Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

MECCANICA E SCIENZE AVANZATE DELL'INGEGNERIA

Ciclo XXXIII

Settore Concorsuale: 09/C1-MACCHINE E SISTEMI PER L'ENERGIA E L'AMBIENTE

Settore Scientifico Disciplinare: ING-IND/08-MACCHINE A FLUIDO

Employment of Real-Time/FPGA Architectures for Test and Control of Automotive Engines

Presentata da:

Marco Abbondanza

Coordinatore Dottorato

Prof. Marco Carricato

Supervisore

Prof. Enrico Corti

Co-Supervisore

Prof. Nicolò Cavina

Esame finale anno 2021

Abstract

Nowadays the production of increasingly complex and electrified vehicles requires the implementation of new control and monitoring systems. This reason, together with the tendency of moving rapidly from the test bench to the vehicle, leads to a landscape that requires the development of embedded hardware and software to face the application effectively and efficiently. The development of application-based software on real-time/FPGA hardware could be a good answer for these challenges: FPGA grants parallel low-level and high-speed calculation/timing, while the Real-Time processor can handle high-level calculation layers, logging and communication functions with determinism. Thanks to the software flexibility and small dimensions, these architectures can find a perfect collocation as engine RCP (Rapid Control Prototyping) units and as smart data logger/analyser, both for test bench and on vehicle application. Efforts have been done for building a base architecture with common functionalities capable of easily hosting application-specific control code. Several case studies originating in this scenario will be shown; dedicated solutions for protype applications have been developed exploiting a real-time/FPGA architecture as ECU (Engine Control Unit) and custom RCP functionalities, such as water injection and testing hydraulic brake control.

CONTENTS INDEX

SYMBOLS & ACRONYMS

ACU	Aerodynamics Control Unit
ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
ATDC	After Top Dead Centre
BTDC	Before Top Dead Centre
Cal	Abbreviation for Calibration
CL	Closed Loop
DI	Direct Injector / Injection
DSP	Digital Signal Processor
DVR	Data Value Reference
ECU	Engine Control Unit or Electronic Control Unit
EM	Electric Motor
EMS	Engine Management System
EOI	End of Injection
ESP	Electronic Stability Program
EVC	Exhaust Valve Closing
FPGA	Field Programmable Gate Array
FW	Firmware
GPIO	General Purpose Input Output
HS	High-Side
HW	Hardware
I/O	Input / Output
ICE	Internal Combustion Engine
IDE	Integrated Development Environment
IVO	Intake Valve Opening
LS	Low-Side
LUT	Look-Up Table
MAir	Abbreviation for MultiAir
MAPO	Maximum Amplitude Pressure Oscillation
MFB(X)	Mass Fraction Burned X %
MCU	(Electric) Motor Control Unit
Meas	Abbreviation for Measurement
NTC	Negative Temperature Coefficient
OBI	On Board Indicating system (product from Alma-Automotive)
OEM	Original Equipment Manufacturer
OL	Open Loop
OS	Operative System
P&H	Peak and Hold drivers
PFI	Port Fuel Injector / Injection
pRail	Abbreviation for Rail Pressure
PTC	Positive Temperature Coefficient
PWM	Pulse-Width Modulation

RCP	Rapid Control Prototyping
RPM	Revolution per Minute
RT	Real-Time
SCAM	Camshaft Teeth Wheel Signal
SMOT	Crankshaft Teeth Wheel Signal
SOI	Start of Injection
SW	Software
TCU	Transmission Control Unit
TDC	Top Dead Centre
VI	Virtual Instrument (LabVIEW Function)
VVL	Variable Valve Lift
VVT	Variable Valve Timing

1. Introduction to Test and Control of Automotive Engines

Because of the complexity of modern vehicle systems and engines, the implementation and integration of control and monitoring systems is crucial. Additionally, the automotive industry is forced to move rapidly from the test bench to the vehicle and for these reasons fast and effective prototyping tools are required to face the applications development on embedded hardware.

During the first part of these projects, time and cost constraints usually orient the hardware choice toward an **RCP (Rapid Control Prototyping)** system, programmable by means of high-level software, oriented to control, acquisition and data analysis. Obtaining an openly programmable and flexible (open and customizable control over every sensor and actuator) control unit for conducting research is fundamental.

On the other hand, the counterpart of an RCP is a **HIL (Hardware-in-the-Loop)**: testing can begin when the vehicle / engine is in the early stages of development and not yet available for testing. This accelerates the development process, allowing rapid changes to be made based on early feedbacks, and improving the quality of the final product.

Both solutions can be also combined to test control models running on an RCP against a simulated plant running on HIL.

These prototyping and testing systems provide one of the quickest paths from simulation to real-time testing directly on the hardware. The toolchains and the dedicated hardware are generally based on a high-level programming approach with a particular orientation to control and acquisition. Different kinds of hardware and software platforms are available on the market, ranging from simply programmable ECUs based on low-power microcontrollers and with limited functionalities, to multi-core CPUs and FPGA based architectures fully customizable in the I/O and fully programmable in the software. Differently from production dedicated solutions or in-house developed embedded controllers, these prototyping systems provide straight forward route to custom applications implementation, with the assurance of I/O expansion (with additional I/O modules or migrating to a more capable HW) and software portability for future development. Easy configuration, connection and effective debugging tools complete the landscape that indicates the powerfulness of these systems as a fast route for testing and control in the automotive industry [1] [2].

At the end of the prototyping phase, some RCPs platforms are also suitable for small production volumes or in some cases the toolchain allows the software portability on production hardware.

1.1 **RCP SYSTEMS**

The platforms dedicated to Rapid Control Prototyping allows engineers to spend their efforts on the implementation of control, acquisition and processing algorithms. Several solutions are available on the market spacing from development ECUs and additional prototyping hardware from ECUs manufacturers, motorsport ECUs and powerful general-purpose controllers. Even if with a different range of computational power and flexibility of customization (over HW and SW), every platform is intended to operate coherently to some common objectives: save costs, reduce risks and shorten development time. Required features are:

- Real-Time control, tunable over parameters
- Configurable I/O to meet specific project needs
- Wide range of I/O that can be expandable in case of modular architecture
- Fast route from algorithm concept to code implementation and testing
- Debugging tools
- Code portability in case of RCP platform change within the same producer
- Possibility to continuously try new ideas and concepts using a production independent development platform

An RCP platform can be resumed as an easy-programmable real-time machine capable of reading sensors, drive actuators and communicate with other devices and the development and calibration environment.

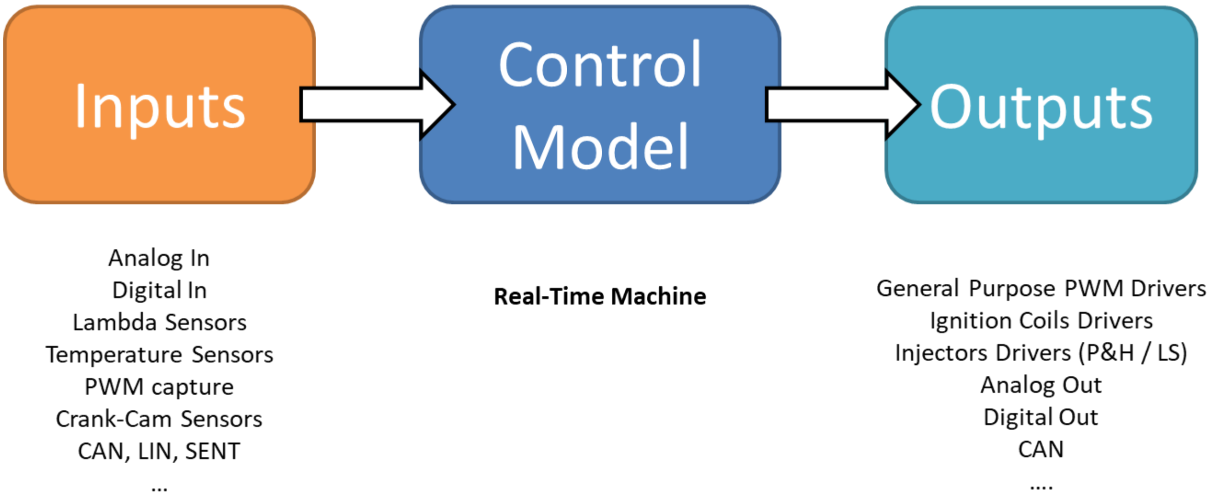


Figure 1: Typical automotive RCP system requirements and main functionalities overview

The image above shows a standard functional representation of an RCP, but the same scheme can be used and integrated in different ways depending on the purpose.

1.1.1 *Functional Bypass*

The functional bypass allows to verify control strategies defined by the calibrator in order to validate templates that replace a function, or part of it, compared to path implemented in the ECU.

This allows rapid and immediate results to develop innovative strategies to change ECU control.

In the scheme below the **function F2 is bypassed by function F2*.**

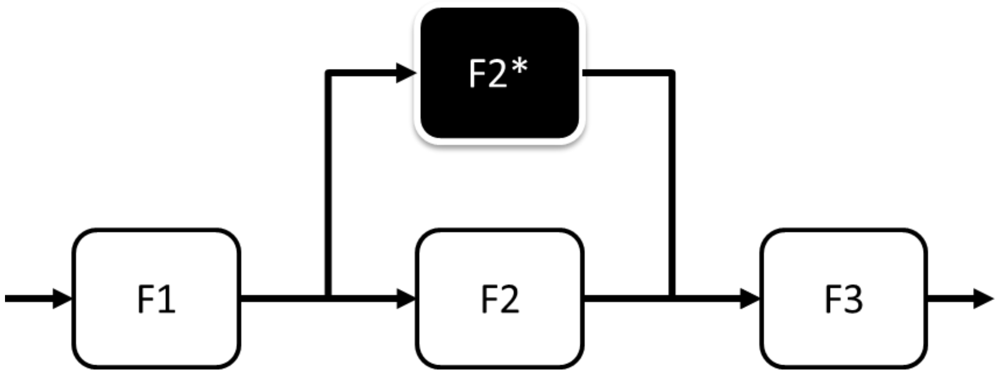


Figure 2: Functional Bypass diagram

1.1.2 *Added Functionality*

This functionality allows adding control strategies defined by the user, in order to actuate additional HW and validate functions not implemented in the ECU.

This allows rapid and immediate results in the development of innovative strategies to be added to ECU control.

In the scheme below the **function F4 is added to the controller.**

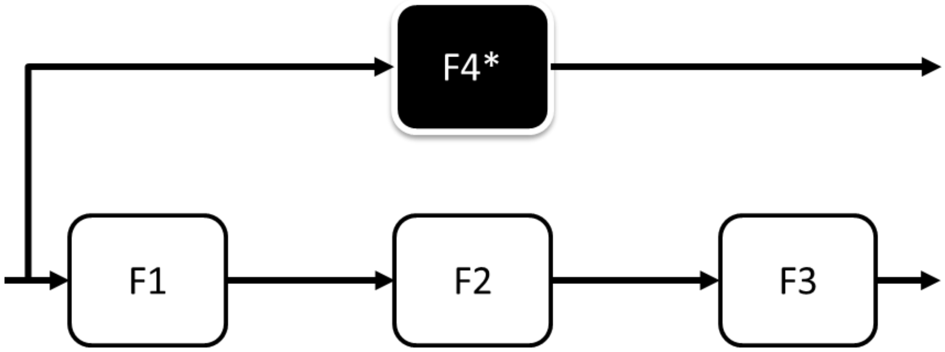


Figure 3: Added Functionality diagram

1.1.3 ***Full Bypass***

Full bypass allows to verify control strategies defined by the calibrator in order to validate templates that replace a function, or part of it, compared to the corresponding path implemented in the ECU.

The approach enables rapid and immediate results, crucial in the development of innovative strategies to change ECU control.

In the scheme below the **full control (F1+F2+F3) is replaced by a totally new control model (F1*+F2*+F3*+F4*).**

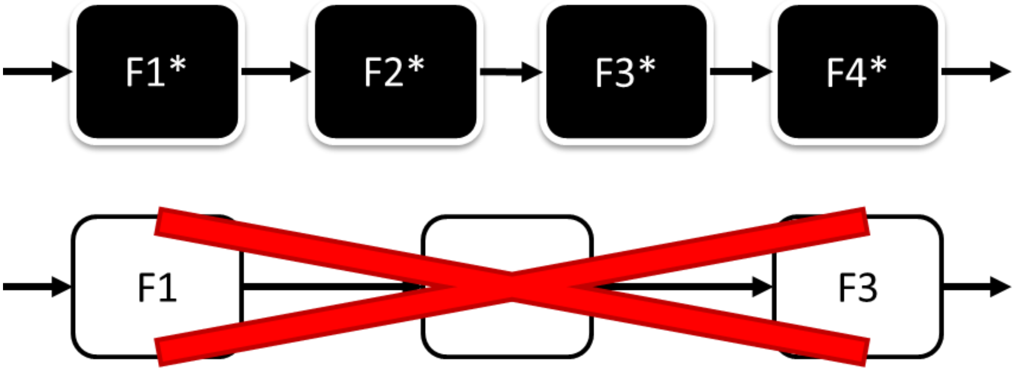
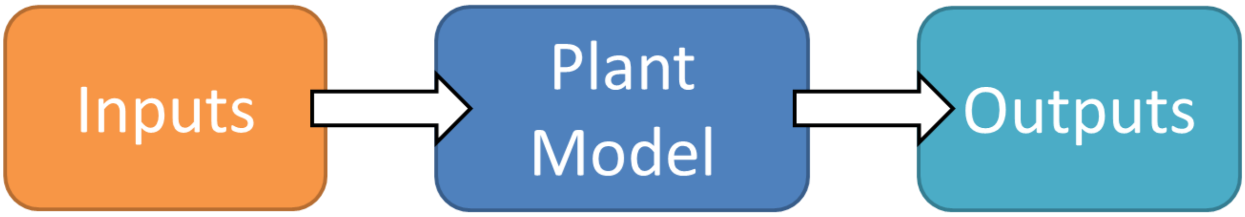


Figure 4: Full Bypass diagram

1.2 HIL SYSTEMS

Despite being based on the same type or similar hardware of RCPs, these facilities are intended to drive tests for validating control algorithms and hardware exploiting a virtual real-time environment to simulate the physical system to control. As for the RCPs the main goals of an HIL based approach are costs savings, risk reduction and time to market reduction. Requirements, that came up to be intended as “benefits” over a more traditional approach, are:

- Real-Time simulation, tunable over parameters
- Configurable I/O to meet specific project needs.
- Wide range of I/O that can be expandable in case of modular architecture
- test the behaviour of control algorithms without physical prototypes
- Perform tests beyond the range of normal parameters or plant capabilities without the risk of damage critical equipment or precious prototypes
- Continuously try new ideas, even when the actual modified plant is not available
- Traditional sequential workflow replaced by a circular one that enables continuous verification and validation at designs at the earliest possible stages



Analog In
Digital In
PWM capture
Loads In (Ign / Inj / pwm / ecc)
CAN, LIN, SENT
…

Real-Time Machine

Analog Out
Digital Out
PWM generation
Configurable Resistance
Sensors Emulators (resolver, Lambda,…)
Fault Insertion
CAN, LIN, SENT
….

Figure 5: Typical automotive HIL system requirements and main functionalities overview

An HIL platform can be resumed as an easy-programmable real-time machine capable of capture and interpret actuations, simulate sensors and communicate with other devices and the development and calibration environment.

The HIL approach is not the only solution that can be used to test and validate a control software / hardware. A full set of options that fit different steps and capabilities of simulation is resumed below:

Model-in-the-loop (MIL): pure simulation over a host computer of both control and plant, exploiting the capabilities of built-in simulation environment.

Software-in-the-loop (SIL): pure simulation over a host computer of both control and plant, but in this case, simulation is accomplished using a compiled version of the control code. The simulation is still disconnected form the embedded hardware that will be used.

Processor-in-the-loop (PIL): focuses the simulation on the use of an evaluation board of the embedded controller, connected to the host PC on which executes the plant model. At this point the control code is compiled for architectures comparable to the final embedded device.

Hardware-in-the-loop (HIL): provides the combination of executing the embedded software on the actual ECU hardware in combination with a model of the physical system (plant). HIL systems may not involve just a simple situation when a single and complete plant is simulated under the control of a single controller:

- The plants and the control units under test can be multiple. In modern vehicles and engine is common to have multiple control ECUs that directs different subsystems: is useful testing the integration and correct problems at early stage
- Not always the entire vehicle / engine system (plants + controls) is involved
- Not always the full plant is simulated: some HW can be connected to the ECUs
- Not always the full control runs on ECUs: some ECUs can be missing

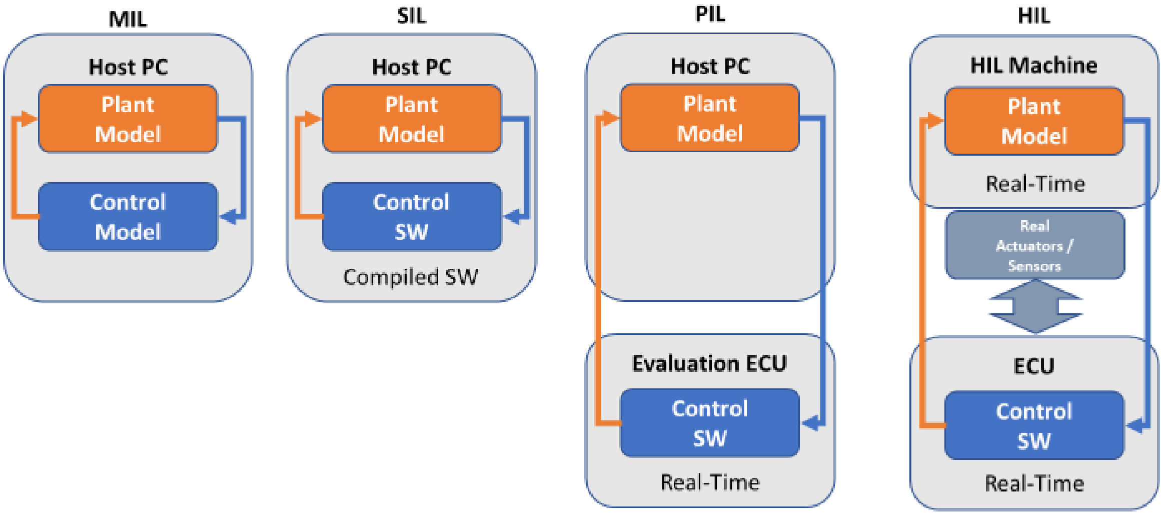


Figure 6: Schematic representation of the differences of MIL, SIL, PIL and HIL

1.3 PhD Aim and Activity Overview

The approaches presented in the previous sections, with particular attention to RCP systems, have been investigated in the potentiality and applied to the automotive engines, but are not exclusive of this environment. Each industrial sector that involves control systems and plants to be controlled can benefit from them: aerospace, energy production, automation are just a few examples of fields were these kinds of approaches are widely used [1] [2].

The main topic of the research activity has been the development and definition of a modular base SW architecture with common functionalities. The efforts have been concentrated since the first year of PhD for building a reusable framework for Real-Time / FPGA platforms based on the toolchain available from National Instruments. The purpose of the work has been the realization of a platform suitable for a wide range of custom solutions capable of granting the independence of the development of the projects from closed ecosystems and the tools for a quick expansion of the control logics and easy integration with other systems. As presented below, the application fields have been pure and applied research projects in engine testing field and the framework is projected to all the innovation and research activities that focus on complex systems integration and control such as hybrid and electrified powertrains. For this reason, this PhD thesis focuses on the description of the concept solutions of the framework and application and validation on the test field in terms of ease of implementation, effectiveness and adaptability to different projects. The research activities supported by the developed solutions, mainly conducted by colleagues, have been left on the background of the exposition to concentrate on the functional solutions and their validation on the field.

As a result, taking advantage of the flexibility of the software approach, several RCPs for different applications have been developed and are presented as validation examples on real testing field: RCPs for complete engine management, for added functionality and test bench management. The development and the success of the project have been possible thanks to the research and test campaigns conducted at the test bench by our research group to which the framework gave support.

During the first PhD year two main challenges have been faced. The first one has been the management of RCCI combustion on one “laboratory” cylinder out of 4 cylinders of an automotive diesel engine. The aim of the activity was reaching independence on load and control parameters for the laboratory cylinder, gaining advantage from the stability guaranteed by the other three. The second has been the development of the management system for a two cylinder turbocharged engine, equipped with the MultiAir system [3] [4] for aeronautical use. The main challenge was the control of the valve lift system with precision and the integration in the air subsystem of the torque structure.

The second year focused mainly on the development of a base software structure and the application of this new architecture to different case of studies. The “new” platform has been built in accordance with the need of setting up quickly new RCPs without rewriting code from a white sheet and having a wide set of common functionalities ready to run. During the second year the completion and the commissioning of the Multiair engine have been reached and in parallel also a water injection RCP and Variable Valve Lift controller have been developed.

During the third year, several improvements on the base architecture and functionalities have been realised and exploited in the development of a complete spark ignition engine management system, including a wide range of actuators control capabilities: PFI and DI injection, GDI pump control, VVT control and an enhanced throttle control with dedicated calibration tool [5] [6]. Since combustion indexes are frequently used for feedback control actuation parameters [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17], an independent SW module for combustion analysis has been integrated directly in the ECU, exploiting the advantage of having the indicating analysis and the control algorithm all on the same device, which is not a common RCP feature. Moreover, a colleague exploited the architecture and the previously developed diesel control as a base for the development of RCP strategies supporting a specific research on low temperature combustions [18] [19]. Additionally, a hydraulic brake controller based on a RT-FPGA ECU has been implemented for testing purpose starting from a Simulink model preliminary work [20] [21] [22].

A research period has also been attended at Aston Martin Lagonda Ltd. facilities focusing on the implementation of a hardware in the loop simulation for hybrid powertrains on Simulink Real-Time hardware.

1.4 Test Facilities and Engines

The research and test activities has been conducted mainly in the Unibo Hangar-Laboratory of Via Seganti 103 in Forlì, which is equipped with three test benches. As introduced in 1.3 different engines have been involved during the PhD.

As a reference for modern spark ignition engines two engines from FCA has been adopted the small 0.9 litre two-cylinder TwinAir engine and the performant 2.0 four-cylinder GME engine.

0.9 TwinAir	
Strokes per Cycle	4
Geometry	2 cylinders in line
Intake System	Turbocharged with waste-gate valve and equipped with MultiAir system on the intake valves
Injection System	Gasoline PFI
Displacement (cm^3)	875
Stroke (mm)	86
Bore (mm)	80.5
Compression Ratio	10:1
Number of valves	4 per cylinder

Max Torque (Nm)	145 @ 2000 rpm
Max Power (kW)	77 @ 5500 rpm

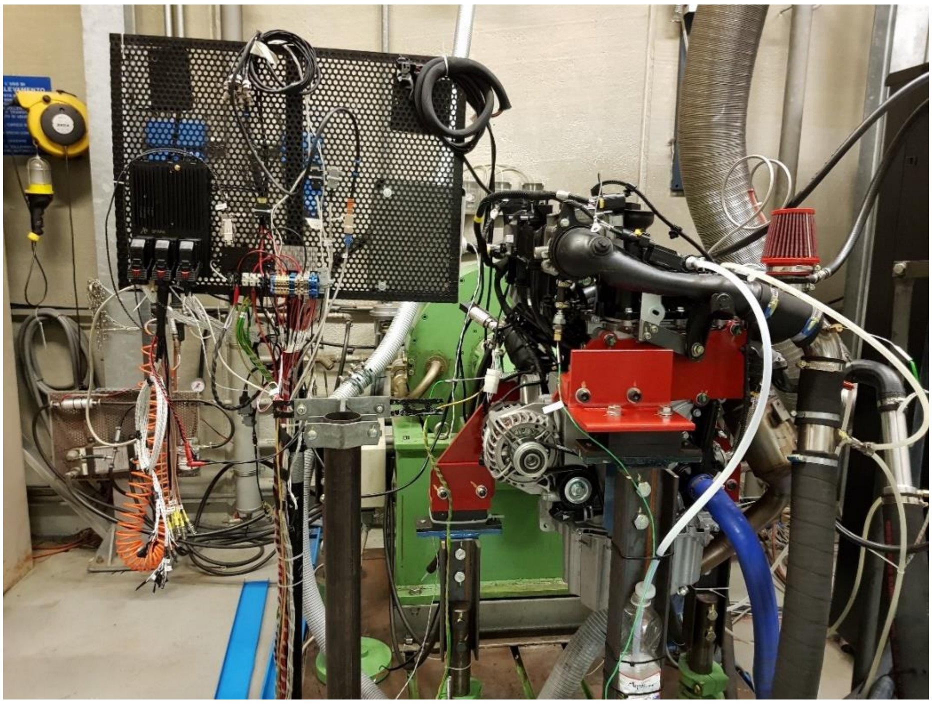


Figure 7: TwinAir engine installed on test bench 1 in Forlì.

2.0 GME	
Strokes per Cycle	4
Geometry	4 cylinders in line, firing order 1-3-4-2
Intake System	Turbocharged with waste-gate valve and equipped with MultiAir system on the intake valves
Injection System	Gasoline DI, high pressure pump
Displacement (cm^3)	1993
Stroke (mm)	90
Bore (mm)	84

Compression Ratio	10:1
Number of valves	4 per cylinder
Max Torque (Nm)	415 @ 2000 to 4800 rpm
Max Power (kW)	209 @ 5200 rpm

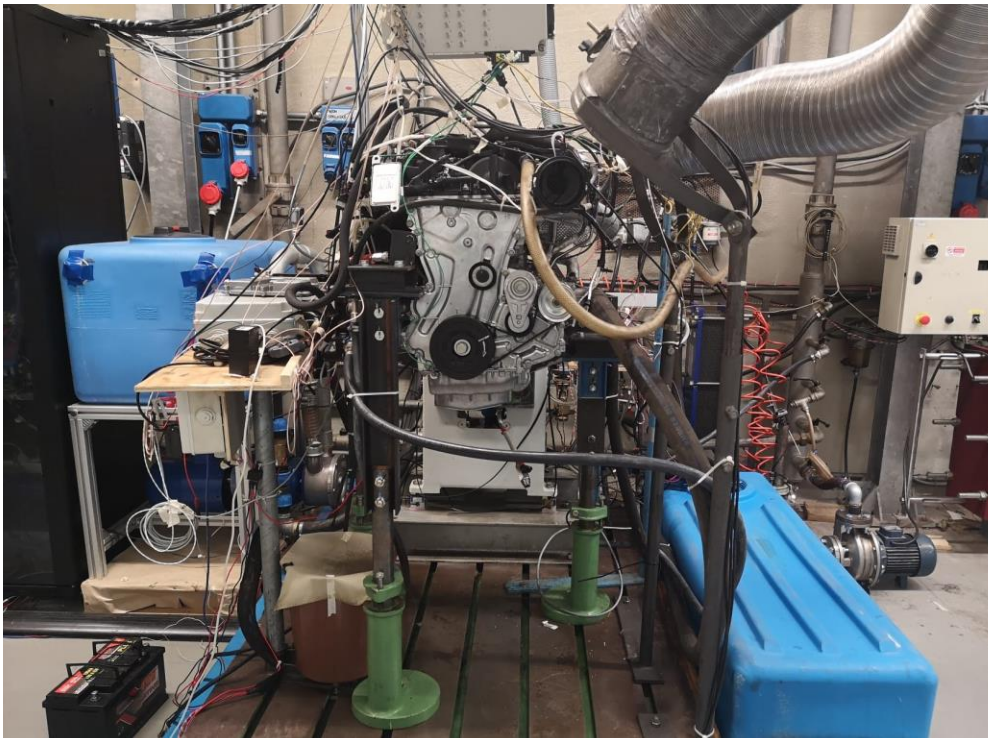


Figure 8: GME engine installed on the test bench 1

For the advanced combustion control research, the FCA four-cylinder 1.3 litre MultiJet engine has been over the years adapted to several configuration.

1.3 MultiJet	
Strokes per Cycle	4
Geometry	4 cylinders in line, firing order 1-3-4-2
Intake system	Turbocharged with variable geometry turbine, HP, liquid cooled EGR
Injection system	Common Rail, MultiJet

Displacement (cm3)	1248
Stroke (mm)	82
Bore (mm)	69.6
Compression Ratio	16.8:1
Number of valves	4 per cylinder
Maximum Torque (Nm)	200 @ 1500 rpm
Maximum Power (kW)	70 @ 3800 rpm

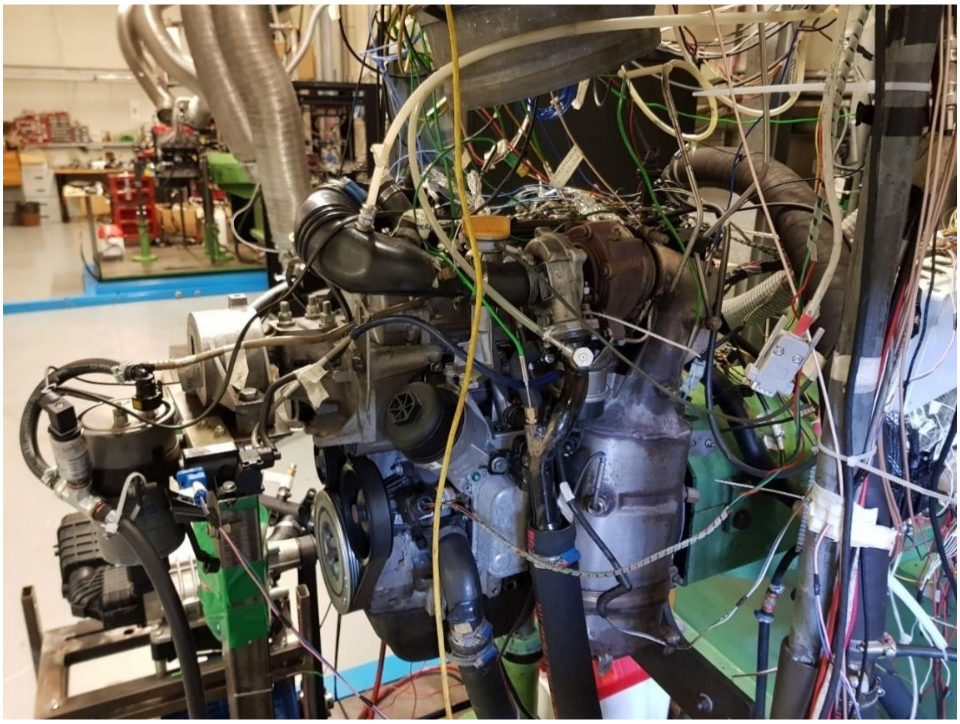
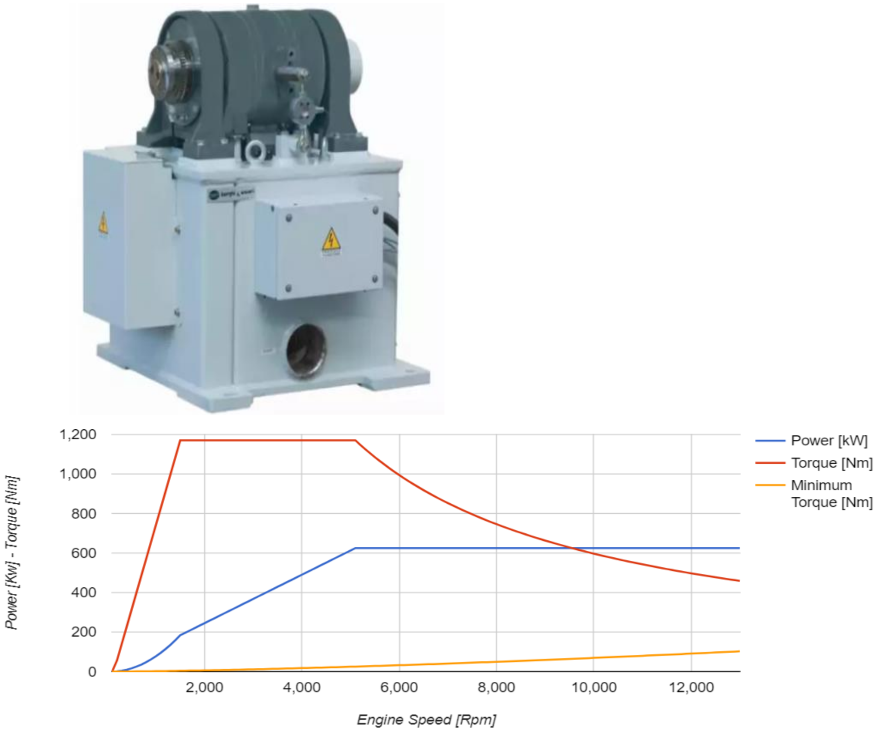


Figure 9: 1.3 MultiJet installed on bench 3 in Forlì.

