Analog Out 1* for analog voltage generation;
  - 7 *Analog Out 3* for transformer-coupled voltage output;
  - 8 *Analog Out 4* for voltage or current generation, configurable as a current sink;
  - 28 *Digital Out 1* for digital voltage generation, configurable as a low-side or high-side switch, or for push/pull mode;
  - 12 *Resistance Out 1* for resistance simulation.
- **Specific channels for simulating lambda probes**:
  - 2 *Analog In 2*, high-speed ADC channels;
  - 2 *Analog Out 2*, high speed DAC channels;
  - 2 *Load 1*, component channel. Each high-speed channel can be combined with a *Resistance Out 1* for resistance simulation.

The connection between the DS2680 I/O Board of the SCALEXIO simulator and the ECM is performed by analyzing the electrical scheme of the engine control system provided by the carmaker. This study leads to understanding what type of connection every actuator or sensor needs, hence in which channel of the simulator the ECM pin must be connected.
As previously mentioned, some complex actuators with strong non-linear behaviour are not simulated by the simulator, but external physical components are connected to the SCALEXIO system, which behaves as an interface for the ECU connection. The list of all the connections between ECM and simulator is called **wiring harness**. It is created to use in the most appropriate configuration all the available channels

of the SCALEXIO system.
The purpose of the simulation is to recreate the same environment of the real engine control system that will be connected to the tested ECM.
In the following paragraphs, the most used channels for ECM connection and useful for the purposes of this thesis are explained with circuit diagrams.

Power switches

The DS2680 I/O Unit provides six *Power Switch 2* channel type to switch the battery simulation supply voltage to the ECM or other connected external devices. The power switch channels can be switched on or off via software, and they can also be used by the central FIU to simulate short-circuits to ECU ground or battery voltage.
Each channel has four VBAT pins. Each VBAT pin can be used in the range 0 to +60 V and 0 to +6 A_{RMS}

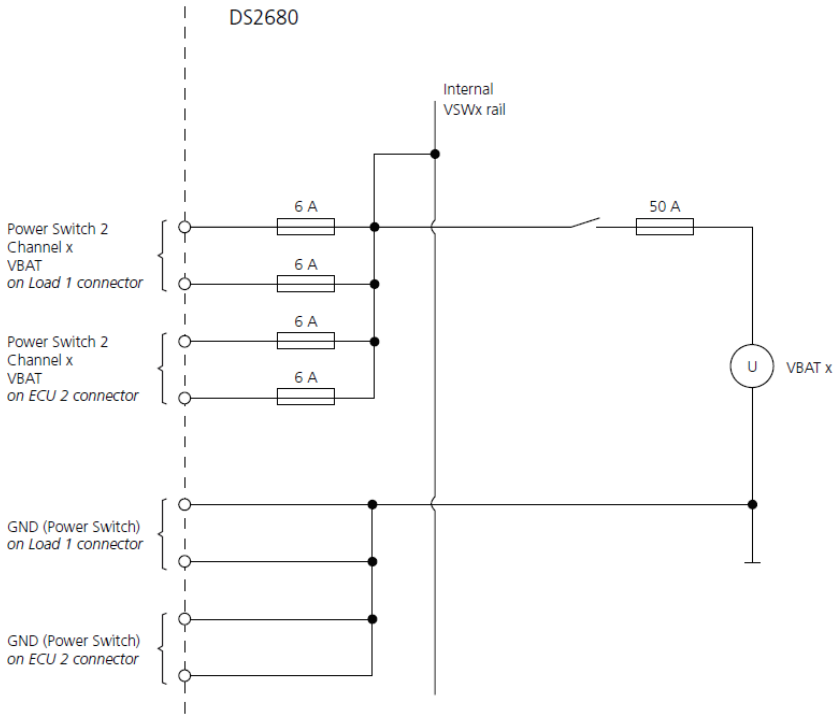

Figure 2.9: Circuit of one Power Switch 2 channel type

Signal measurement channels

The DS2680 provides different signal measurement channels to measure the analog and digital output signals of an ECU. Channel multiplication is allowed for these

type of channels: it is possible to connect them in parallel for current enhancement till a maximum overall current of 50 A_{RMS}.
It is possible to connect internal and/or external loads to each signal measurement channel in order to provide the electrical characteristic of an actuator to the ECM. Each channel provides a failure routing unit (FRU) to simulate electrical failures for the measured signals.

- **Analog In 1 channel type** is used to measure an analog voltage signal in the range 0 V to +60V. Each channel provides 3 pins: one for signal measurement, one to provide signal reference and one to switch the analog voltage on the load board.

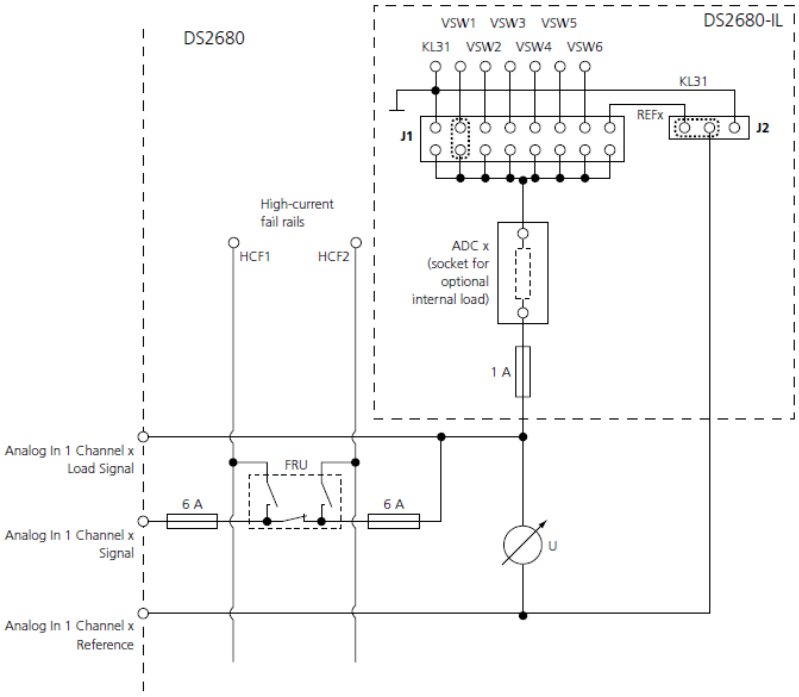

Figure 2.10: Circuit of one Analog In 1 channel type

- **Digital In 1 channel type** is used to measure a digital voltage signal in the range 0 to +60 V with a threshold voltage from 0 to +23.8 V. Each channel provides three pins: 1 for signal measurement, one to provide signal reference and one to switch the analog voltage on the load board.

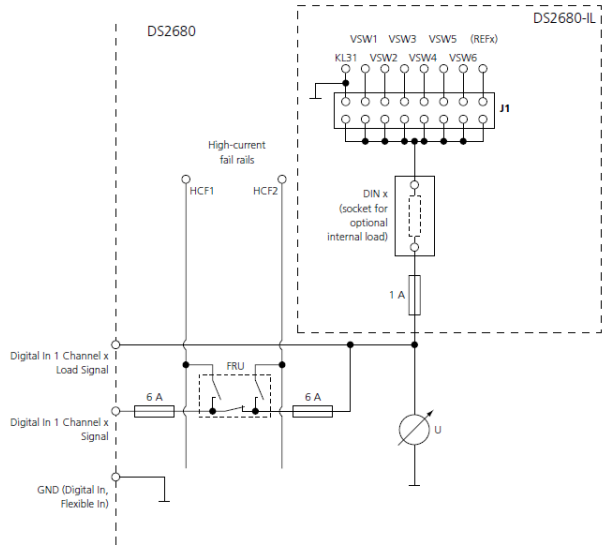

Figure 2.11: Circuit of one Digital In 1 channel type

- **Flexible In 2 channel type** is used to measure digital voltage, digital current or analog current signal. Each channel provides three pins: one for signal measurement, one to provide signal reference and one to switch the analog voltage on the load board.

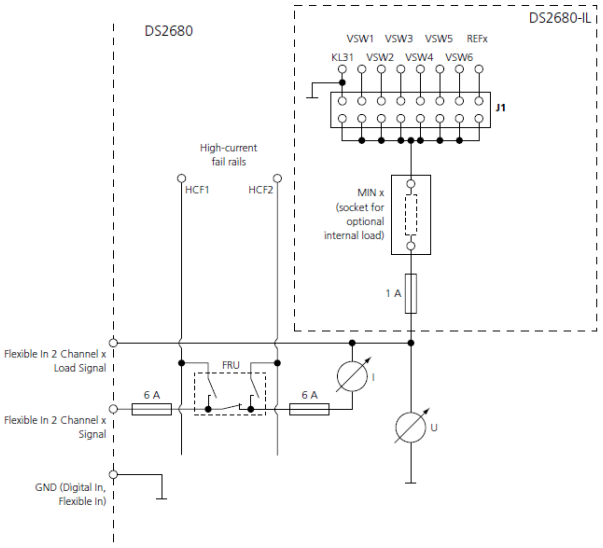

Figure 2.12: Circuit of one Flexible In 2 channel type

Channels for signal generation

The DS2680 I/O Unit can generate signals to simulate not only digital ECU input signals but also digital and analog sensors, resistors and potentiometers. This type of channel can also generate pulse-width modulated (PWM) and pulse-frequency modulated (PFM) signals. Current and/or voltage enhancement is allowed connecting the channels in parallel and/or series: the maximum voltage must not exceed 60 V, and the maximum current must not exceed 50 A_{RMS}.

- **Analog Out 1 channel type** is used to output voltages for sensor simulation. It has two pins: one to provide the voltage signal and one for the signal reference.

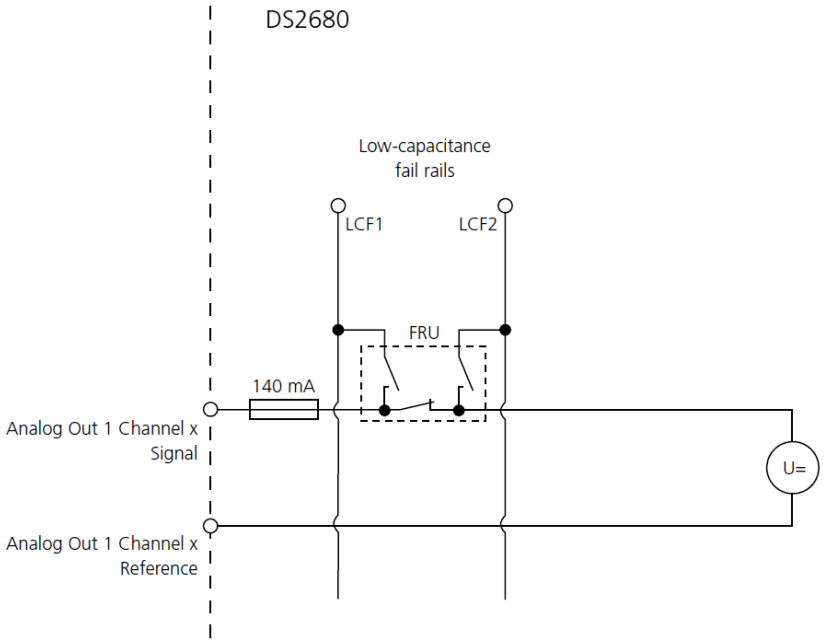

Figure 2.13: Circuit of one Analog Out 1 channel type

- **Digital Out 1 channel type** is used to simulate a digital output stage of 0V or in the range between +5 V and +60 V. The channel is configurable as a low-side or high-side switch or for push/pull mode. Each channel has a common pin for connecting the external reference voltage. The low-side switch can only switch to GND. The high-side switch can be set to an internal reference voltage (indicated as VBAT) or an external reference voltage greater than +5V.

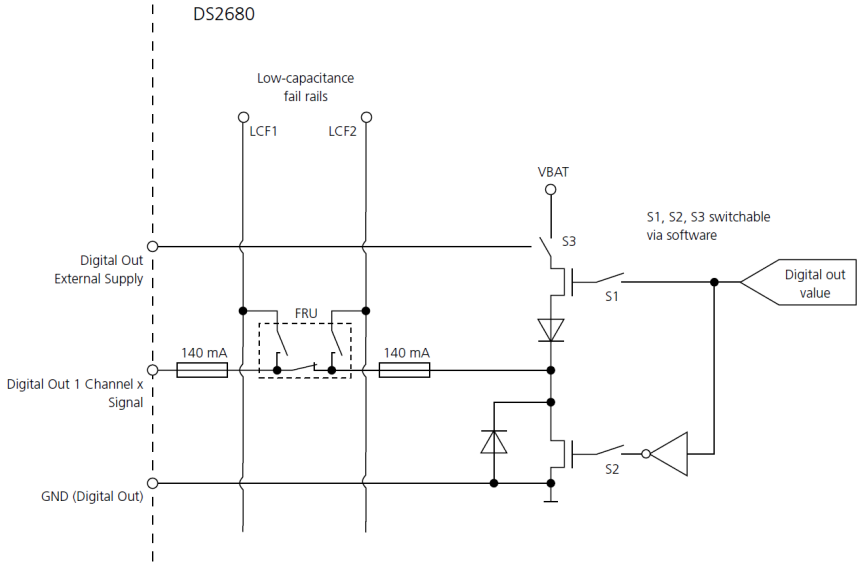

Figure 2.14: Circuit of one Digital Out 1 channel type

- **Resistance Out 1 channel type** is used for resistance simulation. Each channel has two pins: one for the output signal and one for the reference signal.

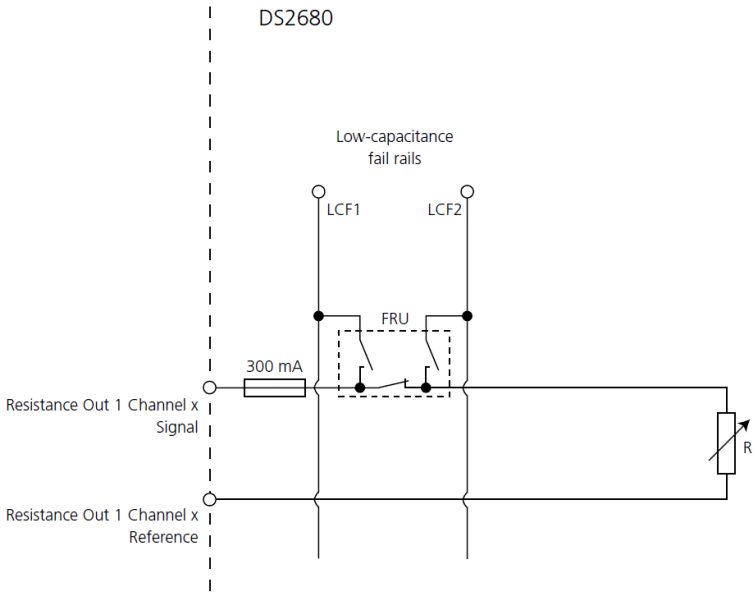

Figure 2.15: Circuit of one Resistance Out 1 channel type

Chapter 3

Engine simulation

One of the most important simulation that a SCALEXIO simulator can perform is the four-stroke piston engine's behaviour. The virtual engine sends realistic sensor signals to and receives realistic actuator signals from a real ECM. Some engine components can be real and are part of the SCALEXIO simulator.

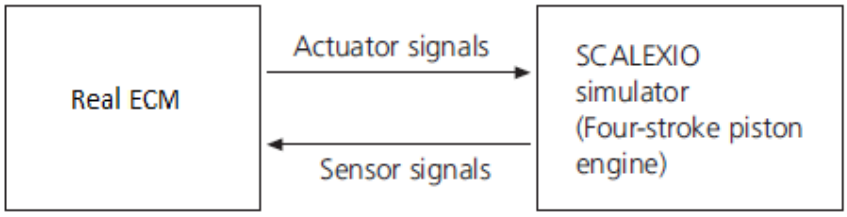

Figure 3.1: A conceptual overview of an Engine simulation in SCALEXIO simulator

In ConfigurationDesk, a real-time application of the virtual engine is built. It consists of two models:

- **I/O model**: it is built in ConfigurationDesk and can contain the following engine I/O functionality: crankshaft and camshaft signal generation; injection and ignition pulse measurement; lambda signal generation; knock signal generation.
- **Behavior model**: it is built in MATLAB/Simulink and mimics the engine's behaviour.

The **Engine Simulation Setup** function block is crucial for the configuration of virtual piston engines as it defines piston-specific characteristics. It provides the *Piston Position* function port, which outputs the current angle positions [°] of the

virtual cylinders to the behaviour model.
An **Angular Clock Setup** function block must be assigned: it provides a connection to the APU and the basic configuration data of the virtual engine. The APU computes the virtual engine's current angular position as a function of the rotation speed of the virtual crankshaft provided by the behavioural model. The Angular Processing Unit is implemented as a 16-bit angle counter. The step size of the angle counter is 0.011°. Its exact range is from 0° to 719.989°, and if the upper limit is overtaken, the counter is reset to 0°. The APU can count forward or backwards depending on the signal's sign at the *Velocity* function port of the Angular Clock Setup function block.
The DS2680 I/O board of the SCALEXIO system can be connected with six usable angle units of the SCALEXIO Processing Unit.

3.1 Crank and Cam

The main objective of this master thesis is to generate a crank signal and a cam signal through a simulator implemented with a microcontroller evaluation board. For this reason, a description of the characteristic of Crankshaft and Camshaft Sensors is fundamental.

3.1.1 Operation Principle of Crankshaft and Camshaft Sensors

The reciprocating linear piston motion of four-stroke piston engines is translated into crankshaft rotation. The crankshaft then transmits the torque to the wheels via the gearbox. One or even multiple camshafts are mechanically coupled to the crankshaft and control the intake and exhaust valves. One engine cycle of a four-stroke piston engine corresponds to two crankshaft revolutions and one camshaft revolution.

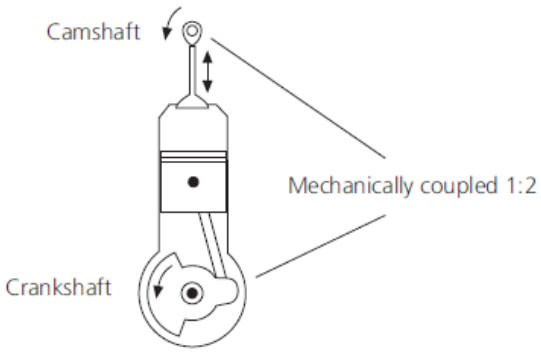

Figure 3.2: Crankshaft and Camshaft Coupling

The angular position of the crankshaft is linked with the position of the piston. However, to identify the exact stroke, the camshaft's angular position must also be known. The **crankshaft sensor** is used to measure the angular position and the rotational speed of the crankshaft, which the ECM uses to control the fuel injection and other diesel engine parameters. It is an optical, or inductive, or magnetic sensor that scans the radial surface of the rotating crankshaft wheel and detects teeth' presence. Indeed, an even-toothed wheel is flanged to the crankshaft and is typically composed of "60 - 2" teeth (there are 60 teeth, but two of them are missing to indicate a defined position).
The **camshaft sensor** is an optical, or inductive, or magnetic sensor that scans the radial surface of the rotating camshaft wheel and detects the presence of teeth. This wheel is flanged to the camshaft and has a particular sequence of teeth depending on the carmaker choice. The combination of a crankshaft position sensor and a camshaft position sensor is used by the ECM to monitor the relationship between the pistons and valves in the internal combustion engine. Therefore, the synchronization of the four-stroke engine depends on these two sensors, which allow the ECM to know when to inject the fuel.