On the test bench side, apart from traditional eddy current brakes (Borghi&Saveri FE350S) installed in all the three benches in the laboratory, has been controlled with a dedicated RCP (Hydraulic Brake Control4.4) the hydraulic dynamometer from Borghi&Saveri FE23.



Max Speed [rpm]	13000
Max Power [kW]	625.0
Max Power [CV]	850
Max Torque [Nm]	1170
Inertia [kg m2]	0.0900

Figure 10: Hydraulic Brake Borghi&Saveri F23

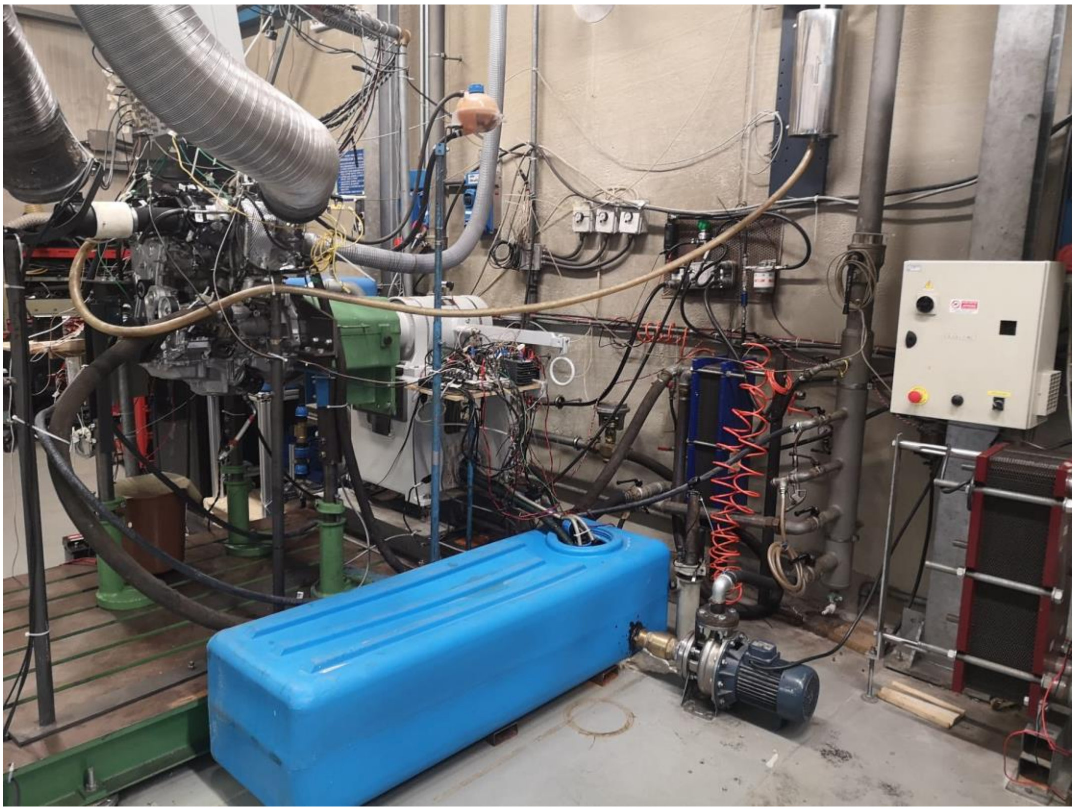


Figure 11: GME installed on bench 1 together with the Hydraulic-Brake FE23 and the RCP control system.

2. Real-Time / FPGA Development Platforms and Environments

In the landscape described in chapter 1, a FPGA/Real-Time architecture offers software flexibility and small dimensions, suggesting using them naturally as Rapid Control Prototyping for control applications or as a smart data logger and analyser. The FPGA grants true hardware parallel low-level and high-speed calculation and timing, while the Real-Time processor can handle the high-level calculation, the logging and the communication functions with determinism.

With a proper Real-Time/FPGA RCP verification, software development and field testing can be done immediately after design or even in parallel [1] [2] [23]. Previous case studies demonstrate the capability of this approach towards applications such as RCCI combustion control [7] [8], multi-spark ignition [9], water injection [24], combustion phase control and optimization [10] [11] and complete engine management fast prototyping [25].

A typical application can be displaced on three levels (with RT and FPGA typically gathered in a single device) that resembles three different objectives in terms of timing and functions complexity.

FPGA

- Hardware implementation of the code: true determinism
- Time domain of the tens of MHz for a single loop iteration
- Low-level algorithms: code complexity limited by the hardware implementation constraints
- Physically limited resources
- High compilation time high, due to hardware routing and placing
- Very limited debugging features
- Typical tasks:
  - Signals acquisition, filtering and low-level / high-speed processing (ex. Phasing algorithms)
  - Signals generation (PWM, phased actuation commands etc)
  - Provides most of the I/O for the whole controller.

Real-Time

- Processor that runs a Real-Time operating system: time critical tasks management, low jitter.
- Time domain of Hz / kHz for a single loop iteration
- High-level control algorithms: all the time critical control tasks that need to run in time.
- Computational power dependant on the CPU unit: from embedded controllers compact and rugged, but with limited resources, to high end machines with multicore CPUs .
- Typical tasks:
  - Custom control algorithms (engine control for example)
  - Read signals acquired form FPGA
  - Calculation and writing to FPGA of signals generation parameters
  - Communication: CAN, UDP/TCP transmission
  - Calibration and measurement slave
  - On-board data logging

Host PC

- Machine that runs a general-purpose operating system: low level of determinism, high jitter
- Independent device from the embedded RT / FPGA controller
- Time domain of tens of Hz for a single loop iteration
- High-level processing algorithms: tasks that do not require a high level of determinism
- Computational power generally not critical: high-end general-purpose machines with multicore CPUs.
- Typical tasks:
  - Calibration and measurement master
  - Reception, processing, visualization and logging of data sent from RT/FPGA target
  - Post processing algorithms

All these three “layers” can be managed with different toolchains and programming languages depending on the device compatibility and the objectives of the application to be realized. In sections 2.3 and 2.4 a couple of different development environment solutions used in the research have been briefly described.

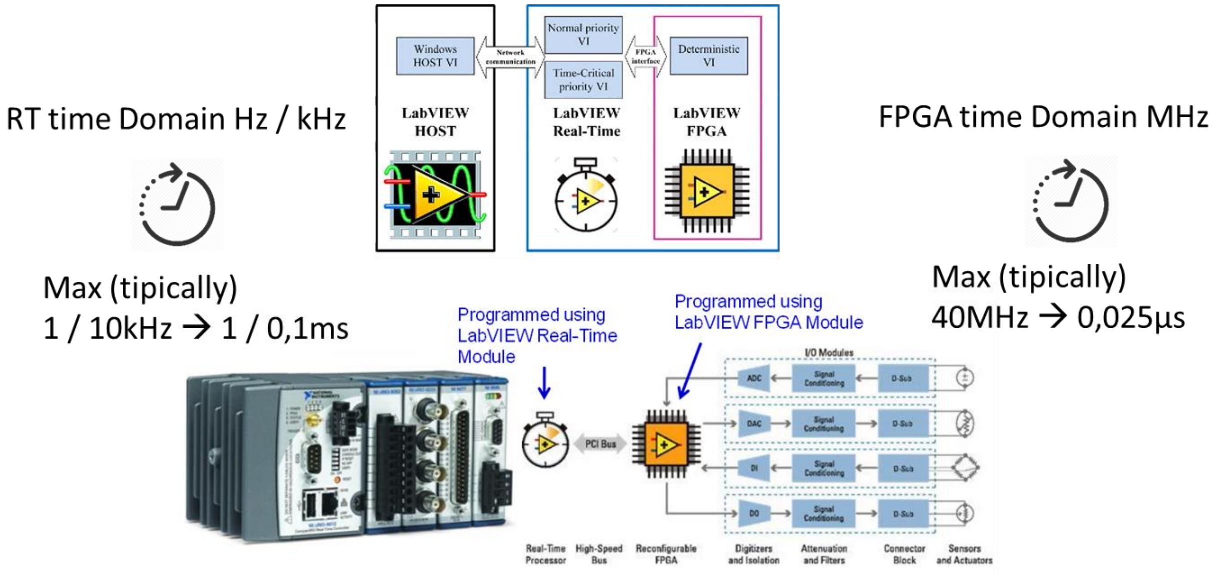


Figure 12: A typical NI cRIO system based on Real-Time / FPGA architecture.

2.1 Real-Time system

To be considered **"real-time"**, an operating system must have a known maximum time for each of the critical operations that it performs (or at least be able to guarantee that maximum most of the time) [1].

Determinism: timing can be guaranteed within a certain margin of error.

Jitter: The amount of error in the timing of a task over subsequent iterations of a program or loop. Real-time operating systems are capable to provide a low amount of jitter: a task will take very close to the same amount of time for each iteration.

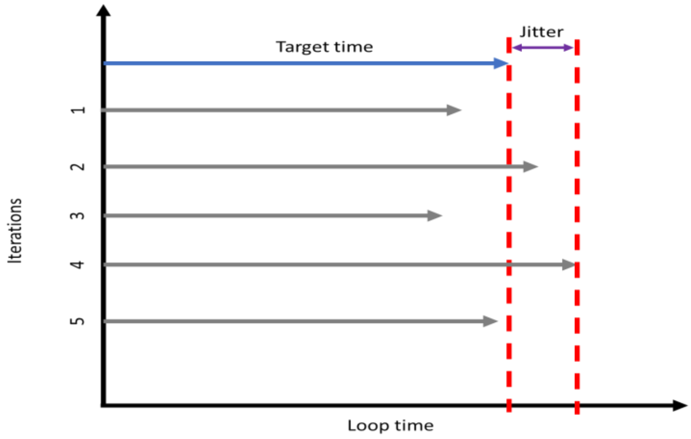


Figure 13: Loop Time jitter rapresentation

The main point of an RTOS, if programmed correctly, is the very consistent timing that can guarantee to a running program. Developers are provided with a set of functionalities that allows control over how tasks are prioritized and the possibility of timing monitoring.

Most operating systems of any type allow the programmer to specify the priority for an application or the tasks within it, and these priorities are used by the system to determine which tasks must be run in comparison to concurrent tasks. In contrast to general-purpose OSs that make sure that all tasks receive at least some CPU time, an RT OS is much stricter and if a high-priority task is using the processor no other low-priority task will run until the high priority has finished. Also interrupt response time is bounded to maximum value and this assures critical event latency is taken under control. A careful definition of time critical and not code sections is required for an application to work properly.

Most popular general-purpose operating systems, for example for personal computer use, are not designed to run critical applications with a precise and reliable timing. This kind of operating systems are more intended to assure responsiveness of the user interface and a fair resource sharing between multiple programs and services running in parallel.

Real-Time operating systems may or may not increase the speed of execution, they can provide much more precise and predictable timing characteristics than general-purpose operating systems.

In a typical RCP system, the **Real-Time processor** handles the control models with all its subsystems, the actuations management, the sensors reading and the high-level functions such as the connectivity. Being real-time assures the control task to be executed at the expected rate. Some operations must be accomplished in time for the next cycle: an interrupt for the calculation can be sent by FPGA.

2.2 **FPGA**

Field Programmable Gate Arrays are integrated circuits composed by several reprogrammable logic blocks. The adoption of an FPGA chip is driven by their flexibility, hardware-time speed, reliability and parallelism [1]. They incapsulate the hardware speed typical of application specific integrated circuits (ASICs) gaining the advantage from the capability of being programmable. The higher cost but consistently lower development time make them ideal for custom application, small production series or rapid prototyping.

Since for an FPGA code is represented on hardware, the processing operations are truly parallel and different sections of code, accomplishing different tasks, are physically independent being assigned to dedicated locations on the chip. As a result, after compilation, the behaviour is deterministically fixed and two independent parts of the application cannot affect each other performance.

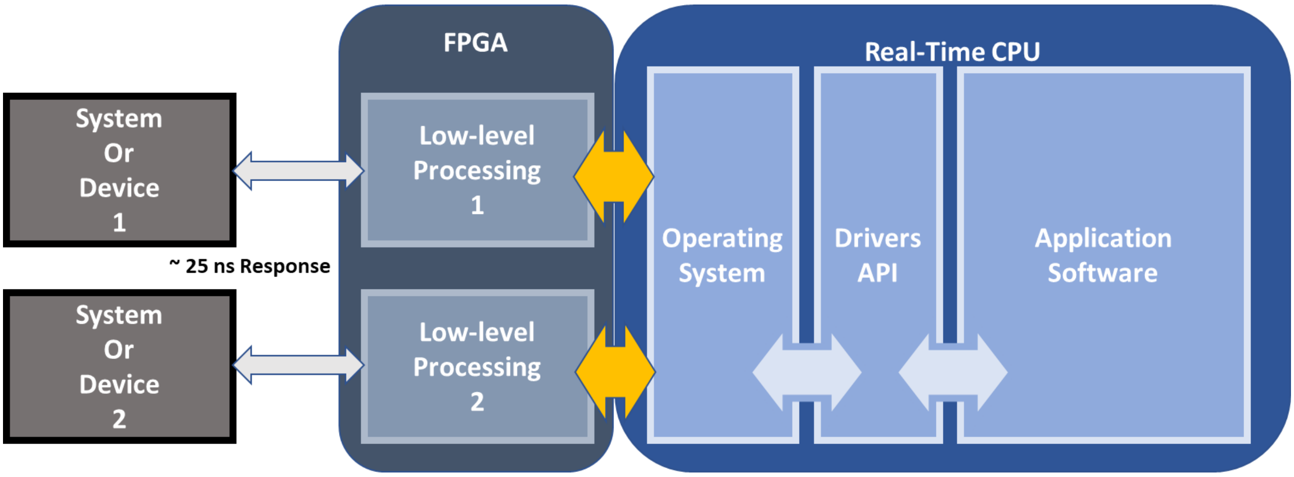


Figure 14: Real-Time to FPGA independent processing example scheme

An FPGA chip is made up of a finite number of predefined resources and programmable interconnection to allow the blocks being configured to execute logic operations and access I/O resources. The main types of resources are configurable logic blocks (CLBs), fixed function logic blocks (ex. multipliers) and embedded block RAM. **CLBs** (also called slices or logic cells) are the fundamental unit of the FPGA and it is composed of **Flip-Flops** and **Lookup Tables**: different hardware families may have a different way of packaging them together. Understanding the resource type and number is important for comparing FPGA for a specific application.

Flip-Flops are binary shift registers used to synchronize logic and retain logical states between clock cycles.

Lookup Tables are memory truth tables used to represent combinatorial logic (AND, OR, NAND, XOR etc.). Each output value of the LUT is the predetermined result for a specific set of inputs.

Multipliers and DSP Slices: in order to reduce the amount of flip-flops and LUT to achieve “simple” operations such as multiplications and other math / signal processing application, FPGAs have pre-built circuitry dedicated to this operations.

Block RAM: Memory resources are a key specification to store data sets or pass values between parallel tasks without consuming precious logic such as flip-flops and LUTs. Especially referring to "large" data as arrays, embedded block RAM is much more effective doing the job since the memory capability of flip-flops is generally orders of magnitude lower and intended for logic implementation.

Math operations in the FPGA take up slices and some operations are very expensive in terms of FPGA space: divisions take quite a bit more room than multiplications. As a result, implemented equations should be optimized for FPGA use and pre-calculated values must be used to minimize computation space and eliminate non-supported operations [11] [26].

In a typical RCP system, the **FPGA** accomplishes the lower-level operations such as phasing, actuations timing and signals acquiring, exploiting the parallel high-speed calculation and timing potential.

2.3 NI LabVIEW-Based Platforms

National Instruments provides hardware and software fully integrated in a development environment (LabVIEW) dedicated to engineering projects with acquisition, testing and control purpose [1]. The toolchain provides support and tools for fast graphical programming and testing workflow that exploits embedded Real-Time/FPGA as well as desktop computer machines. Since the research topic has been on Real-Time / FPGA architectures, attention has been paid to the range of different solution proposed for application development. The I/O possibilities ranges from analog and digital input / output, industrial communication to custom I/O such as specific injector driver modules.

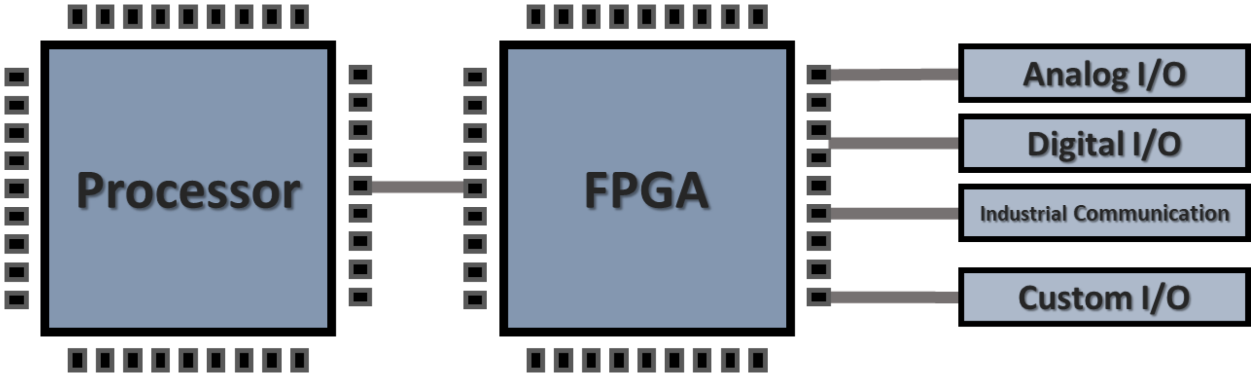


Figure 15: Standard NI Real-Time / FPGA and I/O acrhitecture

CompactRIO platform is a rugged compact Real-Time / FPGA controller that has the possibility of being expanded in the I/O capabilities with a wide range of I/O modules including injector drivers and CAN communication interfaces. The robust design is suitable also for hard-environment tests and measurements.

Figure 16: NI Compact RIO (cRIO) platform and C series modules

The PXI platform offers a similar modular approach, but more oriented to high performance laboratory applications such as hardware in the loop systems or testing and acquisition systems.

Figure 17: NI PXI platform and modules

Single boards and systems on module are also provided, with the same RT/FPGA architecture, for custom hardware realization exploiting the full compatibility with the LabVIEW toolchain.

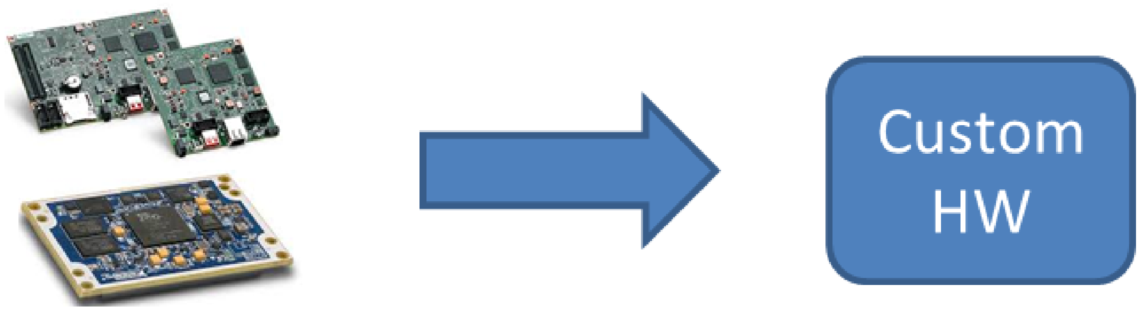
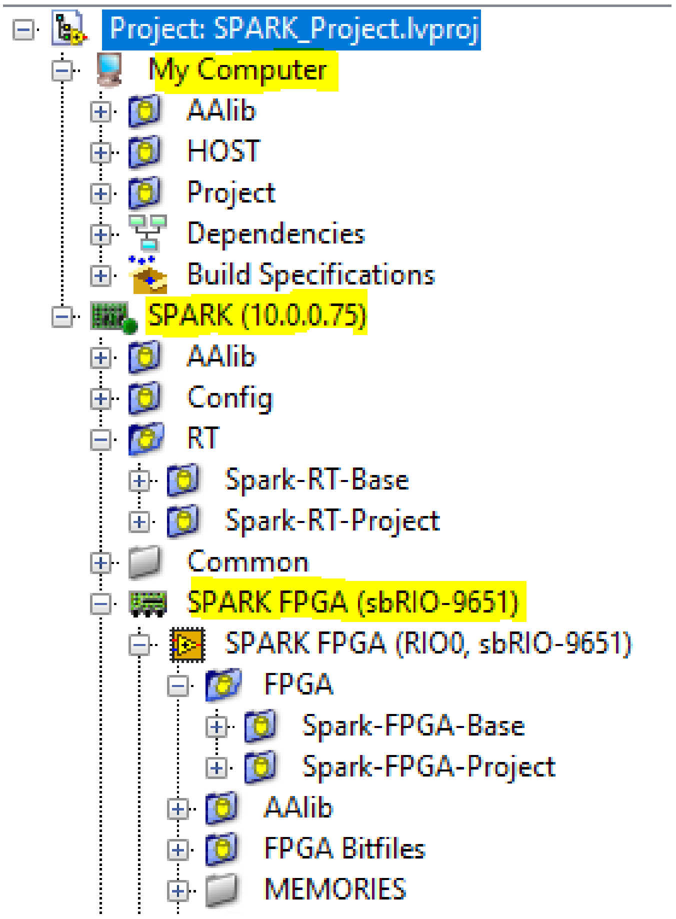


Figure 18: NI sbRIO / SoM platform allows the building of custom HW connected to the core module

Below in Figure 19 an example of LabVIEW development project is presented. Host PC, Real-Time and FPGA programming levels are included natively and the toolchain provides natively development, compilation and debugging tools.



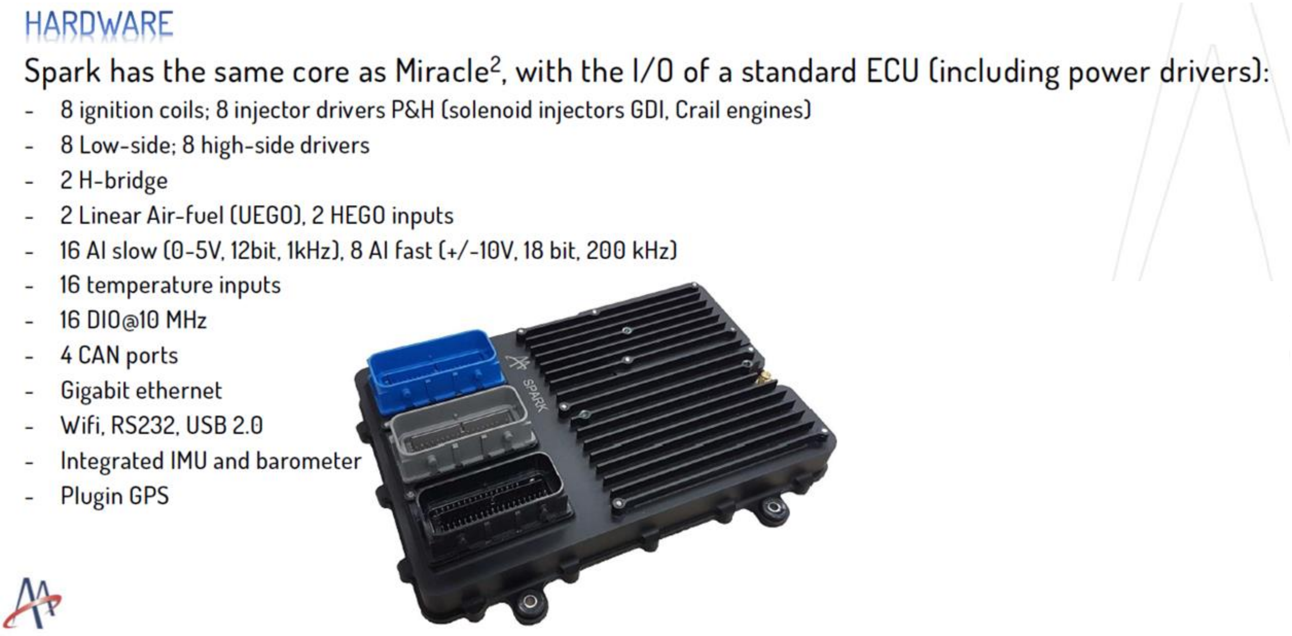
- Project: SPARK_Project.lvproj
  - My Computer
    - AAlib
    - HOST
    - Project
    - Dependencies
    - Build Specifications
  - SPARK (10.0.0.75)
    - AAlib
    - Config
    - RT
      - Spark-RT-Base
      - Spark-RT-Project
    - Common
    - SPARK FPGA (sbRIO-9651)
      - SPARK FPGA (RIO0, sbRIO-9651)
        - FPGA
          - Spark-FPGA-Base
          - Spark-FPGA-Project
        - AAlib
        - FPGA Bitfiles
        - MEMORIES

Figure 19: LabView project for SoM based platform.

Spark Open ECU

The **main development board used** has been the Spark ECU from Alma-Automotive whose core is a National Instruments System On Module (SOM) sbRIO-9651, powered by a Dual Core ARM Real-Time Processor, an Artix-7 FPGA and 512MB RAM. The ECU makes available a wide range of inputs/outputs: 8 coil drivers, 8 injectors drivers (peak and hold current profile), 8 low-side and 8 high-side (PWM actuations), 2 H-Bridge for electric servo-motors, 2 wide band lambda controllers, 24 ADC channels (8 at 200kHz, and 16 at 1 kHz), 16 temperature sensor pins and 16 GPIO.

Because of NI environment potential and a great amount of experience within the research group, Spark has been the main development ECU for most of the research activity. The toolchain chosen for software writing has been LabVIEW, gaining advantage from hardware complete compatibility and from previous experiences on the platform such as Unibo FSAE ECU [27].



HARDWARE

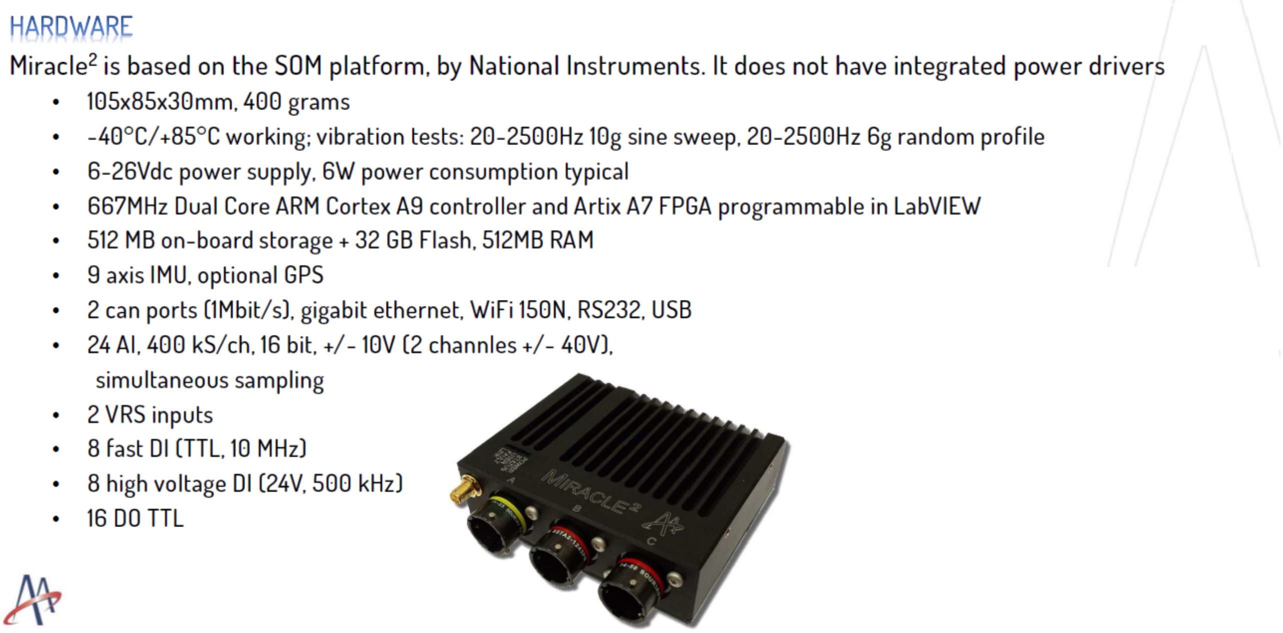
Spark has the same core as Miracle^2, with the I/O of a standard ECU (including power drivers):

- 8 ignition coils; 8 injector drivers P&H (solenoid injectors GDI, Crail engines)
- 8 Low-side; 8 high-side drivers
- 2 H-bridge
- 2 Linear Air-fuel (UEGO), 2 HEGO inputs
- 16 AI slow (0-5V, 12bit, 1kHz), 8 AI fast (+/-10V, 18 bit, 200 kHz)
- 16 temperature inputs
- 16 DIO@10 MHz
- 4 CAN ports
- Gigabit ethernet
- Wifi, RS232, USB 2.0
- Integrated IMU and barometer
- Plugin GPS

Figure 20: Spark Datasheet

Miracle2 RCP Platform

For specific applications that did not require power drivers (4.4 Hydraulic Brake Control) Miracle2 from Alma-Automotive has been used. This small ECU is based on the same SoM platform as Spark, so all the software structure designed for RCPs on Spark has been ready-to use just with a small amount of low-level drivers adaption. The extended amount of high speed analog and digital I/O and the really small form-factor make this target suitable for smart data logging and control RCPs also for on-board applications or where a clean installation is required.



HARDWARE

Miracle^2 is based on the SOM platform, by National Instruments. It does not have integrated power drivers

- 105x85x30mm, 400 grams
- -40°C/+85°C working; vibration tests: 20-2500Hz 10g sine sweep, 20-2500Hz 6g random profile
- 6-26Vdc power supply, 6W power consumption typical
- 667MHz Dual Core ARM Cortex A9 controller and Artix A7 FPGA programmable in LabVIEW
- 512 MB on-board storage + 32 GB Flash, 512MB RAM
- 9 axis IMU, optional GPS
- 2 can ports (1Mbit/s), gigabit ethernet, WiFi 150N, RS232, USB
- 24 AI, 400 kS/ch, 16 bit, +/- 10V (2 channles +/- 40V), simultaneous sampling
- 2 VRS inputs
- 8 fast DI (TTL, 10 MHz)
- 8 high voltage DI (24V, 500 kHz)
- 16 DO TTL

Figure 21: Miracle2 Datasheet

2.4 MATWORKS SIMULINK-BASED PLATFORMS

It is possible to exploit the powerful development environment of Matlab and Simulink on native hardware from Speedgoat [2]: drivers blocksets allow the programmer to develop and compile code directly into the target form Simulink. Different Real-Time machines are proposed, with different computational power and expandability possibilities. The architecture is based on a desktop PC processor, running an RT operating system, and expansion boards for I/O and connectivity. FPGA chassis can be added to the machines in the form of programmable (possibility of writing FPGA code) or reconfigurable devices (possibility of choosing pre-programmed functions). As for NI platforms the I/O possibilities ranges from analog and digital input / output, industrial communication, to fault insertion units and so on.

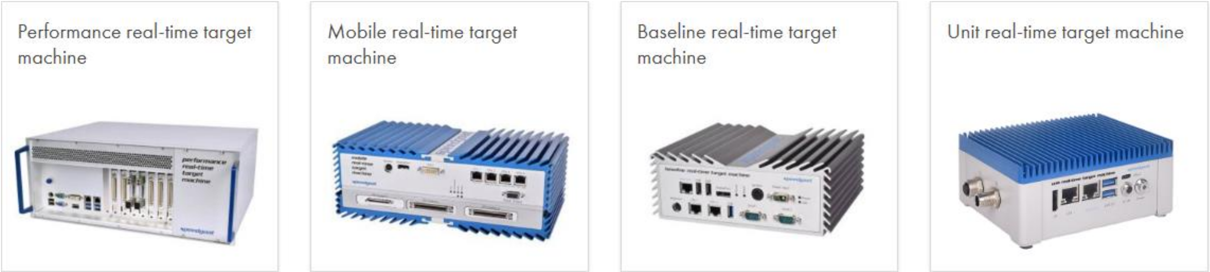

Performance real-time target machine
Mobile real-time target machine
Baseline real-time target machine
Unit real-time target machine

Figure 22: Speedgoat Simulink Real-Time native hardware

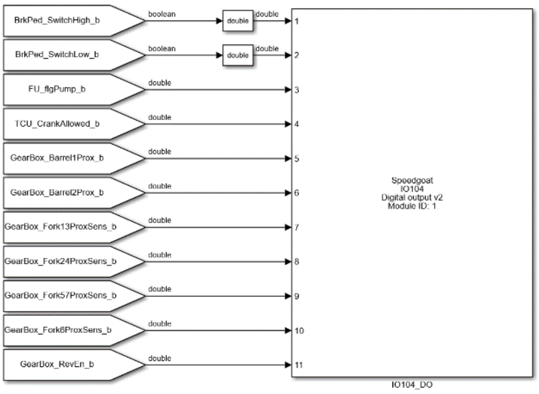


Figure 23: Simulink drivers blockset for Speedgoat targets.

The native integration with Simulink and the high computational power make these platforms a good choice for a fast HIL implementation. Simulink model-based approach has been used in combination with a performance real-time target machine to face a hybrid powertrain HIL implementation during the research activity.

3. Base Software Architecture and Functionalities

Since the main aim of the development was building an FPGA/RT structure reusable for different applications sharing common functionalities (reading RPMs, acquiring a sensor or driving an injector are common requirements), they have to be prebuilt features with simple set up.

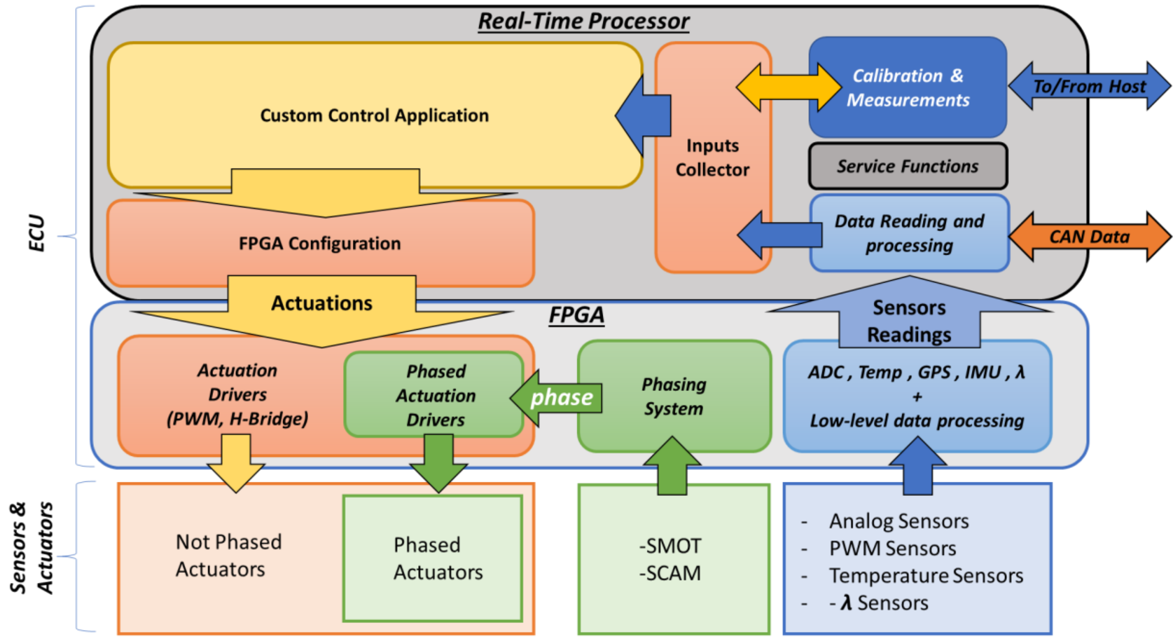


Figure 24: Base Software Architecture with the main functions. Custom control code can be easily hosted.

A time-triggered software architecture has been implemented granting the efficiency and the flexibility on the application specific part of code. Many features implementation and optimizations have been improved during the research, making the configuration flexible and improving the resources usage of the RT and the FPGA.

The software is organized and developed in modules, each one with its own set of variables with a common nomenclature structure. The most common functions have been developed as basic VIs sets with fixed interfaces, in order to be compatible with custom code and make the ECUs I/O quickly accessible both from RT and FPGA. Referring to the Spark platform, below is presented a synthetic list of the standard features implemented by new or exploiting and integrating pre-existent libraries.

These base modules are exportable and compatible on every NI based platform, except for the adaption to the hardware platform of the lower level I/O implementation: a generic cRIO, M2 or Spark have different I/O and different implementations of the interaction with the external world.

Real-Time & FPGA base main features:

Configurable Acquisition Drivers:

- ADC (moving average on angular window, etc…)
- Temperatures
- PWM read

GPIO drivers:

- PWM/Frequency read
- PWM generation
- Crank and Cam signals reading
- Wheel-Speed sensors reading
- GPO write from RT
- GPI read from RT

Configurable Actuators Drivers:

- PWM generation (LS and HS)
- H-Bridge
- Analog Out

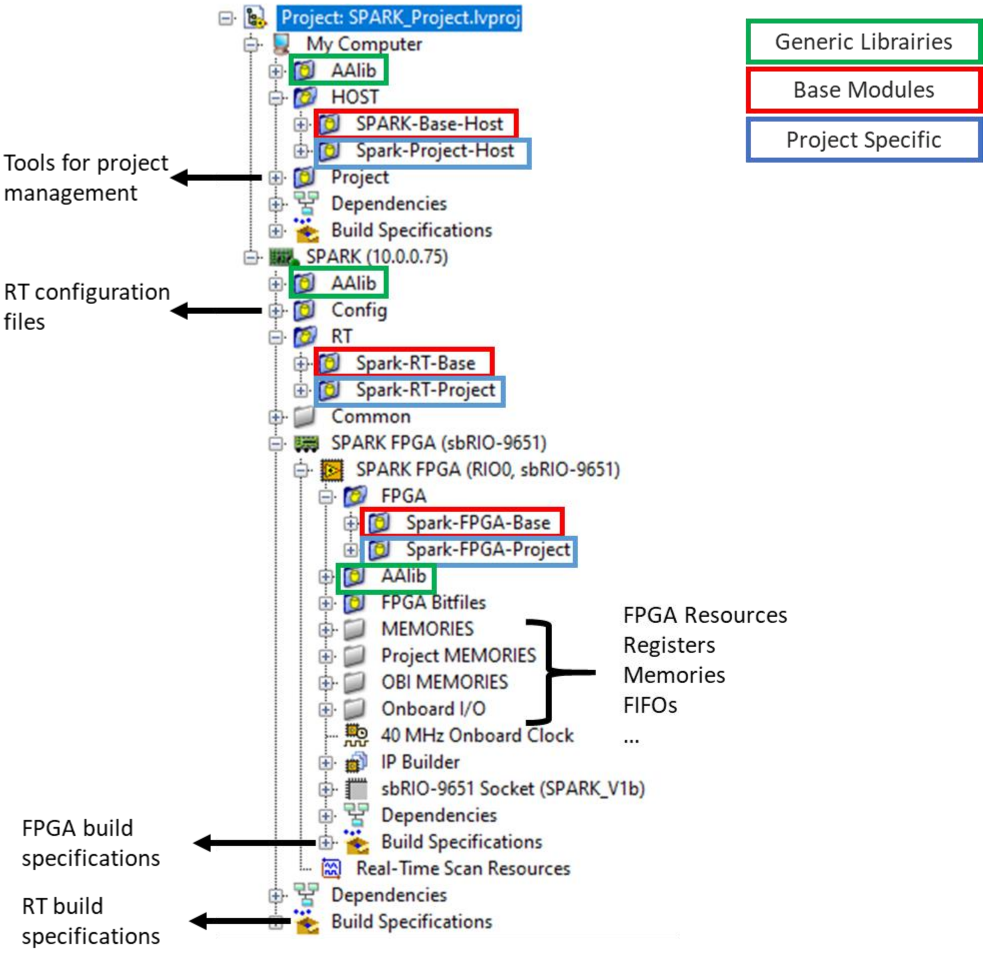
Phased Actuators Drivers: angle to angle or angle and time instructions.

- Peak&Hold Injectors Drivers (single and multi)
- GPO Injectors Drivers (single and multi)
- Ignition Drivers (single and multi)
- Ignition GPO Drivers (single and multi)

Phasing System (teeth identification)
IRQs (calculation trigger, configurable)
Angle Extrapolation (for simple angle referenced actuations)
VVT control
MultiAir control
Throttle control
XCP Communication (10, 50, 500, sync rasters)
CAN Communication configurable form database
PID, LUT, Sensors Linearizations, ecc. standard libraries
Base SW modules configuration and loops
Data Recording
Events Logging

3.1 Project structure

First of all, the variables and the project structure are split into **base part and custom part**. This is done for all the three levels of LabVIEW code Host, Real-Time and FPGA.



```
Project: SPARK_Project.lvproj
  My Computer
    AAlib
    HOST
      SPARK-Base-Host
      Spark-Project-Host
    Project
    Dependencies
    Build Specifications
  SPARK (10.0.0.75)
    AAlib
    Config
    RT
      Spark-RT-Base
      Spark-RT-Project
    Common
    SPARK FPGA (sbRIO-9651)
      SPARK FPGA (RIO0, sbRIO-9651)
        FPGA
          Spark-FPGA-Base
          Spark-FPGA-Project
        AAlib
        FPGA Bitfiles
        MEMORIES
        Project MEMORIES
        OBI MEMORIES
        Onboard I/O
        40 MHz Onboard Clock
        IP Builder
        sbRIO-9651 Socket (SPARK_V1b)
        Dependencies
        Build Specifications
      Real-Time Scan Resources
    Dependencies
    Build Specifications
```

Figure 25: LabVIEW project architecture overview

Additionally, the software structure includes some tools for the project management, files for the real-time configuration, the FPGA resources definition (registers, memories, FIFOs), the specifications for the building RT and FPGA code into stand-alone executables and generic libraries not linked to a particular hardware environment.

BASE:

Contains the base modules and libraries related to ECU specific functions.

- All the SW modules necessary for ECU base functionalities or services
- All the standard and already developed SW modules

PROJECT:

Contains the project specific contents, including the “main” application.

- All the SW modules developed or modified for the specific application
- All the VIs / typedefs that need to be linked directly to the specific application

Generic Libraries:

Contains generic libraries or frequently used functions, not strictly related to the ECU and independent from the hardware environment of programming: PID lib, LUT lib, Boolean logic, time and date functions etc.

The idea behind this architecture is to decouple the reusable parts of the project from the specific ones, allowing to develop specific SW personalities without having to set up from scratch the whole project every time, and gaining advantage of the possibility of updates of old projects.

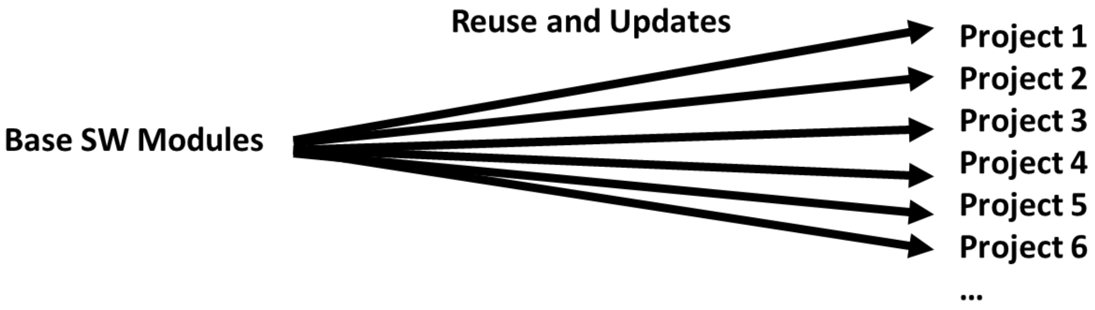
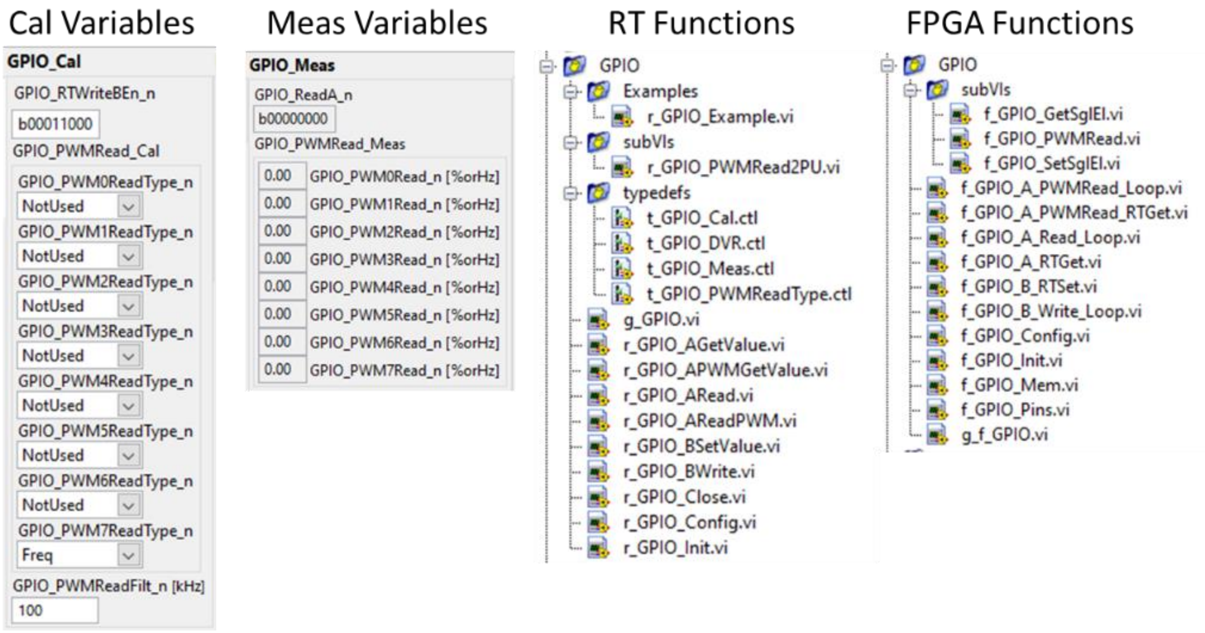


Figure 26: Base SW modules reuse between multiple projects

Another important aspect of the project is the coherency of the nomenclature of variables and VIs. This simple aspect allows a better modularization and a better user experience not only form the aesthetic point of view: variables and functions can be recalled or searched easily by name within the development environment or the calibration software. System, subsystem, physical type, description and data type are as standardized as possible. Functions and variables description and comments also helps keeping the software integrity.

<table>
<tr><th colspan="7">VI Name</th></tr>
<tr><td>Hardware Enviroment</td><td>_</td><td colspan="2">System_Subsystem</td><td>_</td><td colspan="2">VI Name</td></tr>
<tr><td>Host</td><td>h</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Real-Time</td><td>r</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>FPGA</td><td>f</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><th colspan="7">Variables Name</th></tr>
<tr><td>SubSystem</td><td>_</td><td colspan="2">VariableName</td><td>_</td><td>Data type</td><td>[unit_Xaxis_Yaxis_]</td></tr>
<tr><td></td><td></td><td>physicalType</td><td>VariableDescription</td><td></td><td></td><td></td></tr>
</table>

Table 1: Naming convention for functions and variables



Cal Variables

GPIO_Cal

GPIO_RTWriteBEn_n: b00011000

GPIO_PWMRead_Cal

- GPIO_PWM0ReadType_n: NotUsed
- GPIO_PWM1ReadType_n: NotUsed
- GPIO_PWM2ReadType_n: NotUsed
- GPIO_PWM3ReadType_n: NotUsed
- GPIO_PWM4ReadType_n: NotUsed
- GPIO_PWM5ReadType_n: NotUsed
- GPIO_PWM6ReadType_n: NotUsed
- GPIO_PWM7ReadType_n: Freq

GPIO_PWMReadFilt_n [kHz]: 100

Meas Variables

GPIO_Meas

GPIO_ReadA_n: b00000000

GPIO_PWMRead_Meas

- 0.00 GPIO_PWM0Read_n [%orHz]
- 0.00 GPIO_PWM1Read_n [%orHz]
- 0.00 GPIO_PWM2Read_n [%orHz]
- 0.00 GPIO_PWM3Read_n [%orHz]
- 0.00 GPIO_PWM4Read_n [%orHz]
- 0.00 GPIO_PWM5Read_n [%orHz]
- 0.00 GPIO_PWM6Read_n [%orHz]
- 0.00 GPIO_PWM7Read_n [%orHz]

RT Functions

```
GPIO
  Examples
    r_GPIO_Example.vi
  subVIs
    r_GPIO_PWMRead2PU.vi
  typedefs
    t_GPIO_Cal.ctl
    t_GPIO_DVR.ctl
    t_GPIO_Meas.ctl
    t_GPIO_PWMReadType.ctl
  g_GPIO.vi
  r_GPIO_AGetValue.vi
  r_GPIO_APWMGetValue.vi
  r_GPIO_ARead.vi
  r_GPIO_AReadPWM.vi
  r_GPIO_BSetValue.vi
  r_GPIO_BWrite.vi
  r_GPIO_Close.vi
  r_GPIO_Config.vi
  r_GPIO_Init.vi
```

FPGA Functions

```
GPIO
  subVIs
    f_GPIO_GetSglEl.vi
    f_GPIO_PWMRead.vi
    f_GPIO_SetSglEl.vi
  f_GPIO_A_PWMRead_Loop.vi
  f_GPIO_A_PWMRead_RTGet.vi
  f_GPIO_A_Read_Loop.vi
  f_GPIO_A_RTGet.vi
  f_GPIO_B_RTSet.vi
  f_GPIO_B_Write_Loop.vi
  f_GPIO_Config.vi
  f_GPIO_Init.vi
  f_GPIO_Mem.vi
  f_GPIO_Pins.vi
  g_f_GPIO.vi
```

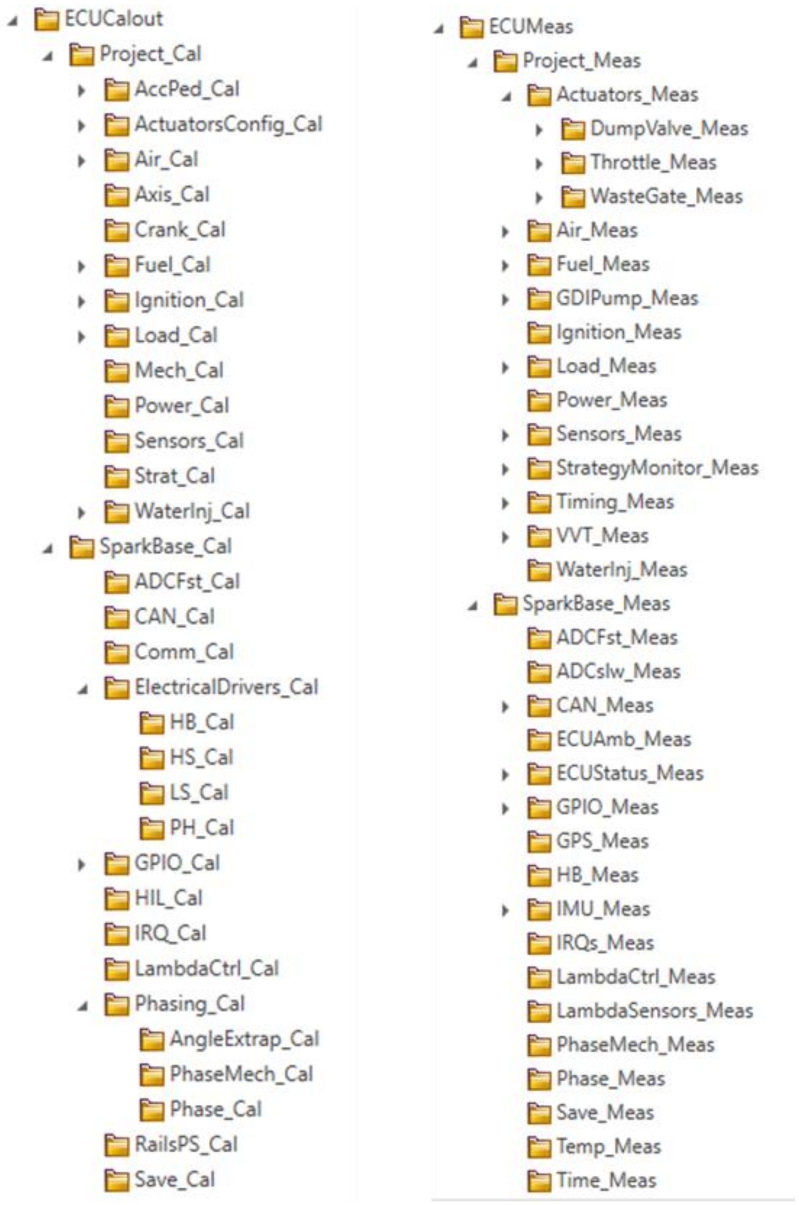
Figure 27: Example of the GPIO software module in all its parts, following the naming coherence and modularity

3.1.1 ***ECU Calibration and Measurement Variables***

Real-Time variables are managed with two main collection of data called "**Calibration**", with all the configuration parameters, maps and axis, and "**Measurement**", which contains all the process variables that the software updates and uses while running. In both of them, custom code variables have been decoupled from the base ones, in order to enhance modularization: every application has a common configuration part that sets up phasing system, acquisition and actuations.

The two data collection are grouped in the software by means of two "clusters" (LabVIEW data type), additionally divided in the two main sub-clusters that collect the base and the project variables. Going deeper through the levels, nested sub-clusters define a hierarchy grouping the variables per modules and sub-modules of functionality in the software.

The main sub-clusters, that collects the variables (Cal and Meas) of a specific module are generally defined in the project as "typedef" (LabVIEW), locking in an independent file the definition. In this way each module has its own set of variables, that can be included dragging and dropping the variables in a new project application. For example, the phasing module has its own set of calibration parameters that defines the algorithm implementation and its own set of measurement variables that are the outputs (speed, cycles, phasing condition etc.).



- ECUCalout
  - Project_Cal
    - AccPed_Cal
    - ActuatorsConfig_Cal
    - Air_Cal
    - Axis_Cal
    - Crank_Cal
    - Fuel_Cal
    - Ignition_Cal
    - Load_Cal
    - Mech_Cal
    - Power_Cal
    - Sensors_Cal
    - Strat_Cal
    - WaterInj_Cal
  - SparkBase_Cal
    - ADCFst_Cal
    - CAN_Cal
    - Comm_Cal
    - ElectricalDrivers_Cal
      - HB_Cal
      - HS_Cal
      - LS_Cal
      - PH_Cal
    - GPIO_Cal
    - HIL_Cal
    - IRQ_Cal
    - LambdaCtrl_Cal
    - Phasing_Cal
      - AngleExtrap_Cal
      - PhaseMech_Cal
      - Phase_Cal
    - RailsPS_Cal
    - Save_Cal

- ECUMeas
  - Project_Meas
    - Actuators_Meas
      - DumpValve_Meas
      - Throttle_Meas
      - WasteGate_Meas
    - Air_Meas
    - Fuel_Meas
    - GDIPump_Meas
    - Ignition_Meas
    - Load_Meas
    - Power_Meas
    - Sensors_Meas
    - StrategyMonitor_Meas
    - Timing_Meas
    - VVT_Meas
    - WaterInj_Meas
  - SparkBase_Meas
    - ADCFst_Meas
    - ADCslw_Meas
    - CAN_Meas
    - ECUAmb_Meas
    - ECUStatus_Meas
    - GPIO_Meas
    - GPS_Meas
    - HB_Meas
    - IMU_Meas
    - IRQs_Meas
    - LambdaCtrl_Meas
    - LambdaSensors_Meas
    - PhaseMech_Meas
    - Phase_Meas
    - Save_Meas
    - Temp_Meas
    - Time_Meas

Figure 28: ECU Cal and ECU Meas grouping example

Calibration variables are written by the calibration manager (XCP protocol or directly from LabVIEW) while are read all over the software to be used as inputs for the application functions.

Measurement variables are read by the calibration manager (XCP protocol or directly from LabVIEW) while can be read all over the code as inputs or written as outputs of the functions. Since the measurement variables are globally accessible all over the software, to avoid race conditions and maintain control over the code, it is a good practice to update the value of a measurement just in a single place in the code, or to be sure that the variable is not written concurrently form two different places.

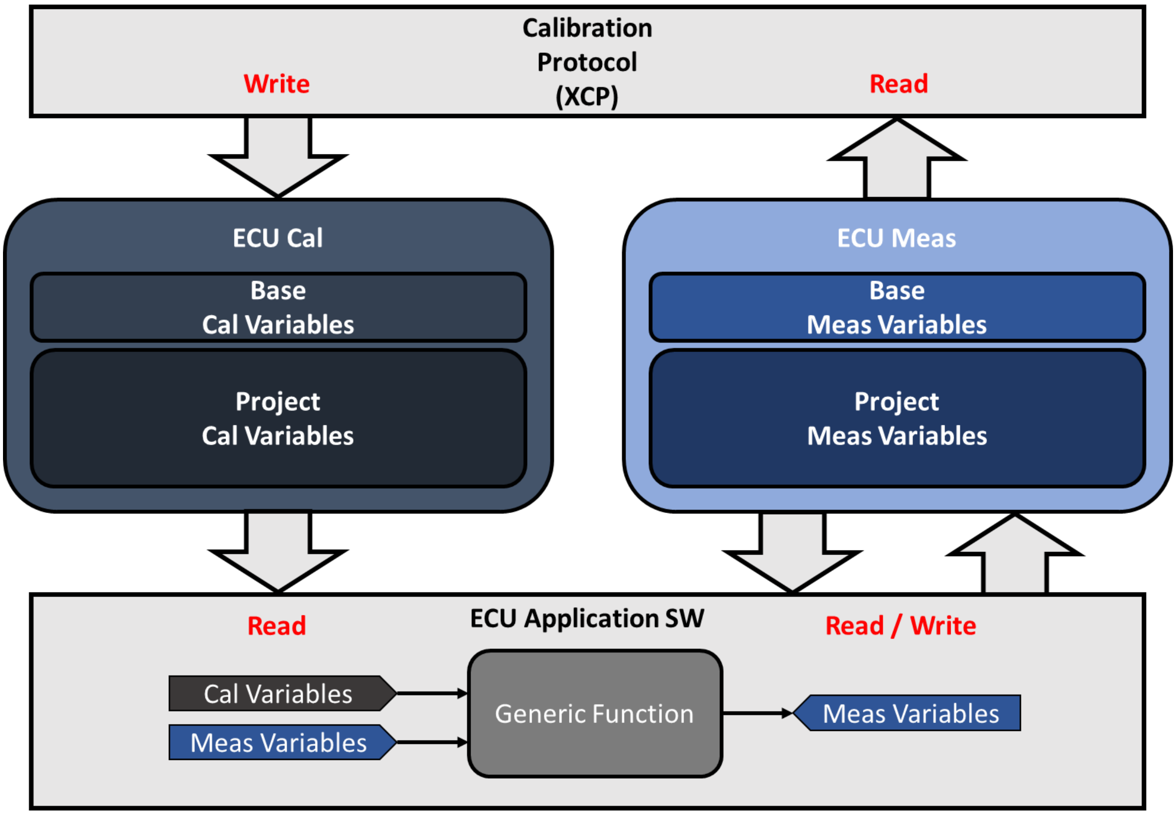


Figure 29: ECU Cal and ECU Meas variables interaction with the SW

Having the variables defined in two hierarchical accessible structures has been a requirement for an easy application software implementation. Although abuse and inappropriate use of global variables is considered bad programming, for a large control software architecture it is crucial. The set of the main variables needs to be accessible from everywhere in the code:

- Adding new variables reads and writes in a nested function will not cause the modification of the function (VI) interface and will not degenerate in a refactoring of all the nested code.
- Adding new functions will not cause the refactoring of the transportation mechanism of the variables.
- Variables needs to be accessed and updated by different tasks at different timing rates and in different part of the code. Full flexibility is required. This consideration is well declined in the calibration and measurement management, that has to update and collect parameters and variables all over the application.

Strict control on reading and writing of the variables is fundamental to avoid incurring in unexpected behaviours due to race conditions: updating a variable in a single place and reading it form multiple places avoid errors.

Data Value Reference implementation

Form the implementation point of view, variables have been made accessible through the software storing them with the **Data Value Reference** (DVR) library that allows the creation of portions of memory with a defined type (the clusters) and the access to them by reference, simply linking the DVR address and directly pointing to the variables of interest.

Data are accessed inside an "in-place structure" that grants the reading and writing of the variables to be done **without making additional copies** inside the structure. This enhances greatly the performance of a big software with hundreds or thousands of variables and maps. It is possible to enable the **parallel read** of the variables form different part of code, but the **writing is always protected**: when a DVR is accessed in writing mode, other parts of the software cannot access it. This ensures data integrity but must be taken in account when running multiple tasks accessing the same DVR, because some may result locked.

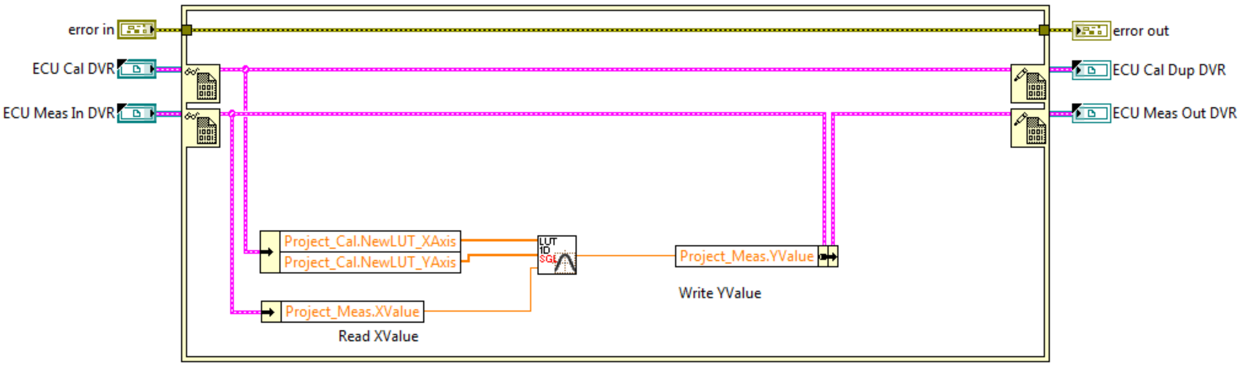


Figure 30: Calibration and Measurement variables access through the DVRs for a simple LUT calculation

3.1.2 *Base and Project Modules*

As presented in previous chapters, to enhance the reuse of common code the software has been divided into main categories base modules and project specific modules. Base modules contain all the standard and debugged code related to the common I/O and functionalities that are necessary to build rapidly a control project. The modules are built as much as possible in an abstracted way, replicating, if possible, the pattern that includes initialization, functions (configuration, read, write etc..) and closing: the developer can call the functions (LabVIEW VIs) and interact with the module through the calibration and measurement variables, defined in specific definition files, almost without the need of thinking about the implementation. Since to allow reusability the base modules need to be independent and de-referenced from the rest of the software, generally there are no references to shared resources among the project, such as the DVRs or the main clusters. When references to common project items are necessary to the functionality, a ready to use example is provided.

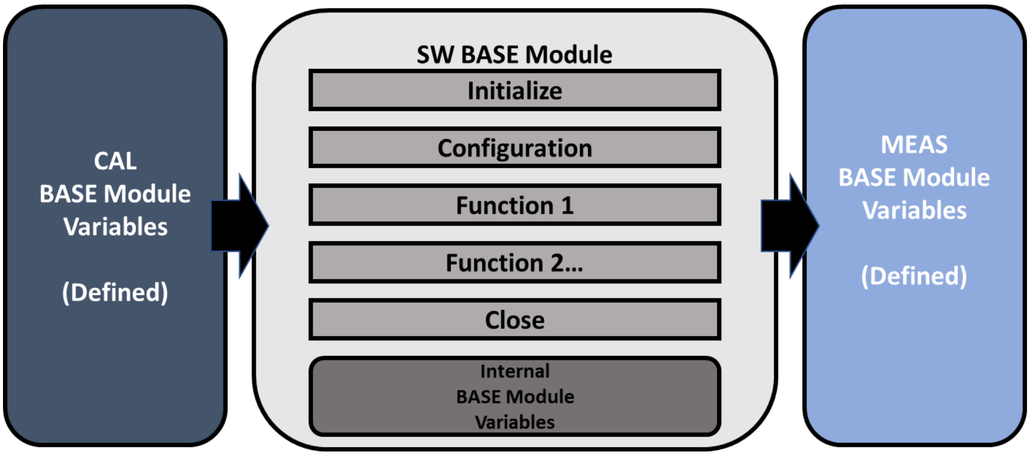


Figure 31: Base Module generic structure

The project section instead is dedicated to the application-specific modules developed for the single control project or as a customization of a standard functionality. The functions and the variables related to a module are grouped and independently defined as well, but generally they contain a direct reference to the shared resources of the project This is predisposed to easily interact with the other modules, and ease of future expansion: for example, an engine control model of the fuel control interacts with the air control, that interacts with the load control, and each module needs information about the RPMs from the phasing module.