It is possible to find two types of crankshaft and camshaft sensors:

- *Passive sensors* (inductive sensors) which generate an analog signal. A rising tooth edge on the wheel corresponds to the signal crossing zero from positive to negative signal level.

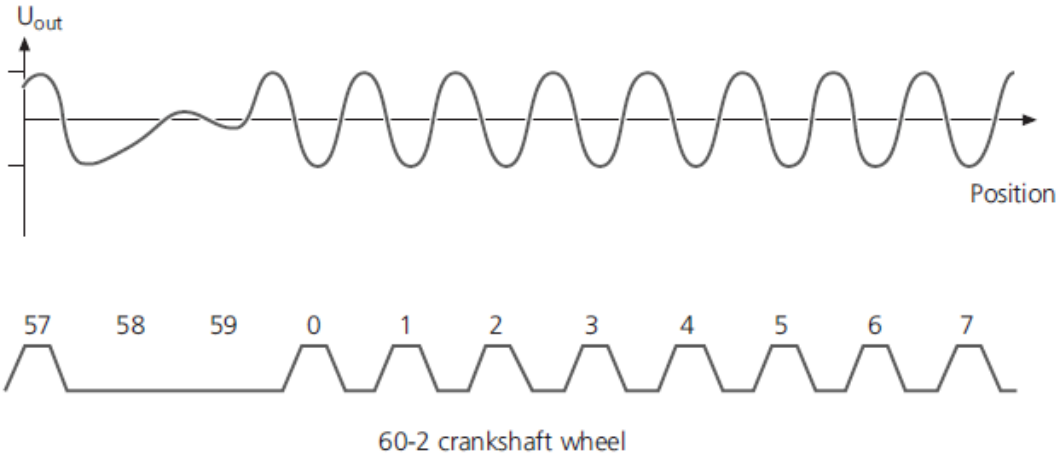

Figure 3.3: Passive crankshaft signal

- *Active sensors*, typically the Hall-effect sensors, generate a digital signal. A rising tooth edge on the wheel corresponds to a falling signal edge.

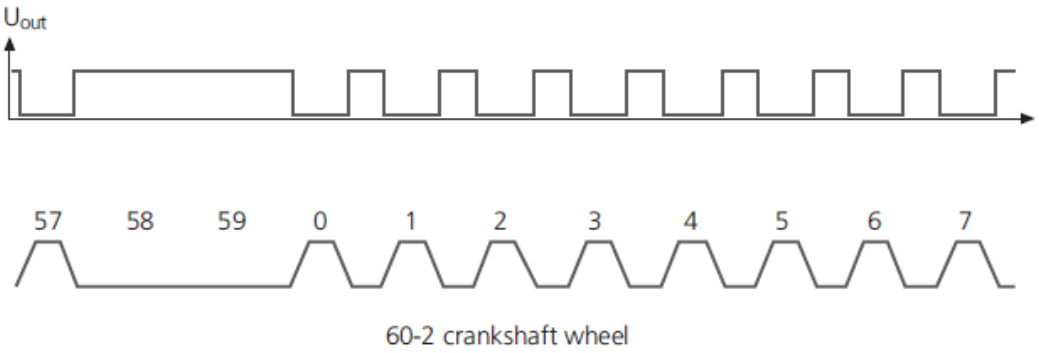

Figure 3.4: Active crankshaft signal

Reverse crankshaft sensors

In some cases, for example, in modern Stop&Start automotive technology, it is required that the crankshaft performs a reverse rotation. Standard crankshaft sensors can not detect this condition, but **reverse crankshaft sensors** are needed. These sensors generate pulse signals, whose temporal pulse length depends on the current rotation direction (forward: tf=45 μs, reverse: tr=90μs).

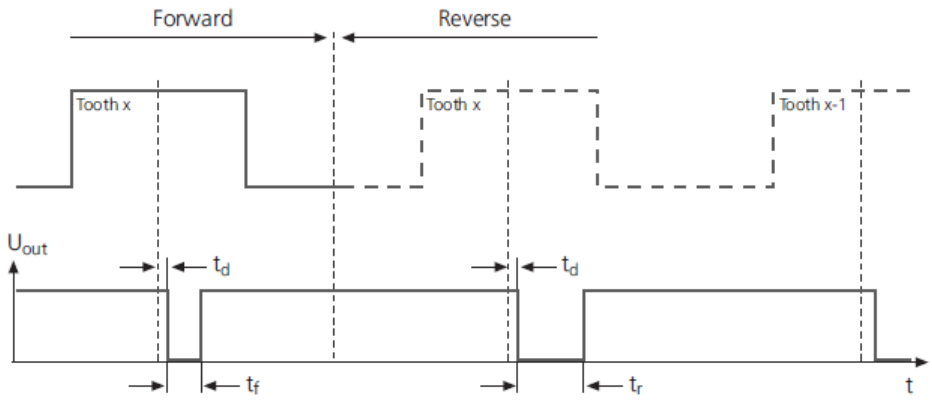

Figure 3.5: Example of reverse crankshaft signal

The trigger point is usually the centre of the tooth. Td is a time delay that represents the sensor's reaction time between trigger and pulse.

3.1.2 Crank and Cam Wavetables

The SCALEXIO system generates crankshaft and camshaft signals useful for the ECM starting from a numeric list called **wavetable**. Crankshaft wavetables are 1-dimensional arrays of float type values in the range between -1.0 and +1.0, stored in CSV files (CSV files are ASCII files containing entries separated by a comma, where no additional data or text is included). The array comprises $2^{16} = 65536$ values: this number is given by the step size of the angle counter, which in this case, it is equal to 0.011°.
The active wavetable is accessed like a look-up table during an engine simulation. The current value of the angle counter acts as the index: as the angle counter continuously increments during engine simulation, the wavetables values are read out consecutively. In the reverse crankshaft and camshaft rotation simulation, the wavetables values are read out backwards.
The output signal value depends on the simulated sensor type:

- in passive crankshaft sensor simulation, the wavetable values are multiplied by the *Amplitude* function port, set before in ConfigurationDesk, giving several output signal values;
- in active crankshaft sensor simulation, the wavetable values are interpreted as 0 if they are less or equal to 0.0, else are 1.

Wavetables are stored in the wavetable memory of the used I/O board when the real-time application is downloaded to the target hardware. The DS2680 I/O board can contain up to 24 wavetables.

3.1.3 Function Block Configuration

The function blocks are elements of the dSpace function library and implement the I/O functionalities. Each block type has unique features and must match the electrical characteristic of its related signal to realize the interface between simulator and model. Hence, in ConfigurationDesk, after the definition of Engine Simulation Setup block and of Angular Clock Setup block, it is possible to simulate crankshaft and camshaft sensor signals using different *Crank/Cam Out* function block types:

- Crank/Cam Voltage Out function block type simulates passive sensors and supports Analog Out 1, Analog Out 3 and Analog Out 4 channel types available on DS2680;
- Crank/Cam Digital Out and Crank/Cam Current Sink function block types simulate active sensors and reverse crankshaft sensors. Crank/Cam Digital Out supports Digital Out 1 channel type, and Crank/Cam Current Sink

supports Analog Out 4 channel type: both channel types are the ones available on DS2680.

Other channel types are supported by these function blocks but are available in the I/O board different from DS2680. The function blocks provide signal ports and function ports depending on the specified settings.

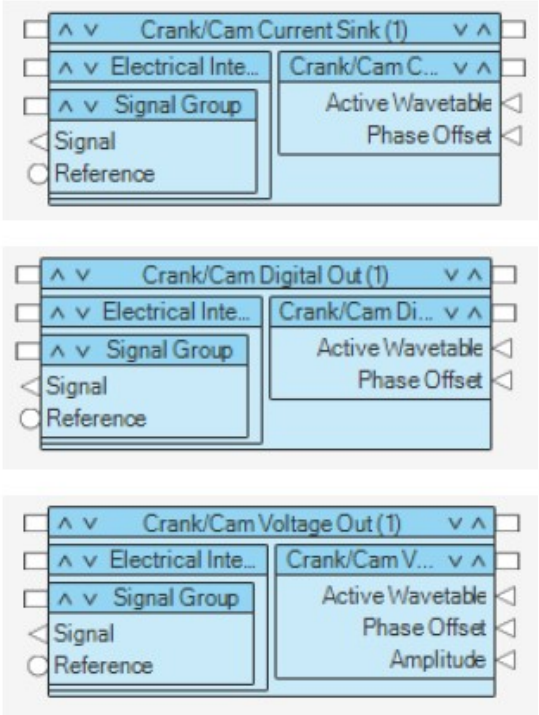

Figure 3.6: Crank/Cam function blocks with their default settings

Additionally, they can provide particular configuration features which influence the behaviour of the basic functionality: for example, referencing of the virtual piston engine and the wavetables, setting a phase shift for the camshaft, specifying generation of reverse crankshaft signals.

Chapter 4

Crank/Cam Simulator Hardware

As previously mentioned, the automotive electronic control systems (particularly the ECM) are increasing their complexity to meet more and more stringent requirements: reducing pollutant emissions and fuel consumption and making the system functional safe. Therefore, the development and testing of embedded software need a strong effort in terms of time and, consequently, costs.
Automotive industries are continuously searching for new methods for testing ECU in a faster way, keeping the same security standard. In this context, the work of this master thesis is created for companies that make some specific ECU tests without the need to buy an expensive and complex simulator, as the whole SCALEXIO system. In particular, the core of this thesis is the design and realization of an open-loop simulator generating crankshaft and camshaft sensors signals. These signals are taken as references for simulating the four-stroke piston engine's behaviour and for synchronizing the injection or ignition engine phase.
An **open-loop simulator** is created since the user sets the desired engine speed as an input, and the corresponding output signal is generated independently from the controller action on the output itself.
The main hardware components of the developed simulator are the following:

- A **microcontroller evaluation platform**, used to develop the application and generates the signals;
- A **memory touch display**, used to implement the HMI and to control the microcontroller;
- A **rotary encoder**, used as an input device to regulate the engine rotational speed value.

4.1 Microcontroller overview

In order to generate a fast response for generating crankshaft and camshaft signals at high engine speed, a high-performance automotive-grade microcontroller is needed. For this reason, it is chosen a highly integrated microcontroller: the **Hercules™TMS570LC43x LaunchPad™** developed by Texas Instruments Corporation.
This microcontroller is a cache-based evaluation platform implementing two ARM Cortex-R5F Floating-Point CPUs, and it is designed to aid in the development of ISO 26262 and IEC 61508 functional safety applications. The ARM architecture is based on a 32-bit reduced instruction set computer (RISC) developed by ARM Holdings. The load or store instructions used by **RISC** processors are small and highly-optimized thanks to the aligned memory access. In particular, the ARM Cortex-R5F processor is chosen since it targets real-time solutions and can fulfil more scalable tasks: it offers an efficient 1.66 DMIPS/MHz and can run up to 300 MHz providing up to 498 DMIPS [7]. The device supports the big-endian [BE32] format.

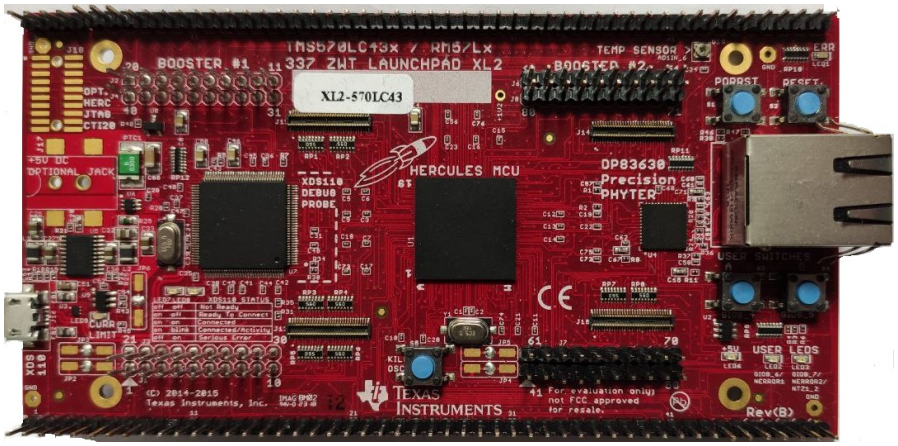

Figure 4.1: TMS570LC43x LaunchPad™

A USB micro cable provides the power to the microcontroller, and it is also the connection to a laptop for USB debugging of the program.
Good knowledge of the microcontroller architecture and memory usage is required to choose the correct strategy and use the needed microcontroller peripherals to develop the simulator.

4.1.1 Architecture

The microcontroller is based on the TMS570 Platform architecture consisting of two main subsystems, which behave as safe islands since they are separated: the CPU Interconnect Subsystem and the Peripheral Interconnect Subsystem.

- The *CPU Interconnect Subsystem*, which connects bus masters and bus slaves directly related to the CPU. It contains a high degree of safety diagnostics on the bus system and the memories, thanks to the Error Correction Code (ECC) used inside the CPU or parity detection. During application run-time, data going in and out are checked against their expected behaviours in the Safety diagnostic logic, which is built into the CPU Interconnect Subsystem. In addition, self-test logic is built in order to diagnose possible faults.
- The *Peripheral Interconnect Subsystem*, which connects the rest of the masters and slaves in the device. Also, in this safety island, ECC or parity protection ensure safety diagnostic on the peripheral memories.

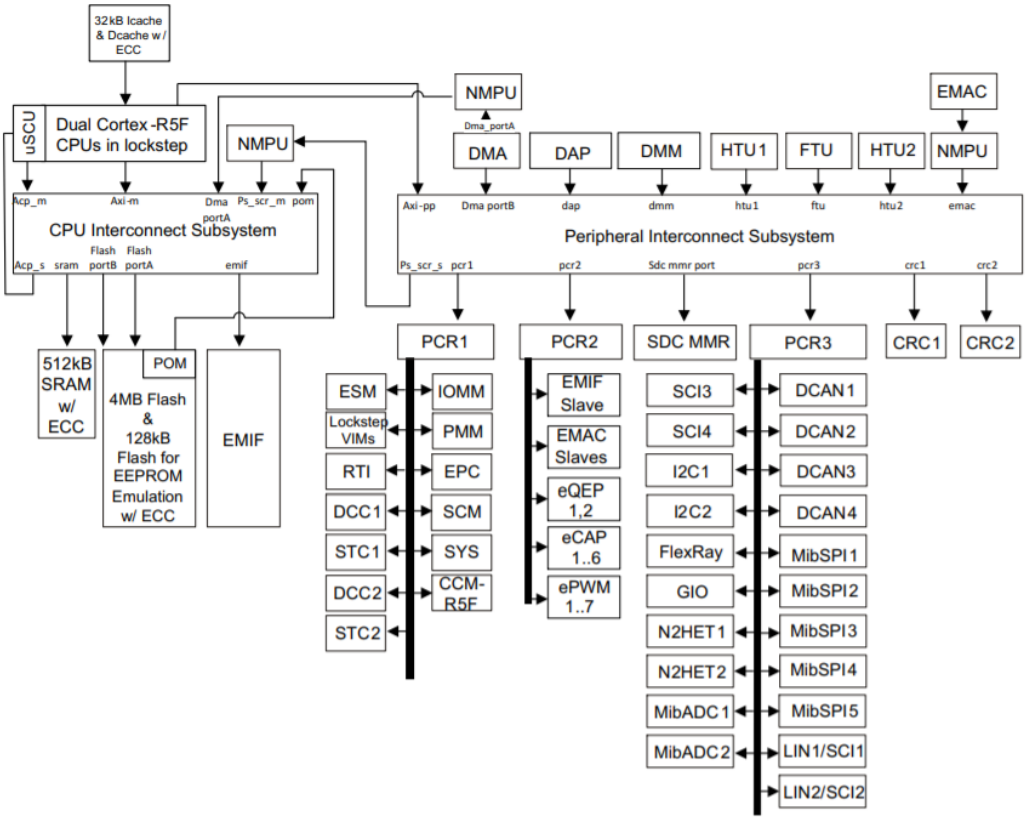

Figure 4.2: TMS570LC43x Architectural Block Diagram

4.1.2 Memory organization

The whole memory space available in the TMS570LC43x launchpad is equal to 4GB. This space is divided into several regions, each addressed by different memory selects. The ARM Cortex-R5F processor core starts execution from the *reset vector address* of 0x00000000 whenever the core gets reset: this address is also the starting address of the **Flash Memory**, a nonvolatile, electrically erasable and programmable memory implemented with a 64-bit-wide data bus interface. In the microcontroller, 4 MB of Flash memory is implemented and operates on a 3.3V supply unit for all read, program and erase operations. SRAM memory is used as program memory, so it hosts the application code.

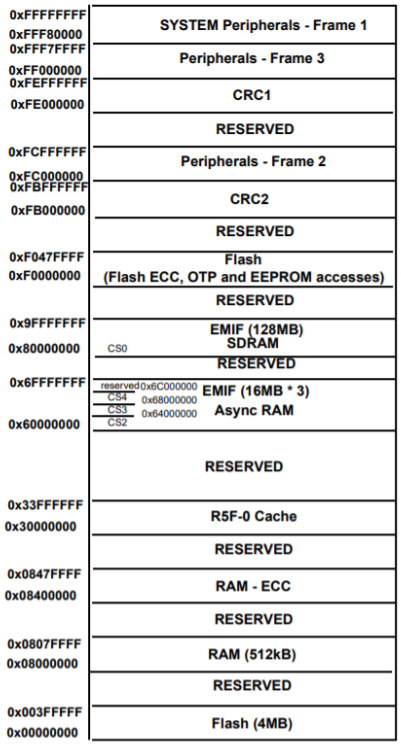

Figure 4.3: TMS570LC43x Memory Map

Up to 512KB CPU data *RAM* is implemented with single-bit error correction and double-bit error detection. The SRAM operates with a system clock frequency of up to 150 MHz, and its starting address is at 0x08000000 by default. Several SRAM modules are implemented on the device to support the functionality of the modules included.
The remaining memory space is reserved or occupied by the control or status registers as indicated in the memory-map table.

4.1.3 Peripheral overview

Hercules microcontrollers can implement several functions targeted towards varied applications. Several domains can be turned on or off by toggling their clocks in order to accomplish the application's requirements. Dynamic current inside the switched off module is virtually reduced to zero, but leakage will remain the same since in this device physical power switches are not implemented to isolate a domain from its core supply. Almost all I/O pins have either pullup or pulldown internal resistors, so the unused I/O pins can be configured as outputs and left unconnected or configured as input with enabled internal pull.
The microcontroller is particularly suitable for automotive tasks since it features a large number of peripherals for real-time control-based applications:

- two Next Generation High-End Timer **N2HET** module with up to 64 total I/O terminals, which provide sophisticated timing functions for real-time applications.
- the Real-Time Interrupt **RTI** which provides timer functionality for operating systems.
- the Enhanced Pulse Width Modulator **ePWM** module can generate complex PWM signals with minimal CPU overhead or intervention. It is ideal for digital motor control applications.
- the enhanced capture **eCAP** module, which accurately measures external signals.
- the enhanced Quadrature Encoder Pulse **eQEP** module directly interfaces with a linear or rotary incremental encoder to get the position, direction, and speed information from a rotating machine as used in high-performance motion and position-control systems.
- two 12-bit-resolution **MibADCs** analog to digital converters.

The device has multiple communication interfaces:

- five **MibSPIs**, which provides an efficient serial interaction for high-speed communications between similar shift-register type devices.
- four **UART (SCI)** interfaces, two with **LIN** support.
- four **CANs**, which support the CAN 2.0B protocol standard. They use a serial, multi-master communication protocol that efficiently supports distributed real-time control with robust communication rates of up to 1 Mbps. It is ideal for automotive applications since it can operate in noisy and harsh environments that require reliable serial communication or multiplexed wiring.

- two **I2C** modules provide an interface between the microcontroller and an I2C-compatible device through the I2C serial bus. The I2C module supports speeds of 100 and 400 kbps.

- one **Ethernet** controller.

- one **FlexRay** controller, which uses a dual-channel serial, fixed time base multimaster communication protocol with communication rates of 10 Mbps per channel. A FlexRay Transfer Unit (FTU) enables autonomous transfers of FlexRay data to and from main CPU memory.

4.2 Control and visualization interface

The crankshaft and camshaft signals simulator requires an interface to visualize the rotational speed set and change some settings. A simple solution to accomplish this feature is to connect a programmable touch display to the system. The device used is the Nextion®display NX4827T043_011, a Human Machine Interface (HMI) solution providing a 4.3" touch display combined with an on-board processor, 16 Mb of Flash memory, 3.5 KByte of RAM. Nextion display communicates via a TTL Serial port (5V, TX, RX, GND) with the microcontroller SCI peripheral module, and its connection is explained in Table 4.1 at the end of this chapter.

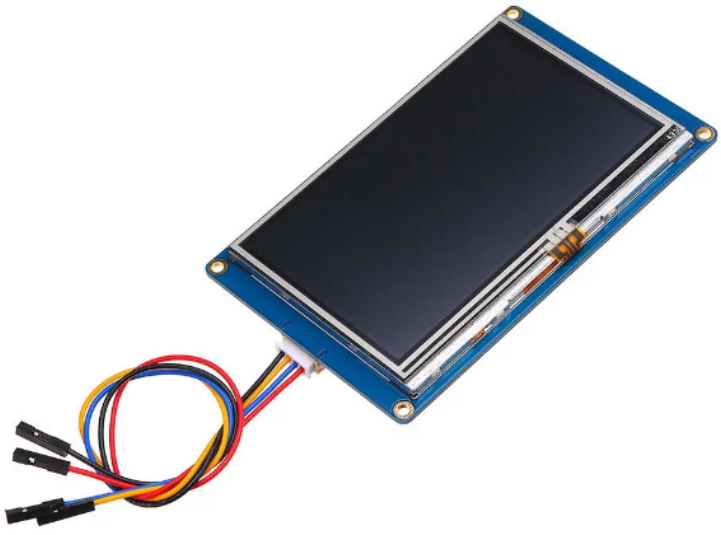

Figure 4.4: Nextion®touch display

It is possible to develop the Graphical User Interface with Nextion Editor software quickly: function components are added by drag-and-drop, and the interaction at the display side with the user is regulated by coding ASCII text-based instructions. A micro SD card is needed in order to upload the GUI developed via Nextion Editor. The output file must be copied to the previously FAT32 formatted SD card, and it must be the only file present on the card. Then, the micro SD card is inserted into the SD card slot on the rear side of the display: by connecting the display only with power, the file will be uploaded in about 5 seconds, and progress will be shown on display. At the end of the uploading process, the power can be disconnected, and the card is taken out. From this moment, the Nextion display can be connected to the microcontroller, and it will be able to communicate via SCI protocol communication.

4.3 Rotary Encoder

The simulator user can regulate a rotary encoder to increase or decrease the simulated engine's rotational speed.
Rotary encoders are divided into two main types: absolute encoders and incremental encoders. The first ones are devices that determine the absolute current shaft position without continually keeping track of its movement. The second ones provide real-time position changes, detecting also the direction of movement. For these essential features, incremental encoders are commonly used in applications that require precise measurement of position and velocity. Also, to regulate the value of the crankshaft rotational speed, an incremental encoder is chosen.
The **incremental encoder** used is a rotary input device composed of a knob that you can turn clockwise or counterclockwise without ever getting to a blocking limit. While the knob is turning, the user can feel that the angular position is changed thanks to a click per step. This rotary encoder also has a push button that can be toggled when pressing the knob in the rotation axis direction. The model of the used rotary encoder is shown in the following figure.

Figure 4.5: KY-040 Rotary Encoder

The incremental encoder produces two square-wave signals in quadrature (one 90° phase-shifted with respect to the other); for this reason, it is also called **quadrature encoder**. The phase difference between the two signals will be +90° for clockwise rotation and -90° for counter-clockwise rotation, or vice versa, depending on the device design. The signals are generated by two mechanical switches, either opened or closed, regulating the connection between the two pins and the ground. The available pins of the device are five:

- CLK: the pin that outputs the first square-wave;
- DT: the pin that outputs the second square-wave, shifted of 90° from CLK;
- SW: the pin that manages the push switch;
- +: the pin connected to the +5V voltage supply;
- GND: the pin connected to the ground reference.

When turning the knob clockwise, the signal on line CLK will first have a rising edge before line DT also goes up, followed by a falling edge on signal CLK and later by a falling edge on line DT. Considering 0 as the low level of the signal and 1 as the high level, a sequence from one-click position to the next can be determined for both CLK and DT signals. The sequence for a clockwise movement is then 00, 10, 11, 01, 00, while for a counter-clockwise movement the order is 00, 01, 11, 10, 00. Hence, the direction of the rotation can be determined by tracking these sequences of signals.

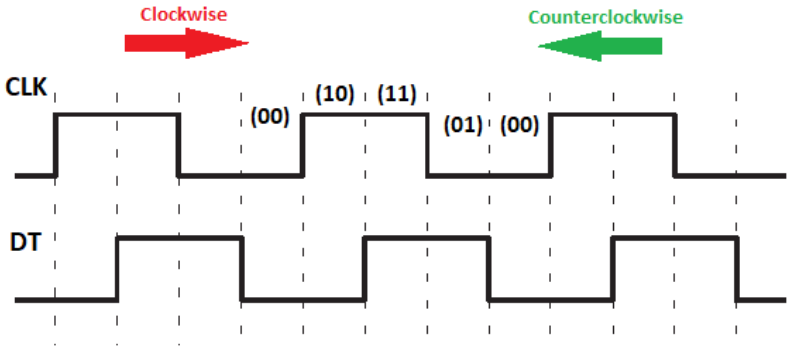

Figure 4.6: Quadrature encoder signals

4.4 Wiring Harness

The different components of the crankshaft and camshaft simulator must be appropriately connected in order to run the application governing the I/O peripherals. The simulator core is the evaluation board, and all the peripherals are connected to it, according to the pin configuration given by the manufacturer in the launchpad schematic.

CONNECTED PERIPHERAL		**TMS570LC43x MICROCONTROLLER**		
	PIN	***PIN***	***TYPE***	***FUNCTION***
KY-040 Quadrature Encoder	CLK	J1-7	EQEP1A	Encoder position
	DT	J5-8	EQEP1B	Encoder position
	SW	/	/	/
	+	J10-1	+5V	Power
	GND	J10-2	GND	Ground
NX4827T043_011 Nextion Touch Display	5V	J9-1	+5V	Power
	TX	J5-3	SCI1RX	Input data receiving from display
	RX	J5-4	SCI1TX	Output data transmission to display
	GND	J9-2	GND	Ground
Oscilloscope	CH1+	J10-36	GIOA-2	Digital Crank Signal
	CH1-	J10-32	GND	Ground
	CH2+	J10-7	GIOA-0	Digital Cam Signal
	CH2-	J10-4	GND	Ground

Table 4.1: Wiring Harness of the developed simulator

The table also shows the microcontroller pins that are chosen to generate the digital output signals of the crankshaft and camshaft sensors. These pins are part of the general input-output module GIO in port A. They can be connected to the channels of an oscilloscope in order to plot the two different signals. In the final configuration used for executing ECM tests, these pins generating the crankshaft and camshaft signals will be connected to the ECM employing a proper connection that will be explained in the last chapter of this thesis.

Figure 4.7 shows how the evaluation board is connected with touch display and quadrature encoder.

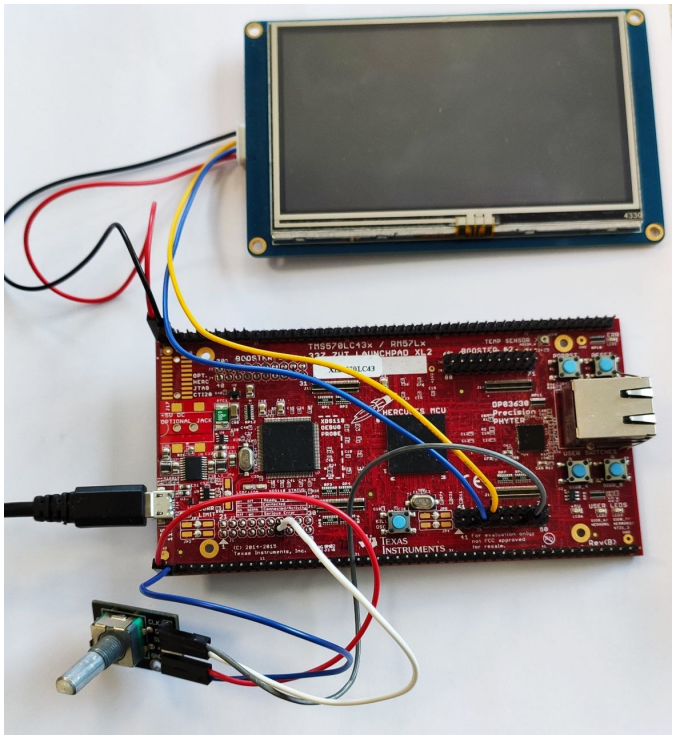

Figure 4.7: Crank/cam sensors simulator hardware components connections

Figure 4.8 shows the front view of the realized simulator. Nextion touch display is placed in the centre and quadrature encoder (controller) on the right side.

Figure 4.8: Crank/cam sensors simulator front view

Chapter 5

Crank/Cam Simulator Software Development

The implemented hardware of the open-loop simulator requires an application code in order to perform the task. The user application is developed on top of the basic software, which contains all the services useful to manage microcontroller memory, CPU, Files, and peripherals. Therefore, between the device and the application software, it is needed a class of software that provides a programming platform for low-level control of the hardware. Texas Instruments gives a tool for the software stack of its microcontrollers through HALCoGen.

5.1 Hardware Abstraction Layer

HALCoGen™ (Hardware Abstraction Layer Code Generator) is a software tool that provides a graphical user interface to easily generate the low-level drivers for component modules on Hercules microcontrollers. The peripheral configuration is simplified since the user needs to have only a high abstraction level of the system modules and peripheral modules: the tool provides drivers that are optimized for both performance and flash impact.
HALCoGen automatically generates error-free C source code of the basic software. This feature makes more efficient the configuration and initialization of the system, control and peripherals modules: software developers do not need to go into detail about the register levels and can mainly focus on application development. Furthermore, the generated source code is clear and structured, and can be customized by the user at a later time. Once the TMS570LC43x is configured using HALCoGen, the generated code can be imported into Code Composer Studio (CCS) to develop the application.
Therefore, a new project must be created through HALCoGen software tool and

it must be tailored with the used microcontroller evaluation platform: from the home page it is possible to create the new project, select the TMS570LC43x device among all the available Hercules devices in the list. Once chosen the project name and selected the Texas Instrument tool, the hardware block diagram of the device will be available.

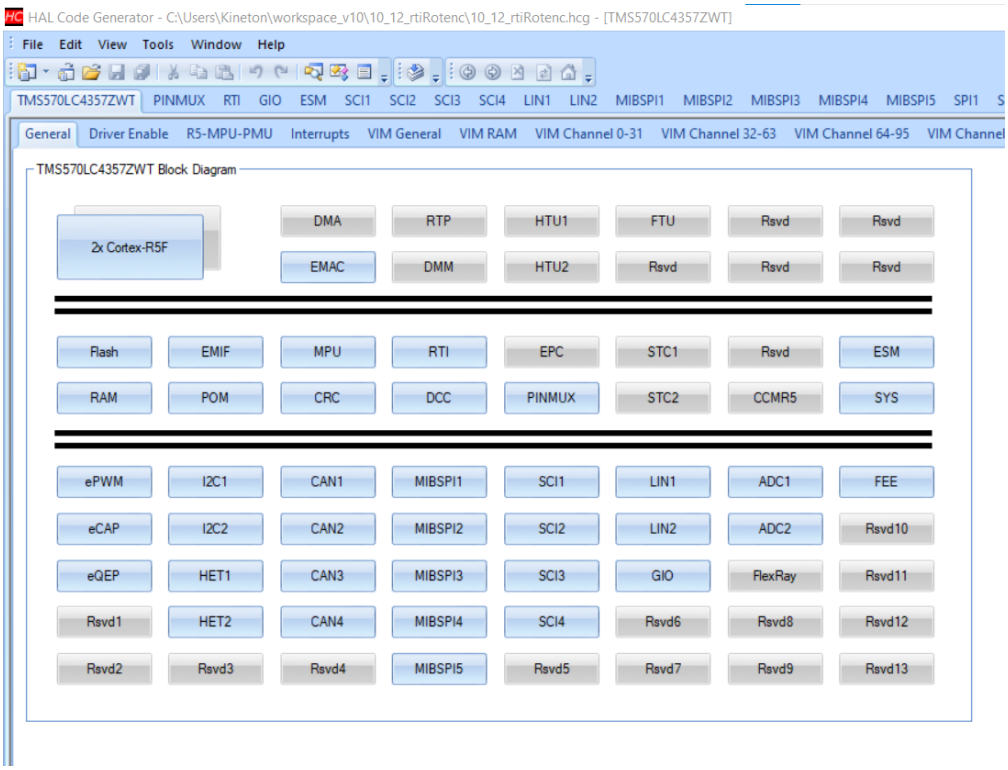

Figure 5.1: General block diagram from HALCoGen

Starting from this main page, the programmer can select the supported drivers to configure the device and check how the silicon vendor configured it by default. For this thesis's purposes, some drivers are used, and they are enabled in the section *Driver Enable*: RTI driver, GIO driver, SCI1 driver, EQEP1 driver. Some TMS570LC43x modules use the same communicating path; therefore, conflicts are possible in case of simultaneous usage: in order to avoid this problem, the HALCoGen *Pin Muxing* page allows to select all the modules and check if conflicts are listed. Whether a conflict is present, one of the chosen modules must be changed with another of the same type using another connection path.
The enabled drivers are explained in the following paragraphs of this thesis.

5.1.1 Clock configuration

The clock sources govern almost the totality of the drivers. TMS570LC43x devices support up to seven clock sources that can be independently enabled or disabled: oscillator, PLL1, External 1, LPO Low, LPO High, PLL2, External 2. The clocking is divided into multiple **clock domains** for flexibility in control and clock source selection. There are ten clock domains on this device: they are obtained by combining some dividers to different clock sources in *GCM* block. Details of the clock configuration and their frequency limits are shown in Figure 5.2.

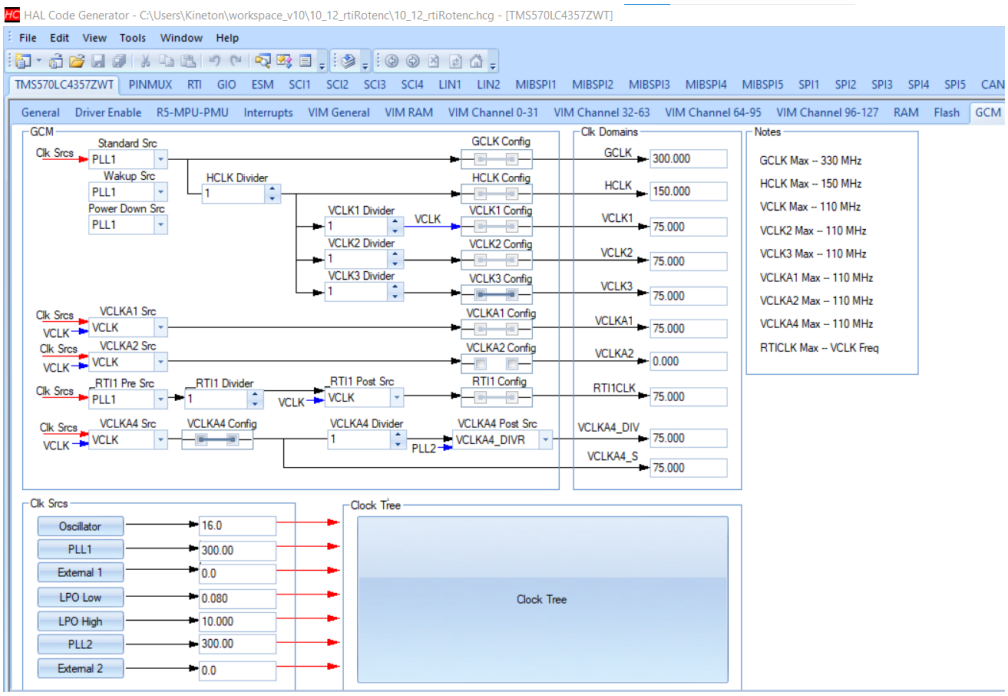

Figure 5.2: GCM clock driver

Particularly worthy of note domains are the CPU clock (GCLK), the system clock (HCLK) and the peripheral clocks (VCLKx).
In this project, all the clock configurations have the default value.

5.1.2 SCI peripheral module

One of the most important peripherals connected to the microcontroller is the **Nextion touch display**. It is a device communicating with a UART (Universal Asynchronous Receiver Transmitter) protocol, so one particular microcontroller type of module must be used: the **SCI module**. TMS570LC43x provides four SCI modules, each consisting of two external pins (SCIRX and SCITX), and the

module used for this project is SCI1.
Configuration of this module via HALCoGen can be done correctly only if its architecture and working principle are clear to the user. SCI module is composed of three main blocks:

- *Transmitter (TX)*, which performs double buffering with two main registers. CPU loads data to the transmitter data buffer register (SCITD), which transfers them to the transmitter shift register (SCITXSHF). Then data are shifted from SCITXSHF onto the SCITX pin, one bit at a time.

- *Baud Clock Generator*, which produces a baudrate scaled from VCLK peripheral clock through a 24-bit baud select register.