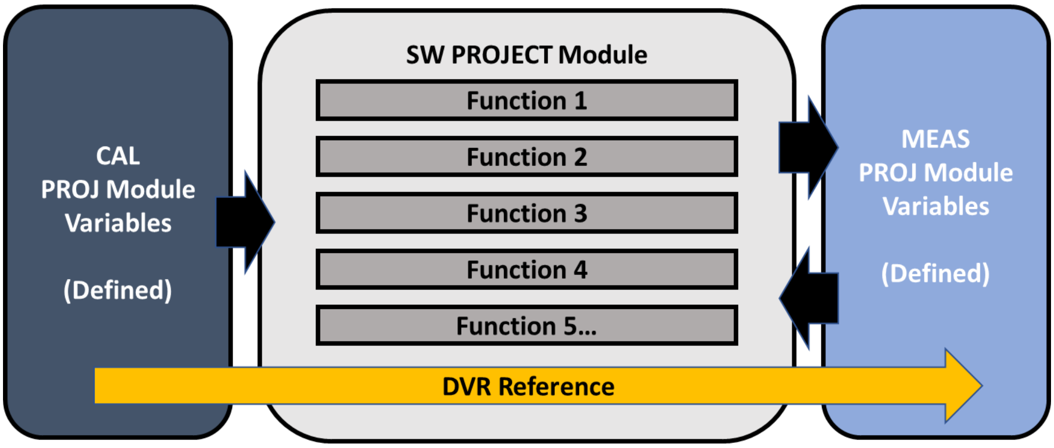


Figure 32: Project Specific Module generic structure

3.1.3 *Tasks organization*

The software architecture, apart from initialization and closing sequences, is divided in time-triggered or interrupt-triggered tasks. Most of the tasks execute at fixed rate, for example 2 / 4 / 10 / 100 / 500 milliseconds, but there is also the built-in possibility to trigger the loop frequency with an FPGA interrupt (IRQ) synchronously with the combustion frequency, for engine cycle-based calculation. Except from some predesigned base functions and libraries, such as CAN, logging, XCP or ECU services, that are gathered in independent tasks, all the custom acquisition, control and actuation functions can be organized to meet the user needs.

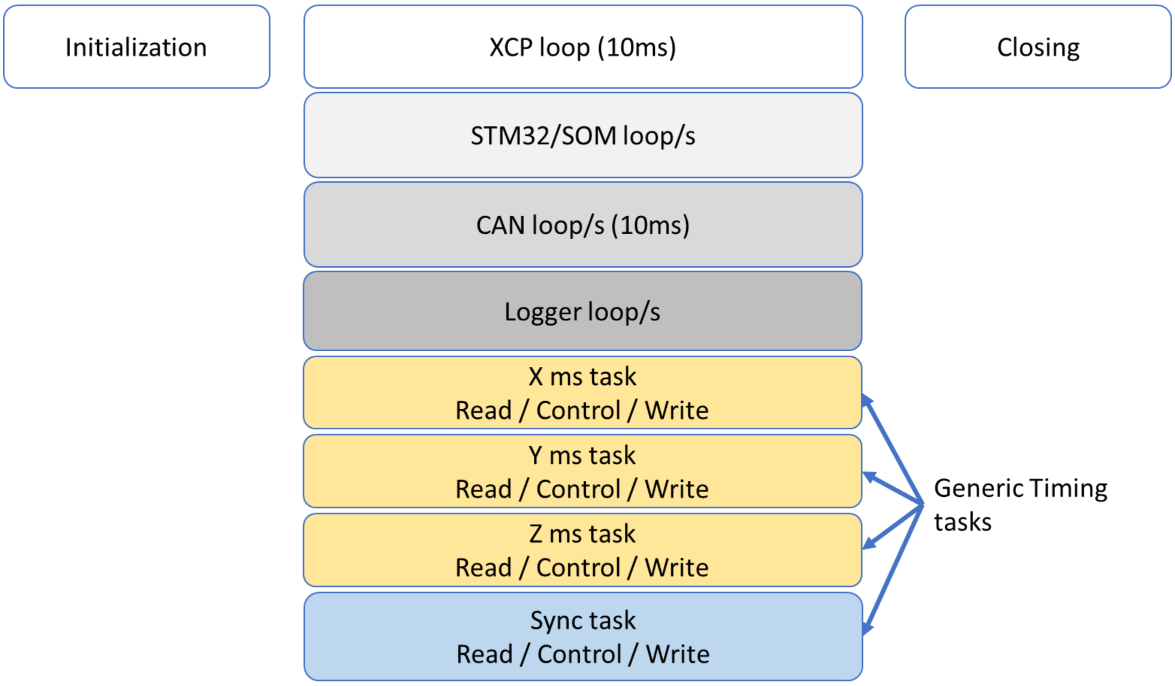


Figure 33: Software architecture tasks overview

Tasks are implemented as timed loops, and it is possible to assign them the execution priority and the processor (in case of multicore). Even if fully customizable, the structure of a generic task is designed as in Figure 34:

1. Execution Time Monitoring: evaluates and stores the loop time and the code execution time.
2. Read Functions: contains all the functions that perform reading form the ECU inputs or acquisition and linearization of sensors or data.
3. Control Functions: contains all the functions that perform control logic, strategies and calculations.
4. Write Functions: contains all the functions that perform writing to the ECU outputs.

5. Additional Functions: additional services that need to be executed within the assigned task time: for example, send data to the logger, send synchronous CAN messages and send synchronous XCP data.

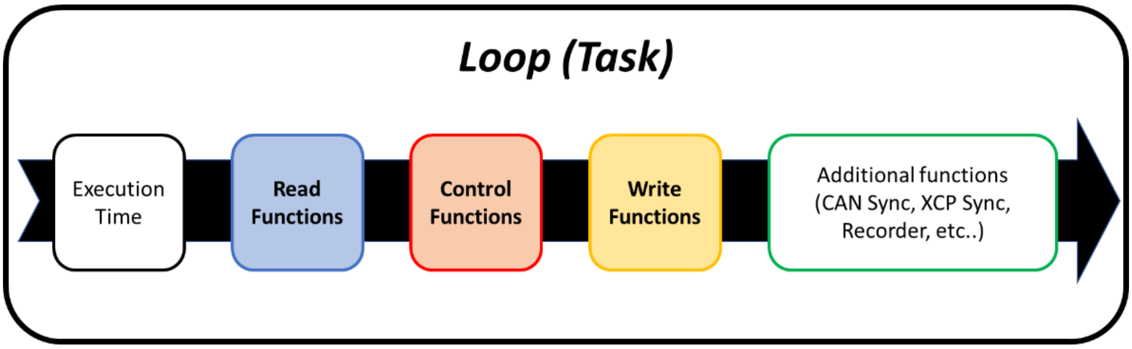


Figure 34: Generic Task Structure

3.2 Base Modules

The next paragraphs will present the logic behind the implementation choices of the base functionalities of the software architecture, with particular attention to the Spark platform that represented the main development board. In case of different platforms, the lowest layers of software will be adapted.

Some modules are present just in the Real-Time layer while others are also implemented in the FPGA layer. Since the reference to access the FPGA is shared among all the software, to improve the independence of the single libraries, each VI including a call to the FPGA resources also contains the global FPGA reference type-casted into the specific reference needed by the SW module.

In this way **the FPGA resources (input and outputs variables, FIFOs etc.) needed by the module are inherently defined inside the module**, and changes in the global reference will not affect the code integrity of the module.

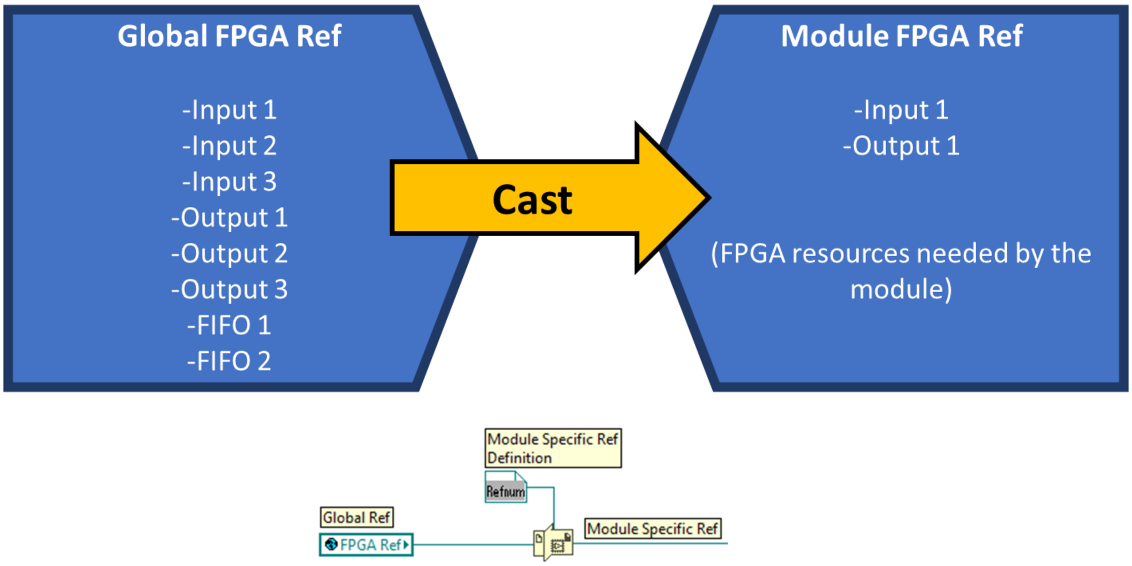


Figure 35: Type-Cast form global FPGA Reference to Module Specific FPGA Reference

If the actual global FPGA reference does not contain the needed resources by the module the functionality will not be available and an error will be returned in the log file.

3.2.1 *Configuration*

The configuration pattern is one of the key elements of the modularity of the framework. Almost each base module has a configuration section were the user accessible variables are converted in low-level parameters or actions to be accomplished. A collection of base modules configuration functions is set as ready to use in the framework, while a section is reserved for user custom and project specific configurations.

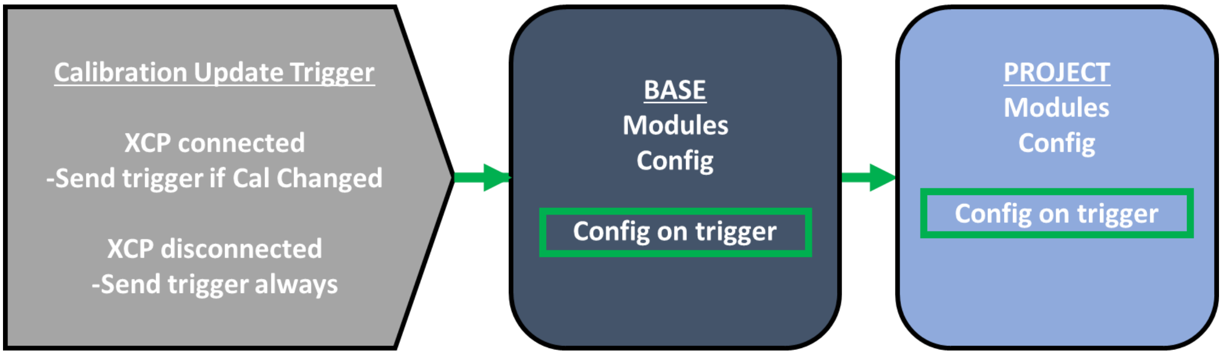


Figure 36: Configuration update scheme

The configuration can involve the Real-Time layer, the FPGA one or both.

For the **Real-Time** side, the structure is quite simple and involves the Real-Time configuration function/s of the specific module. For example, the configurable termination resistors for the CAN are set through a group of Boolean variables sent to the co-processor: the configuration VI simply writes those variables with the values defined by the user. As a result of internal calculations, in some cases variables are returned from the configuration VI or are written to internal memory allocations (DVRs) for private use inside the SW module.

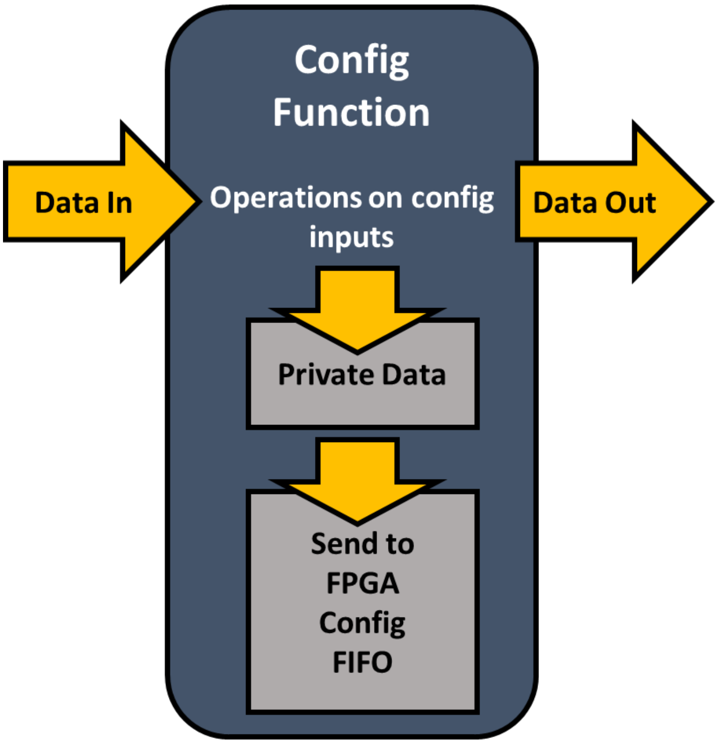


Figure 37: Generic configuration function pattern

For the **FPGA** a dedicated configuration data transfer structure has been realized. Different transfer mechanisms are available between the RT processor and the FPGA chip, and the two main and most used are:

-**Direct Read / Write to FPGA registers**: low latency and big resource usage for synchronization and data transfer. Each variable has its own direct access and it is suitable for small amount of data.

-**FIFO data transfer:** higher latency and smaller resource usage. It is a monodirectional communication channel with a fixed data type and First In / First Out queue mechanism. It is suitable for large or high frequency lossless data transfer.

The first implementation exploited the registers direct writing, with each module accessing its own variables through the FPGA read / write interface, but as a result a large amount of FPGA resources were used just for data transfer. With the software growing, not enough space on the chip for logical operations was available. For this reason, a FIFO-based refactoring has been realized for configuration functionalities, leaving the direct access only for actuators control and sensor reading that really benefit from low-latency in the data transfer.

The transfer mechanism is based on packages composed with an identifier number and data to be sent. The packages are "assembled" in the Real-Time environment by the configuration functions of each module, which has a set of ID numbers associated, and received and disassembled into FPGA configuration variable in the FPGA based on the ID number of the package. The packages are sent over the Real-Time environment with a queue to the FIFO transfer mechanism sending to the FPGA all the packages at fixed intervals (generally at 500ms). Each module including the configuration part has a Real-Time function for configuration and packages creation and a FPGA function for packages disassembling.

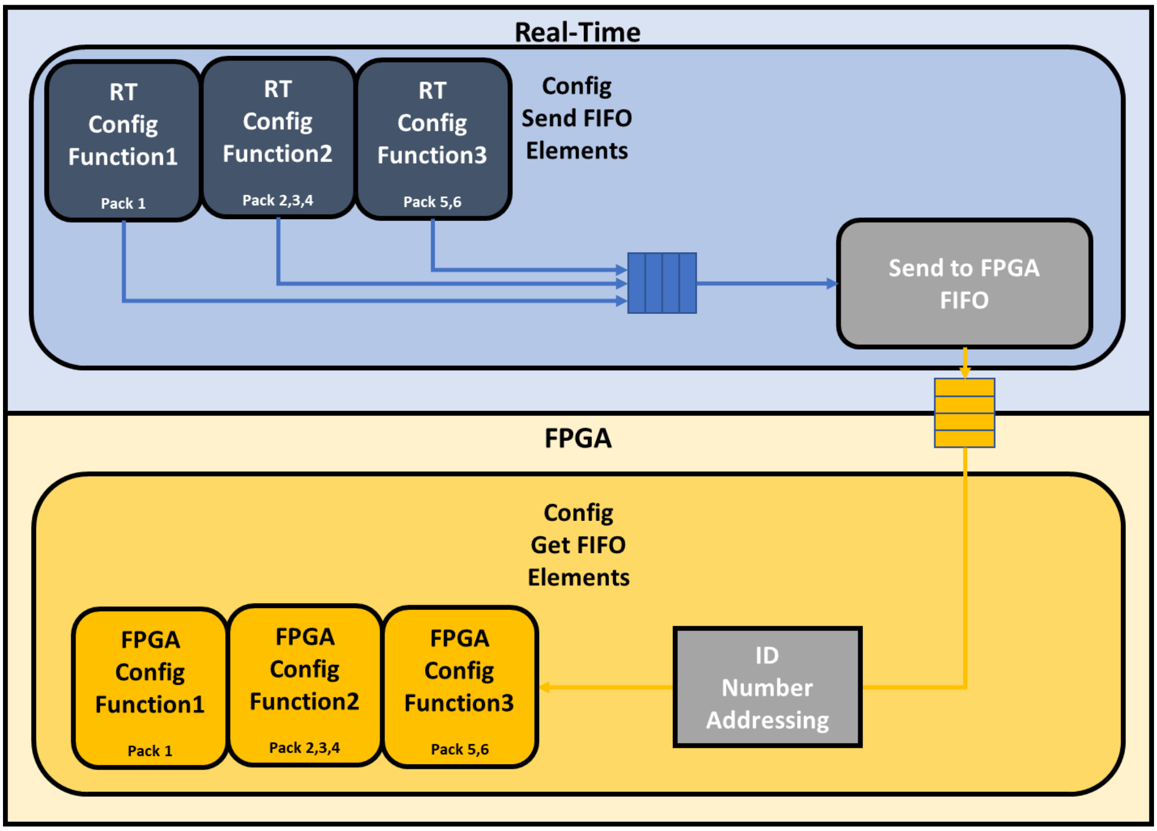


Figure 38: FPGA configuration structure example

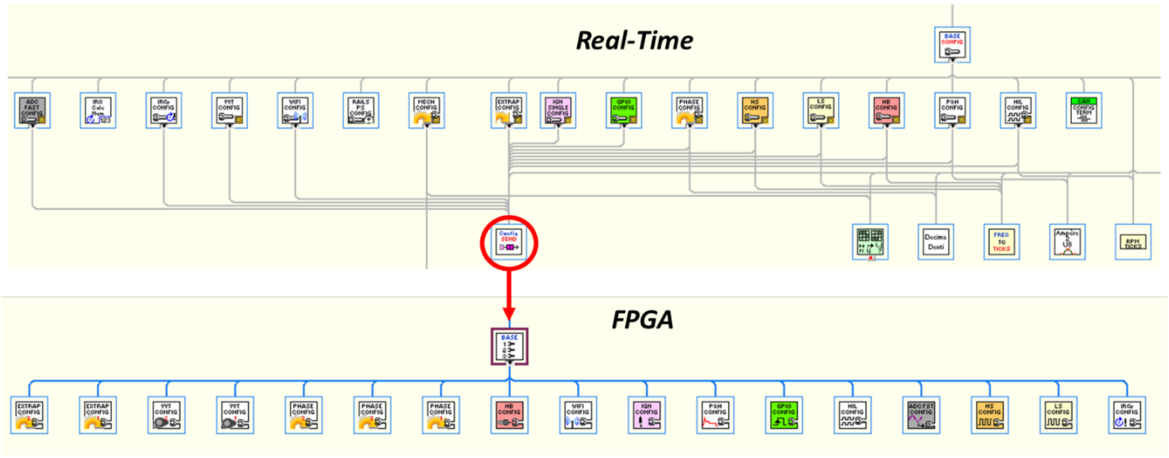


Figure 39: Real-Time and FPGA configuration base VIs hierarchy

3.2.2 *Phasing System*

The phasing system is one of the key modules for an engine control system. It is responsible not only of the RPM reading, but especially of the correct and accurate positioning of the phased actuations like injection and ignition. Since the module needs to process signals from teeth-wheels at high frequency and execute logic with hard determinism, the implementation of the algorithms has been realized entirely in the FPGA, apart from the configuration and reading function to interact with the Real-Time processor. Engine phasing algorithms for 4-strokes engines require a **crankshaft signal**, for the position referencing and a **camshaft signal**, for the hemi cycle identification. The module is divided in several submodules fulfilling different complementary functions.

Get Input Signals

This submodule gathers the input signals from the different sources of the ECU and allows the user to choose between them. VRS and GPIO inputs can be used as boolean source for the crank and cam signals variables that are passed into the other modules. The acquisition is done at high frequency in a loop running at 40MHz.

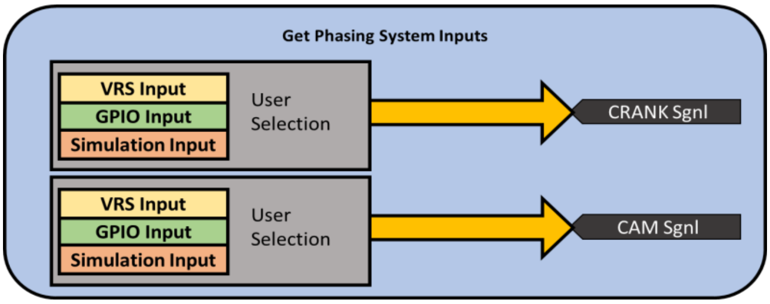


Figure 40: Get Phasing System Inputs scheme

Teeth Counter and Phase Determination

This module accomplishes the fundamental task of counter of the teeth of the crank signal and the phase determination based on the cam signal. Algorithm adaption are predisposed for different patterns of the signals: with one or two missing teeth holes, race wheel (no missing teeth), associated to a cam signal in case of a four-stroke engine or without in case of a two stroke. A user selectable integral filter is applied to the input signal to avoid false transitions in case of noisy signals.

The teeth counter counts the physical teeth on cycle basis, while the phase signal is used to determine where the 0 of the counter must be placed. Referring to a standard 60-2 teeth-wheel plus camshaft signal, the zero is placed at the 2nd teeth after the hole with the high cam signal. Falling or rising edge reference is user selectable.

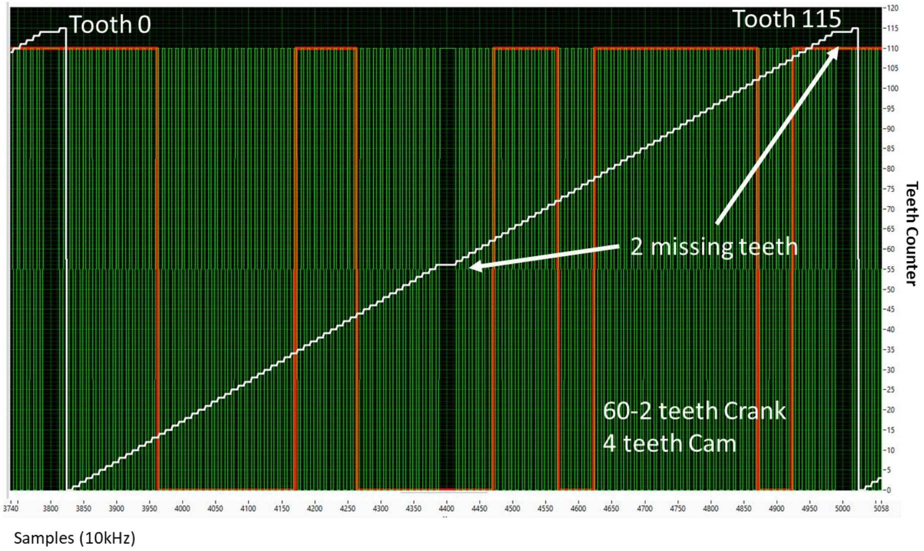


Figure 41:60-2 Crank and Cam signals acquired at 10kHz together with the teeth counter

This module has been refactored into a simpler subsystem starting from an already developed algorithm.

Teeth Extrapolation

Since the previous module counts the "real" teeth, in order to have an equal high level managing of the teeth count over every configuration, a teeth extrapolation algorithm has been realized to fill the missing teeth in the holes when included in the crank signal pattern. When the teeth counter assumes the value of the expected missing teeth, immediately previous tooth time is "locked" and replicated for N times, where N is the number of the missing teeth. The tooth time and teeth counter variables are copied into two new “extrapolation” variables and modified accordingly with the filling algorithm:

- The teeth extrapolation algorithm is executed at each crank tooth edge (loop on interrupt).
- If crank teeth counter is 0, then also the extrapolated counter is set to 0, otherwise it is incremented by 1. Parallelly, the extrapolated tooth time is set equal to real tooth time.
- If the selected mode includes missing teeth and the teeth counter is equal to the tooth number locating the missing teeth (2 times per cycle for a 4 stroke), then the additional following steps are executed.
- A for loop with a wait function is executed M times (M being the number of missing teeth): in each iteration, after a waiting time equal to the last measured tooth time, the extrapolated teeth counter is incremented by 1.
- The cycle after the previous event, the extrapolated tooth time is calculated as the real tooth time (including the missing teeth) minus M times the tooth time used for teeth filling in the for loop.

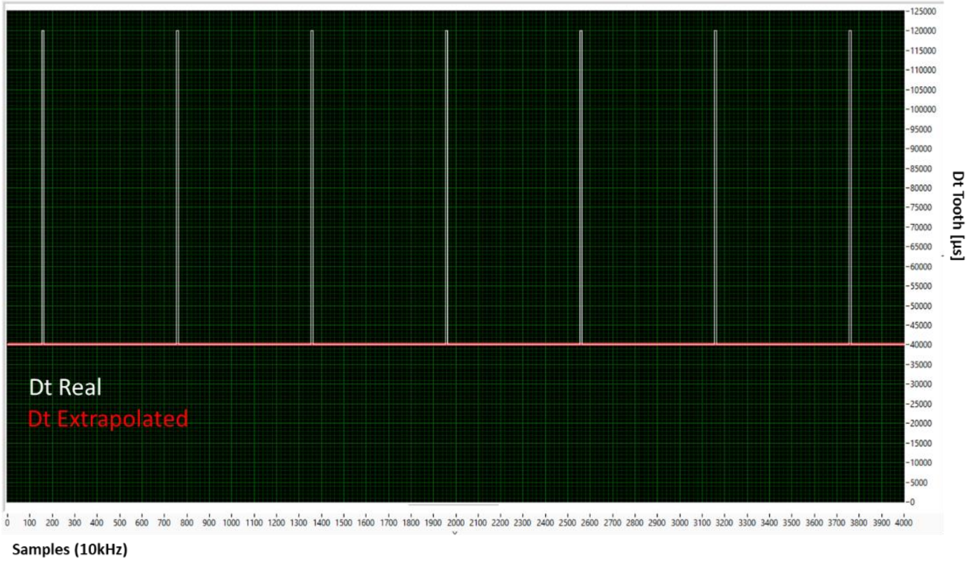


Figure 42: Tooth time extrapolation for the missing teeth

For the previous example of a 60-2 teeth-wheel the extrapolated teeth counter is shown in the figure below.

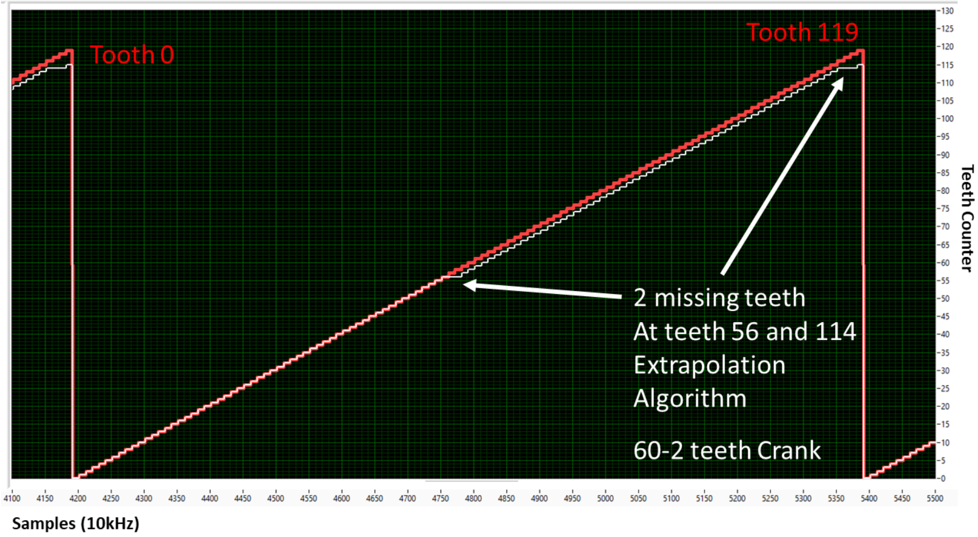


Figure 43: Teeth extrapolation result

Angle Extrapolation

While the position interpolation between two teeth is the approach selected for a precise angle-based analysis algorithm, extrapolating the position after the last seen tooth is necessary for a control application. Having an extrapolated angle variable is crucial to drive precise angle-based actuations and build new phased actuations drivers easily and rapidly. Starting from a previous implementation experience relying on the teeth counter and a delay time for actuations phasing, a new algorithm has been implemented in the FPGA to support the quick development of angle-based control and analysis algorithms.

The old angle reference mechanism (Figure 44) used the teeth countera and a delay time from the tooth as position extrapolation. This meant that for each angle-based actuation two parameters had to be specified: the tooth and the delay from that tooth. Moreover, since the actuations definition is generally demanded to the real-time high-level algorithms, the delay was calculated as an angle conversion on the basis of the mean cycle time, leading to quite big approximations due to the instantaneous speed fluctuations. For each actuation a checking mechanism for teeth and delay had been implemented.

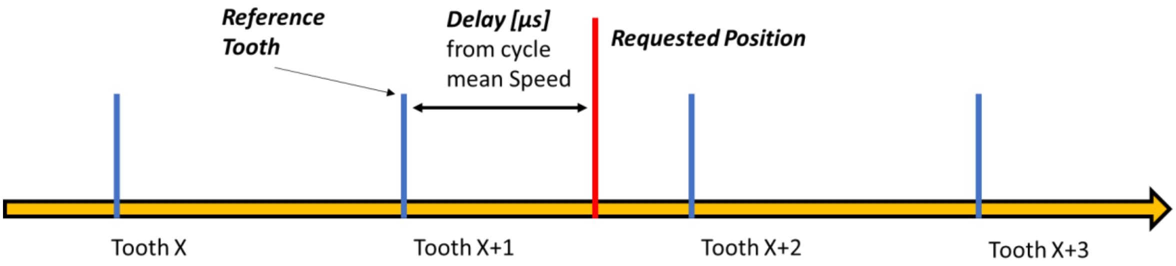


Figure 44: Old position referencing system

The new angle-based system exploits the direct angle referencing, making simpler the specification of the requested position: just one parameter, the requested angle, must be specified. This also makes the FPGA resource usage more efficient and the check mechanism simpler. On the implementation side, it must be reminded that floating point numbers cannot be used, while divisions use a very high quantity of resources on the FPGA, making them almost not usable. For this reason, a fixed-point representation of the angle has been chosen and an FPGA compatible algorithm has been implemented, exploiting the best capabilities of the environment: simple operations accomplished with fast speed. Instead of extrapolating the foreseen angle position after a fixed time step, which involves the computation of speed and so a division, the implemented solution waits for the necessary amount of time to reach the next angle step.

The fixed-point representation of the angle is 16 bits with 10 bits for the unsigned integer part. This leads to a maximum angle value of 1023,984375 (720 is max requested value) and a resolution in the representation of 0,015625°, much more than sufficient for the control purpose. This is the precision of the variable accessible to the other modules for position referencing. The internal angle variables of the extrapolation module have a 24 bits representation with a higher resolution of $6{,}10352 * 10^{-5}$ degrees.

Each tooth is divided in a selectable power of two number of parts (64, 128, 256, 512, 1024). This allows to calculate the corresponding time from the tooth time as a bit-shift (of 6, 7, 8, 9, 10 bits) in the FPGA. Low partition numbers are suitable for application characterized by high-speed or high number of teeth, in order to avoid losing precision at higher speed when the tooth time measured on 40MHz basis can become too small to be divided with a sufficient approximation on an integer number.

Starting from the extrapolated teeth counter described in the previous module, the FPGA angle extrapolation algorithm is defined below and graphically represented in Figure 45:

- The algorithm is executed in a loop running as fast as possible in FPGA
- The teeth counter is multiplied by the degrees per tooth, obtaining the actual tooth angle. The last tooth time is divided (bit-shift) to represent the time necessary to cover the respective tooth partition. The loop waits for this time before stepping next.
- After waiting, the actual tooth angle is summed with the partitioned tooth angle value that represents the angle step for each iteration. The angle integrated at each iteration apart from when the teeth counter changes value: in that case the integral is reset. The Integration cannot in any case go beyond the value of the single tooth equivalent angle.
- The value of the angle obtained by the integration is written to the 16 bits variable accessible from the other modules.

In this algorithm, the last instantaneous speed (tooth time) is used for angle extrapolation, so the speed fluctuations are chased better than using a cycle average approach. More refined solutions, such as an average on the last teeth (a power of two division can be easily calculated) or a speed trend prevision can also be implemented, but the tested solution, despite its simplicity has demonstrated to be valid in the faced applications.