- *Receiver (RX)* performing double buffering with two main registers in the opposite direction of the transmission. Data entering one bit at a time from the SCIRX pin are shifted in the receiver shift register (SCIRXSHF), which then transfers completed data into the receiver data buffer register (SCIRD).

The SCI receiver and transmitter can be enabled from the user as well as their interrupts, and they can operate independently or simultaneously in full-duplex mode.
The SCI uses a **frame format** that can be programmable through the bits in the SCIGCR1 register. Both receive and transmit data lines are at logic high since they are in nonreturn to zero (NRZ) format. All frames begin with a start bit at a logic low; then, data are sent and received from the least significant bit till the most significant bit, with one to eight data bits length. An address bit and/or a parity bit can be present or not, depending on the address-bit mode and PARITY ENA bit configured by the user. In order to ensure synchronization between communicating devices, the end of the frame is indicated by one or two stop bits, which are always high level.
The TIMING MODE bit in the SCIGCR1 register can set the SCI module either in asynchronous or isosynchronous timing mode. Since the Nextion display uses the standard UART protocol, **asynchronous timing mode** is set: each bit in a frame has a duration of 16 SCI baud clocks periods. The SCI receiver detects a valid start bit only if the four samples after a falling edge on the SCIRX pin are of logic level 0 in order to prevent interpreting noise as a start bit. When a valid start bit is detected, the SCI determines the value of each bit by sampling in the middle of the bit and taking the majority vote of the seventh, eighth, and ninth samples. Thanks to this strategy, the determined value stored in the SCI receiver shift register is far from errors caused by possible propagation delays and data line noises.

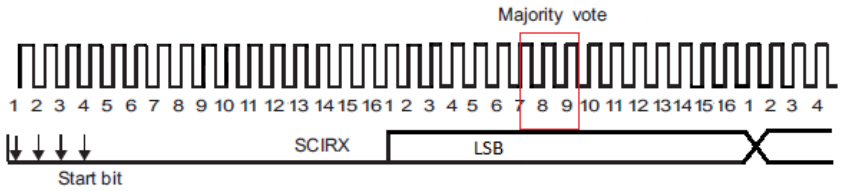

Figure 5.3: Asynchronous timing while receiving a data frame

Also in the transmission of data frames, each bit has a duration of 16 SCI baud clocks periods: the value in the first clock period is shifted by the transmitter on the SCITX pin and then hold for 16 SCI baud clock periods.
The baudrate of the SCI module in asynchronous timing mode is generated internally and determined by the peripheral VCLK and the prescaler BAUD value in the BRS register:

$$baudrate = \frac{VCLK}{16 * (BAUD + 1)}$$

The specifications given by the datasheet of the touch display used in this project indicate that the device interfaces correctly with a baudrate range of 9600 bps [8]. Therefore, via Halcogen, the SCI1 module is set in asynchronous mode, with an internal clock (VCLK), and the baudrate value is 9606 thanks to a prescaler value of 487 applied to the clock. The bit frame length is 8, stop bits are two, and no parity is needed. The direction of the SCITX transmission pin must be enabled as output by checking the box TX direction.

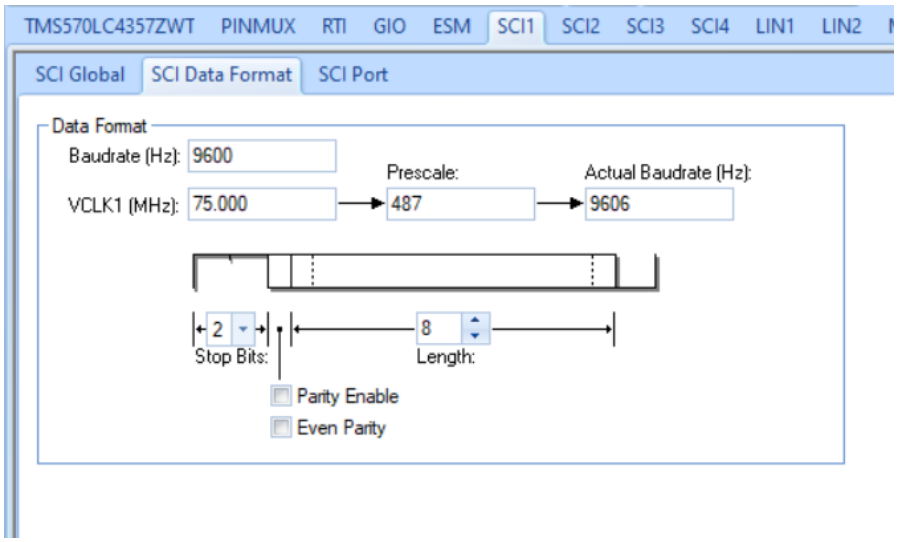

Figure 5.4: SCI Data format for Nextion display

With these register configurations set via HALCoGen, the microcontroller and the touch display are ready to interface, but the data to transmit and receive are strictly related to the display user's commands. Indeed, the Nextion display must be programmed so that the user can easily interface with the microcontroller and control the system.

Touch Display Programming

The Nextion touch display is the HMI used in this project to allow the user to interact with the simulator. The main objective for this project is to have a menu where the user can choose some options by touching the display:

- a dashboard page, where a digital signal can be controlled via two dual-state buttons used to either turn on or off the battery supply and key signal of the simulated vehicle. Furthermore, it is possible to visualize the engine speed value displayed in a display box and a tachometer.
- a temperature page, where can be checked the temperature sensors;
- a pressure page, where can be checked the pressure sensors;
- a switches page, where can be set the switches controlling the connection between ECU and sensors;
- an actuators page, where can be set the actuators connected to the ECU;
- a settings page, where the user can choose the type of crankshaft and camshaft sensors used and the signals' pattern depending on the engine selected.

Since this project deals with the generation of active crankshaft and camshaft sensors signals, the user needs only the dashboard and settings pages; the other pages are implemented for future works.
The realization of the display interface needs a programming part realized via **Nextion Editor** software tool: it must be installed on a laptop, and a new project can be created in a .hmi file according to the display model and orientation selected. Therefore, it is chosen NX4827T043_011 basic device with a display direction of 0° horizontal. Nextion Editor main page is composed of several panes located around the display of the page to be modified:

- *toolbox* pane on the left of the window has a list of functional components to add to the page by drag-and-drop;
- *page* pane on the right side is the place where all the needed pages are listed;

- *attribute* pane contains the component drop-down, where all the components included within the current design page are listed. Selecting a component will display the component's available attributes (component name, identifying value, value) that can be very useful to control its function;

- *event* pane, which can contain instructions related to the page components. Simple ASCII text-based instructions can be written in this pane and can control a possible event of the component or page, i.e., when it is pressed or released.

In Figure 5.5 is shown the Nextion Editor main page with the pages and components of this project, and the red boxes highlight the most used panes. The visualized page in the center of the page is the menu page, which will be uploaded into the touch display and then will be available for the user after the loading of the splash page.

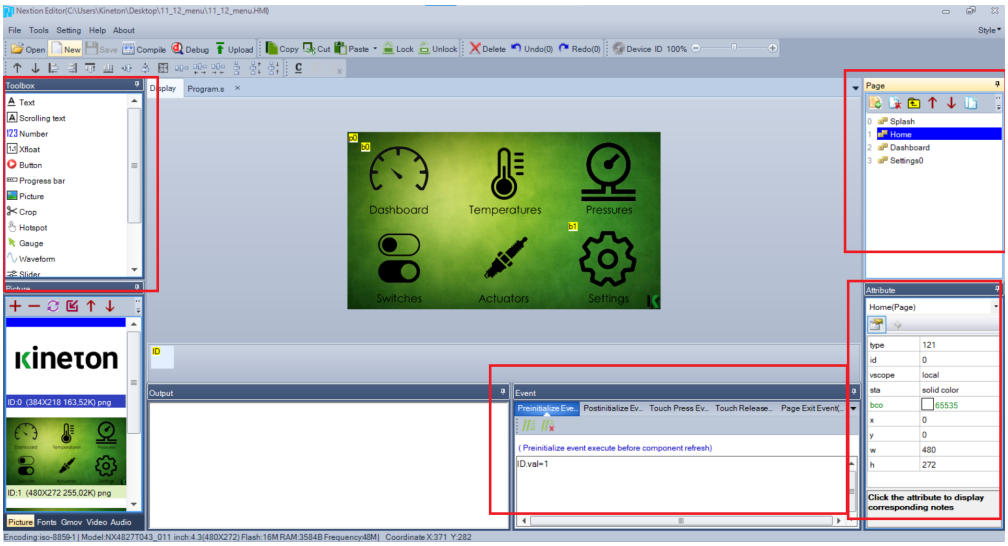

Figure 5.5: Nextion Editor main page

The pictures used in this project are realized previously through a raster graphics editor, and their resolution is equal to the display resolution (480x272 pixels) if they are full-screen. Then, they are uploaded in the *picture* pan and added to the corresponding page through the picture tool.

ASCII text-based instructions for coding how components interact on the display side are listed in the Nextion instruction set web page [9]. From the toolbox pane, two button components are added to the menu page: b0 covering the dashboard symbol, which will send the user to the dashboard page if pressed, thanks to the command *page 2* written in the touch press event pane; b1 covering the settings

symbol, which will send the user to the settings page if pressed, thanks to the command *page 3* written in touch press event pane. The changing page will be performed since an ID value is added to all the pages (2 is dashboard ID, 3 is settings ID).
Dashboard page is full of components:

- one dual-state button bt0 over battery switch image, which will turn on or off a digital signal. Every time a release event happens, it will send a particular raw formatted data over serial to microcontroller thanks to *print* and *prints* commands set in the event pane;
- one dual-state button bt1 over key switch image, which will turn on or off another digital signal. As bt0, *print* and *prints* commands will generate a particular raw formatted data that will be sent over serial to microcontroller at each release event;
- one text t0, which will show the crankshaft rotational speed value in rpm, by changing its text according to the value set by the user via rotary encoder;
- one gauge z0, which will move a needle according to the crankshaft rpm value;
- one button b0, which will allow the user to come back to the menu page. A page command is used in the touch press event.

Figure 5.6 shows dashboard components configuration and details of print commands used for bt0 in the event pane.

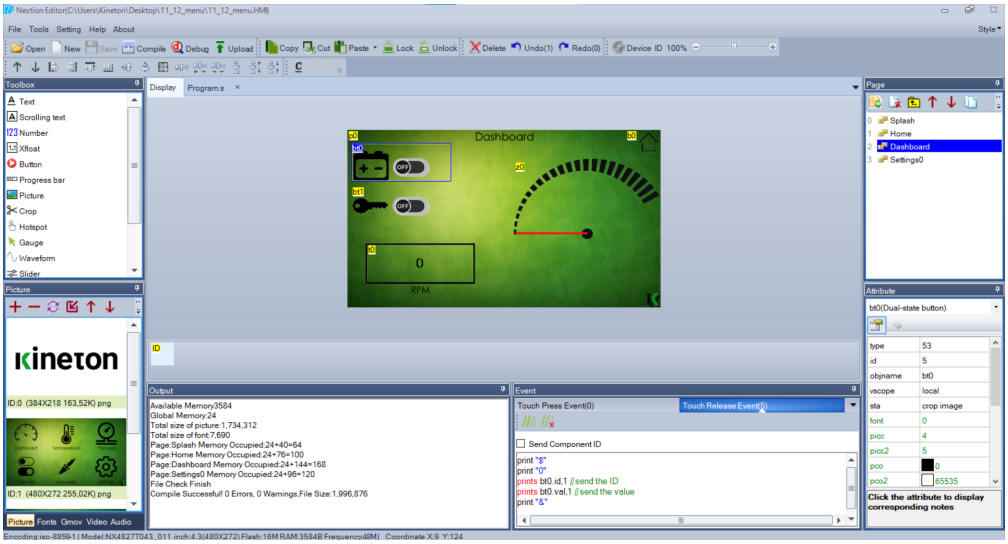

Figure 5.6: Dual-state button configuration in dashboard page

Nextion display also provides a *compile* tool, which checks for errors in the currently loaded project and shows it in the output pane. After verified that compiling is correct, it is possible to simulate if the project works as expected with the tool *debug*. An essential feature of debugging is that it is possible to see how a component either sends or receives ASCII code when used: in this way, it is possible to control its behavior in the final application code. Indeed, debug is used in this project in order to see how the button and dual state buttons react when pressed and what serial code they send every time they are pressed, as shown in the red box of Figure 5.7.

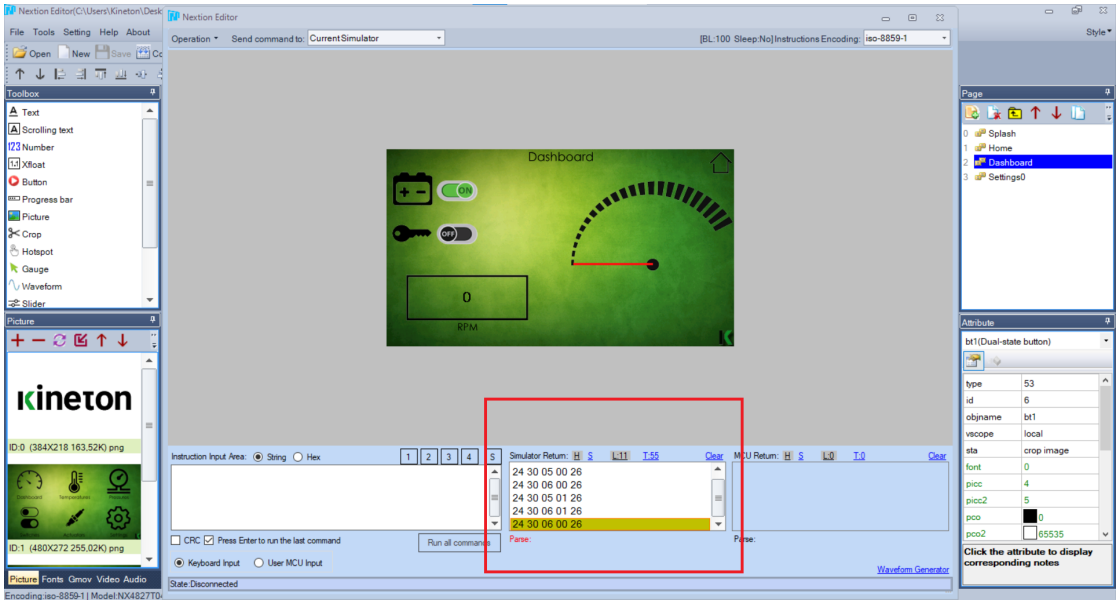

Figure 5.7: Nextion Editor debug

Once finished to program the Nextion project, it must be uploaded via microSD card into the display. By using the function *TFT file output*, a .tft file is generated and saved in a laptop folder, and then it must be copied into a microSD card. It is needed that the microSD card is less than 32GB in size, it must be Windows formatted as FAT32, and the .tft file must be the only file in the root folder. The power to Nextion device must be off when inserting and removing the microSD card but must be on when firmware updates are required.
Now, the Nextion project is saved into the touch display device: it runs when the power is on, and it is ready to interface with the user when it is connected to the SCI module of the microcontroller and the application code is created entirely.

5.1.3 Digital Signal I/O

TMS570LC43x evaluation platform provides several I/O bidirectional and bit-programmable pins for digital signals: they are grouped in *GIO* general-purpose input/output module. This module can be used as a digital output channel to generate the crankshaft sensor signal and camshaft sensor signal, as well as digital output to control the battery voltage supply switch and key-on signal switch.
The GIO module in this microcontroller consists of 64 terminals that can be independently configured as input or output and configured as required by the application. This module can generate interrupts whenever a rising or falling edge or any toggle is detected on up to 32 GIO pins in input mode. Multiple registers control the various aspects of the input and output functions, and they can be easily configured via HALCoGen software tool. In the thesis' project, the GIO port A is used:

- *Bit 0* is chosen as generator of camshaft signal. Therefore, the data direction must be set as an output by checking the bit 0 DIR box in HALCoGen (this selection will change the GIODIRA register bit value accordingly). Also PSL box must be checked (GIOPSLA register bit 0 will be 1) since a pull-up resistor inside the microcontroller must be enabled: pull-up resistor guarantees that the pin is driven to a logic high when enabled, and the logic is low when the pin is not enabled. PDR box is not checked since open drain functionality is not required (the pin is not needed at a high impedance state). DOUT value is 0 since the initial output value of port A bit 0 must be null at the start condition.

- *Bit 2* is chosen as the generator of crankshaft signal. The boxes checked via HALCoGen are the same as those in bit 0 since the functionality is like the camshaft. The registers will be configured considering bit 2 at value 1 in case of a checked box.

- *Bit 6* is chosen as a digital channel to drive the battery voltage supply switch: when the logic is high, the battery +12V will be available to supply the ECU, otherwise it will be disconnected. Also in this case, since it is a digital output channel, the checked boxes are DIR and PSL.

- *Bit 7* is used to simulate the digital key signal useful for the ECU. Bit 7 in GIODIRA and GIOPSLA are set to 1.

Figure 5.8 shows HALCoGen configuration for GIO port A bit 0.

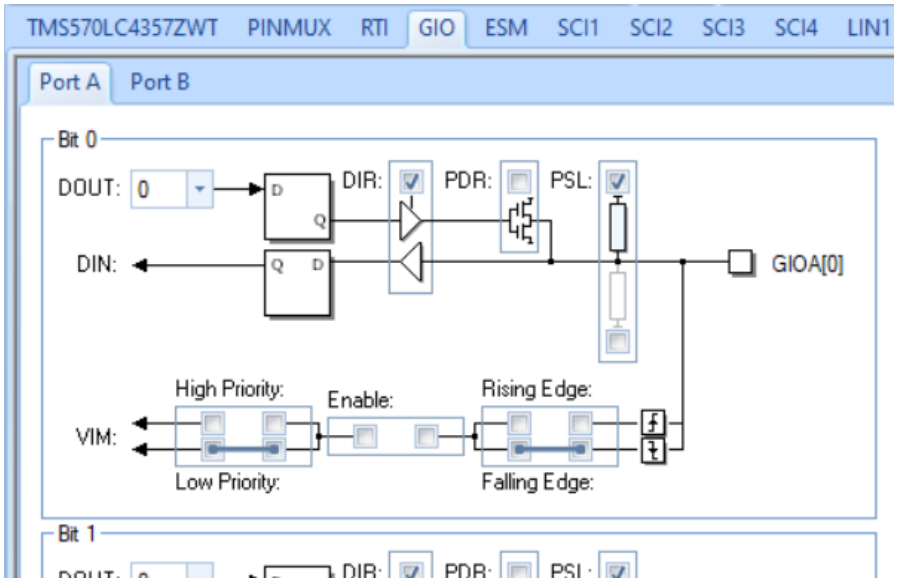

Figure 5.8: GIO HALCoGen configuration

It is also important to check if there are conflicts between GIOA and other microcontroller modules, so in the HALCoGen PINMUX section, the GIOA box is checked.

5.1.4 eQEP module

The rotary encoder is connected to the EQEP1 module of the TMS570LCx microcontroller. This is one of the two available EQEP (enhanced quadrature encoder pulse) modules used to get the position, direction, and speed information of a linear or rotary incremental encoder.
In Table 4.1 all the connections between rotary encoder and microcontroller are listed, and it is possible to notice that the EQEP input pins used are only two: QEPA and QEPB. These are the pins providing two square wave signals 90 electrical degrees out of phase used to determine the direction of rotation and the position information. EQEP module also provides other two pins (Index and Strobe), but they are not used since the model of the rotary encoder is very simple, and it needs only QEPA and QEPB.
The general configuration of the module is based on the position counter mode, which is the **quadrature-count mode** since it is used as a quadrature encoder. Therefore, in the HALCoGen project must be configured the EQEP1 section by selecting QUADRATURE_COUNT in position counter mode drop-down list: QDECCTL register will be configured by setting [QSRC] bits to 0. The most important step for controlling the rotation of the quadrature encoder is the choice

of *direction decoding logic*. The binary sequences generated by QEPA and QEPB determine if an increment or a decrement is counted by QPOSCNT register value due to a clockwise or counter-clockwise rotation of the encoder knob. The direction decoding logic updates the direction information in QEPSTS[QDF] bit (0 if counter-clockwise rotation, 1 if clockwise rotation) according to the detected leading sequence.
EQEP external clock rate is imposed with RESOLUTION_2X in HALCoGen configuration (QDECCTL[XCR] bit is set to 0) since it must be a quadrature clock (QCLK) counting both the rising and falling edge. Indeed, the frequency of the generated clock must be four times that of each complete input sequence in order to detect every binary change. Quadrature direction is imposed at a high level when the direction is clockwise, so the QDIR parameter in HALCoGen is CLOCKWISE.

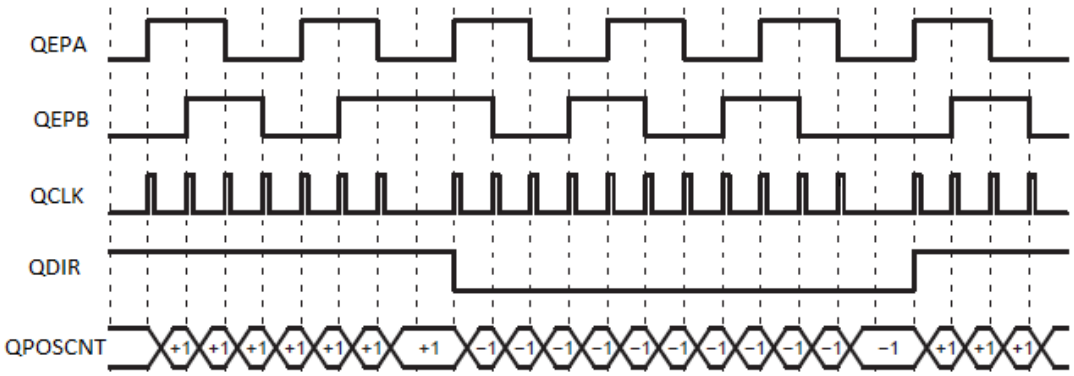

Figure 5.9: Direction decoding logic in quadrature counting mode

The other check-boxes in EQEP1 general configuration are not used, as well as all the compare output configurations since the project does not need those functionalities.
The EQEP module includes a **position counter**, which provides position information of the quadrature encoder with respect to an imposed reference. It can be configured to be reset in four modes: reset on Index event; reset on maximum position; reset on the first Index event; reset on Unit Time Out Event. Since the Index pin is not present in the used quadrature encoder and reset on unit time out event is not suitable to keep the value for a long time, *reset on maximum position* is chosen. The position counter value is intended to be the rpm value to control via quadrature encoder, therefore 0 is selected as initialize position count value. Instead, the hexadecimal value of 0x1F40 (which is equivalent to 8000 in integer) is set as the maximum position count (QPOSMAX register) since it can be the maximum rotational speed of a supercar crankshaft. Position counter is reset to QPOSMAX register value when position counter counts down after 0 (underflow),

and it is reset to 0 when the position counter counts up after QPOSMAX (overflow). Each underflow or overflow event is indicated in the interrupt flag register (QFLG) bit 5 and 6, respectively.
In the EQEP module, an interrupt mechanism can generate up to eleven interrupt events. Among the several interrupts, in the project of this thesis is used only the **unit time out interrupt**: it allows to call periodically an interrupt service routine, and this characteristic is used to check the position counter value and to detect a new value eventually. Unit period register (QUPRD) value selected via HALCoGen is 0xFFF: the interrupt will be set in flag bit QFLG[UTO] when the 32-bit unit timer QUTMR (which is clocked by VCLK3) matches this QUPRD value. Hence, since VCLK3 is 75 MHz, the interrupt period will be 54μs, a shorter period than that of the RTI at maximum frequency: this will ensure a fast update of crankshaft and camshaft frequency when the user will change the position of the rotary encoder.

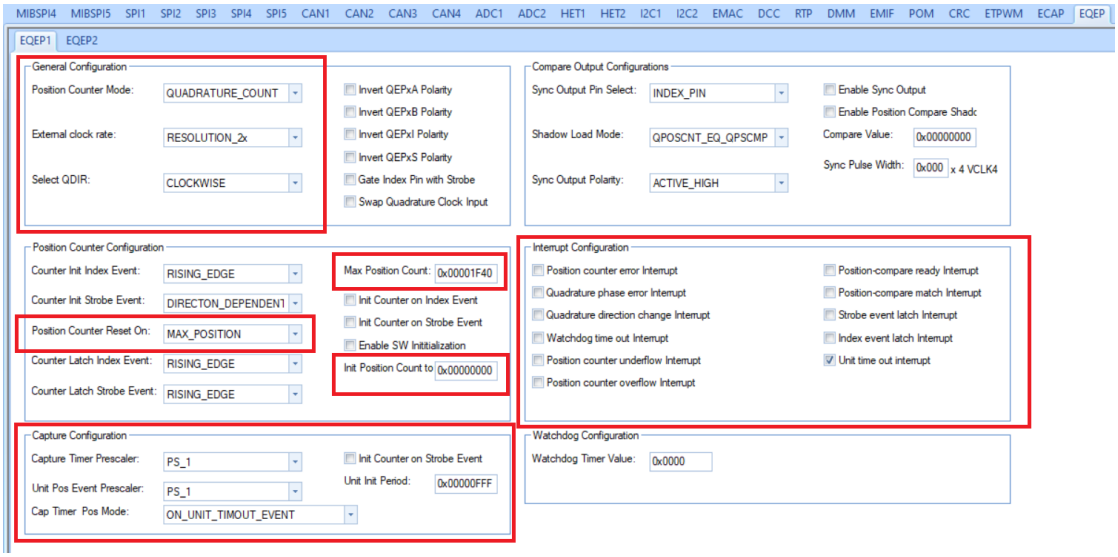

Figure 5.10: EQEP module configuration

5.1.5 Real-Time Interrupt Module

In this thesis, the crankshaft sensor model considered is a Hall-effect sensor scanning the radial surface of a "60 - 2" even-toothed wheel. A **Periodic Interrupt Timer (PIT)** is used to simulate a call function every time the crankshaft sensor detects the rising or falling edge of a tooth. PIT occurs with a period inversely proportional to engine rpm:

$$P = \frac{60}{2 * n * rpm} \tag{5.1}$$

Where n indicates the number of teeth (by taking into account also the missing teeth) and rpm is engine speed. Multiplier 2 represents the ratio between PIT period and tooth period: indeed, tooth angular displacement is 3°, and the angular displacement between two teeth is 6°. Therefore, in the particular case of 60 - 2 wheel:

$$P = \frac{1}{2 * rpm} \tag{5.2}$$

This relationship shows how important it is to generate a periodic interrupt that can change its period value correctly when the crankshaft's rotational speed increases or decreases. In order to accomplish this critical feature, a microcontroller module able to generate periodic interrupts must be used: RTI module.
The **real-time interrupt (RTI) module** is a TMS570LCx module incorporating two independent 64-bit counters that define timebases needed for scheduling in the operating system and for performing periodic tasks. A compare unit compares either counter block 0 or counter block 1 with programmable values and can generate up to four independent interrupt requests on compare matches. Figure 5.11 shows the RTI high-level block diagram configured in this project through HALCoGen.

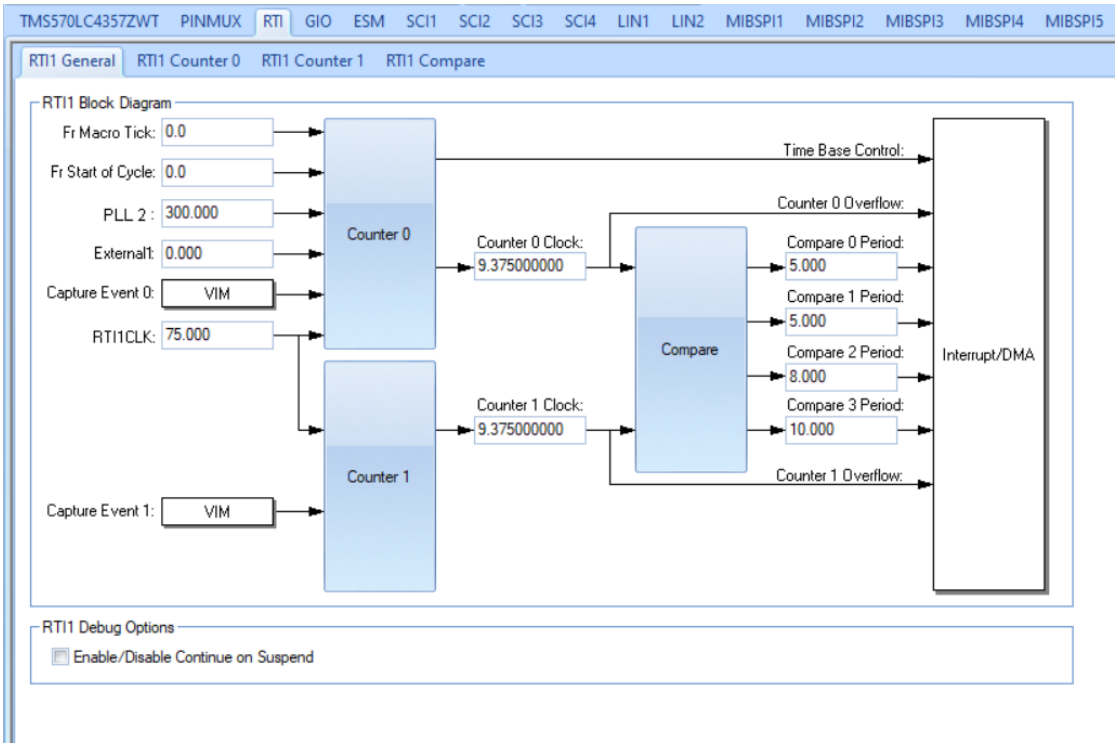

Figure 5.11: RTI general block diagram

Since the generation of one periodic interrupt is required, only RTI Counter 0 block is used. It consists of one prescale counter (RTIUC0) and one free-running counter (RTIFRC0). A clock of 75 MHz given by the RTICLK drives the RTIUC0, which counts up until the compare up value in the compare up counter register (RTICPUC0) is reached. This counting operation defines the frequency of the free-running counter RTIFRC0 since it is incremented every time the compare matches and then RTIUC0 is reset to 0:

$$f_{RTIFRC0} = \frac{f_{RTICLK}}{RTICPUC0 + 1} \tag{5.3}$$

In this project, the free-running counter frequency is equal to 9.375 MHz since the RTICPUC0 value is 7. If RTIFRC0 overflows, an interrupt is generated to the vectored interrupt manager (VIM), which is the microcontroller module controlling all the interrupt resources present. The overflow interrupt is intended only to reset the free-running counter and not to generate the time base for the crankshaft signal.

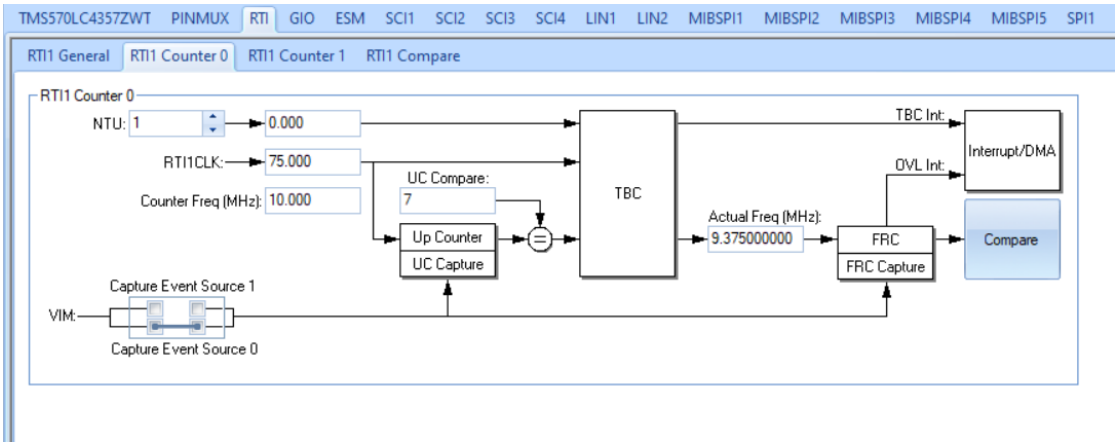

Figure 5.12: RTI counter 0 block

The interrupt request to the VIM needed for periodic interrupt function is generated by one of the four RTI compare registers: RTICOMP0. It is configured to be compared to RTIFRC0, and when the counter value matches the compare value, an interrupt is generated. Via HALCoGen must be enabled VIM channel 2 of the VIM module to IRQ (interrupt request), otherwise RTI compare 0 will not generate interrupts.
Furthermore, in the update compare register (RTIUDCP0) is stored a value that can be added to the compare value in RTICOMP0 after a compare is matched in order to allow periodic interrupts. The period of the generated interrupt request

can be calculated as follows:

$$t_{COMP0} = t_{RTICLK} * (RTICPUC0 + 1) * RTIUDCP0 \tag{5.4}$$

Considering an RTIUDCP0 value of 46875, the RTI compare 0 period will be 5ms, which is useful to generate a periodic interrupt for a crankshaft rotating at 100 rpm. In the application code, RTIUDCP0 and RTICOMP0 values will be changed according to the engine speed set by the user.

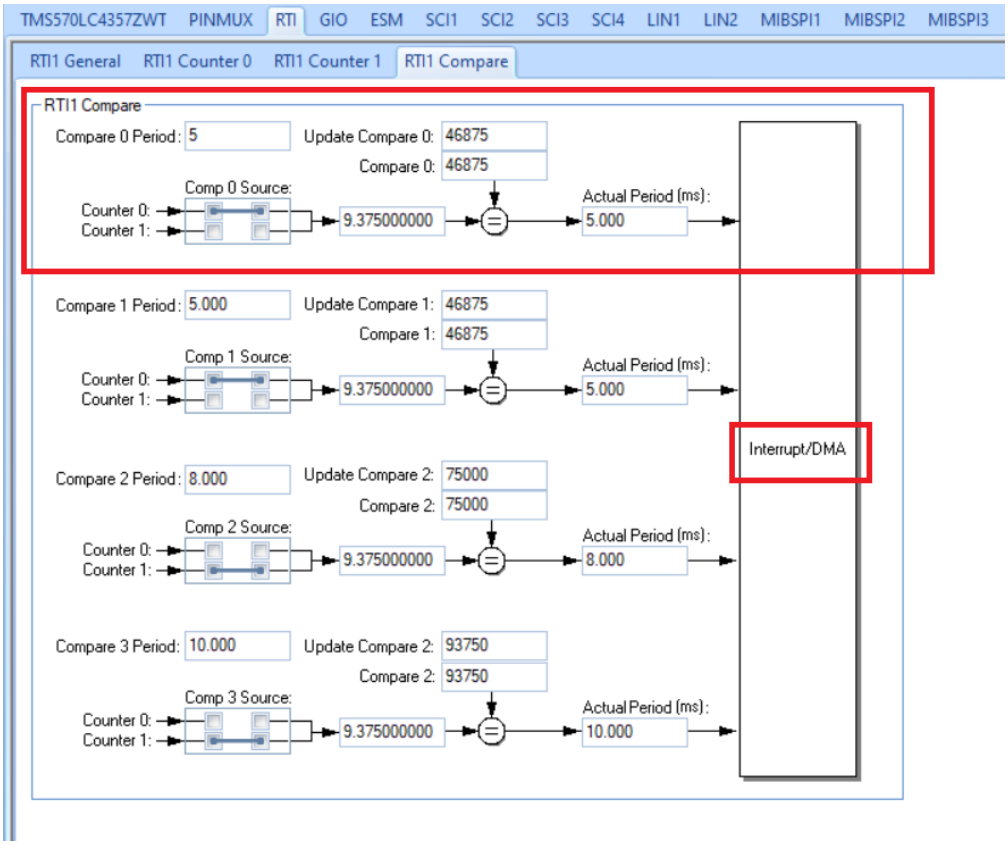

Figure 5.13: RTI compare block

At this point, the basic software configuration, considering peripheral and system modules, has been completed. Source files derived from the configuration of the module are automatically generated thanks to the HALCoGen option *Generate Code*. This function will create all the header .h files and source .c files needed for the programmer to create the application code and also an empty main file called *'sys_main.c'*. Possible configuration changes can also be performed later, when programming the application may lead to settings edits in order to improve the functionality of the system: source code can be generated again, and application code will remain unchanged.

5.2 Details of the Application Code

The application code, describing the task that must be performed for the generation of crankshaft and camshaft sensors signals, is created by editing the automatically generated code given by the HALCoGen software tool. Indeed, the project folders containing the generated source files can be shared with **Code Composer Studio™** integrated development environment (IDE) for Texas Instruments' microcontrollers. Thanks to this software, it is possible to edit, build and debug the application and then run it into the TMS570LCx development board connected via a USB debug probe.
A new CCS project has been created with the same name of the HALCoGen project previously described, and the TMS570LCx device is selected with the connection XDS110 USB debug probe: the empty project is chosen since all the source files will be imported later. The project is stored in a workspace, which is the main working folder for Code Composer Studio and HALCoGen. Once the project is created, CCS edit perspective will be available: it includes an optimizing C/C++ compiler, Project Explorer, Editor, and Problems view and many other features.
In Project Explorer view, it is possible to see all the projects into the workspace, therefore also the project created for this thesis. The project contains the source codes build via HALCoGen since they are automatically added to it, but the generated Include folder is not present, so it must be added manually: by right-clicking the project folder in Project Explorer and selecting properties, the HALCoGen generated Include folder can be added to the #include search path under the compiler Include Options. Now the HALCoGen project is completely coupled with the CCS project.
The main file where the application is developed is called *HL_sys_main.c*: at the beginning, it was empty, and it has been coded with the application code. Since many functions from different source codes are used, all the declarations are added in the main file thanks to the including of all the header files needed. Figure 5.14 shows the part of HL_sys_main.c in which header files are included in the Code Composer Studio compiler.