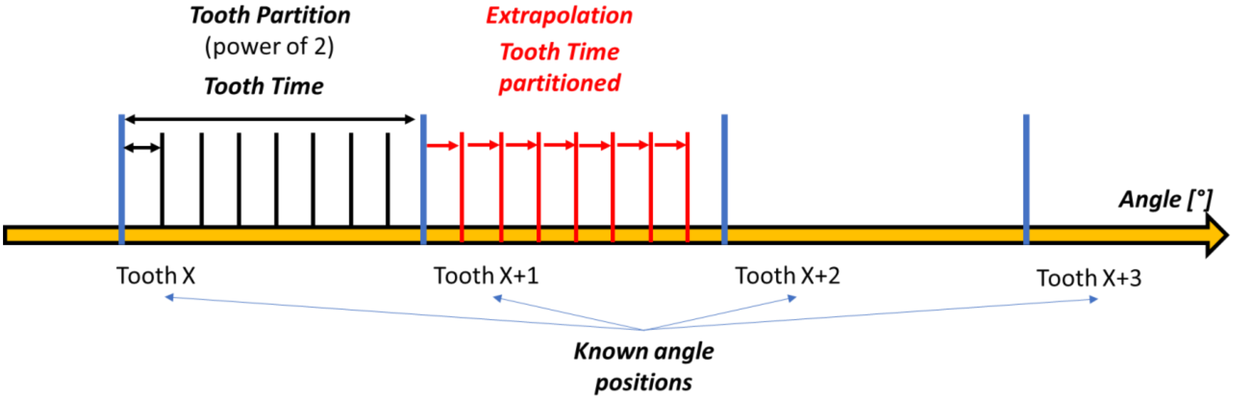


Figure 45: Angle extrapolation based on tooth partition

Angle Check Functions

The availability of a simple variable representing the actual angle in the engine cycle allows building a set of functions for checking when a target angle is reached. A tolerance, set as a configuration parameter, is used to return a valid acceptable output also in case of extrapolation under and over estimation and to deal with numeric approximations.

The actual angle is compared with the input target angle and the tolerance. If the actual angle lays between the target and the target plus the tolerance the output of the function is set to true, notifying that the searched angle has been reached. The angle check logic is graphically represented in Figure 46. Wrapping logic is included to accept and work without errors with any value included between 0 and 719,984375° as target input.

Different functions on the same logic have been built to return a rising edge trigger or a true / false persistent state or to perform the algorithm logic on integer angles.

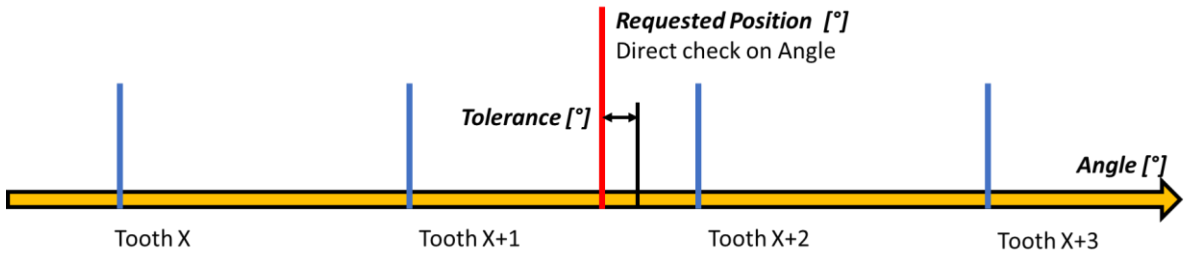
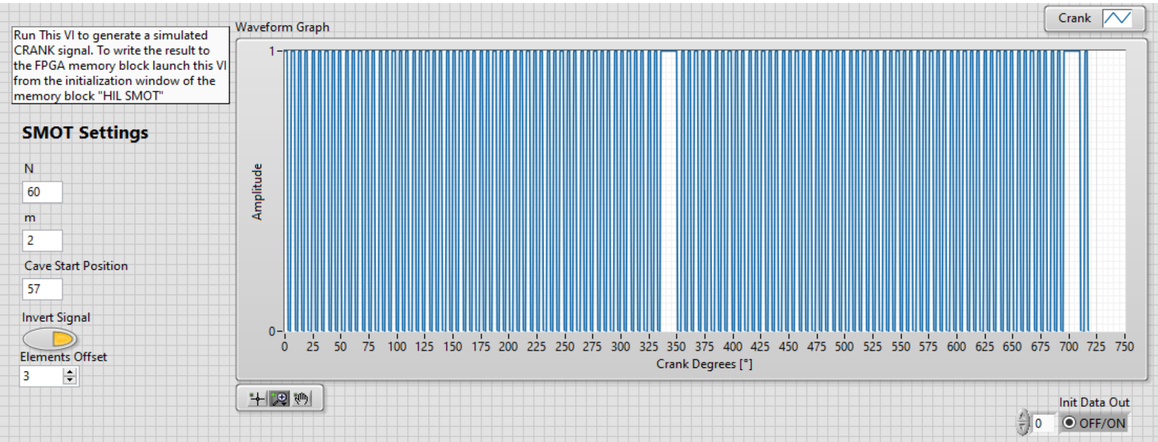


Figure 46: Angle-based position check

Signals Simulation

In order to test the phasing algorithms and the phased actuations (injection, ignition etc.) also when physical teeth-wheels signals are not available a simulation module has been developed. The module can simulate one crank signal and up to 4 cam signals that must be predefined at the compilation of the FPGA. The signals are generated in the FPGA on dedicated boolean variables, while the speed is configured from real-time.

The signals generation is provided through a while loop, where a set of memories, initialized with the signals profiles on angle basis, are read element per element at a variable and configurable rate depending on the speed to be simulated. In figures Figure 47 and Figure 48 are presented the configuration interfaces of the simulation crank and cam signals.



Run This VI to generate a simulated CRANK signal. To write the result to the FPGA memory block launch this VI from the initialization window of the memory block "HIL SMOT"

SMOT Settings

N: 60

m: 2

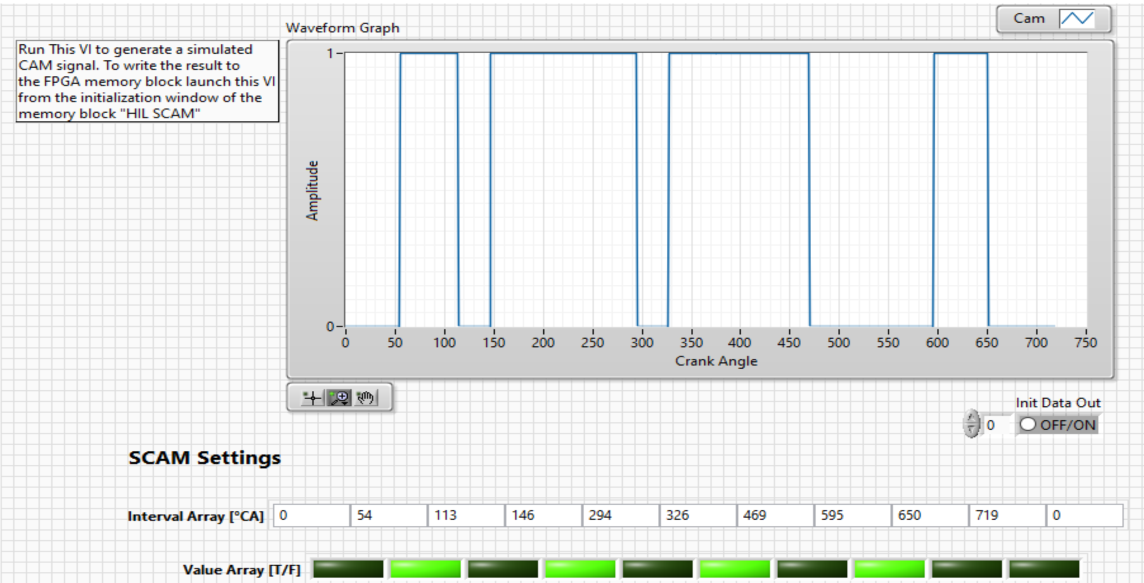
Cave Start Position: 57

Invert Signal

Elements Offset: 3

Init Data Out: 0 ● OFF/ON

Figure 47: Simulated crank signal generation VI



Run This VI to generate a simulated CAM signal. To write the result to the FPGA memory block launch this VI from the initialization window of the memory block "HIL SCAM"

Init Data Out: 0 ○ OFF/ON

SCAM Settings

Interval Array [°CA]	0	54	113	146	294	326	469	595	650	719	0
Value Array [T/F]	■	🟩	■	🟩	■	🟩	■	🟩	■	■	

Figure 48: Simulated cam signal generation VI

The possibility of emulating the VVT actuation is included allowing the user to shift the cam signals in comparison to the crank. A simple offset on the memory addresses to be read is configurable from RT.

3.2.3 *AI*

Analog input channels are one of the most used interfaces to acquire sensors. The developed module allows the easy access to the ECU hardware resources, without the need of re-writing low-level code. Taking in account the implementation on Spark hardware there are two analog input sources at two different rates. The resource access and the processing are different, but the high-level access of the resource has been made standardized and "ready-to-use" as presented in the functional scheme in Figure 49.

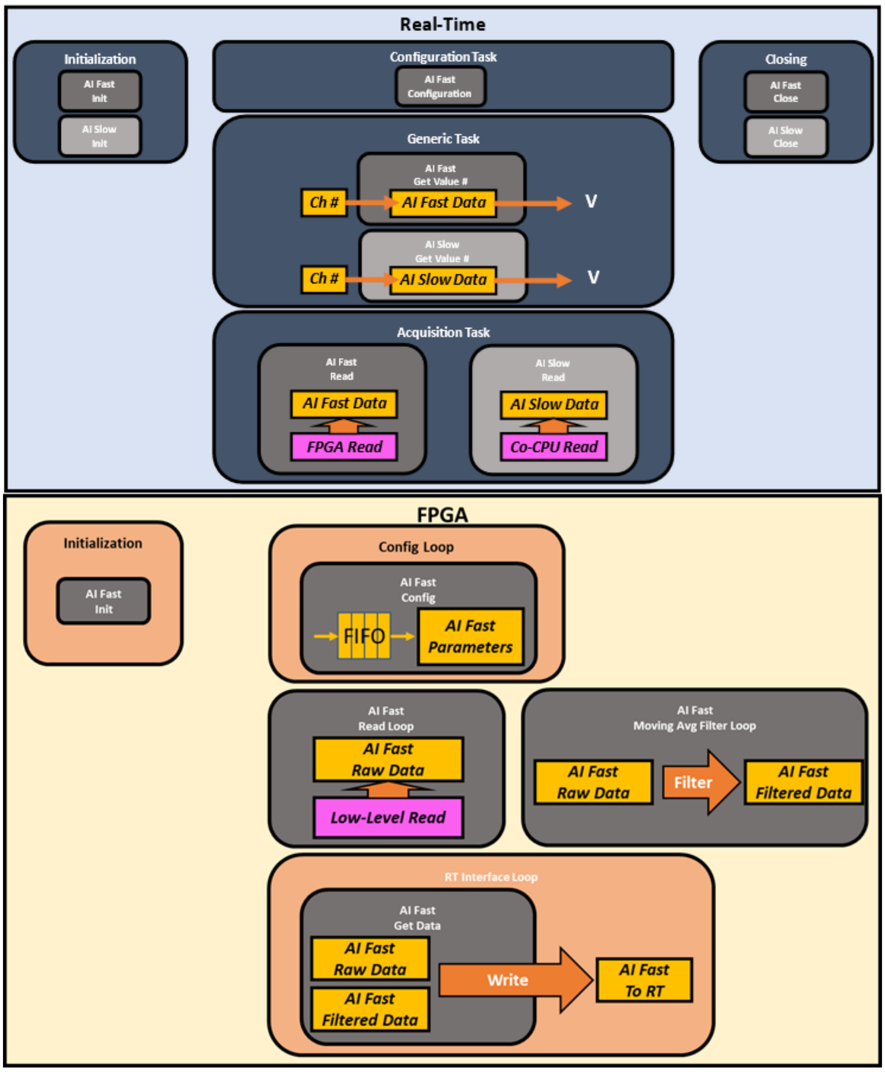


Figure 49: FPGA and Real-Time AI module functional scheme

Analog Input Fast: 8ch with 18bits resolution, acquired in FPGA at a maximum frequency of 200 kHz.

The raw data are acquired in the FPGA in a function that contains a loop running at 200 kHz and stored in specific registers. These registers can be filtered with a standard low-pass Butterworth filter function or with a moving average filter on a selectable angular window. Finally, the AI values are made available to the Real-Time as direct registers read through a specific function that selects the filtered or raw data.

The high frequency capability of these inputs makes them suitable for the processing of signals such as cylinders pressure, knock accelerometers or for custom processing. For example a common task is to acquire the manifold air pressure sensor or the boost pressure sensor of an internal combustion engine and remove the cycle frequencies oscillations that can generate aliasing when acquired by the Real-Time controller at a lower frequency than the sensor bandwidth. Since main oscillation patterns are produced on cycle base (depending also on the number of cylinders) a moving average on a configurable angle window is capable of effectively remove the cycle trends and allow a stable reading for control purpose (cylinder air mass estimation and boost control): in Figure 50 is presented the effected of the filter on the acquired boost pressure as example of application. From the implementation point of view a specific calculation algorithm is needed to meet the FPGA limitations and requirements.

- The moving average algorithm loop is triggered each 1°, exploiting the angle extrapolation variables. Narrower acquisition rates are possible but 1° is a good trade-off between algorithm effectiveness and FPGA resource usage.
- To allow the possibility of a moving window on angular basis, a memory block has been implemented for each AI channel to store up to 720 sampled elements (1 sample per degree).
- At each iteration, the oldest element stored in the memory is removed and a new sampled element is added. The memory addresses are managed based on the configured angular window width.
- A 32bits variable is used to evaluate the channel average value on the angular window, storing the sum of the elements within the selected angular window.
- The 32bits value is the value accessed form the RT: it is divided by the number of elements of the configured window and then converted to volts. Managing the division in the RT allows saving resources in the FPGA.

$$AI_{sum}(Ang_0) = AI_{sum}(Ang_{-1}) + AI_{value}(Ang_0) - AI_{value}(Ang_{0-Wndw}) \rightarrow \text{Calc on FPGA}$$

$$AI_{mean}(Ang_0) = \frac{AI_{sum}(Ang_0)}{Wndw} \rightarrow \text{Calc on RT}$$

Ang=integer angle value

$Wndw$= Angular moving average window number of elements (1° = 1 element)

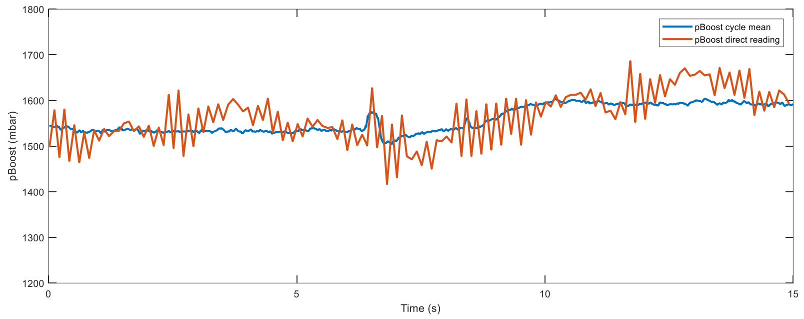


Figure 50: Boost pressure sensor reading form the RT controller (100Hz): direct reading compared with a 720° mean.

On the Real-Time side an initialization function loads a calibration file that takes in account the ADCs offset and gain of the device compared to the nominal values, and a DVR for storing the channels value is initialized. A configuration function sends to the FPGA the processing method and the calibration offsets, while a specific VI reads the raw data form the FPGA registers, converts them into electrical values (V) and stores them as private data in the DVR. At this point the AI voltage values can be accessed all over the software by channel address using the dedicated function.

Analog Input Slow: 16ch with 12bits resolution, acquired by the co-processor and made available in the Real-Time software at a maximum frequency of 1 kHz.

In the real-time software the data are accessed through a dedicated co-processor DVR and are pre-processed with an antialiasing filter. The data are retrieved and stored in a specific memory location in the acquisition loop, then they can be accessed by channel address from anywhere through the dedicated VI ().

Below the functional schematics (Figure 49) and the relative example of the LabView code (Figure 51) implementation and usage of the AI module are presented.

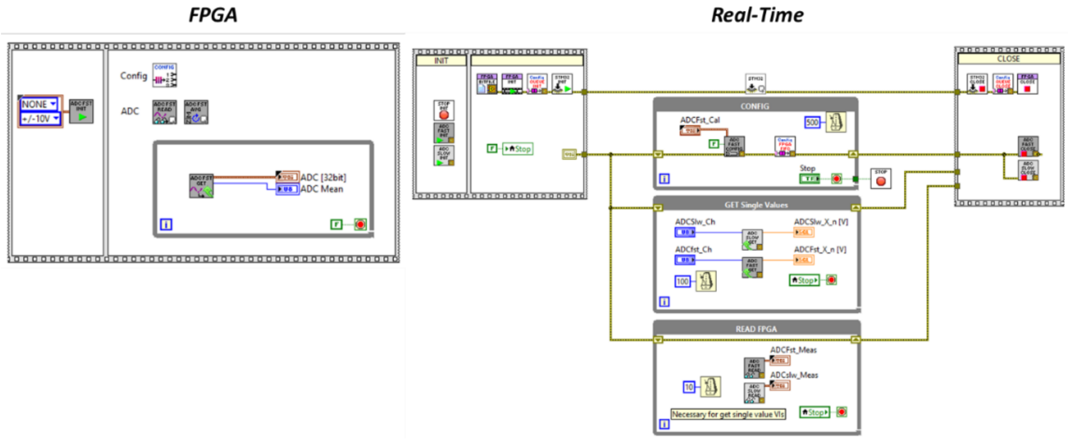


Figure 51: FPGA and Real-Time AI module code implementation example

The data transport mechanism and processing are not strictly dependant on the HW used since the lower-level access to the HW drivers can be adapted and replaced at the need. For example, the AI module has been used on different RCPs on Spark and Miracle hardware, with a different AI configuration (Miracle has only 24 fast AI channels), but exploiting the same framework and the same high level functions and data interfaces.

3.2.4 ***AO***

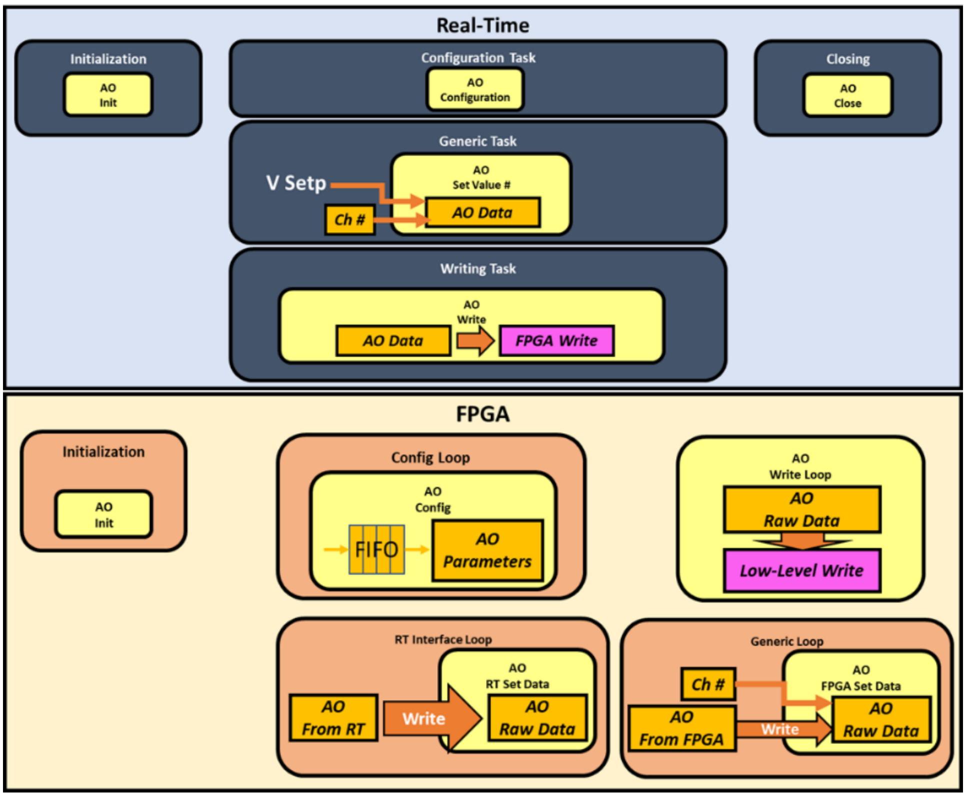


Figure 52: FPGA and Real-Time full AO functional scheme

Analog outputs channels follow mostly the same path as AI but in the opposite sense. On the Real-Time side the library includes a function for AO setpoint definition that can be accessed from everywhere in the code and that internally operates the conversion from volts to raw binary value to be transferred at low level. The Spark hardware has only 2 AO channels, managed by the co-processor that reads the desired setpoint and accomplishes the task independently, while on the Miracle controller 16 high speed channels accessible as FPGA I/O are provided. For these reasons an additional functions subset has been integrated in the AO software module. The functions include the setpoint writing from RT to FPGA and a configuration block: the user is given the possibility to choose the source of the AO setpoint channel by channel, from RT for general control purpose and directly from FPGA, for high-speed generation tasks, such as teeth-wheel replication. On the FPGA side the module includes: real-time setpoint getting, AO setpoint setting from FPGA, configuration acquisition and the output generation function that recalls the HW drivers. In Figure 52 is presented a functional scheme of the AO module.

3.2.5 *GPIOs*

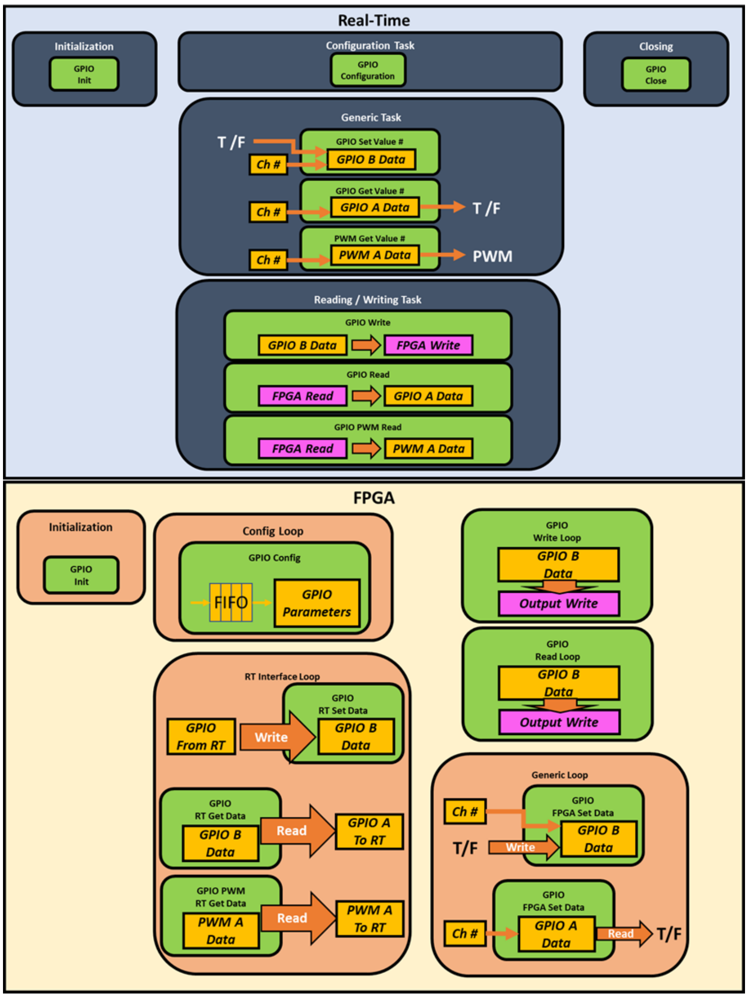


Figure 53: Real-Time and FPGA GPIO module functional scheme

General Purpose Input – Output pins are a key feature for reading digital sensors, such as hall-effect sensors, for driving external powered actuators, such as “smart” ignition coils or switching solid state relays and also for simple high / low state communication between devices. GPIO are generally dived in banks configurable both as input or output, on Spark hardware two 8 channels banks (A and B) are accessible on the FPGA with a 0-5 V logical level. Since they can be used for a wide range of applications the library module developed includes the possibility to use them as fast I/O directly in the FPGA or access them more slowly from the Real-Time. On the Real-Time side a configuration function allows to select which channels are writable from RT or FPGA, while the standard DVR mechanism, viewed also for AI and AO, allows the user to get or set by address the single value (True or False) on the GPIO pin. Then a function placed in a fast loop writes the values to the FPGA, where after the initialization function that sets the direction of the pins, it is possible to access the I/O resources up to 40MHz. By default, a bank is configured in write mode while the other in writing mode and two loops access the HW resources to read and write the values at 1 tick loop time, as shown in Figure 54 where the code of the two FPGA reading and writing loop is presented.

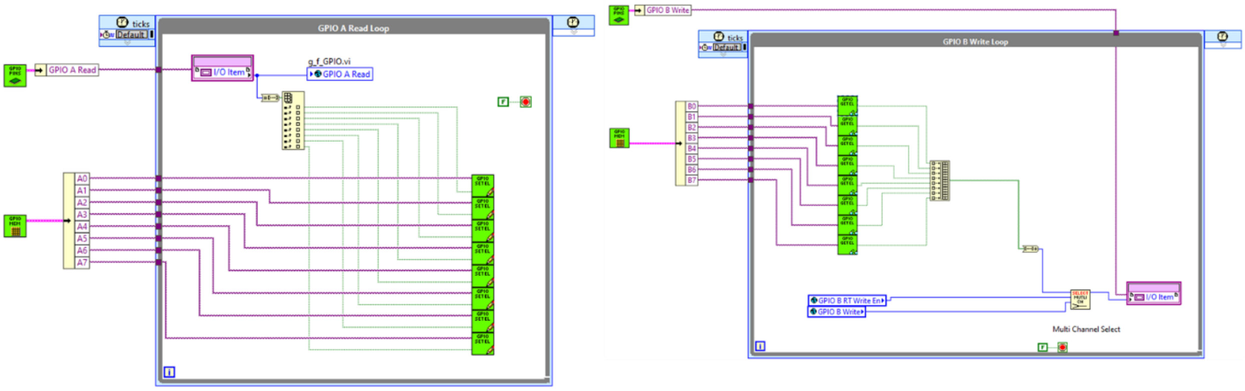


Figure 54: FPGA GPIO read and write functions insight.

Each GPIO channel value is stored in an internal FPGA register (Figure 55) and can be get or set individually by any part of the FPGA code. This is useful to build custom functions that use the GPIOs as inputs or outputs.

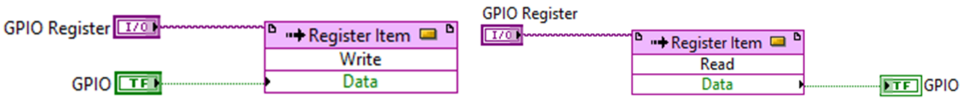


Figure 55: FPGA internal registers read and write for the GPIO module

Additionally to the standard reading and writing, also PWM acquisition and generation VIs have been included in the module. The PWM reading function applies a configurable digital integral filter to the boolean value in input to cut away false transitions and counts the permanence time at the high state and at low state of the input signal. These information are passed to the Real-Time controller where a dedicated function calculates the PWM metrics: the duty cycle and the frequency of the pulsation. This function is useful in particular to acquire those sensors that have a frequency output such as flow meters. PWM generation on the GPIO is also possible.

3.2.6 *LS, HS and HB (PWM)*

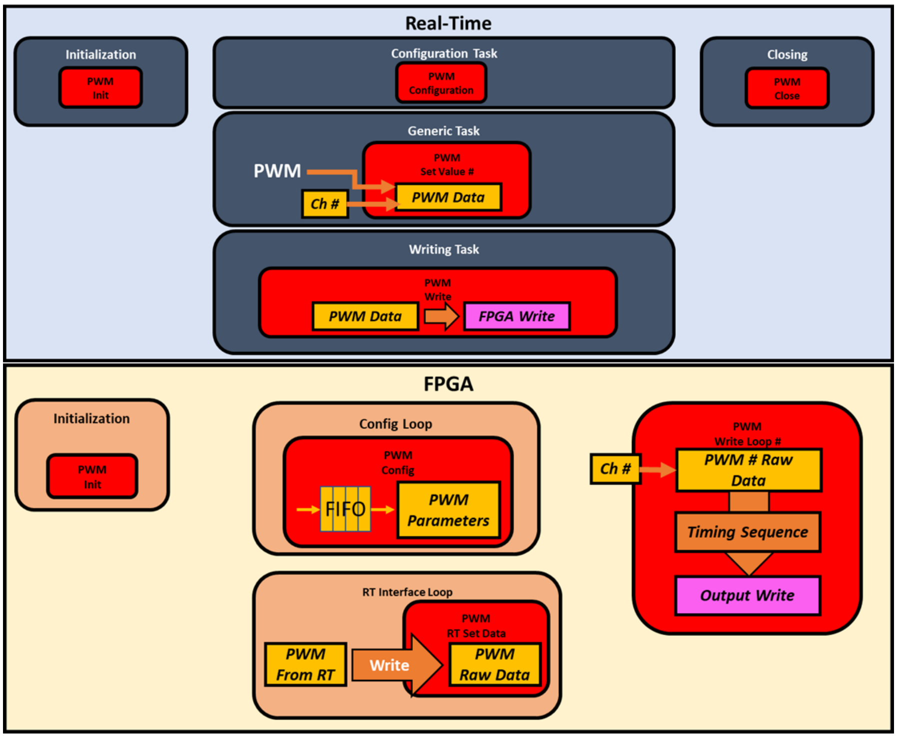


Figure 56: Real-Time and FPGA PWM generic example functional scheme

Standard modules for easily accessing PWM generation on power drivers have been one of the first developed features, since most of the actuators to be controlled apart from ignition and injectors require this type of command. On the Spark hardware three types of general-purpose power drivers are present in the I/O: 8 Low-Side drivers, 8 High-Side drivers and 2 H-Bridge drivers.

Low-Side and High-Side

Low and High side drivers provide the same kind of logical actuation, on and off, and providing a proper high switching frequency it is possible to generate a duty cycle from 0 to 100% on the voltage source. For this reason, the base functions are the same and, on the RT, include the initialization, the closing, the configuration of the PWM frequency, and the single channel PWM setting. Then a writing function in a fast loop acknowledges the FPGA with the required PWM values. On the FPGA side, the drivers are initialized and configured through config FIFO with the proper frequency and the PWM generation is accomplished by a function containing a timed loop (Figure 57):

- The proper PWM channel resource is selected
- The channel specific PWM **period** register, configured through the config FIFO, is read and used to give the timing of the loop

- The channel specific PWM **pulse** register, set by the RT control software, is read and use to specify the high or true time of the PWM output.

The timing resolution for the period and pulse registers is 1 tick (40MHz) that gives an optimal resolution on the PWM = Pulse time / Period time. The parameters are written to the FPGA registers as 32 bits unsigned register.

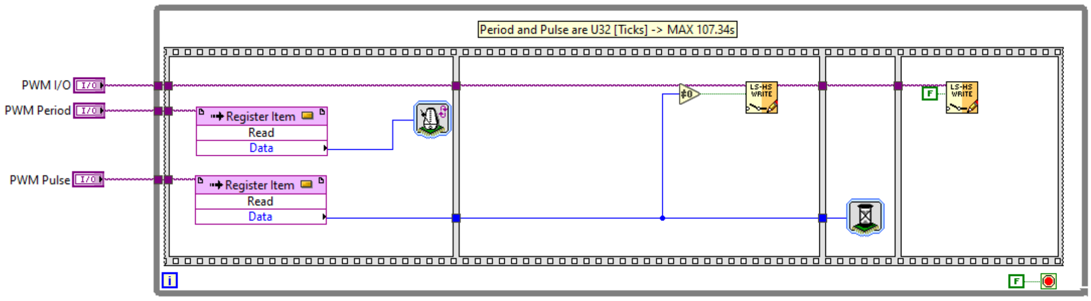


Figure 57: FPGA PWM generation loop example for LS and HS

H-Bridge

Also for the H-Bridge channels the functions are the same, but is possible to generate a duty cycle from -100 to 100% on the voltage source, for actuators that require a bidirectional control of the electric motor, for example an electric throttle body. Only the low-level implementation inside the PWM timing loop is different, since the H-Bridge configuration is not just simply on and off, but requires the management of the "mode" of the driver: forward and reverse supply, short circuit to supply and to ground (brake) and disabled. The mode of the H-Bridge driver is automatically set, based on the duty cycle requested, by the Real-Time in the function that converts and sends the pulse value to the FPGA. The user just needs to set the desired duty cycle and the library has already all the necessary code implemented to drive the requested output.