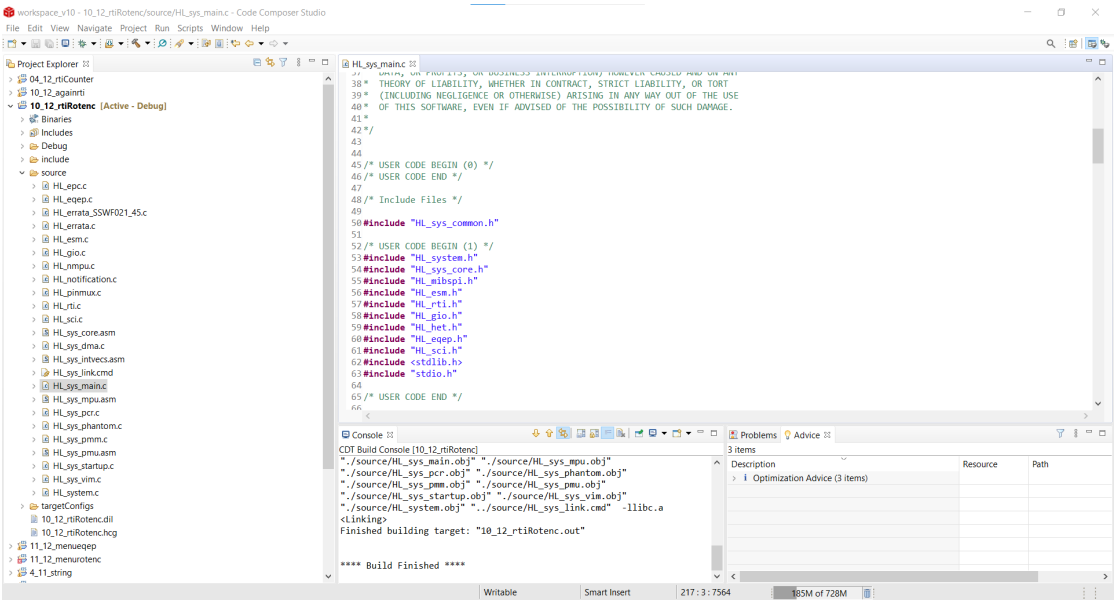

Figure 5.14: CCS edit perspective showing the header files included in the main file

It is possible to notice both system and peripheral microcontroller modules header files.
Then, all the variables are declared: *command* and *gauge* are two strings that will be used to store respectively the value in ASCII of the engine rpm set through the rotary encoder and the same value but reduced for the used gauge function in Nextion touch display. Indeed, as previously mentioned, exchanging serial data between microcontroller and display must be in lowercase ASCII code, following the Instruction Set rules given by the display vendor. Furthermore, every instruction over serial must be terminated with three bytes of 0xFF 0xFF 0xFF hexadecimal variable in order to establish the communication end. Therefore, an 8-bit unsigned hexadecimal array called *Cmd_End* is added.
More variables are declared and also initialized to 0: *speed* is an integer variable indicating the crankshaft rotational speed coming from rotary encoder regulation; *gauge_val* is the integer variable that stores a reduced speed value useful for the regulation of the Nextion display's tachometer needle; *comp* is the integer value to store in RTIUDCP0 and RTICOMP0 registers in order to set the compare value needed for frequency regulation of the generated crank and cam signals; i is an integer value used as an index in order to read out the look-up table describing the crankshaft and camshaft wheel pattern.

```
/* USER CODE BEGIN (2) */

/* Strings for display communication */
unsigned char command[8];
uint8_t Cmd_End[3] = {0xFF, 0xFF, 0xFF};
unsigned char gauge[8];

/* Variables for display Output */
uint32 speed = 0;
uint32 gauge_val = 0;
//uint32 tmp_rpm = 0;

/* Variable used for setting correctly the compare value of real-time interrupt in order to obtain the period for signal generation */
uint32 comp = 0;

/*Index for reading out the wavetable */
int i = 0;
```

Figure 5.15: Main file variables declaration

The next variable is a string made up of 240 characters which describes a "60-2" rotating crankshaft wheel pattern and a "6+1" camshaft wheel pattern used in a Cursor 9 engine. This string defines the sequence of teeth and gaps that a crankshaft and a camshaft sensor can detect during four strokes of a diesel engine, so over a crankshaft rotation of 720° and a camshaft rotation of 360°.
Sixty teeth divide camshaft wheel into 6° pitches. Therefore 60 teeth and 60 gaps with an angular displacement of 3° are present, but the last two teeth are missing, so actually there are 58 teeth and 62 gaps (the last four are connected by making up one single big gap).
Six teeth equally divide 360º into 60º pitches. The first tooth after the additional +1 tooth in the camshaft wheel defines where zero cam angle starts and where the engine intake stroke begins.
The possible ASCII characters composing the string are the following:

- 0, representing the presence of a gap in both crankshaft and camshaft wheels;
- 1, which indicates the presence of a tooth in the crankshaft wheel and a gap in the camshaft wheel;
- 2, which indicates the presence of a gap in the crankshaft wheel and a tooth in the camshaft wheel;
- 3, representing the presence of a tooth in both crankshaft and camshaft wheel.

```
const unsigned char sixty_minus_two_with_cam_Cursor9[240]=
  { /* 60-2 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 1-5 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 6-10 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 11-15, Cam trigger on first half of 12th */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 16-20 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 21-25 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 26-30 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 31-35, Cam trigger on first half of 32th */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 36-40, Cam trigger on first half of 37th */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 41-45 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 46-50 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 51-55, Cam trigger on first half of 52th */
    1,0,1,0,1,0,0,0,0,0,  /* teeth 56-58 and 59-60 MISSING */
    1,0,1,0,1,0,1,0,1,0,  /* Second revolution teeth 1-5 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 6-10 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 11-15, Cam trigger on first half of 12th */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 16-20 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 21-25 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 26-30 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 31-35, Cam trigger on first half of 32th */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 36-40 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 41-45 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 46-50 */
    1,0,3,0,1,0,1,0,1,0,  /* teeth 51-55, Cam trigger on first half of 52th */
    1,0,1,0,1,0,0,0,0,0   /* teeth 56-58 and 59-60 MISSING */
  };
```

Figure 5.16: String variable describing crankshaft and camshaft wheel pattern

ASCII characters are used rather than integer value since each of them occupy only one byte of memory and also in order to manage a look-up table similar to the wavetable stored in CSV files accessed by APU in dSpace SCALEXIO DS2680 for crank and cam simulation.

Then, the **main function** is coded. Firstly some local variables are added:

- *buf* and *buf_g* are two string variables used as a buffer to store the ASCII commands for serial communication with the touch display. They will be sent in order to show the engine rpm value via the display dashboard respectively as an integer value and as tachometer needle position.

- *len* and *len_g* are two integer variables storing the number of characters stored in the buffers previously described.

- *Rx_Data* is an array of unsigned integer values which contain five hexadecimal values describing the condition of the battery and key signal dual-state buttons in Nextion display each time they are pressed.

```
int main(void)
{
/* USER CODE BEGIN (3) */

    /* buffer containing the display command for visualizing the rpm value in the text box */
     char buf[50];
     /* value of the number of character of the buffer above*/
     int len = 0;
     /* buffer containing the display command for visualizing the rpm value in the speedometer */
     char buf_g[50];
     /* value of the number of character of the buffer above */
     int len_g = 0;
     uint8_t Rx_Data[5];
```

Figure 5.17: Variables declared into the main function

Next step is represented by **peripheral initialization**: *gioInit()*, *rtiInit()*, *sciInit()*, *QEPInit()* are functions defined in the respective source .c file that now are called into the main in order to configure all the microcontroller settings made via HALCoGen. As an example, Figure 5.18 shows a part of gioInit function defined in *HL_gio.c*, in which peripheral registers are initialized according to the GIO module configuration decided via HALCoGen tool.

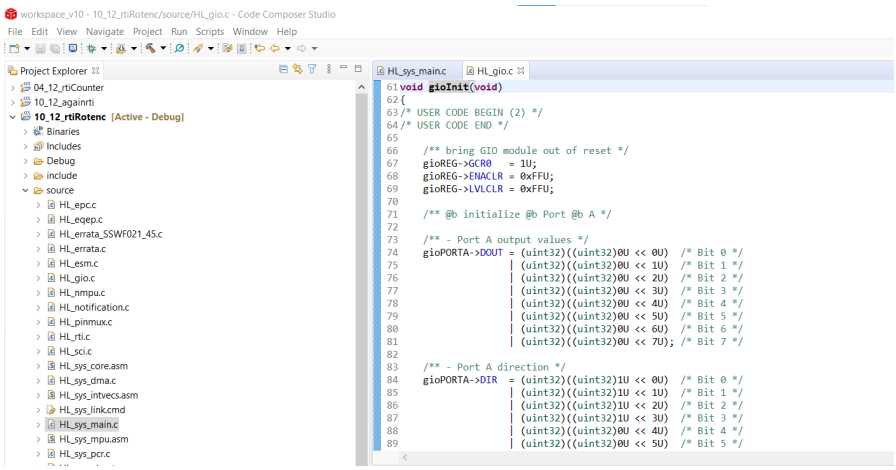

Figure 5.18: Part of GIO initialization function defined in HL_gio.c file

Other functions are called into main function:

- *eqepEnableCounter(eqepREG1)*, which enables EQEP1 position counter by setting QEPCTL register bit 3 to 1. It is declared in HL_eqep.c file;

- *eqepEnableUnitTimer(eqepREG1)*, which enables EQEP1 unit timer by writing 1 to QEPCTL register bit 1. It is declared in HL_eqep.c file;

- *eqepEnableCapture(eqepREG1)*, enabling EQEP1 capture unit by setting QCAPCTL register bit 15 to 1. It is declared in HL_eqep.c file;
- *_enable_IRQ_interrupt_()*, which enables IRQ interrupt mode in CPSR register. It is declared in HL_sys_core.h file;
- *rtiEnableNotification(rtiREG1,rtiNOTIFICATION_COMPARE0)*, which enables interrupts in RTI module and activates a notification when an interrupt due to a Compare 0 match is pending. This function calls another function (*rtiNotification*) at every compare 0 match.
- *rtiStartCounter(rtiREG1,rtiCOUNTER_BLOCK0)*, which starts RTI counter block 0 (RTIUC0 and RTIFRC0), by writing 1 to RTIGCTRL register bit 0.

Before enabling IRQ interrupt mode, RTICOMP0 and RTIUDCP0 registers are initialized with the variable comp equal to 0 since the starting condition of the simulator will be at a null rpm value. However, then it will be updated according to the rotary encoder regulation.

```
    /* Initialization */
    gioInit();
    rtiInit();
    sciInit();
    QEPInit();

    /* Enable Position Counter */
    eqepEnableCounter(eqepREG1);
    /* Enable Unit Timer. */
    eqepEnableUnitTimer(eqepREG1);
    /* Enable capture timer and capture period latch. */
    eqepEnableCapture(eqepREG1);


    /* Setup rti Registers */
    rtiREG1->CMP[0U].COMPx = comp;
    rtiREG1->CMP[0U].UDCPx = comp;

    /* Enable IRQ - Clear I flag in CPS register */
    _enable_IRQ_interrupt_();

    /* Enable RTI Compare 0 interrupt notification */
    rtiEnableNotification(rtiREG1,rtiNOTIFICATION_COMPARE0);

    /* Start RTI Counter Block 0 */
    rtiStartCounter(rtiREG1,rtiCOUNTER_BLOCK0);
```

Figure 5.19: Functions called into main function in HL_main.c file

Inside the main function, an endless loop function is added in order to perform the program task continuously, and it is called *while(1)*.
Firstly, a function *SciReceive* is used to allow the SCI1 module of the microcontroller to receive information about battery and key signal condition imposed by the user

via Nextion touch display.
Battery dual-state button used in dashboard page has been programmed to send a sequence of five hexadecimal values. The third element (*Rx_Data[2]*) is equal to 0x05, and *Rx_Data[3]* changes according to the button state: when the button is off Rx_Data[3] value is 0, so GIOA bit 6 is set to 0; when the button is on Rx_Data[3] value is 1, so GIOA bit 6 is set to 1.
Key signal dual state button holds the same conditions of the battery dual state, but *Rx_Data[2]*) is equal to 0x06, and Rx_Data[3] sets GIOA bit 7.

```
sciReceive( sciREG1, 5, Rx_Data);
        if(Rx_Data[2] == 0x05) {
            if(Rx_Data[3] == 0) {
                        gioSetBit(gioPORTA, 6, 0);
            }
            if(Rx_Data[3] == 1) {
                        gioSetBit(gioPORTA, 6, 1);
            }
        }
        if(Rx_Data[2] == 0x06) {
            if(Rx_Data[3] == 0) {
                        gioSetBit(gioPORTA, 7, 0);
            }
            if(Rx_Data[3] == 1) {
                        gioSetBit(gioPORTA, 7, 1);
            }
        }
```

Figure 5.20: SciReceive function controlling Nextion display battery and key dual-state buttons

Then an interrupt service routine ISR is called into while(1) function, and it is invoked when EQEP1 unit position event flag is activated (when QEPSTS register changes its bit 7 from 0 to 1). Therefore, this interrupt is invoked periodically following the unit time out interrupt described in paragraph 5.1.4. The position-counter value obtained from the rotation of the quadrature encoder knob is latched into the QPOSLAT register on unit time out event, so it is passed in the integer variable *speed*. The PIT was configured with a value of 5ms (it gives 100 rpm), and RTICOMP0 and RTIUDCP0 value is 46875: this means that 1 rpm value requires a *comp* value of 4687500. Therefore the following equation gives the comp value depending on the current *speed*:

$$comp = \frac{4687500}{speed} \tag{5.5}$$

comp represents the value that must be stored in RTICOMP0 and RTIUDCP0. Since the tachometer visualized in the Nextion display dashboard has its maximum position with an angle of 151° over 360° and the maximum rpm can be 8000 rpm,

1° will correspond to about 53 rpm. Therefore the following equation will be considered:

$$gauge\_val = speed/53 \tag{5.6}$$

RTI registers must be updated according to the new crankshaft rotational speed, so *comp* value is passed to RTICOMP0 and RTIUDCP0 registers, and it will change the period of PIT. Free running counter RTIFRC is set to the maximum value since, in this way, it will be ready to restart and count for compare matches.
Speed value is sent to Nextion touch display, and it is shown in a text box of the dashboard: it must be converted from integer variable into ASCII characters through *ltoa* function and then added in*buf* buffer after the instruction "t0.txt=", by following the Nextion Instruction Set rules. Then, the hexadecimal value of all instruction characters is sent over SCI1 module thanks to *sciSend* function and they are terminated with another *sciSend* containing *Cmd_End* array.
Almost the same happens for the instruction moving the tachometer needle visualized in touch display. Indeed *gauge_val* is added in another buffer (*buf_g*) after the ASCII command "z0.val=", which will change needle position. Then, the hexadecimal value of all instruction characters is sent over the SCI1 module again thanks to the *sciSend* function, and they are terminated with another *sciSend* containing *Cmd_End* array.
At this point, the ISR is complete, and the unit position event flag in the QEPSTS register is cleared in order to be ready to detect the next interrupt. The endless loop function and the main function are complete too.

```
/* Status flag is set to indicate that a new value is latched in the QCPRD register of quadrature encoder */
if((eqepREG1->QEPSTS & 0x80U) !=0U)
{
   //tmp_rpm = eqepREG1->QPOSLAT;
   speed = eqepREG1->QPOSLAT;

   /* formula used to match the compare funtion of rti in the desired signal period */
   comp=4687500/speed;
   /* formula used to visualize correctly the speedometer needle in the display */
   gauge_val = speed/53;

   // Setup rti Registers
   rtiREG1->CMP[0U].COMPx = comp;
   rtiREG1->CMP[0U].UDCPx = comp;
   /* Reset of the rti free running counter for keeping interrupt ready to run */
   rtiREG1->CNT[0U].FRCx = 0xFFFFFFFFU;

   /* Sending the rpm value to the display in order to read the value */
   ltoa(speed,(char *)command, 10);
   len = sprintf(buf, "t0.txt=\"%s\"",command);
   sciSend( sciREG1, len, (uint8_t *)buf);
   sciSend( sciREG1, 3, Cmd_End);

   /* sending the rpm value to the display in order to set the speedometer */
   ltoa(gauge_val,(char *)gauge, 10);
   len_g = sprintf(buf_g, "z0.val=%d",gauge_val);
   sciSend( sciREG1, len_g, (uint8_t *)buf_g);
   sciSend( sciREG1, 3, Cmd_End);

   /* Clear the Status flag. */
   eqepREG1->QEPSTS |= 0x80U;
}
```

Figure 5.21: ISR inside *while(1)* function in HL_main.c file

The last part of the HL_main.c code is the declaration of *rtiNotification* function, which will be called every time the interrupt request due to RTI compare match is pending. It is able to read each character stored in the string describing the crankshaft and camshaft wheel pattern and set the corresponding GIOA bits:

- in case of char 0 detections, both GIOA bit 2 (crankshaft digital signal) and bit 0 (camshaft digital signal) will be at logic low;
- in case of char 1, GIOA bit 2 (crankshaft digital signal) will be at logic high and bit 0 (camshaft digital signal) will be at logic low;
- in case of char 2, GIOA bit 2 (crankshaft digital signal) will be at logic low and bit 0 (camshaft digital signal) will be at logic high;
- in case of char 3, both GIOA bit 2 (crankshaft digital signal) and bit 0 (camshaft digital signal) will be at logic high;

Digital logic state imposed by *gioSetBit* function will be held over the whole PIT period. Then, index *i* is increased at every match by guaranteeing that a new character of the look-up table will be analyzed next time rtiNotification is invoked. Furthermore, index i will be reset to zero when it reaches the last character in order to read out the look-up table in endless-loop.

```
void rtiNotification(rtiBASE_t *rtiREG, uint32 notification)
{
    if(sixty_minus_two_with_cam_Cursor9[i] == '\x00'){

        gioSetBit(gioPORTA, 2, 0);
        gioSetBit(gioPORTA, 0, 0);

    }
    else if(sixty_minus_two_with_cam_Cursor9[i] == '\x01'){

        gioSetBit(gioPORTA, 2, 1);
        gioSetBit(gioPORTA, 0, 0);

    }
    else if(sixty_minus_two_with_cam_Cursor9[i] == '\x02'){

        gioSetBit(gioPORTA, 2, 0);
        gioSetBit(gioPORTA, 0, 1);

    }
    else if(sixty_minus_two_with_cam_Cursor9[i] == '\x03'){

        gioSetBit(gioPORTA, 2, 1);
        gioSetBit(gioPORTA, 0, 1);

    }

    i++;
    /* command to reset the index at the end of the wavetable */
    if(i == 240){
        i = 0;
    }
}
```

Figure 5.22: *rtiNotification* function declaration in HL_main.c file

The application code is composed of several source files that must be linked together into one executable file. For this reason, the **build** action must be performed in Code Composer Studio: *compiler* accepts C/C++ source files and produces assembly language source files, which are translated into machine language relocatable object files by the *assembler*. Then, a *linker* combines object files into a single executable object file. If any coding error is present, *Problem view* will provide a summary of errors and warnings encountered during project build.
If no error is found in the building phase, the next step is **debug** process: the application can be loaded to the targeted microcontroller evaluation board connected via USB, and its behaviour can be evaluated. This process allows to detect errors or modify behaviours of the application. When a debug session is started, Code Composer Studio will automatically switch to the CCS Debug perspective, which by default contains views associated with debugging. Once the launch process is finished, the interaction with the crankshaft and camshaft simulator hardware can be done by issuing commands such as resume button (it runs the program), suspend button (it halts the program), terminate button (it ends the debug session and return to the edit perspective), and buttons for stepping. Breakpoints can be added to inspect variables, expressions, registers at a precise running point.