For example, in the case of the M2 platform, due to the lack of power drivers, PWM generation VIs are just available as GPO outputs with the structure as for LS and HS drivers. Below in the functional scheme it can be noticed that the structure exploits pretty the same mechanisms as for the AO, GPIO and also other modules.

In case of a different usage for this power drivers different from just a PWM, a simple FPGA write function is provided in the library to build custom functions.

3.2.7 *Multi and Single Injection*

Automotive RCPs almost always require the control of a set of injectors that can be for direct or indirect injection, requiring different electrical actuations and different control modes. For this reason, the developed injection module tries to cover the most frequent situation that can be encountered and faced with the available hardware. Referring to Spark hardware the output resources that have been included in the described module are:

- **Peak and Hold drivers**: used to generate a fast current profile generally required by solenoid direct injectors. They can be used also to drive a traditional PFI injector setting properly the voltage and the required current level. An independent sub-module manages the configuration of the voltage (0 to 100V) of the P&H rail and allows the developer to easily set the requested current level of the single driver by means of ready to use functions.
- **Low-Side drivers**: mainly used to drive solenoid indirect injector. The library includes the possibility to use as injector driver the standard outputs for the ignition coils (low-side). This can be useful for running a configuration with both DI (high voltage P&H) and PFI (12V LS) injectors.
- **GPIO output**: general purpose outputs can be used to command external power drivers when the standard I/O of the used ECU is not sufficient or must be expanded. An example is the case of piezo-electric injectors that require specific power drivers.

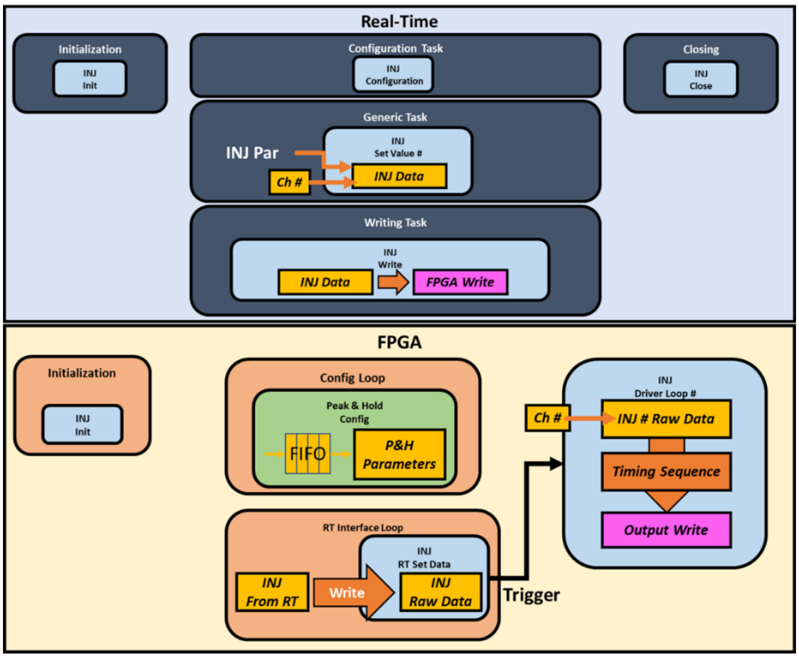


Figure 58: Real-Time and FPGA Injection Module functional scheme

The module both implemented in RT and FPGA exploits the already used structure architecture (Figure 58). The main difference is in the timing of the actuation that needs to be phased with the engine cycle and for this reason it is strictly dependant on the angle reference provided by the phasing module. As it will be described later, the injection sequence relies on the search of the start angle of the actuation. Form the FPGA perspective the Real-Time engine control can update the injection parameters at any time and for this reason, to avoid the changing of the parameters during the injection sequence a pre-triggering mechanism has been built (Figure 59):

- The injection parameters registers are updated reading the values from RT continuously.
- The SOI (Start of Injection) angle of the **first** injection of the cycle is compared quickly with the current angle.
- If the current angle is nearer than a fixed angle (5° is the default value) a trigger is raised on for the involved injector. These operations involve a simplified and less precise angle check procedure that consumes less resources.
- The trigger rising is seen by the involved injector driver loop and the effective injection sequence begins. Once the trigger is received the parameters are locked for the current injection sequence.

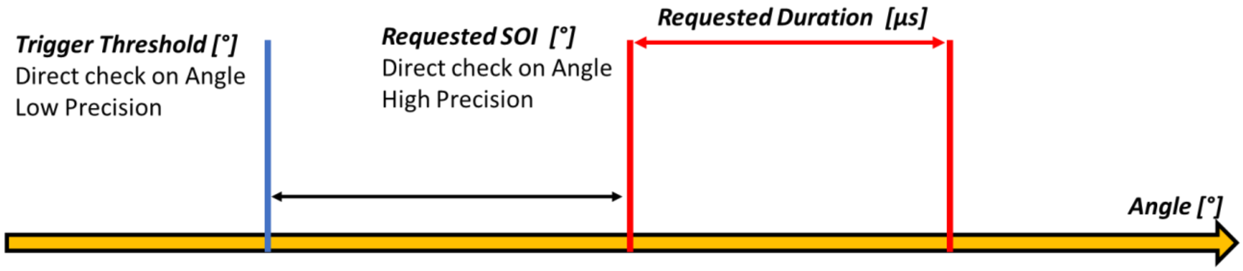


Figure 59: Angle based Trigger search and SOI + Duration actuation

This mechanism is necessary also to grant that each injection actuation is done with the most recent values commanded form the RT engine control, otherwise as soon as an injection sequence is completed a new one would start searching for the same parameters, causing every time two engine cycles with the same actuations. Differently, the pre-trigger search allows to wait as much as possible for a changing in the injection parameters before locking them and start the sequence.

The functional scheme is the same for all the output drivers described above in this chapter except for the output functions that recalls different power drivers. The injection module is divided into two different groups of functions: multiple and single injection.

Multiple Injection:

Up to five injections per cycle are possible for each injector. For each injection the following parameters must be set from the engine control in RT:

- The number of the selected injector
- The number of the selected injection
- Injection time [µs]
- The SOI [°BTDC]
- The TDC [°] on which the SOI is referred

These information are collected and converted in the RT software into FPGA compatible instruction for each injector. For each injector an array of 5 32bits elements is built, where the 32bits represent the requested start angle on the absolute reference (16bits) and the duration of the actuation (16bits). These parameters are then passed into the FPGA through direct writing and stored in dedicated registers. Each injector has its own independent function containing a loop to manage the timing of the actuation, the steps are described below:

- If the injector is disabled or the angle referencing system is not phased, no action is taken and the loop runs empty.
- The injection trigger is waited
- Injection SOI [°] and duration [µs] are retrieved from the registers and processed in a for loop, injection per injection:
    - If injection time is 0, then the actuation is skipped.
    - The SOI angle is waited through the angle check function
    - Actuation is set to active state. In case of a boolean actuation it is set to TRUE while in case of a P&H the sequence of the current levels is run.
    - After the proper duration the actuation is set to the not active state.
    - The for loop iterates to the next injection
- The loop restarts

In case of a peak and hold actuation a dedicated sequence of activation of the power driver is accomplished, represented in Figure 60 and described by the parameters:

- **P Current [A]:** peak current
- **P Time [us]:** peak current duration time
- **PH Current [A]:** "peak-hold" current. This current level is an intermediate level between "peak" and "hold" phases.
- **PH Time [us]:** "peak-hold" duration time
- **H Current [A]:** hold current is maintained as long as the actuation duration requires.

A truncation mechanism is included, so at any point in the P&H sequence the current be set to 0 if the total actuation time comes up to be shorter than the fixed time phases.

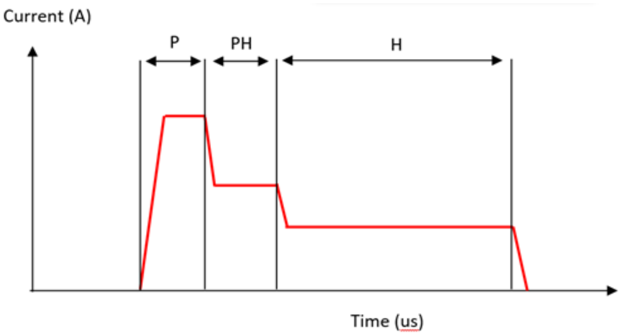


Figure 60: Peak and Hold sequence

In order to prevent the overlapping or the missing of some actuation due to unrealizable SOI and duration requests, on the Real-Time side a dedicated function checks the and corrects the SOI of the 5 injections if a minimum distance between them is not respected. The actuations are subsequently shifted forward until the minimum distance (time constraint) is respected between all the actuations. An example of 5 consecutive P&H actuations observed on the oscilloscope is shown below in Figure 61.

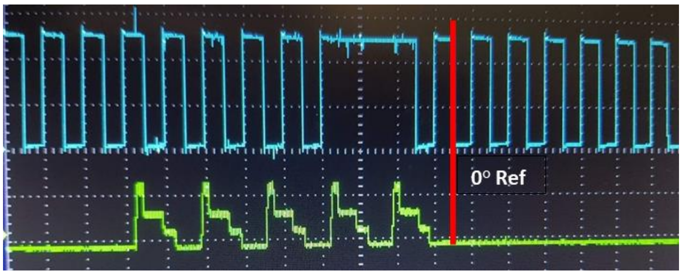


Figure 61: Example of 5 Peak and Hold actuation observed on the oscilloscope referred on the crank signal (0° reference is highlighted)

The ready to use outputs are the P&H and the GPO, but any kind of output can be ideally included just replacing the calls to the writing functions.

Single Injection:

Also a library for single injection per cycle is provided and can be used also in parallel to the multi injection one. It exploits the same structure described previously and, apart from the number of injections, it differs only for the possibility to select a SOI or EOI specification for the actuation. This sub-module can be useful to drive additional PFI injectors together with DI ones.

The SOI version of the drivers expects the specification of the following parameters to be converted into FPGA instructions:

- The number of the selected injector
- Injection time [µs]
- The SOI [°BTDC]
- The TDC [°] on which the SOI is referred

The EOI (End of Injection) version of the drivers requires additionally the average cycle speed of the engine. The difference is purely in the function interface because the low-level implementation is the same and necessarily the actuation is searched on a SOI basis.

- The number of the selected injector
- Injection time [µs]
- The EOI [°ATDC]
- Average Cycle Speed [°/s] → needed to convert the EOI to SOI
- The TDC [°] on which the SOI is referred

The ready to use outputs are the Ignition Coil Low-Side and the GPO, but any kind of output can be ideally included just replacing the calls to the writing functions.

3.2.8 *Ignition*

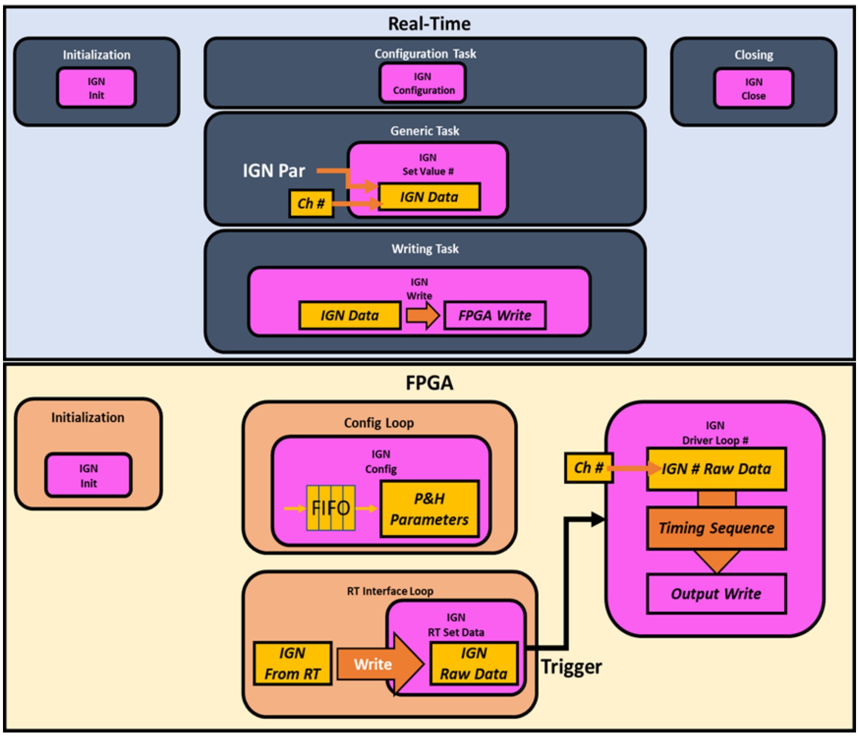


Figure 62: Real-Time and FPGA Ignition Module functional scheme

The ignition coil phased actuation is one of the most important tasks that the ECU has to accomplish when the controlled plant is a spark ignition engine. Referring to Spark hardware the output resources that have been included in the described module are:

- **Ignition Coil Low-Side drivers**: dedicated low-side that can be used to drive traditional ignition coils.
- **GPIO output**: the general purpose outputs can be used to drive modern “smart coils” that incorporate independently all power circuitry for spark generation.

The module structure (Figure 62) is very similar to the injection one and it exploits also the same phasing and timing mechanisms such as the trigger for the ignition sequence activation and the structure of the sequence itself. The main difference is in the nature of the actuation and in the precision requested for the SA (Spark Advance): the coil has to be charged and then the discharge (and so the spark generation) happens on the falling edge of the control command that must be referred with the maximum angular precision (Figure 63). Because of the development of the angle extrapolation module this task has been accomplished simply as an angle to angle actuation.

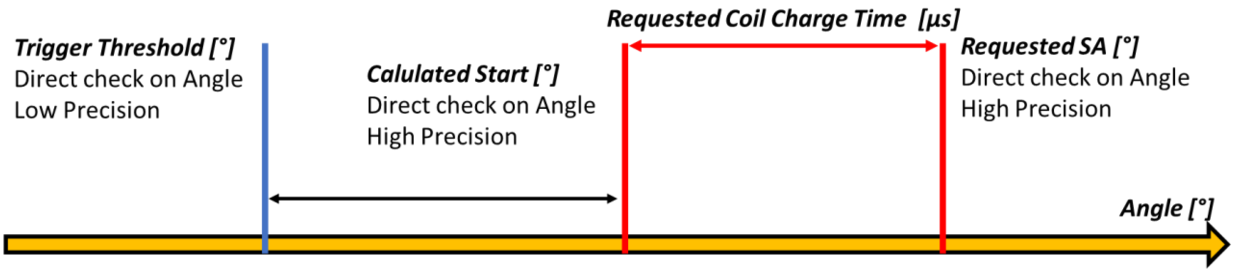


Figure 63: Angle based Trigger search and Angle to Angle Ignition command actuation

On the user side the drivers expect the specification of the following parameters to be converted into FPGA instructions:

- The number of the selected ignition coil
- Coil charge time [µs]
- The SA [°BTDC]
- Average Cycle Speed [°/s] → needed to convert the coil charge time in a calculated start charge angle.
- The TDC [°] on which the SA is referred

Even if not yet implemented, a multi-spark per cycle solution could be rapidly developed applying a for loop similarly as in the multi-injection sub-module. In Figure 64 an example of ignition command is shown on the oscilloscope together with injection command, crank signal and cylinder pressure.

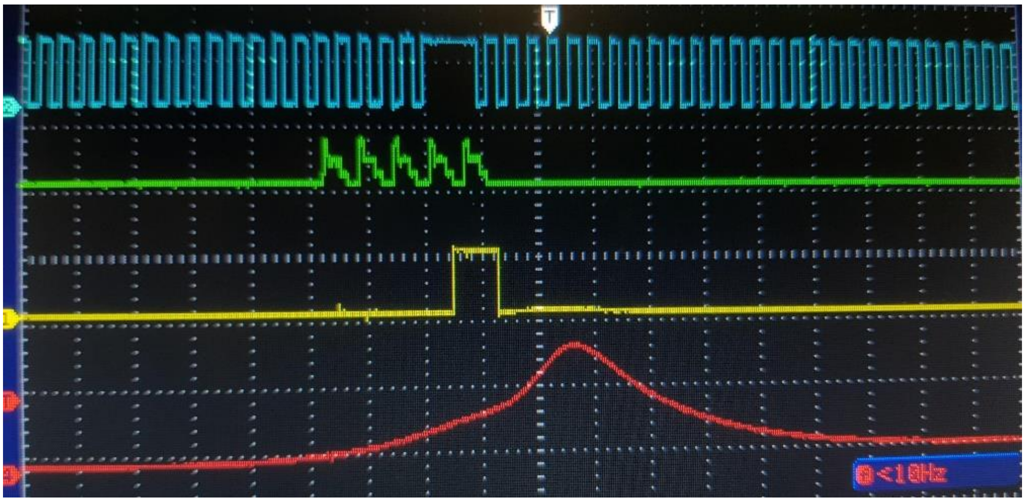
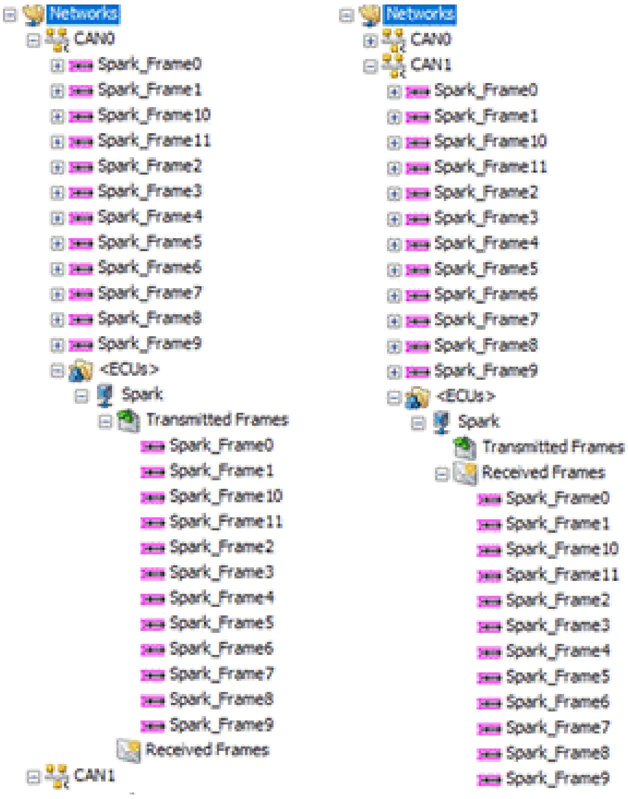


Figure 64: Injection (green), Ignition Digital Command (yellow) and cylinder pressure (red) referred on the crank signal (blue) in a oscilloscope snapshot

3.2.9 ***CAN***

A CAN communication module has been developed with the objective of providing a fast configuration and an easy integration of the data sent and received through the bus. The library giving access to CAN I/O of the hardware is a standard NI module, but is intended just for the direct writing and reading of raw CAN frames: any high-level functionality, such as the usage of a database for encoding and decoding the signals, is not included. For this reason, a flexible framework has been implemented, integrating high-level functions blocks in a prebuilt architecture capable of adapting to the communication requirements with the minimum programming effort. To permit the usage of a CAN database and the automatic conversion of the raw frames into signals the NI X-NET library has been integrated in the framework. The library has been structured for an object-like programming, with a CAN object for each physical CAN collecting all the information and the parameters necessary for the correct working:

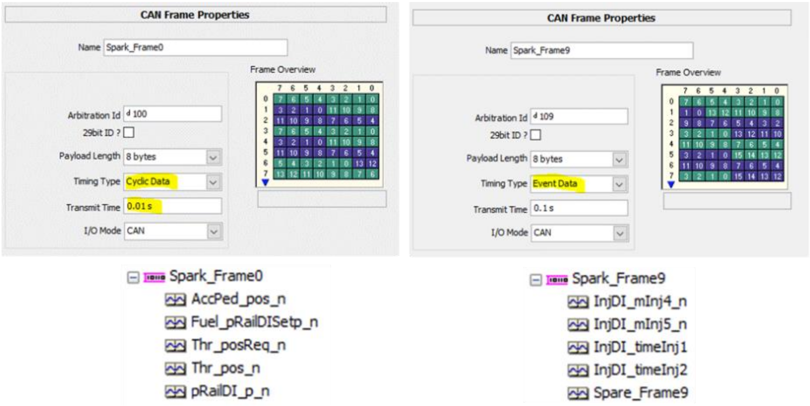
1. The reference of the CAN interface selected
2. CAN Enable / Disable
3. The state of the communication
4. The conversion session for the read signals (all the information to convert the CAN frames to signals)
5. The read queue reference
6. The conversion session for the cyclic write signals (all the information to convert the signals to CAN frames)
7. The conversion session for the event write signals (all the information to convert the signals to CAN frames)
8. The write queue reference



- Networks
  - CAN0
    - Spark_Frame0
    - Spark_Frame1
    - Spark_Frame10
    - Spark_Frame11
    - Spark_Frame2
    - Spark_Frame3
    - Spark_Frame4
    - Spark_Frame5
    - Spark_Frame6
    - Spark_Frame7
    - Spark_Frame8
    - Spark_Frame9
    - <ECUs>
      - Spark
        - Transmitted Frames
          - Spark_Frame0
          - Spark_Frame1
          - Spark_Frame10
          - Spark_Frame11
          - Spark_Frame2
          - Spark_Frame3
          - Spark_Frame4
          - Spark_Frame5
          - Spark_Frame6
          - Spark_Frame7
          - Spark_Frame8
          - Spark_Frame9
        - Received Frames
  - CAN1

- Networks
  - CAN0
  - CAN1
    - Spark_Frame0
    - Spark_Frame1
    - Spark_Frame10
    - Spark_Frame11
    - Spark_Frame2
    - Spark_Frame3
    - Spark_Frame4
    - Spark_Frame5
    - Spark_Frame6
    - Spark_Frame7
    - Spark_Frame8
    - Spark_Frame9
    - <ECUs>
      - Spark
        - Transmitted Frames
        - Received Frames
          - Spark_Frame0
          - Spark_Frame1
          - Spark_Frame10
          - Spark_Frame11
          - Spark_Frame2
          - Spark_Frame3
          - Spark_Frame4
          - Spark_Frame5
          - Spark_Frame6
          - Spark_Frame7
          - Spark_Frame8
          - Spark_Frame9

Figure 65: CAN Database example for loopback test



CAN Frame Properties

Name: Spark_Frame0

Arbitration Id: d 100

29bit ID ? □

Payload Length: 8 bytes

Timing Type: Cyclic Data

Transmit Time: 0.01 s

I/O Mode: CAN

Frame Overview

	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	3	2	1	0	11	10	9	8
2	11	10	9	8	7	6	5	4
3	7	6	5	4	3	2	1	0
4	3	2	1	0	11	10	9	8
5	11	10	9	8	7	6	5	4
6	5	4	3	2	1	0	13	12
7	13	12	11	10	9	8	7	6

- Spark_Frame0
  - AccPed_pos_n
  - Fuel_pRailDISetp_n
  - Thr_posReq_n
  - Thr_pos_n
  - pRailDI_p_n

CAN Frame Properties

Name: Spark_Frame9

Arbitration Id: d 109

29bit ID ? □

Payload Length: 8 bytes

Timing Type: Event Data

Transmit Time: 0.1 s

I/O Mode: CAN

Frame Overview

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

- Spark_Frame9
  - InjDI_mInj4_n
  - InjDI_mInj5_n
  - InjDI_timeInj1
  - InjDI_timeInj2
  - Spare_Frame9

Figure 66: Definition of a cycling frame and an event based one

The CAN database is the core of the automatic encoding to and decoding from raw frames and the module has been developed in order to extract and use the maximum quantity of information from it:

- The definition of the CAN lines as “clusters”. For the SoM based hardware like M2 or Spark the software expects 2 lines named CAN0 and CAN1
- The list of the frames for each CAN line.
- For each frame the definition of:

- ○ The ID number of the package
- ○ The frame transmission timing: **cyclic** or **event** based. In case of cyclic transmission, the transmit time has to be properly specified as a multiple of the base frequency of the CAN module (default 100Hz).
- ○ The signals encoded in the frame: bits number, position, conversion gain and offset

Read and written elements must be included in the database ECU Specification, correctly importing the frames under "Transmitted Frames" and "Received Frames". In Figure 65 and Figure 66 an example of X-NET database for loop-back test between CAN0 (only writing in this case) and CAN1 (only reading in this case).

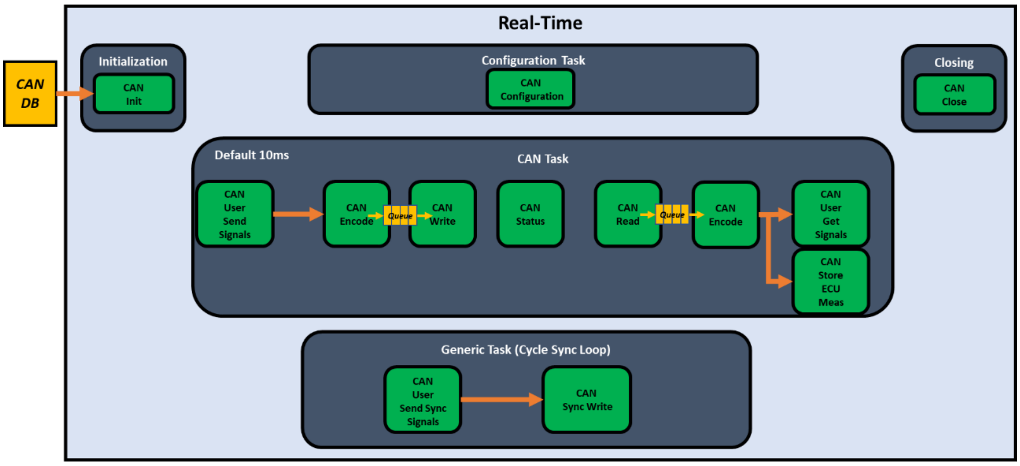


Figure 67: Real-Time CAN Module functional scheme

The module architecture (Figure 67, Figure 68) operates his main methods (functions) on the CAN object as specified below:

- **CAN Initialization**: initializes the CAN hardware interface and creates the conversion sessions based on the selected CAN database
- **CAN Closing**: closes the can hardware interface and all the conversion sessions.
- **CAN Configuration**: sets the CAN terminations state.
- **CAN State**: returns the state of the CAN transceiver and in case of errors resets the interface.
- **CAN Frame Encode**: converts the list of the selected cyclic signals to be written over CAN into raw frames and sends them into the transmission queue. The "encoding" is done accordingly to the description of the signals in the CAN database. The cyclic frames are transmitted automatically at the frequency specified by means of a decimation algorithm.

- **CAN Write**: send over CAN all the raw frames in the transmission queue
- **CAN Write Sync**: converts the list of the selected signal to be written over CAN into raw frames and transmits them synchronously with the call of the “Write Sync” function itself. This feature allows to have information transmitted for example once per cycle or once per combustion and not only at a fixed time rate. The “encoding” is done according to the description of the signals in the CAN database.
- **CAN Frame Read**: reads all the frames available from the low-level CAN driver and sends to conversion through the read queue.
- **CAN Decode**: converts all the read frames into meaningful signals. The “decoding” is done according to the description of the signals in the CAN database.
- **CAN Store to ECU Meas**: the read signals are automatically allocated in an array in the ECU Measurements and their variable names are changed in the A2L file according to the naming in the CAN database. This means that if a signal is needed just for visualization or recording through the calibration interface only a database modification is needed.

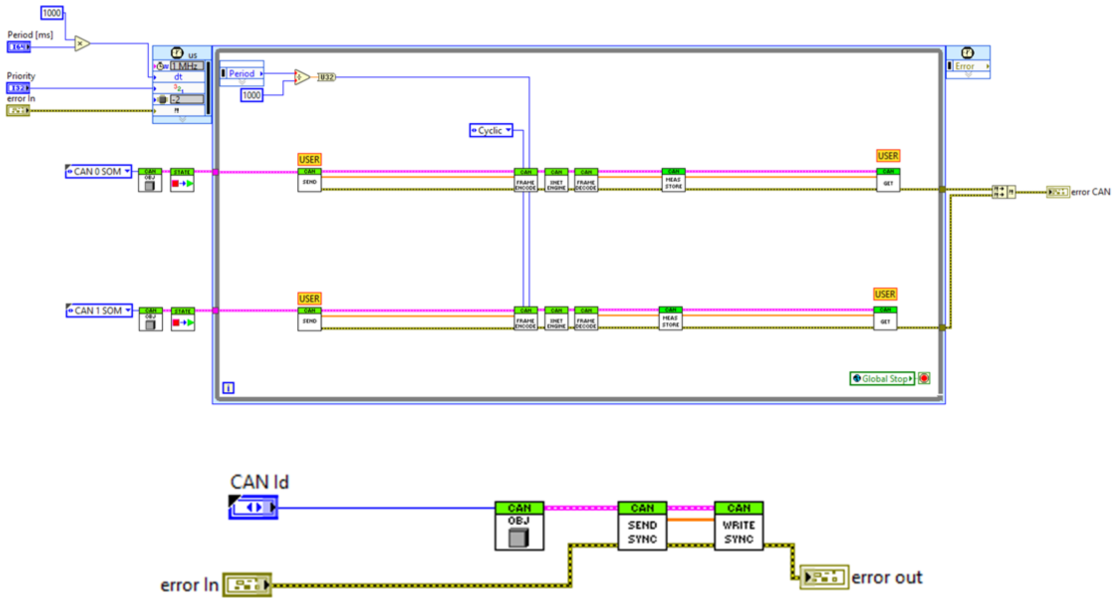


Figure 68: Real-Time CAN module code implementation for cyclic and event receiving and transmission

The only operation that the developer needs to do to make operative the CAN communication, a part from the definition of the database, is the linking of the actual ECU variables to transmitted and received signals (Figure 69). For the transmitted ones, both for cyclic and event data, an array of elements has to be built with the signal order of the database and simply connected to the CAN Write function. This is done extracting the data from the ECU variables of interest. For the received signals an array is returned for each iteration by the CAN Read function and each value can be searched by name or index, according to the database definition, and connected to the proper ECU variable. This allows the for using the signals as inputs of the control software.

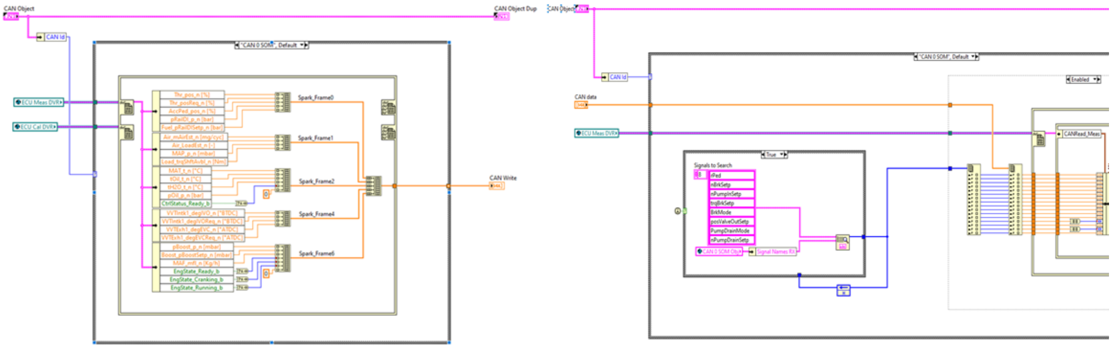


Figure 69: Examples of transmission array building and extraction of signals from the received array.

3.2.10 *XCP library integration*

The XCP library, available from previous projects, has been integrated in the framework of the software in order to require no programming action to achieve a working communication. This is a strong requirement since parameters calibration and measurements visualization and recording through standard XCP tools are essential functions for an RCP system, that for his nature must be developed quickly. The integration with the general software framework exploits the ECU Cal and ECU Meas clusters and their memory allocation in the DVR. This means that all the variables included in these clusters are automatically available to the XCP communication, leading to a good flexibility and scalability in case of modification. The functional example scheme is presented in Figure 70.

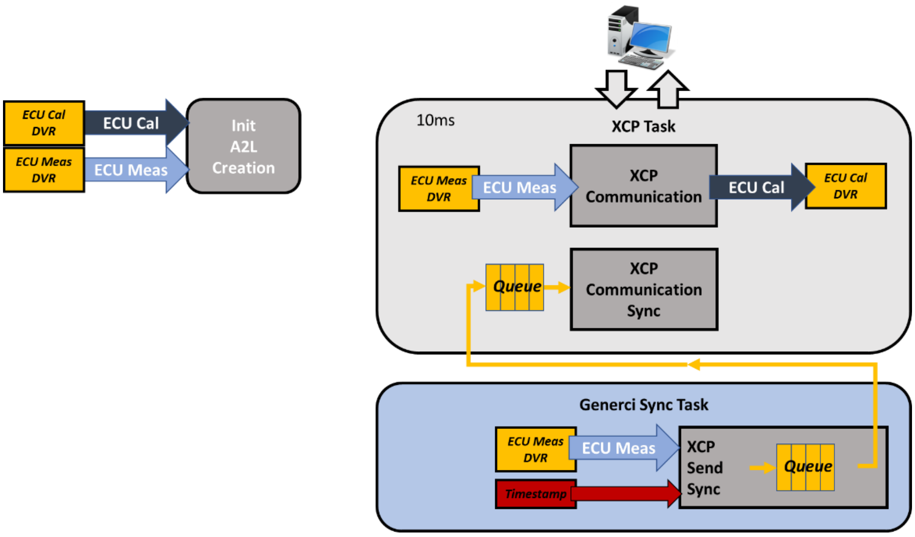


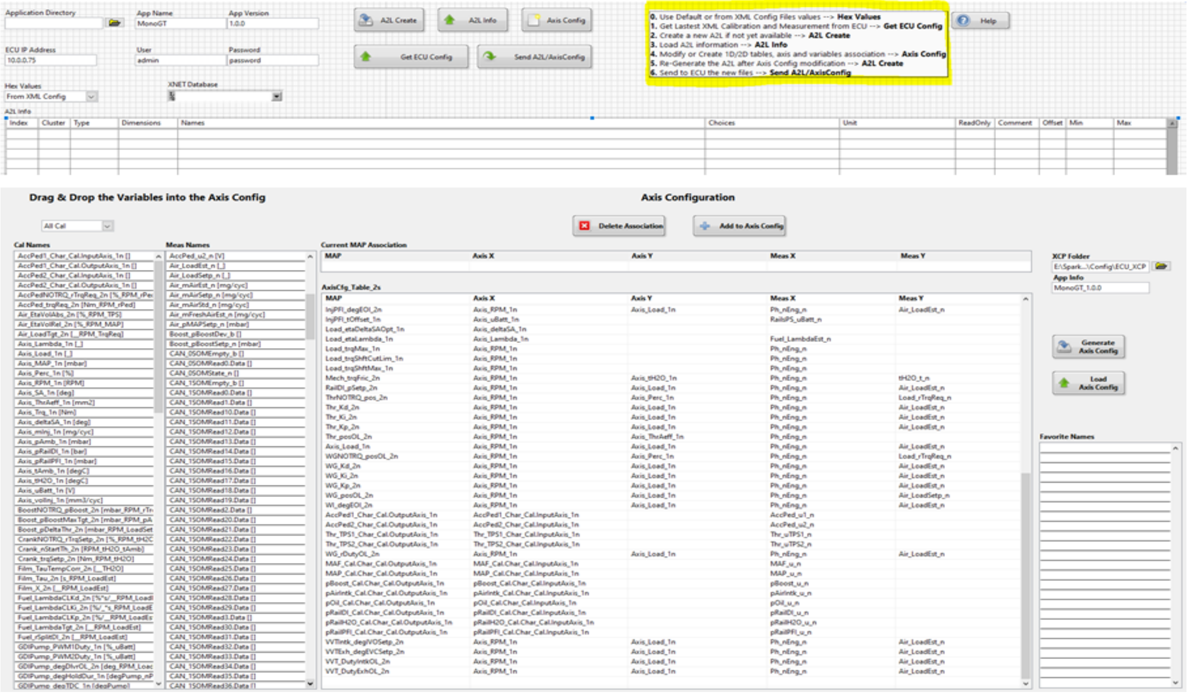
Figure 70: Real-Time XCP Communication module functional scheme

The XCP module is divided into two different sub-modules:

- The XCP library, that operates the communication with the host PC of the calibration tool
- The A2L and Hex file generation library, that converts the LabView variables (ECU Cal and ECU Meas clusters) into files used to define the communication through XCP

The LabView clusters containing the calibration parameters and the measurement variables are used during the initialization to define the A2L file, that describes all the variables interested by the communication, and the HEX files describing the numeric content of the variables. Once uploaded to the host PC (XCP master) calibration tool, these files will allow the XCP communication with the embedded target (XCP slave). An automatic procedure extracts the variables and compiles the A2L file with the instructions that defines the communication protocol and the variables information. A wide set of checks and operations are recursively done and have been improved to allow a correct A2L generation: naming univocity of groups and variables, permitted characters and logic association between maps, axis and variables. Moreover, a specific function block has been developed to integrate the CAN DB specification into the A2L definition: as described in the CAN module paragraph the signals are stored in an array structure in the ECU Meas cluster, these signals are renamed in the A2L file from their generic array names to the names specified in the database. This allows a meaningful visualization and logging of these variables in the calibration tool.

All these operations are accomplished automatically at the boot on the real-time target, but also an offline tool has been developed (Figure 71: Host A2L generation tool interface overview). This tool is dedicated to the association between the maps, the axis and the variables of inputs, the visualization of the information included in the generated A2L, the specification of the used CAN database and the automatic retrieve and inclusion of the LabView variables description (it can be done only if the a2l is generated on a windows machine). Moreover, this tool is capable of retrieving the needed files from the ECU for the A2L and HEX generation and also deploy the generated files back to the ECU.



Application Directory

App Name: MonoGT

App Version: 1.0.0

ECU IP Address: 10.0.0.75

User: admin

Password: password

Hex Values: From XML Config

XNET Database

0. Use Default or from XML Config Files values --> **Hex Values**
1. Get Lastest XML Calibration and Measurement from ECU --> **Get ECU Config**
2. Create a new A2L if not yet available --> **A2L Create**
3. Load A2L information --> **A2L Info**
4. Modify or Create 1D/2D tables, axis and variables association --> **Axis Config**
5. Re-Generate the A2L after Axis Config modification --> **A2L Create**
6. Send to ECU the new files --> **Send A2L/AxisConfig**

A2L Info

Index	Cluster	Type	Dimensions	Names	Choices	Unit	ReadOnly	Comment	Offset	Min	Max

Drag & Drop the Variables into the Axis Config

All Cal

Cal Names	Meas Names
AccPed1_Char_Cal.InputAxis_1n []	AccPed_u2_n [V]
AccPed1_Char_Cal.OutputAxis_1n []	Air_LoadEst_n [_]
AccPed2_Char_Cal.InputAxis_1n []	Air_LoadSetp_n [_]
AccPed2_Char_Cal.OutputAxis_1n []	Air_mAirEst_n [mg/cyc]
AccPedNOTRQ_rTrqReq_2n [%_RPM_rPe	Air_mAirSetp_n [mg/cyc]
AccPed_trqReq_2n [Nm_RPM_rPed]	Air_mAirStd_n [mg/cyc]
Air_EtaVolAbs_2n [%_RPM_TPS]	Air_mFreshAirEst_n [mg/cyc]
Air_EtaVolRel_2n [%_RPM_MAP]	Air_pMAPSetp_n [mbar]
Air_LoadTgt_2n [__RPM_TrqReq]	Boost_pBoostDev_b []
Axis_Lambda_1n [_]	Boost_pBoostSetp_n [mbar]
Axis_Load_1n [_]	CAN_0SOMEmpty_b []
Axis_MAP_1n [mbar]	CAN_0SOMRead0.Data []
Axis_Perc_1n [%]	CAN_0SOMState_n []
Axis_RPM_1n [RPM]	CAN_1SOMEmpty_b []
Axis_SA_1n [deg]	CAN_1SOMRead0.Data []
Axis_ThrAeff_1n [mm2]	CAN_1SOMRead1.Data []
Axis_Trq_1n [Nm]	CAN_1SOMRead10.Data []
Axis_deltaSA_1n [deg]	CAN_1SOMRead11.Data []
Axis_mInj_1n [mg/cyc]	CAN_1SOMRead12.Data []
Axis_pAmb_1n [mbar]	CAN_1SOMRead13.Data []
Axis_pRailDI_1n [bar]	CAN_1SOMRead14.Data []
Axis_pRailPFI_1n [mbar]	CAN_1SOMRead15.Data []
Axis_tAmb_1n [degC]	CAN_1SOMRead16.Data []
Axis_tH2O_1n [degC]	CAN_1SOMRead17.Data []
Axis_uBatt_1n [V]	CAN_1SOMRead18.Data []
Axis_volInj_1n [mm3/cyc]	CAN_1SOMRead19.Data []
BoostNOTRQ_pBoost_2n [mbar_RPM_rTr	CAN_1SOMRead2.Data []
Boost_pBoostMaxTgt_2n [mbar_RPM_pA	CAN_1SOMRead20.Data []
Boost_pDeltaThr_2n [mbar_RPM_LoadSet	CAN_1SOMRead21.Data []
CrankNOTRQ_rTrqSetp_2n [%_RPM_tH2C	CAN_1SOMRead22.Data []
Crank_nStartTh_2n [RPM_tH2O_tAmb]	CAN_1SOMRead23.Data []
Crank_trqSetp_2n [Nm_RPM_tH2O]	CAN_1SOMRead24.Data []
Film_TauTempCorr_2n [__TH2O]	CAN_1SOMRead25.Data []
Film_Tau_2n [s_RPM_LoadEst]	CAN_1SOMRead26.Data []
Film_X_2n [__RPM_LoadEst]	CAN_1SOMRead27.Data []
Fuel_LambdaCLKd_2n [%*s/__RPM_Loadl	CAN_1SOMRead28.Data []
Fuel_LambdaCLKi_2n [%/_*s_RPM_LoadE	CAN_1SOMRead29.Data []
Fuel_LambdaCLKp_2n [%/__RPM_LoadEs	CAN_1SOMRead3.Data []
Fuel_LambdaTgt_2n [__RPM_LoadEst]	CAN_1SOMRead30.Data []
Fuel_rSplitDI_2n [__RPM_LoadEst]	CAN_1SOMRead31.Data []
GDIPump_PWM1Duty_1n [%_uBatt]	CAN_1SOMRead32.Data []
GDIPump_PWM2Duty_1n [%_uBatt]	CAN_1SOMRead33.Data []
GDIPump_degDlvrOL_2n [deg_RPM_Loac	CAN_1SOMRead34.Data []
GDIPump_degHoldDur_1n [degPump_nP	CAN_1SOMRead35.Data []
GDIPump_degTDC_1n [degPump]	CAN_1SOMRead36.Data []

Axis Configuration

Current MAP Association

MAP	Axis X	Axis Y	Meas X	Meas Y

AxisCfg_Table_2s

MAP	Axis X	Axis Y	Meas X	Meas Y
InjPFI_degEOI_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
InjPFI_tOffset_1n	Axis_uBatt_1n		RailsPS_uBatt_n	
Load_etaDeltaSAOpt_1n	Axis_deltaSA_1n			
Load_etaLambda_1n	Axis_Lambda_1n		Fuel_LambdaEst_n	
Load_trqMax_1n	Axis_RPM_1n		Ph_nEng_n	
Load_trqShftCutLim_1n	Axis_RPM_1n		Ph_nEng_n	
Load_trqShftMax_1n	Axis_RPM_1n		Ph_nEng_n	
Mech_trqFric_2n	Axis_RPM_1n	Axis_tH2O_1n	Ph_nEng_n	tH2O_t_n
RailDI_pSetp_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
ThrNOTRQ_pos_2n	Axis_RPM_1n	Axis_Perc_1n	Ph_nEng_n	Load_rTrqReq_n
Thr_Kd_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
Thr_Ki_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
Thr_Kp_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
Thr_posOL_2n	Axis_RPM_1n	Axis_ThrAeff_1n	Ph_nEng_n	
Axis_Load_1n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
WGNOTRQ_posOL_2n	Axis_RPM_1n	Axis_Perc_1n	Ph_nEng_n	Load_rTrqReq_n
WG_Kd_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
WG_Ki_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
WG_Kp_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
WG_posOL_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadSetp_n
WI_degEOI_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
AccPed1_Char_Cal.OutputAxis_1n	AccPed1_Char_Cal.InputAxis_1n		AccPed_u1_n	
AccPed2_Char_Cal.OutputAxis_1n	AccPed2_Char_Cal.InputAxis_1n		AccPed_u2_n	
Thr_TPS1_Char_Cal.OutputAxis_1n	Thr_TPS1_Char_Cal.InputAxis_1n		Thr_uTPS1_n	
Thr_TPS2_Char_Cal.OutputAxis_1n	Thr_TPS2_Char_Cal.InputAxis_1n		Thr_uTPS2_n	
WG_rDutyOL_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
MAF_Cal.Char_Cal.OutputAxis_1n	MAF_Cal.Char_Cal.InputAxis_1n		MAF_u_n	
MAP_Cal.Char_Cal.OutputAxis_1n	MAP_Cal.Char_Cal.InputAxis_1n		MAP_u_n	
pBoost_Cal.Char_Cal.OutputAxis_1n	pBoost_Cal.Char_Cal.InputAxis_1n		pBoost_u_n	
pAirIntk_Cal.Char_Cal.OutputAxis_1n	pAirIntk_Cal.Char_Cal.InputAxis_1n		pAirIntk_u_n	
pOil_Cal.Char_Cal.OutputAxis_1n	pOil_Cal.Char_Cal.InputAxis_1n		pOil_u_n	
pRailDI_Cal.Char_Cal.OutputAxis_1n	pRailDI_Cal.Char_Cal.InputAxis_1n		pRailDI_u_n	
pRailH2O_Cal.Char_Cal.OutputAxis_1n	pRailH2O_Cal.Char_Cal.InputAxis_1n		pRailH2O_u_n	
pRailPFI_Cal.Char_Cal.OutputAxis_1n	pRailPFI_Cal.Char_Cal.InputAxis_1n		pRailPFI_u_n	
VVTIntk_degIVOSetp_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
VVTExh_degEVCSetp_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
VVT_DutyIntkOL_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n
VVT_DutyExhOL_2n	Axis_RPM_1n	Axis_Load_1n	Ph_nEng_n	Air_LoadEst_n

XCP Folder: E:\Spark...\Config\ECU_XCP

App Info: MonoGT_1.0.0

Favorite Names

Figure 71: Host A2L generation tool interface overview

On the other side, while the control software is running the communication tasks are carried by the XCP library. The main XCP loop (10 ms) is demanded to cyclically send the measurement data, retrieved form the Meas DVR, and update the calibration parameters in the Cal DVR. To reduce the cpu usage the calibration, overriding with the new values is accomplished only if an update event has occurred over the XCP. In parallel in a generic task, for example the cycle synchronous loop, measurement data can be sent in a queue together with a timestamp, allowing to enable the visualization and recording of the data also at a rate that is not cyclic.

3.2.11 *Data Recording*

Even if the most frequent usage of an RCP is at the test bench or in any case with the calibration software connected, ready to log the data directly on the host computer, an internal data recorder is useful to cover all the other situation where logging data on the embedded memory is required: on-board applications, fast logging for functionality debugging and anomalous event logging for example. For this reason, a modular data recorder with the following characteristics has been implemented:

1. Data recording in TDMS format: it is a binary National Instruments native file format, easily readable and writable form NI application, custom LabView applications, Microsoft excel and also other high-level programming platform such as Matlab.
2. Maximum recording sample rate in the order of the ms: the logger is intended for the control environment and not for the high frequency data acquisition
3. Possibility to log multiple groups of data with different sample rates. Each group has its own independent definition with a name and all the names of the channels included.
4. The logging groups definition and data sending are independent from the main logging framework: once the group is locally defined and the data sent to REC, no other SW modification must be done.
5. An event triggered acquisition submodule is included in the library. It allows the recording of files collecting data from a predefined amount of time before and after the event. The main usage is for alarms detection recordings.

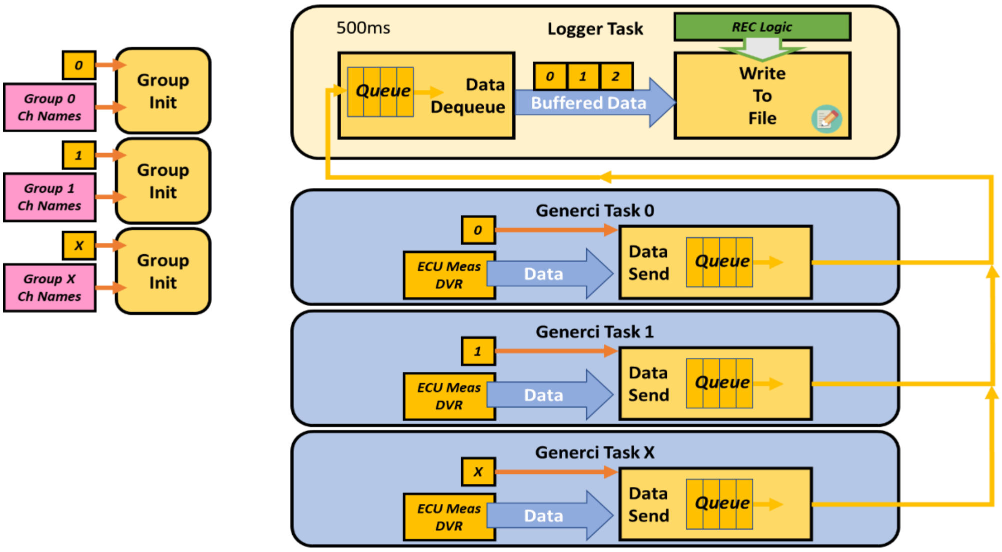


Figure 72: Real-Time Data Logger functional scheme

As shown in the functional scheme (Figure 72) each recording group must be simply defined with an identifier number and a list of channel names through a dedicated VI. Then in a generic loop (periodic or not), an array of data whose elements represent a single point recording for each channel, are sent along with the group identifier number in the recording queue with a specific function. The elements of this queue are then collected, automatically divided and saved by group according with the identifier number in a slow loop. Since the data collection is decoupled from the writing to file with a queue based on identifiable packets, the logging frequency is free and independent for each group.

For the event based recording a similar and independent software structure is provided. The differences are the trigger mechanism, accomplished as an asynchronous notification reporting the event name to be included in the file, and the data transfer mechanism, that exploits the depth of the transfer queue as a rolling buffer for storing the last X elements, with X configurable by the user. The recording is triggerable form multiple points in the software and it is automatically stopped after Y seconds after the event, with Y configurable.

System Event Logger

Additionally, a library for logging basic information is included in the base modules of the framework and is used for reporting system events, errors or warnings. Each day a new txt file is created and each time an event is notified through the dedicated function a new text line with timestamp, type of event and comment (or automatic error) is added. This logger is particularly useful to report system errors or problems such missing files, base drivers faults etc. The application developer can add its own events to the file simply recalling the dedicated functions.

3.2.12 *Sensor Acquisition and Linearization*

Since acquiring sensors is the first task demanded to a control system, a standard Real-Time library for the most common use case has been built, in order to avoid the rewriting of custom, but almost identical, code for each sensor (Figure 73). The requirements included a standard and flexible input and output variables interface, a filtering section, a linearization section and an error checking section. The proposed library includes all these functionalities as sub-functions with the possibility of calibration and configuration for the user.

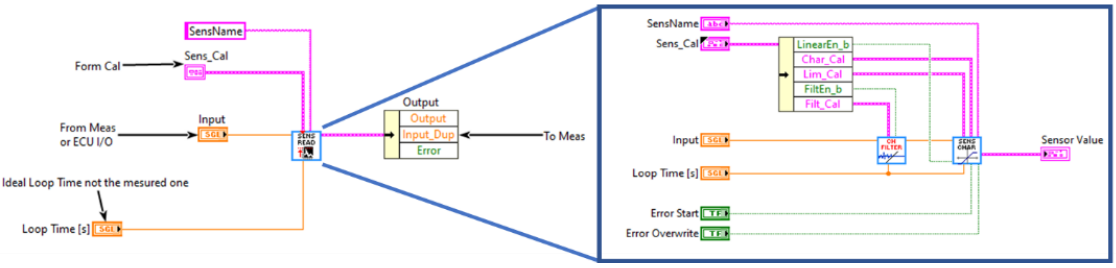


Figure 73: Sensor acquisition and linearization code overview

Input: the input to the sensor read block is a simple single precision variable that can be connected to any data source from the ECU Inputs (AI fast or slow, PWM read, temperature read etc.) or from the ECU measurement cluster (for example CAN data). The input can be a raw electrical value or an already converted physical value. The sensor name, the loop rate and the sensor calibration parameters (from the ECU cal) are passed as well into the block as inputs.

Filter: a configurable filter is then applied to the input signal, the options available are:

1. No filter
2. Moving average on the last X samples, with X configurable
3. Median filter on the last X samples, with X configurable
4. Butterworth filter configurable as low, high or bandpass filter with order and cut-off frequency selectable by the user

The filter runs at the frequency of the loop where sensor reading is called, typically 500 or 250 Hz.

Linearization: this function uses the filter output as input of a user configurable look-up table that converts the data from raw to engineering unit. The lookup table can be bypassed if the input value is yet in engineering unit as for CAN values or temperature channels reading.

Error: together with the linearization a boundary check is applied to the input signal and in case of an out-of-range value (high limit and low limit on the input signal) the error function is triggered. The error flag for the sensor is reported in case of permanence out of the limits for a user defined amount of time. In the same way, the error cleaning happens after a user defined amount of time back in the boundaries. When the error flag is high the sensor output can be saturated from the LUT or set to a default value. Additionally, when the error flag is raised or erased a message is added to the log file including the name of the sensor.

Output: the output of the block collects the linearized output, the duplicated input and the error flag, made available to the whole software updating them in the ECU measurements data store.

3.2.13 ***IRQs from FPGA***

Exploiting the angle-based phasing system a module for angle-based interrupts from FPGA has been developed, giving the possibility to carry out calculation on the Real-Time synchronized (for example) with the cylinder TDCs and not simply executed at a fixed frequency. Cycle-based and cylinder by cylinder calculations are important to allow cycle-based strategies and save CPU time when not running at high RPMs. The library has a standard initialization, configuration and closing structure both on RT and FPGA side, and it is possible to set up to 8 interrupt per cycle (720°) fixed on custom angles (Figure 74). The typical usage is the cycle triggering of fuel and ignition computing setting the IRQs angles in advance of an arbitrary offset compared to the TDCs of the cylinders.

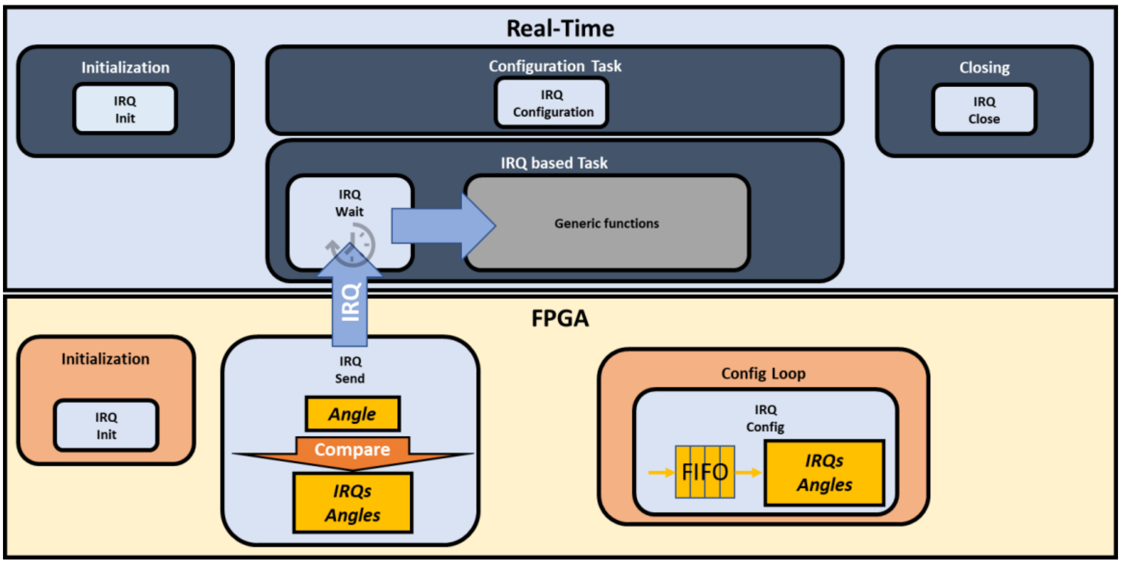


Figure 74: Real-Time and FPGA IRQs functional scheme

3.2.14 ***Accessories modules and libraries***

A variety of minor and auxiliary modules are included to complete the framework functionality for example the library that reads the inertial metering unit (IMU), the GPS coordinates, the temperatures and lambda inputs from the co-processor, the function set that is used to enable the supplies provided by the ECU board, the reading of ECU status and of the integrated pressure and temperature sensor. Together with the modules related to the HW I/O a set of libraries for common and frequently reused functions has been integrated in the architecture: PID functions with advanced functionalities, a LUT library, counters, boolean logic, timing functions etc. Additionally, for debug an acquisition purpose a simple oscilloscope has been built. It allows the real-time visualization on graphs and the recording at high and low frequency of all the inputs of the ECU and of debug internal variables form the FPGA:

- Fast AI channels from FPGA up to 200 kHz
- Slow AI channels (if available) up to 1kHz
- GPIO raw reading from FPGA up to several 10MHz

- Phasing algorithm information form FPGA at high frequency: angle, teeth counter, crank and cam signals etc.
- Ignition, Injection and PWM actuation manual test
- IMU, GPS, ECU status variables, etc.

Below in Figure 75 examples of real time oscilloscope interfaces are presented for angle and actuations (yellow injection and red ignition) and Inertial metering unit.

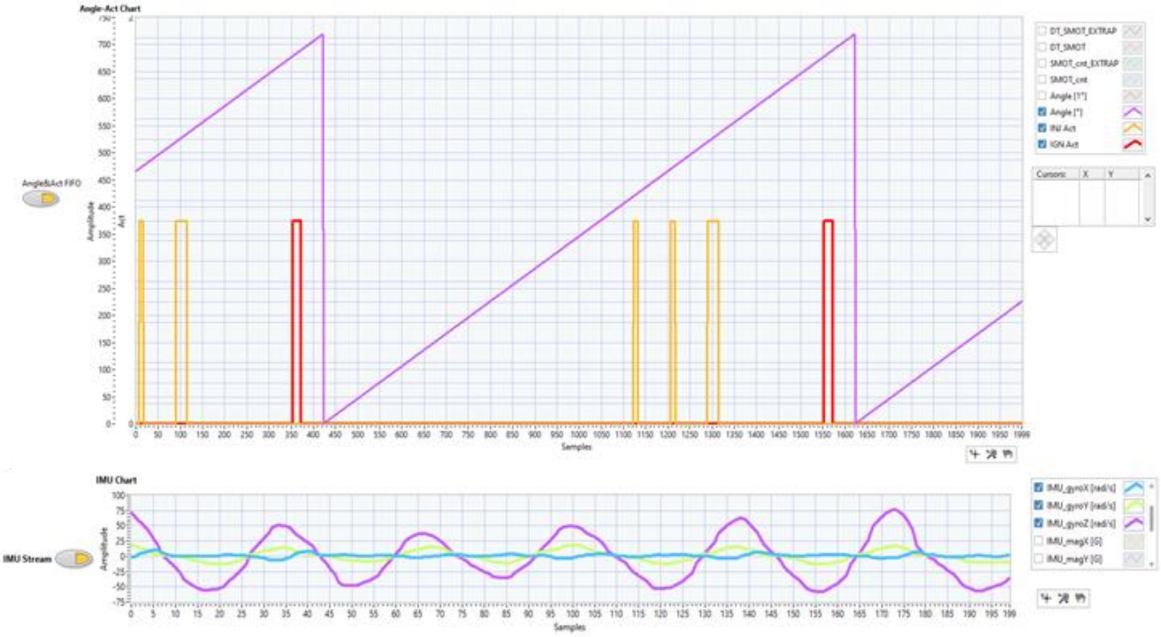


Figure 75: Angle and Actuations oscilloscope and IMU oscilloscope examples

4. RCP APPLICATIONS

4.1 SPARK IGNITION ENGINE MANAGEMENT

The first project completely based on the SW framework has been the development of a full bypass ECU for a two-cylinder turbocharged engine equipped with MultiAir system on the intake valves (Figure 76). The purpose of the activity was the investigation and the prototyping of a control system for the conversion of the engine from automotive to aeronautical use. The requirement was the capability of modifying both the control strategies and the calibration parameters, targets not achievable with the standard manufacturer ECU. Additionally, the proposed full bypass solution enhanced the interaction and integration with other devices and the test bench. Starting from that first experience a complete spark ignition engine management for testing purpose has been developed and integrated with a wide set of functionalities. This architecture and engine management has been reused in more advanced engine testing, including direct injection control and it is planned to be adopted for freely control a 4-cylinder engine turbocharged MultiAir engine.

Integrating together base and specific control functions a complete spark ignition engine management system [28] [29], characterized by a wide range of control and communication capabilities has been realized. Standard and advanced sensors can be accessed for monitoring and for feedback control and expansion is possible through CAN: for example, external lambda controllers for cylinder-cylinder AFR control or high precision flow meter and temperature sensors can be easily integrated in the control system.

A wide sensors equipment is generally the counterpart of a complex actuators layout, so standard and advanced control functions have been developed and integrated in the system. A main addition to the previous spark ignition engine management has been the possibility of implementing both PFI and DI injections: a dedicated GDI pump controller has been developed due to the need of specific actuations. Also, the air path has improved on the electronic throttle control and calibration methods and with the addition of VVT control up to 4 camshafts, both on intake and exhaust. The integration of the water injection control and the accessibility of the combustion metrics from cylinder pressure sensors directly calculated on board, reduce the need of additional RCPs and expands the built-in ready to use functions. The software can be modified through LabView for example to implement cylinder-by-cylinder and cycle-by-cycle strategies on ignition and fuel, for feedback combustion control.

Moreover, real-time calibration support and bidirectional communication with the bench environment are fundamental to make a full bypass RCP controller effective in boosting the research and shrinking the development time. The built-in flexibility of modifying the software with LabView and using the ECU not only as a controller but also as data collector, extends the standard capabilities provided by a production or a development ECU.

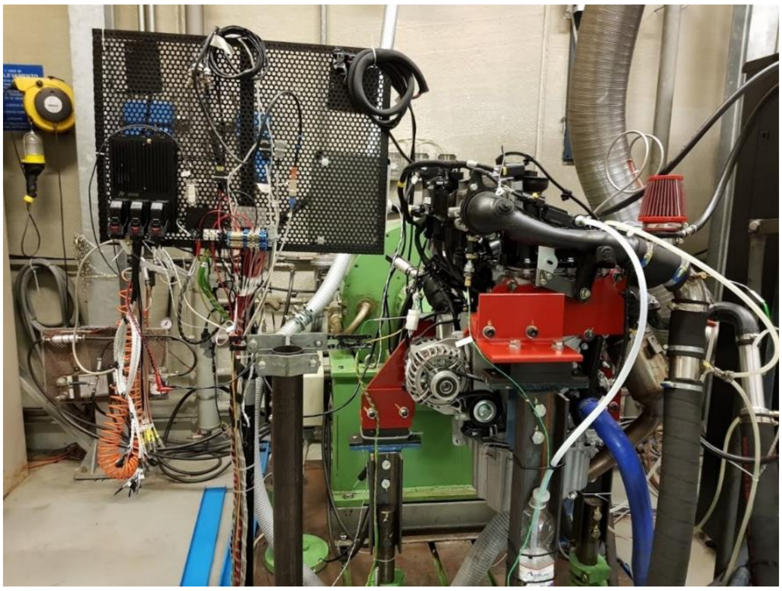


Figure 76: The two-cylinder MultiAir engine installed ate the test bench fully controlled with Spark ECU

4.1.1 ***Control software structure***

The modular framework described in chapter 3 has been fully exploited in each part to accomplish the task of creating a complete, reusable and quickly operative RCP system for spark ignition engine. The base architecture presented in Figure 24 has been integrated with the specific application functions adding them in the spaces dedicated to custom development. Excluding the standard LabVIEW functions, for the Real-Time part the software contains almost 700 functions and subfunctions, with more than 50% coming from the base architecture and libraries, while for the FPGA side the code is made of more than 200 functions with 90% of them part of the prebuilt framework. From these numbers it is clear why a set of standard libraries built in a pre-defined framework architecture is so important to boost the development of an RCP. These approximative statistics are as valid for this spark ignition management RCP as for other applications, and the more the RCP requires to be fast and simple, the more the prebuilt architecture gains importance in the total effort required for the realization: the framework allows the developer to concentrate the effort only on the application core.

The high-level engine control is the core of the application and it is built by different control subsystems governing the main engine subsystems. The variables are passed in and out in the sub-functions by means of the calibration and measurement DVRs architecture described in 3.1.1, while the generic loops, containing the control code executing at different rates, are built according to the standard described in 3.1.3. These variables are available for calibration and recording through the XCP module (3.2.10) and they can be integrated and shared though CAN (3.2.9), as well. Analog and digital sensors are acquired and made available to the application software through the ECU inputs modules (3.2.3, 3.2.5, 3.2.9, 3.2.12, 3.2.14) and the conversion form high level commands to actuators electric inputs is made simple by the output modules (3.2.4, 3.2.6, 3.2.7, 3.2.8). From the functional point of view this is represented in Figure 77, which is the declination of Figure 24 for the specific case of the spark ignition engine management.