Chapter 6

Results

It must be verified that the open-loop crankshaft and camshaft simulator works as expected, and this is done with a debug session that confirms the correct behaviour of the system. When the program runs, the touch display is turned on and, after the splash page, it shows the home page, which the user can interact with.

Figure 6.1: Crank/cam sensors simulator after startup

It is possible to navigate through the menu by selecting the desired option: the most important page is the dashboard, where the battery and key signal can be switched on or off, and it is also possible to read the rpm value. In order to check that crankshaft and camshaft digital signals are generated correctly, an oscilloscope is needed. Channel 1 is connected via oscilloscope probe with the GIO pin generating crankshaft signal and GND, instead channel 2 is connected with the GIO pin generating camshaft signal and GND thanks to another probe (connection details are given in Table 4.1).

6.1 Oscilloscope acquisitions

Dashboard in starting conditions shows a value of 0 rpm, battery and key signal are turned off. The oscilloscope reads two digital signals always in low logic as expected.

Figure 6.2: Crank/cam sensors simulator: dashboard page in starting conditions

At this point, the user can turn clockwise the controller knob in order to raise the simulated crankshaft rotational speed: the simulator can increase correctly the rpm value, which is displayed in the dedicated frame, and to generate also the crankshaft and camshaft position sensors signals. By rotating the controller counter-clockwise, the rpm value and also signals frequency decreases. Both digital signals are generated continuously since the look-up table is read out again from the top when the index variable oversteps the last value.
It is fundamental to verify empirically that the visualized engine rotational speed generates the digital signals with the correct frequency. As discussed in chapter 5.1.5, a periodic interrupt timer is used to manage the digital signal state to impose at each rising or falling edge of the crankshaft wheel tooth. Since in the "60-2" crankshaft wheel both teeth and gaps have an angular displacement of 3°, the period given by the detection of a new tooth is equal to twice the PIT period. Hence, the relationship between "60-2" crankshaft tooth period and rpm value is the following:

$$P_{tooth} = 2 * P_{PIT} = 2 * \frac{1}{2 * rpm} = \frac{1}{rpm} \tag{6.1}$$

The tooth frequency will be:

$$f_{tooth} = \frac{1}{P_{tooth}} = rpm \tag{6.2}$$

Therefore, the crankshaft and camshaft sensors simulator is set at a certain engine speed value. For example, it is taken an rpm value of 200rpm and with a cursor measurement in the time domain, two consecutive crankshaft signal rising edge are chosen. As shown in Figure 6.3 and 6.4, the signal is generated with a frequency equal to 200 Hz, corresponding to a crankshaft rotating at 200 rpm (the yellow signal is the crankshaft and the blue one is the camshaft).

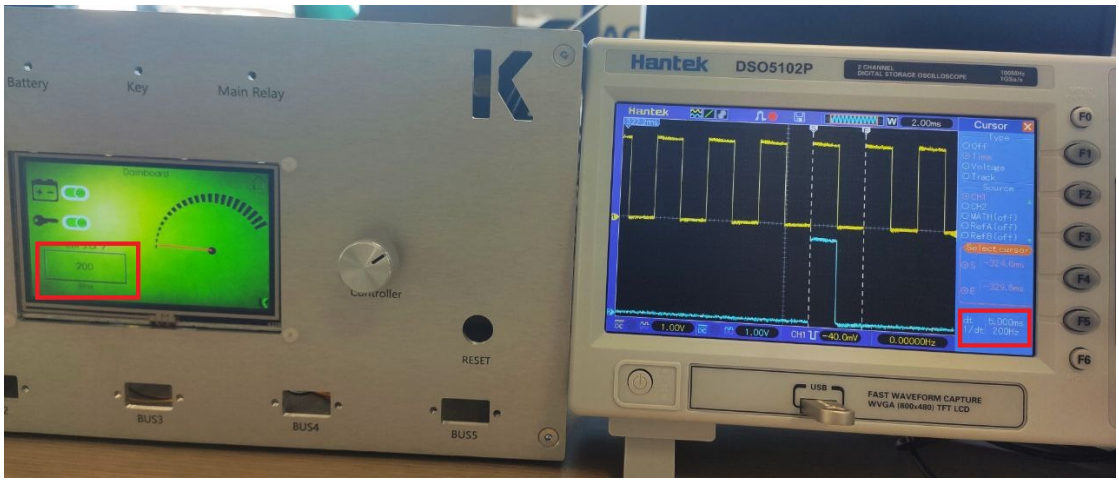

Figure 6.3: Crank/cam sensors simulator set at 200 rpm: oscilloscope in cursor mode

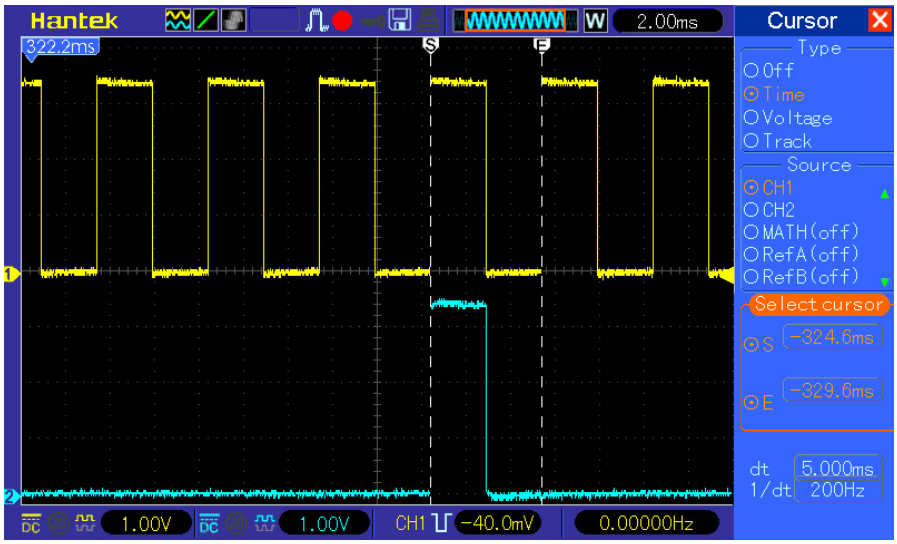

Figure 6.4: Oscilloscope frequency measurement in cursor mode: 200 Hz

In order to better visualize the two signals, the camshaft sensor signal has been

shifted down by using oscilloscope vertical position regulation, but it has the same voltage dimension as the crankshaft signals.
It is possible to notice that the tachometer needle is moved from the rest position according to the rpm value set by the user.

The same verification procedure is performed to check the correctness of the crankshaft wheel teeth period at different rpm value: it has been verified empirically that tooth frequency is correct with all the possible engine speed value. Therefore, the rotary encoder can act on the update frequency of the crank and cams signals in real-time. The following oscilloscope acquisitions show that the simulator works as expected with also some precise rpm values.

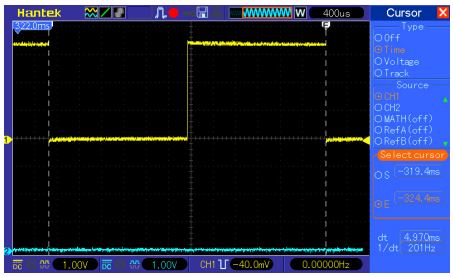

Figure 6.5: Oscilloscope frequency measurement in cursor mode: 201 Hz

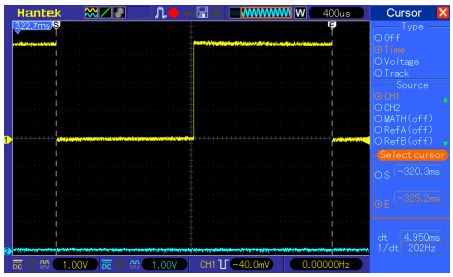

Figure 6.6: Oscilloscope frequency measurement in cursor mode: 202 Hz

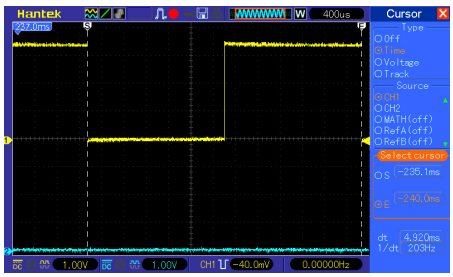

Figure 6.7: Oscilloscope frequency measurement in cursor mode: 203 Hz

After this analysis, another verification is required. As mentioned in chapter 5.2, the look-up table has been realized from a "60-2" rotating crankshaft wheel pattern and a "6+1" camshaft wheel pattern used in a Cursor 9 engine. In this case, cam teeth have an angular displacement of 3° such as those in the crankshaft wheel: they are in correspondence of 12th, 32th, 37th, 52nd crankshaft teeth in the first crankshaft revolution and 12th, 32th and 52nd crankshaft teeth in second crankshaft revolution. To be clear, a model of the patterns has been plotted through MATLAB, and it is shown in Figure 6.8, where blue signal represents crank, and red signal represents cam over a whole engine cycle of 720°.

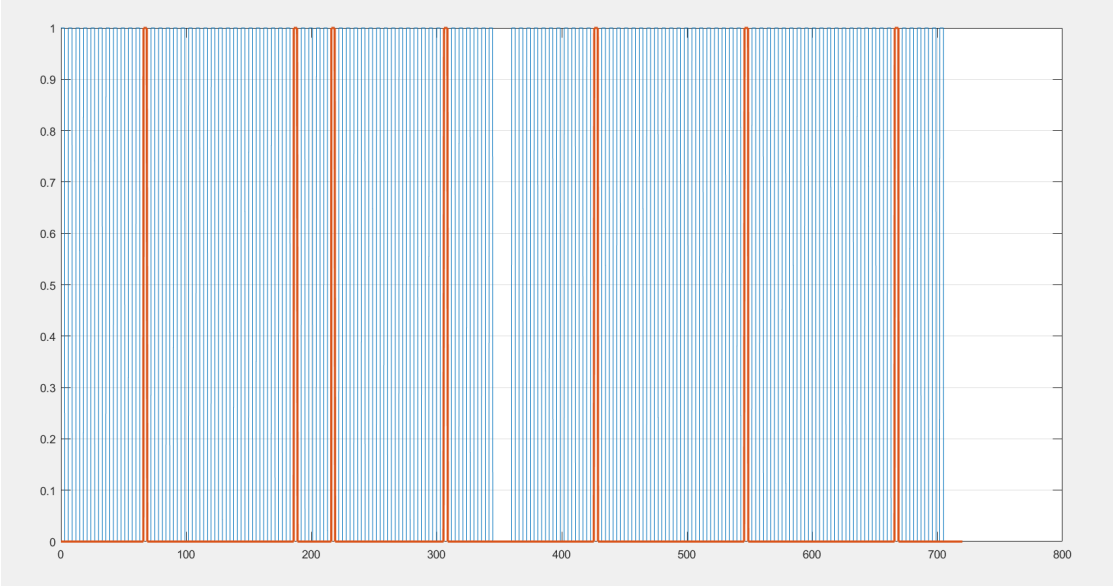

Figure 6.8: Model of "60-2" crankshaft wheel pattern and "6+1" camshaft wheel pattern

These patterns have been reproduced correctly by the simulator. As an example, in Figure 6.9 is shown an oscilloscope acquisition of the signals generated with 200rpm: digital cam signal goes in the high state exactly as it does in Cursor 9 model.

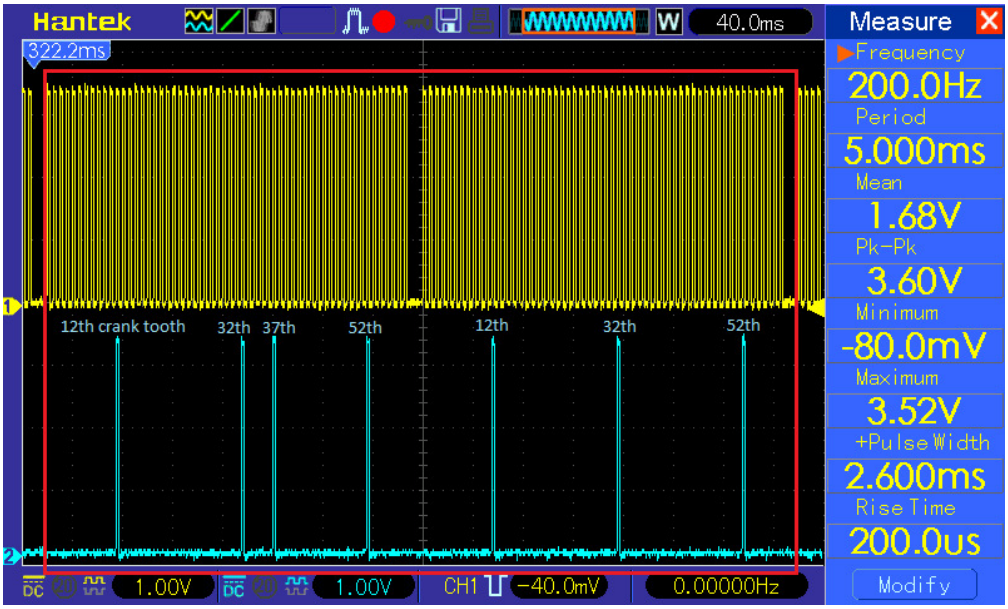

Figure 6.9: Oscilloscope acquisition of crank and cam signals at 200 rpm: 720° crankshaft rotation is framed

Moreover, for the sake of completeness, the simulator has been tested by increasing and decreasing in real-time the rpm value and verifying that output signals were generated with the corresponding tooth frequency. The following figures show the crankshaft and camshaft sensors simulator working as expected at 1000 rpm and 2500 rpm. The tachometer needle also moves correctly.

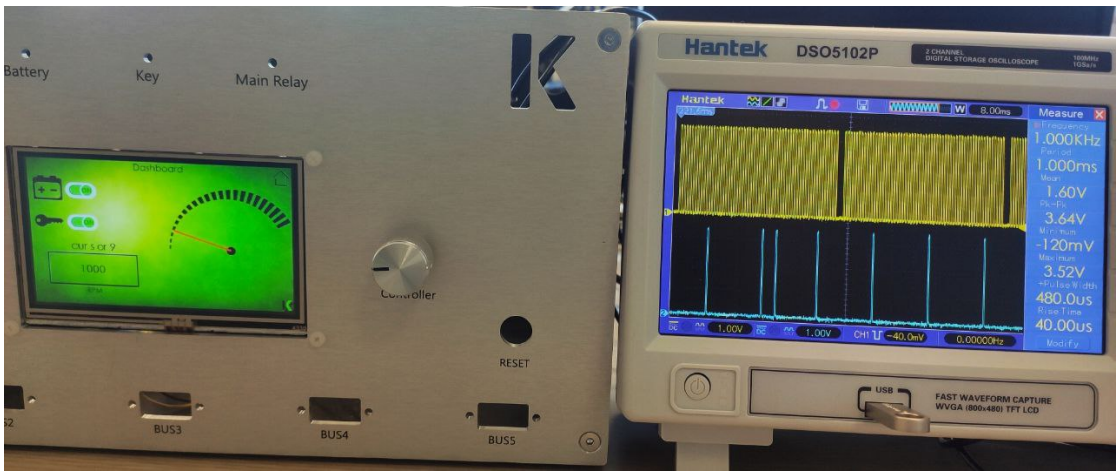

Figure 6.10: Crankshaft and Camshaft sensors simulator set at 1000 rpm: oscilloscope in measure mode

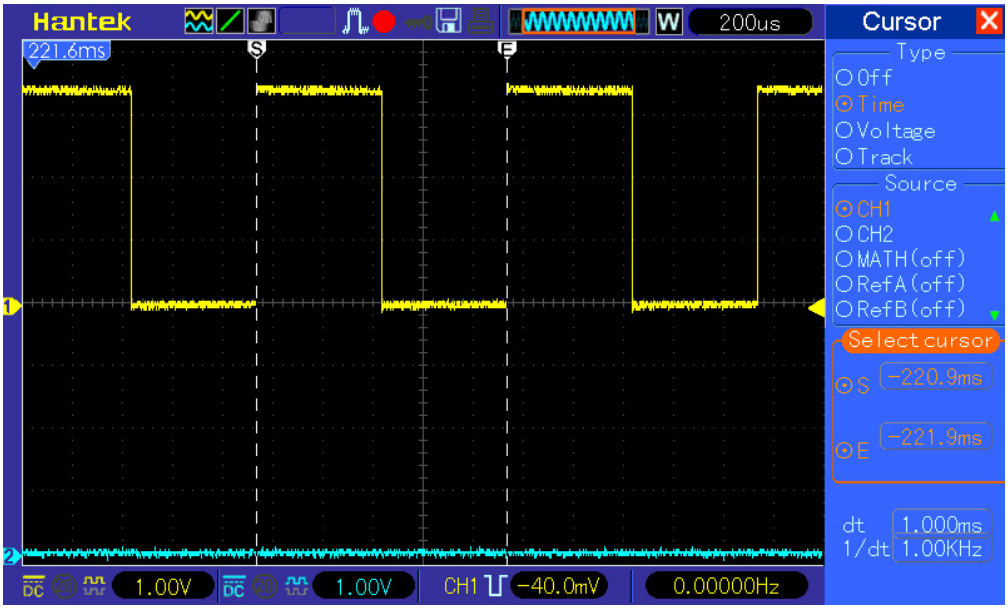

Figure 6.11: Oscilloscope acquisition in cursor mode of crankshaft and camshaft sensors signals generated by the simulator at 1000 rpm

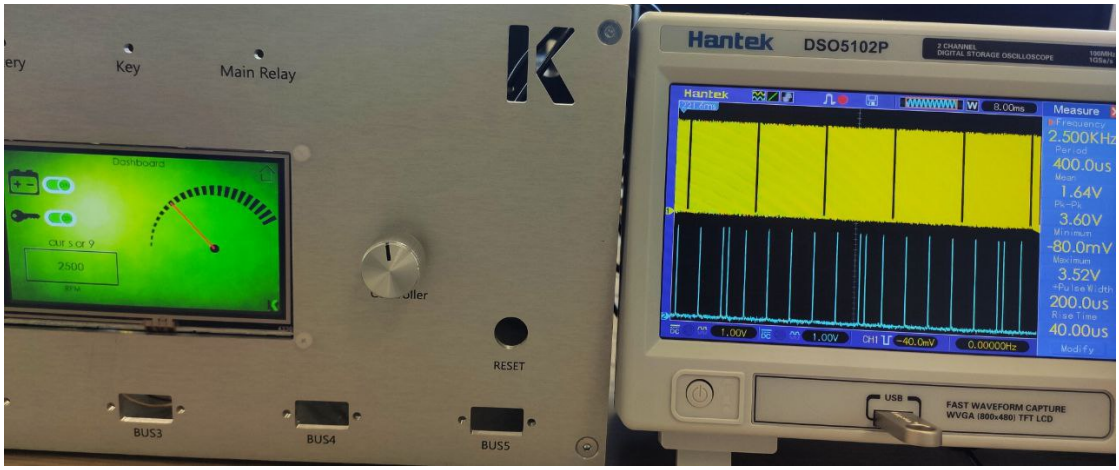

Figure 6.12: Crankshaft and Camshaft sensors simulator set at 2500 rpm: oscilloscope in measure mode

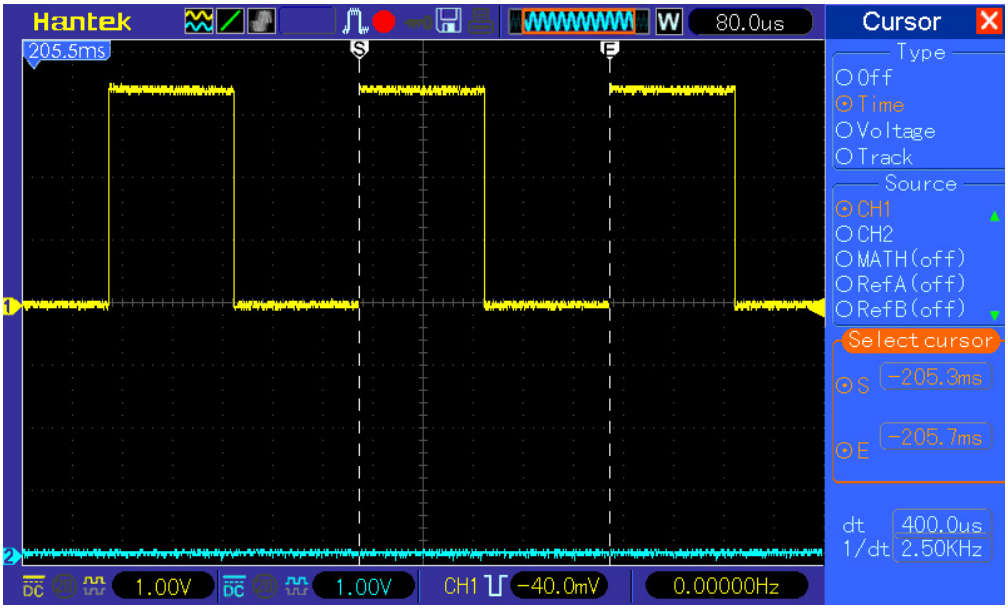

Figure 6.13: Oscilloscope acquisition in cursor mode of crankshaft and camshaft sensors signals generated by the simulator at 2500 rpm

The realized open-loop simulator now can be used to perform ECU tests. The application is stored in SRAM, and it runs every time the simulator is supplied with a voltage of 5V via its USB connector.

6.2 Crank/Cam position sensors simulator connected to a real ECU

As a sketch of the proof, the open-loop simulator is connected to a real ECU: Bosch EDC17C69 running software for Diesel 1.6cc E6D engine.
Firstly, the crankshaft and camshaft wheel patterns related to this simulated engine have been studied. Carmaker provides a model with a "60-2" crankshaft wheel and a 3-teeth camshaft wheel. The first cam tooth spans from an angle of 0° to 123°, the second tooth 180°-357° and third tooth 489°-537°. A MATLAB script file has been created in order to plot the crank and cam patterns.

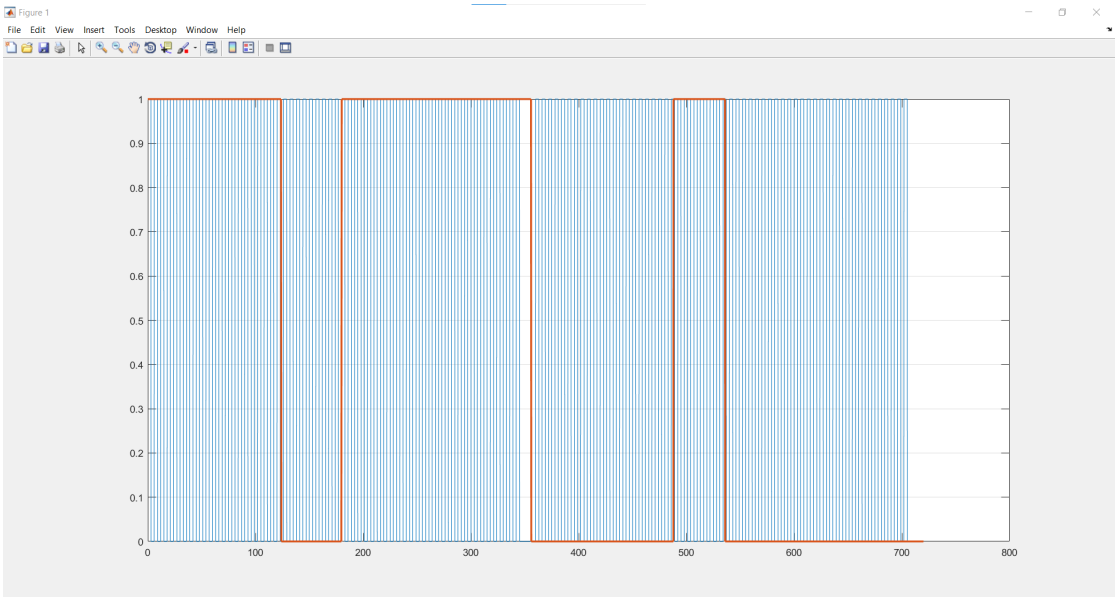

Figure 6.14: Model of "60-2" crankshaft wheel pattern and 3-teeth camshaft wheel pattern

Therefore, the variable describing the look-up table in the application code written in Code Composer Studio has been edited accordingly, as shown in Figure 6.15.

```
/*cam for ECU test*/
 const unsigned char sixty_minus_two_with_cam_Cursor9[240]=
  { /* 60-2 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 1-5 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 6-10 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 11-15 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 16-20 */
    3,0,1,0,1,0,1,0,1,0,  /* teeth 21-25 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 26-30 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 31-35 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 36-40 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 41-45 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 46-50 */
    3,2,3,2,3,2,3,2,3,2,  /* teeth 51-55 */
    3,2,3,2,3,2,2,2,2,0,  /* teeth 56-58 and 59-60 MISSING */
    1,0,1,0,1,0,1,0,1,0,  /* Second revolution teeth 1-5 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 6-10 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 11-15 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 16-20 */
    1,0,1,2,3,2,3,2,3,2,  /* teeth 21-25 */
    3,2,3,2,3,2,3,2,3,0,  /* teeth 26-30 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 31-35 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 36-40 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 41-45 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 46-50 */
    1,0,1,0,1,0,1,0,1,0,  /* teeth 51-55 */
    1,0,1,0,1,0,0,0,0,0   /* teeth 56-58 and 59-60 MISSING */
  };
```

Figure 6.15: Application code variable describing the new crankshaft and camshaft wheel pattern

The updated code has been debugged, and the application has been loaded to the evaluation board governing the simulator. Once again, the oscilloscope has been connected to the output simulator pins, and it has been checked that the simulator works as expected. It has been verified that the generated cam pattern corresponds to model one, as shown in Figure 6.16.

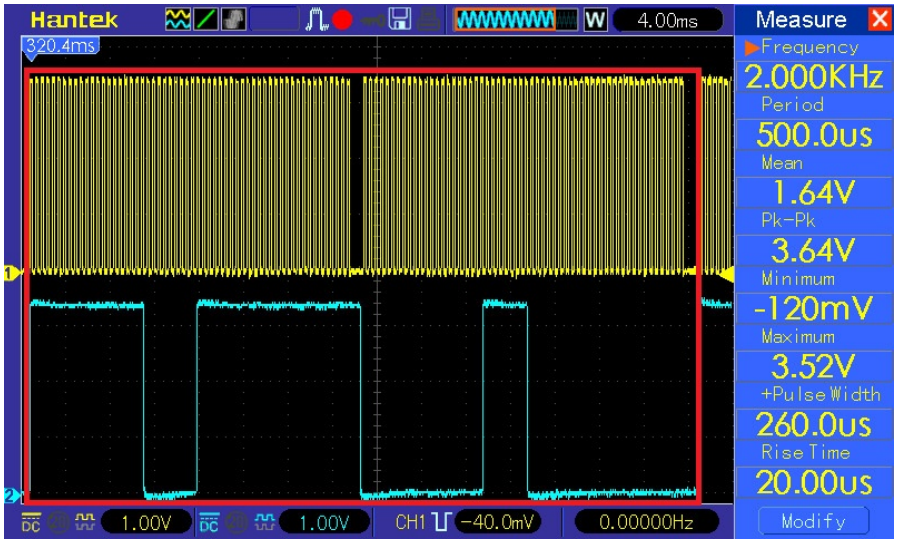

Figure 6.16: Oscilloscope acquisition of crank and cam signals at 2000 rpm: 720° crankshaft rotation is framed

The simulator generates the two digital signals varying in real-time the crank tooth frequency according to the rpm set by the user.
Then, the digital pins outputting the crankshaft sensor and the camshaft sensor signals can be connected to a real ECU. A HIL SCALEXIO simulator configured for automotive engine verification and validation is connected to the ECU, and a Breakout Box is inserted between them in order to connect or disconnect ECU pins or SCALEXIO channels for testing purposes.
From the engine control system electrical scheme provided by the carmaker it has been derived how ECU interfaces with the SCALEXIO simulator: the crankshaft position sensor signal is taken as input by the ECU with pin A20 (GND with pin A21), and camshaft position sensor with pin A51 (GND with pin A52). Therefore, breakout box connections related to these pins have been disconnected since it is not needed the connection with SCALEXIO anymore, but now with the open-loop crank/cam simulator: jumper wires to banana plugs are used to connect the open-loop simulator's crank and cam digital output pins to the Break-Out Box entries connected to the related ECU pins. Figure 6.17 shows wiring connection between breakout box and simulator: blue plug provides crankshaft signal to A20 entry;

red plug provides GND to A21 entry; yellow plug provides camshaft signal to A51 entry; green plug provides GND to A52 entry.

Figure 6.17: Breakout box connecting the real ECU with SCALEXIO simulator: external connection with open-loop crank/cam simulator

The other ECU pins remain connected with the dSpace SCALEXIO simulator, which can simulate the other signals of the engine sensors and receive actuator signals. Figure 6.18 shows a conceptual overview of the system configured.

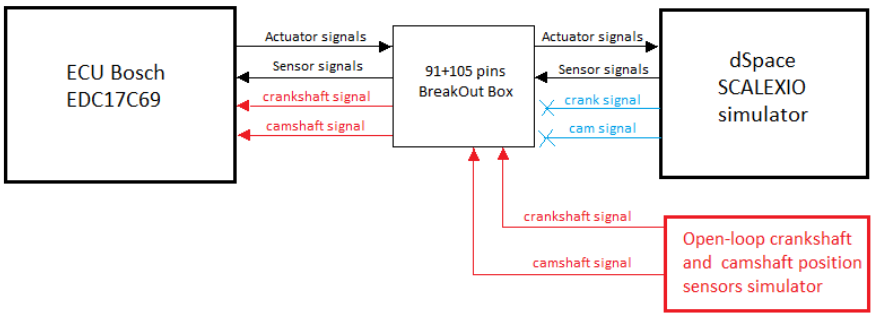

Figure 6.18: A conceptual overview of the exchanging signals between ECU, SCALEXIO simulator and crank/cam simulator

The used ECU has also been connected to an ETAS ES592: it is a hardware interface module used for ECU calibration, diagnostics, flash programming, and bus monitoring. It is connected to a host PC via ethernet, where the INCA software tool can be used for the calibration, diagnostics and validation of automotive electronic systems.
At this point, the test is performed: the real ECU is supplied by the SCALEXIO simulator and receives crankshaft and camshaft signals from the realized simulator. Several rpm values have been imposed as input by the user, and ECU has understood the incoming crankshaft and camshaft signals. Indeed, through INCA, ECU software variables can be monitored, and engine rotational speed corresponds to the value set from the simulator.

Figure 6.19: Open-loop crank/cam simulator connected to a real ECU: crankshaft rotational speed is set at 1000 rpm and the value is correctly read by the ECU, and is visualized via INCA software

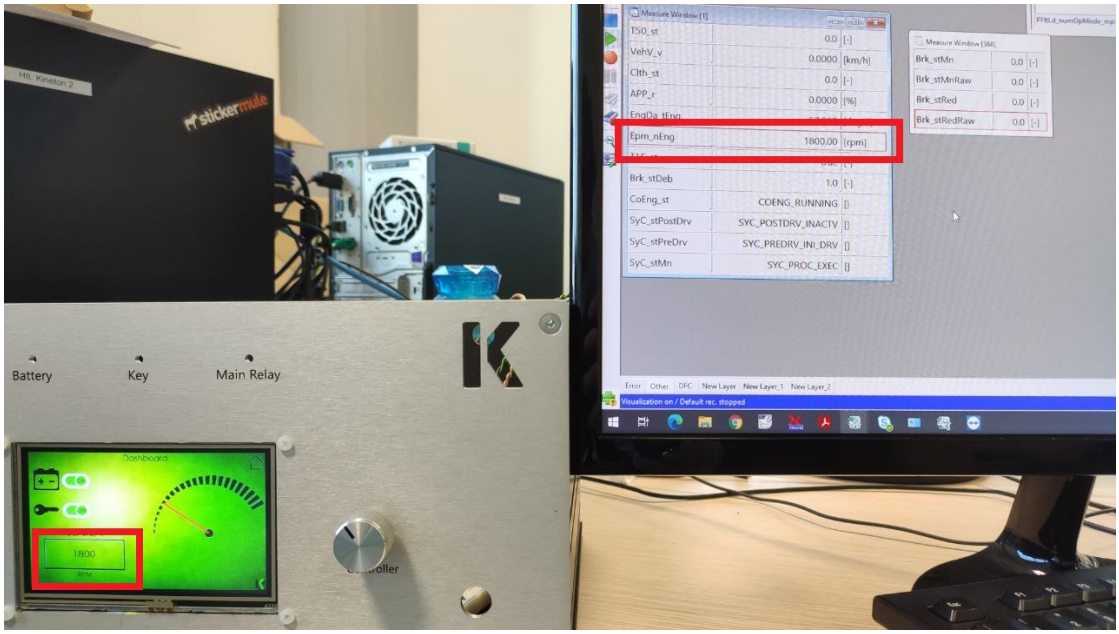

Figure 6.20: Open-loop crank/cam simulator connected to a real ECU: crankshaft rotational speed is set at 1800 rpm and the value is correctly read by the ECU, and is visualized via INCA software

Therefore, the open-loop crankshaft and camshaft position sensor simulator can be used to perform some ECU validation and verification tests.

Chapter 7

Conclusions

Electronic systems in the automotive field are continuously increasing their complexity, and they lead companies to study new strategies in order to improve vehicle performance. ECUs are the main embedded systems that control the electrical systems and subsystems in a vehicle, and ECM is the fundamental one since it controls all the actuators on an internal combustion engine. Developing and testing ECUs may require a big effort in terms of time and consequently of costs; therefore, the automotive industry is interested in reducing workloads, bearing in mind that the safety and validity of the testing process must be guaranteed. In this sense, the proposed open-loop crankshaft and camshaft simulator is developed: ECM software developers and testers can use it to perform a fast simulation of the speed of an internal combustion engine, also providing the ignition timing. Indeed, the crankshaft position sensor detects the relative engine speed, and the camshaft position sensor is used to determine which cylinder is firing to synchronize the fuel injector and coil firing sequence.
The master thesis work has been focused on the realization of an open-loop simulator generating crankshaft and camshaft position sensors digital signals: a customized embedded system has been configured in order to perform the task as desired by the user. The simulator core is the microcontroller evaluation platform (Texas Instruments' Hercules TMS570LC43x board); therefore, the whole logic has been implemented on it. Hardware characteristics and peripheral functionalities have been studied in order to build the most suitable configuration of the system. The most important task is the generation of the digital signals with a period that is determined by the rpm value set by the user. Therefore, a human-machine interface has been implemented by connecting two I/O peripherals to the evaluation board: a memory touch display and a quadrature encoder. The first one has been programmed with a menu to allows the engine speed visualization but also the controlling of some settings; the second one can correctly increase or decrease the speed. Simulator basic software and application code have been programmed

through software tools provided by Texas Instruments. The final executable file is loaded into the evaluation board after a debugging session. The program is stored in microcontroller SRAM memory, and it is launched every time the simulator is powered.
Crankshaft and camshaft wheel patterns are stored in the program as a look-up table which is read out thanks to a periodic interrupt controlled by the user: this leads to the generation of the crankshaft and camshaft digital sensor signals with the correct frequency. Hence, the obtained signals can be used as an actual reference for testing and validating internal combustion engine ECMs, and the simulator can be provided to automotive companies.

7.1 Future Works

The implemented simulator works as expected, but some improvements are foreseen. Firstly, another I/O peripheral (i.e. a USB host shield) can be added to the evaluation board in order to read out directly a look-up table very similar to SCALEXIO wavetables stored in CSV files. User in this way will be able to change crankshaft and camshaft wheel patterns fastly, loading a CSV file from a USB flash drive. Furthermore, the application can be enhanced in order to simulate also analog and reverse crankshaft and camshaft sensors signals by selecting the type from the settings page already developed into the touch display.

Bibliography

[1] N. Navet and F. Simonot-Lion. «Automotive Embedded System Handbook». In: Boca Raton, FL: CRC Press, 2009. Chap. 1 (cit. on p. 1).

[2] R. Palomera M. Jiménez and I. Couvertier. «Introduction to Embedded Systems». In: New York, NY: Springer, 2014. Chap. 1 (cit. on p. 7).

[3] M. Violante. «Operating System Structure». In: Torino: Politecnico di Torino, 2019 (cit. on p. 8).

[4] *ISO 26262-1:2018(en) Road vehicles – Functional safety*. International Standard. 2018 (cit. on p. 11).

[5] M. Violante. «Introduction to ISO-26262». In: Torino: Politecnico di Torino, 2019 (cit. on p. 13).

[6] dSPACE GmbH. *SCALEXIO System Overview*. Paderborn, Germany: dSPACE GmbH, 2015 (cit. on p. 19).

[7] *SPNU563A datasheet*. Dallas, Texas: Texas Instruments (cit. on p. 41).

[8] Nextion Distribution. *NX4827T043_011 datasheet*. 2018. URL: `https://nextion.ca/datasheets/nx4827t043_011/` (cit. on p. 54).

[9] ITEAD STUDIO. *The Nextion Instruction Set*. 2011-2020. URL: `https://nextion.tech/instruction-set/` (cit. on p. 56).