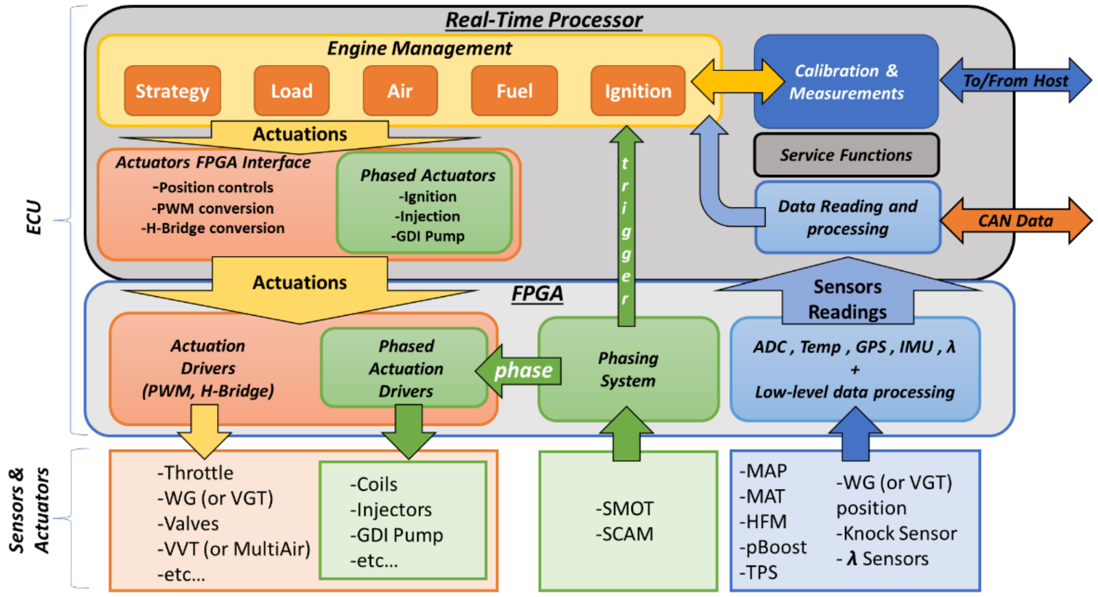


Figure 77: Spark Ignition Engine Management functional overview: developed on the Base Architecture

From the implementation point of view, the loops of some base main functions are included independently in the software, while other submodules functions, like inputs reading and actuation writing are included locally in the reding and writing sections of the proper fixed rate tasks. Similarly, the sensors linearization blocks and the low-level control of the actuators are located in tasks with a rate coherent with the purpose. The high-level engine control is mainly included in two tasks, one at fixed rate (10ms) and the other synchronized with the engine cycle. The differentiation in tasks frequency is necessary to rationalize the CPU time usage, since the computational power of an embedded platform is limited: it would be useless and CPU consuming acquiring a slow temperature sensor at the same rate of the intake pressure sensor, as well as it would be a waste writing the injectors parameters at the same high frequency of the throttle control PWM parameters. This representation is given in Figure 78 that presents the tasks overview of the software and the relative LabVIEW code implementation is shown in Figure 79. CPU core and priority specification are also included in the software. Cycle sync and faster tasks that contain critical sections are set as high priority tasks, while the CPU assignation is mainly driven by the timing affinity: fixed rate tasks are referred to the same core for an easier execution schedule, while the sync cycle task and in general aperiodic tasks are assigned to a different core. The details of all functional and implementations points of view of this specific application would be a mere list of standard strategies or a too specific declination of solutions in response to problems. This is out of the objective of this thesis work, which is the presentation of the potential of a framework for RT / FPGA based architectures for automotive RCPs.

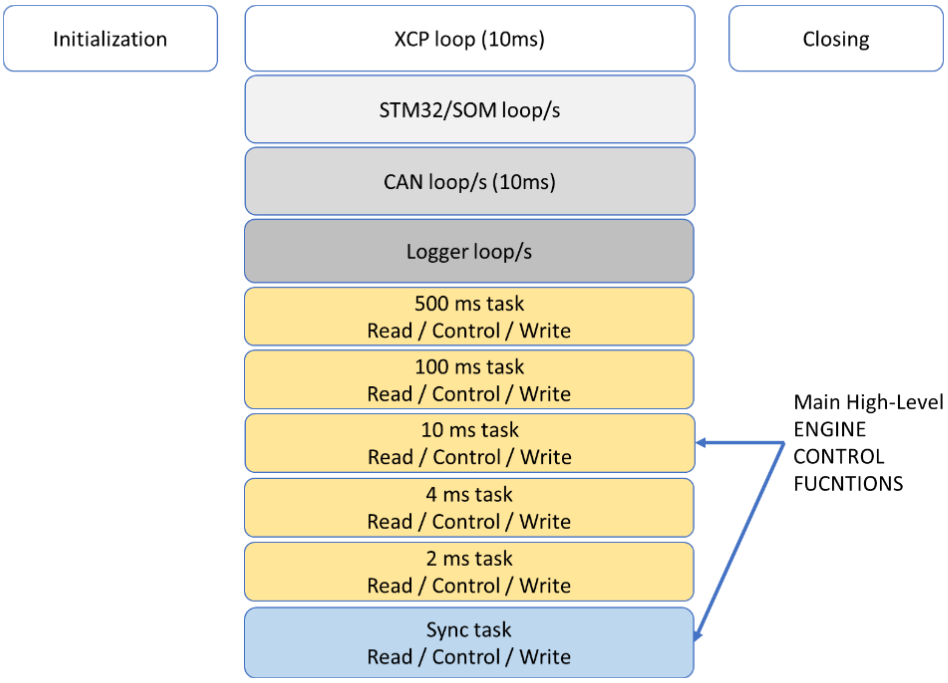


Figure 78: Tasks / Loops structure of the Spark Ignition Engine Management

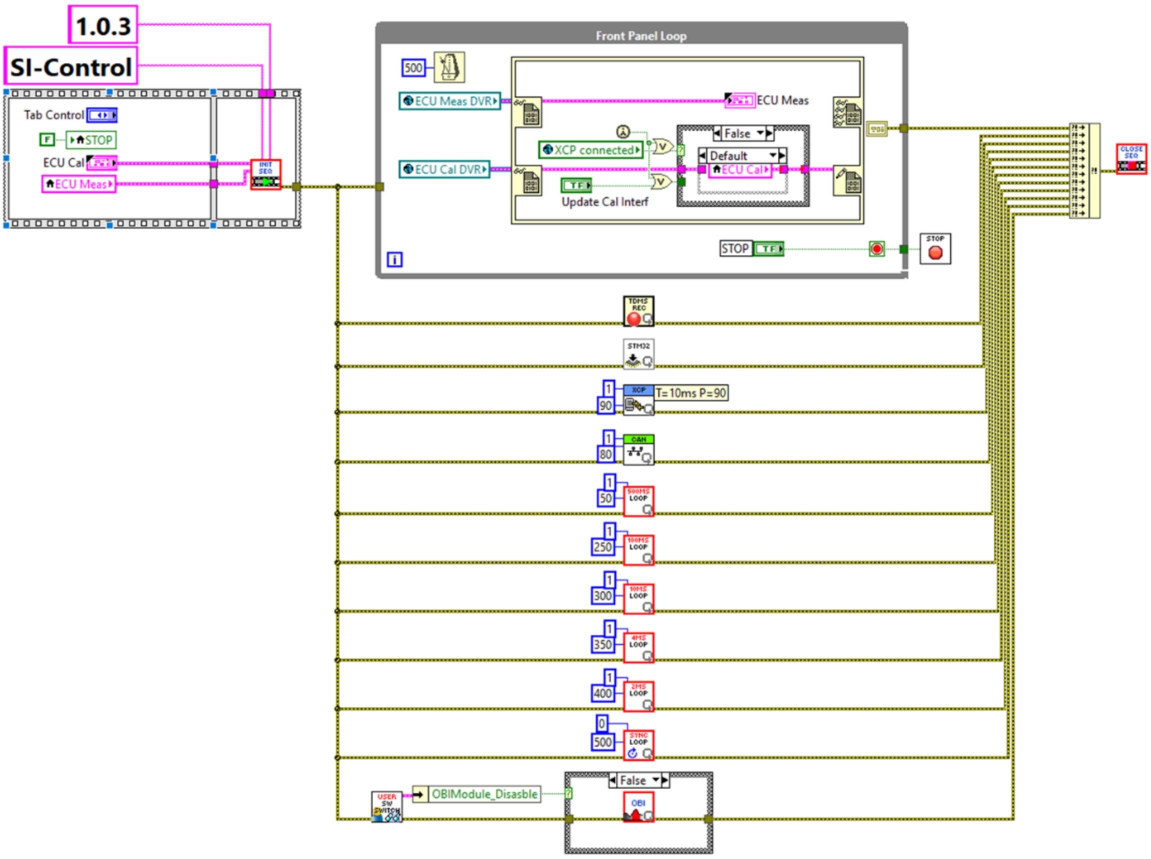


Figure 79: Spark Engine Management LabVIEW main code overview and example of the nested functions

The proposed application software is capable of full control of a modern spark ignition engine and an effective connectivity and integration with the test bench system. A standard configuration is characterized by several inputs:

- Pressure sensors, such as manifold air pressure (MAP), boost pressure, fuel and oil pressure
- Temperature sensors, such as the manifold air temperature, and the essentials water and oil temperatures
- Flow meters, for example the standard air mass flow sensor
- Lambda sensors
- Position sensors, from the pedal to all the actuators feedback such as the throttle or the waste-gate valve.
- Additional CAN sensors of any type, generally not critical for control
- Knock or cylinder pressure sensors, acquired and processed at high speed in the FPGA

On the actuators side, while engines from past generations mostly required simply PFI injection and ignition control, last generation ones involve a wider set of electrically driven actuators:

- Electric throttle body
- Variable air path actuators such as variable valve timing (VVT), variable valve lift (VVL) and MultiAir systems
- Water injectors
- DI injectors and the necessary high-pressure pump
- Electro-pneumatic or fully electric boost actuators such as waste gate valves (WG) or variable geometry turbines (VGT) and dump valves.

The sensor inputs source as well as the actuators output type are configurable to the specific use case according to the hardware capabilities. All these essential sensors and actuators are included in the standard control strategies and their variables used as inputs and outputs from the engine control function blocks (Figure 80). The strategies too, are fully customizable and adaptable the scenario of the tests that have to be faced.

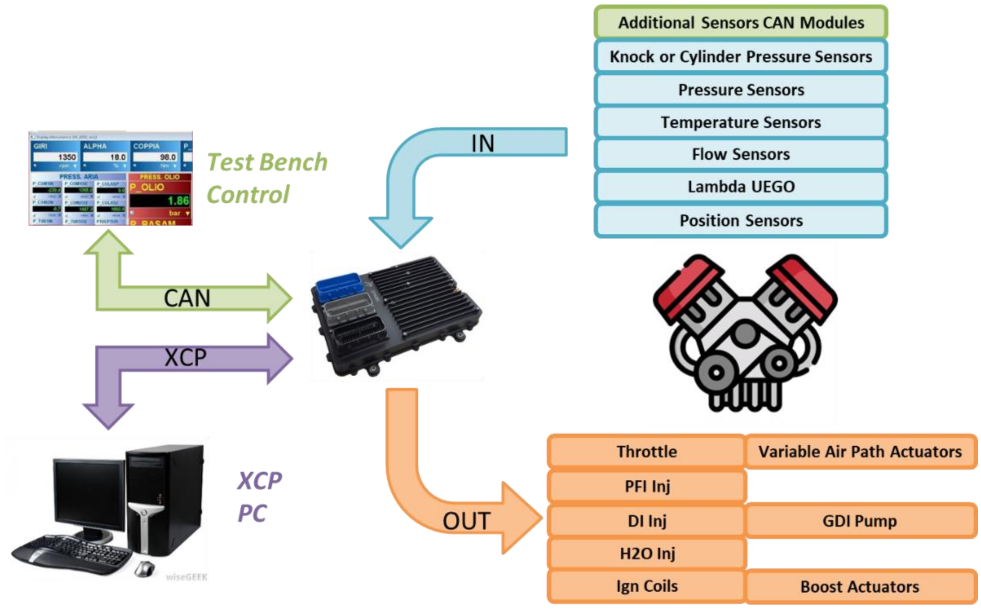


Figure 80: General Inputs / Outputs configuration for a Spark Ignition Engine Management

As previously mentioned, the high-level engine control is displaced into two main loops, one at fixed rate and the other at cycle frequency. These loops contain the management of the main engine subsystems from inputs interpretation to actuations setpoint definition such as the throttle position, the rail pressure or the boost level and the WG position. The “low-level” control of the actuators, such as the throttle PWM definition starting from the request position, or the GDI pump actuation parameters starting from the target fuel pressure, is demanded to specific functions, generally at a faster frequency (2/4ms), that receive the setpoints form the engine control. Interesting examples of actuators control will be presented in the next chapters.

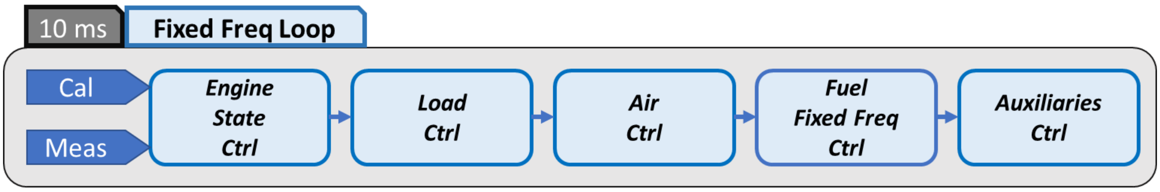


Figure 81: Functional overview of the fixed frequency Engine Control loop

The **fixed frequency loop** (Figure 81) is demanded to manage the subsystems that require control on a “fast” periodic basis: the 10ms frequency has been chosen as the standard control frequency on experience basis, but different choices are possible such as the separation between additional different rates of the modules included.

The **Engine State** subsystem is demanded to determine the current state of the engine between OFF, cranking and running. Moreover, the activation of other high-level strategies such as the warming up after cranking and the shut-off are managed by this subsystem. These engine states and strategies “flags” are responsible of different control choices in the other submodules: for example, during the cranking state air and fuel requests are managed completely independently form the driver inputs. The decision making is done using as inputs the engine speed and other sensors information such as the water and the ambient temperature.

The **Load Control** subsystem is responsible of all the engine load request, coming both from the driver or independent strategies. Two modes are available, a torque-based control and a traditional control based on direct definition of the actuators setpoints. Under torque control an indicated torque setpoint is calculated as sum of a shaft torque request and friction torque: the requested shaft torque is calculated form the pedal map, the cranking map or the idle controller (included in this module), depending on the conditions, while the friction torque derives from maps or a model. Then the requested torque is converted in a torque request for the air path (slow path) in function of the optimal torque map (torque / rpm to air) and of the efficiency calculated form the actual conditions such as the effective lambda and SA. The “traditional” load control is instead based on simple maps and strategies that give as output a percentage of requested load that will be interpreted directly by the air submodule for deciding the position of the actuators.

The **Air Control** subsystem is one of the most extended modules and covers the management of all the actuators of the air path not controlled on cycle basis (MultiAir is cycle based). A dedicated function is responsible for calculating the air mass contained in the single cylinder at standard atmospheric conditions: this mass is used as reference unit in comparison to the actual air mass for measuring the relative load of the engine, a parameter used as input of most part of the standard engine maps together with the engine speed. The determination of the relative load is demanded to the air estimation function that uses one of the methods between Alpha-Speed, Speed-Density and MAF to give an estimation or a measurement of the air mass aspirated by the cylinders. Once the estimation is done, if the torque control strategy is active, a function block provides the conversion from the requested torque for the air path into an air mass setpoint. This setpoint is then used by the throttle control to determine the requested throttle position as a combination of an open loop model (inverted flow through the throttle) and a closed loop control comparing the actual engine load to the requested one. Then, the boost control calculates the boost target as an offset on the required manifold pressure and an open loop plus closed loop controller returns the WG or VGT desired position. In case of standard control, the throttle position and the boost setpoints are simply calculated through maps in function of % of load requested and rpm. Moreover, the air subsystem provides the activation of the dump valve on predefined strategies and calculates though maps the VVT setpoints, both for intake and exhaust.

Finally, the **Fuel Control** subsystem in the fixed rate loop is in charge of calculating the high pressure rail setpoint as a function of the actual conditions, while the **Auxiliary Control** subsystem contains the control strategies of service actuators such as a variable displacement oil pump.

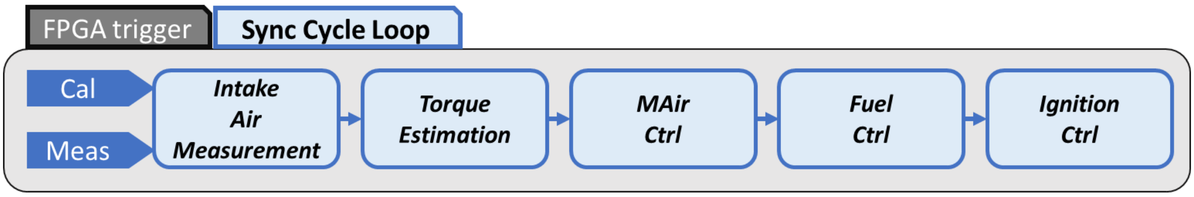


Figure 82: Functional overview of the cycle sync Engine Control loop

The **Cycle Sync loop** (Figure 82) is demanded to manage the subsystems whose actuation is done on cycle basis, such as ignition or injection: this kind of timing avoids unnecessary high-rate calculation, adapting the calculation to the engine speed, and offers the possibility to control the actuated parameters combustion per combustion and cylinder per cylinder. In fact, the loop is triggered by the FPGA through the IRQs module (3.2.13) that is configurable to fire a calculation for each cylinder once per cycle. The module contained in this loop are briefly described below.

An additional copy of the **air estimation** function is demanded to update the air mass and so the relative load estimation on cycle basis, with the most recent readings from the sensors.

The **Torque Estimation** subsystem then converts the estimated air mass into an available torque through the torque model of the engine (air / rpm to torque), the same used in the Load Control subsystem but in the opposite direction. In case of torque-based control, the estimated available torque can be compared with the requested one and drive “fast path” actions, as it will be seen in the Ignition Control subsystem.

The **MultiAir Control** subsystem is in charge of determining the start and end angles of the actuations. This module is placed here in the cycle synchronous loop because of the nature of the MultiAir actuation: it is a phased actuation based on start and end excitation angles, similarly to an ignition command. In case of the predisposition of this module in the SW the dedicated driver blocks to send the actuation instructions to the FPGA are included. The strategy implemented is simple and based on maps, while for more specific consideration the reader is remanded to the dedicated chapter.

The **Fuel Control** subsystem, together with Air Control, is quite extended because of the several actuators and options on which is called to manage. A first function converts the air mass at the intake into a fuel quantity by means of the feed forward calculation based on lambda target and stoichiometric ratio, then next function provides a closed loop correction on the total fuel quantity using the lambda variables available from ECU I/O. At this point the total fuel quantity has been calculated and can be split into PFI and DI injections. The PFI subsystem includes the fluid film compensation, the EOI definition, the conversion from mass to time and the drivers to send the actuation parameters to the FPGA (3.2.7). The DI subsystem provides the split pattern between the multiple injections (up to 5 as standard), and for each injection operates the conversion from mass to volume and form volume to time by means of the injector map (pRail / Inj-Volume to Inj-Time). After the determination of the time for each injection through maps are determined the SOI of these injections: a coercion on the SOI is then applied in case of overlapping or actuations too close to each other. As for the PFI injectors also for the DIs the last function includes the drivers for the FPGA sending of the multi-injection parameters (3.2.7).

Finally, the **Ignition Control** is demanded to calculate the proper spark advance and coil excitation time. In case of a standard control, these parameters are simply the outputs of base maps and corrections, but in case of a torque-based control the SA is used as the "fast path" for the torque production. Starting from a SA base map, in case of an estimated torque higher than the requested one, an offset on the SA is applied in order to reach the ignition efficiency that grants the target torque. Additionally, in case of availability of combustion parameters such as knock intensity or combustion barycenter, advanced control strategies that manipulates the SA to follow MFB or MAPO targets can be included in this module. As for PFI and DI injectors, the last function is the one that gathers the actuation setpoints and send them to the FPGA through the dedicated drivers (3.2.8).

All the parameters of the engine management can be managed both with built-in strategies or manual direct control to allow the maximum control of the working point for test purpose.

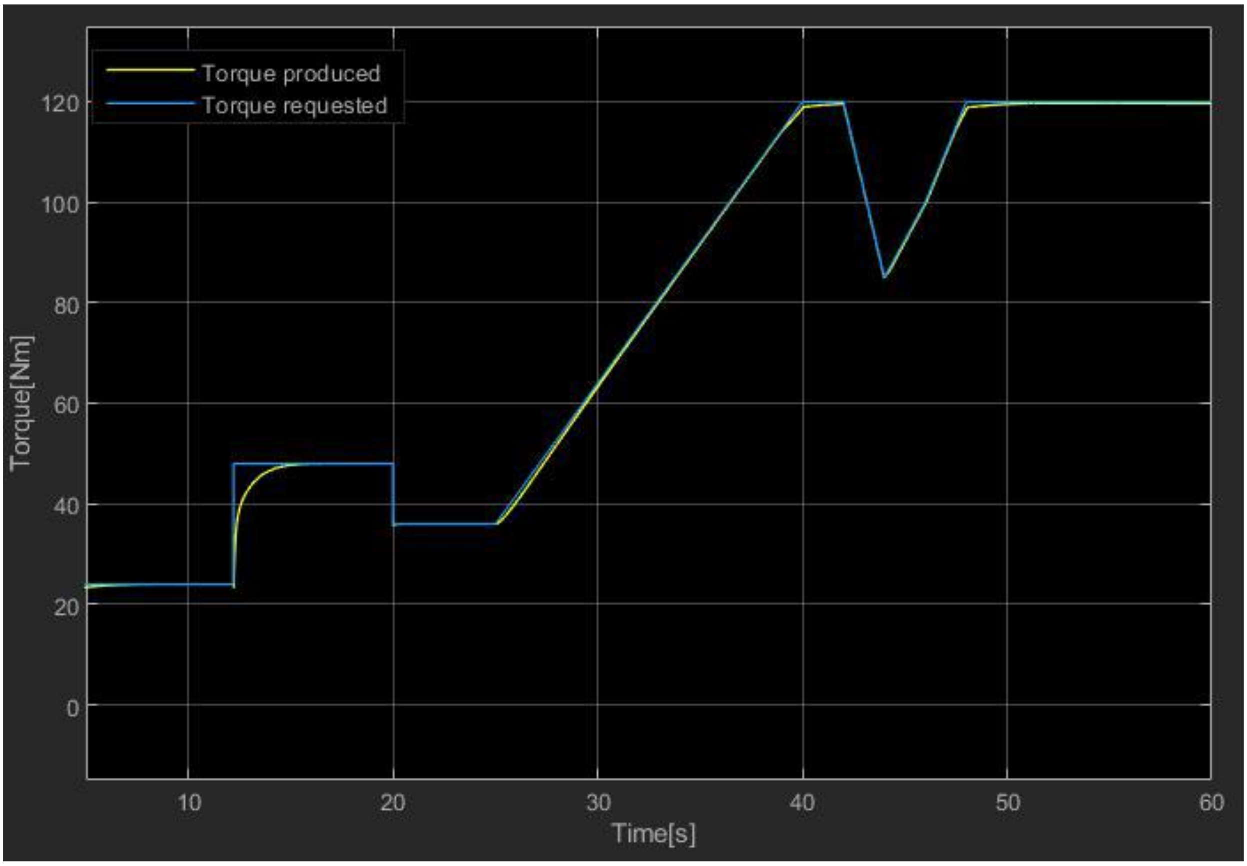


Figure 83: Torque structure test from a model simulation with the torque-based control active

In Figure 84 and Figure 85 is reported as example the results of a gradual ramp from 2000 to 6000 rpm and from 5% to 100% pedal position on the engine after a first attempt mapping

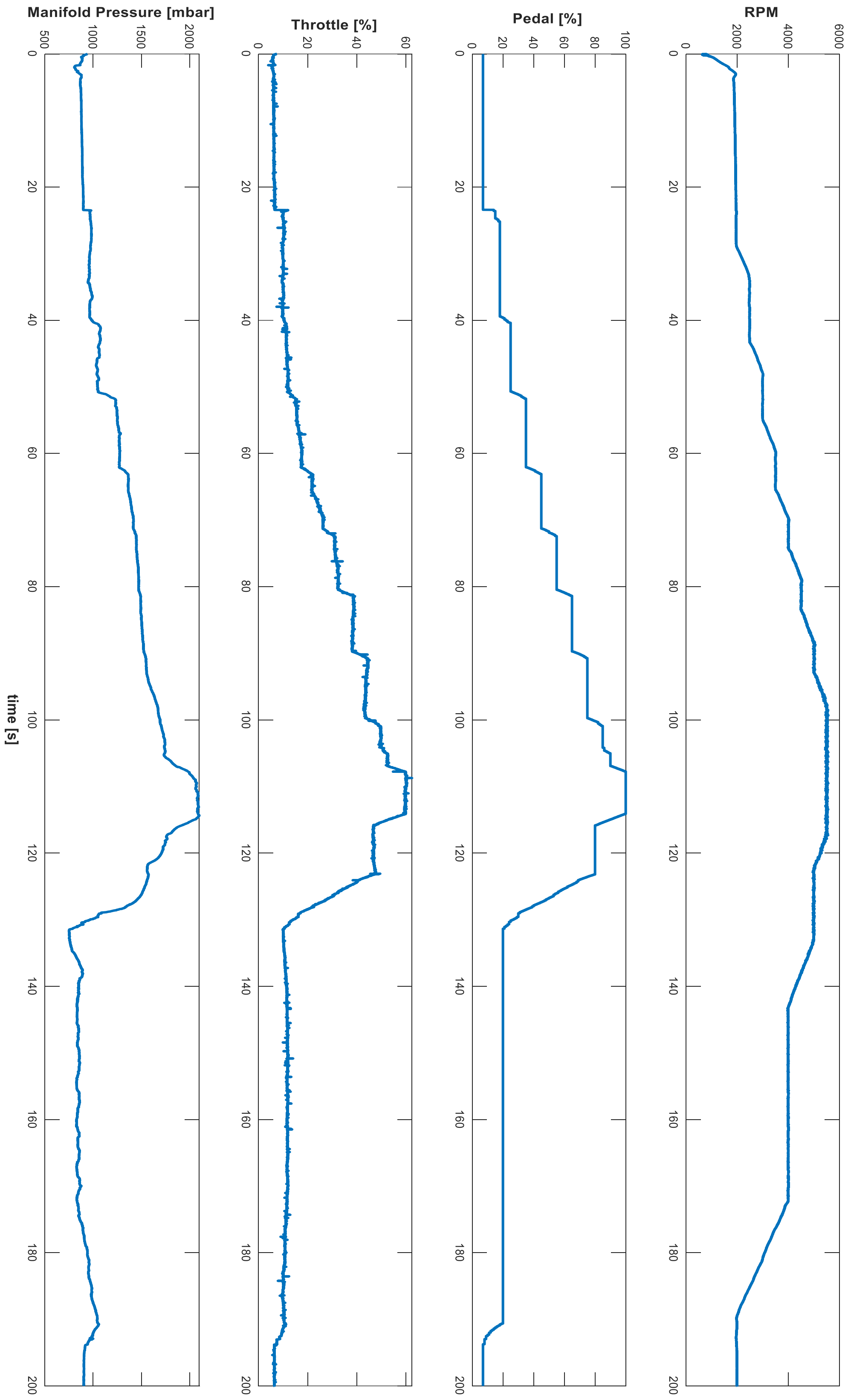


Figure 84: RPM, pedal position, throttle position and manifold pressure during during a test of the two cylinder engine equipped with MAir.

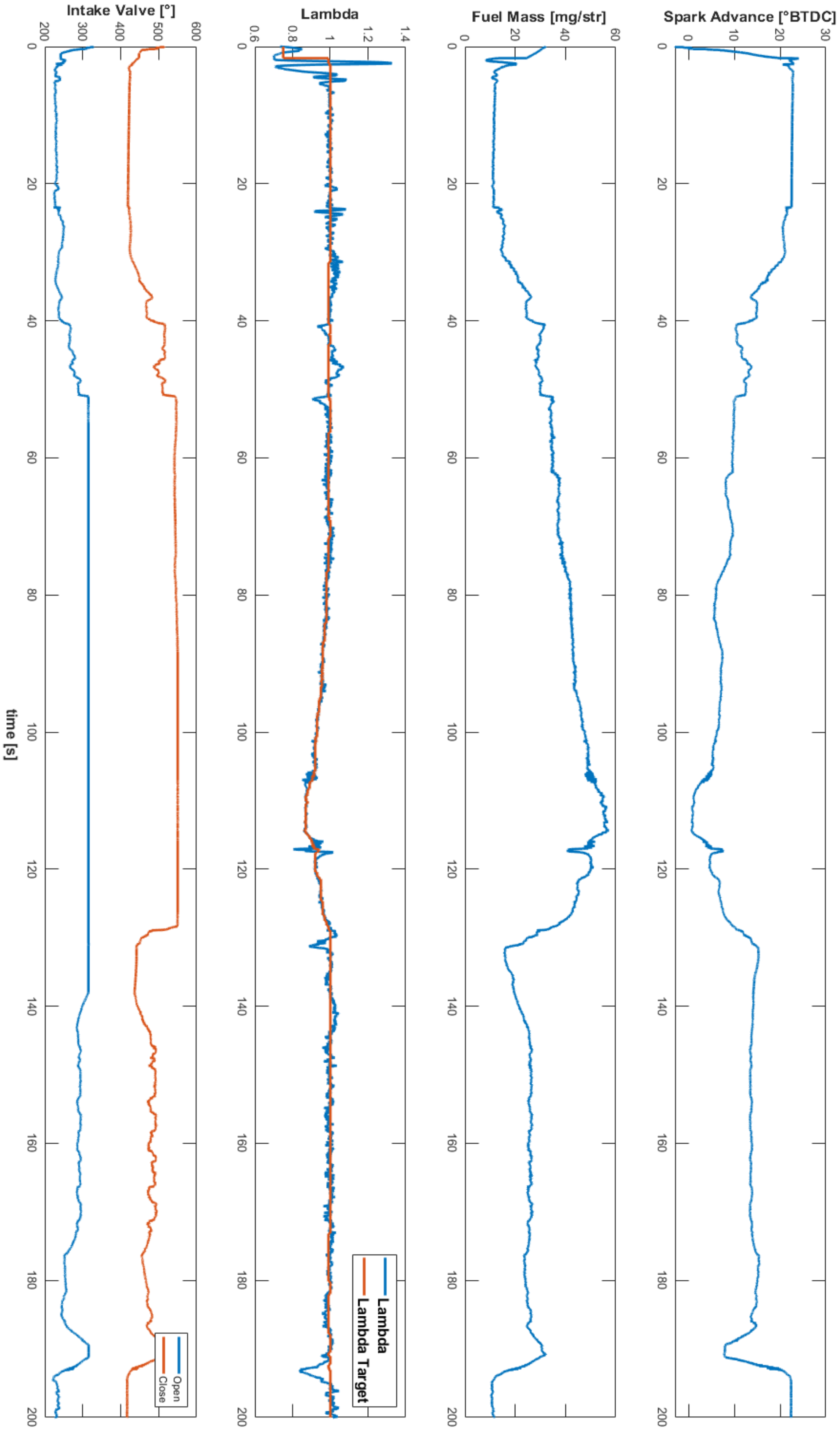


Figure 85: Spark advance, fuel mass injected, lambda value and intake valve control (Multiair) during a test of the two-cylinder engine equipped with MAir.

4.1.2 *Electric Throttle Control*

Precise dosing the air at the intake of a spark ignition engine determines a precise control on the producible torque, for this reason a robust and effective control on the electronic throttle position is so important. From the implementation point of view controlling an actuator like the throttle requires multiple layers of software interacting and passing data each other, including the communication to the FPGA:

1. Throttle position request determination → Engine Management, Air subsystem on RT
2. Determination of the PWM value→ Throttle Duty Actuator controller on RT
3. PWM generation on the HB driver → HB PWM driver on FPGA

Points 1 and 3 have been already presented in 4.1.1 and 3.2.6, while in this chapter will be discussed the point 2, the core of the throttle position control. The determination of the duty cycle of the PWM actuation need to be performed at sufficient rate to ensure stability and at least at the same frequency as the requested position, for this reason, coherently also with the producer datasheet of the throttle body under test, the chosen task rate has been 4ms. Looking at the functions implemented in the throttle actuator control submodule they are grouped in the three sections (divided also by contour colors, blue, red and orange in Figure 86) read, control and write as described for the generic task structure in 3.1.3.

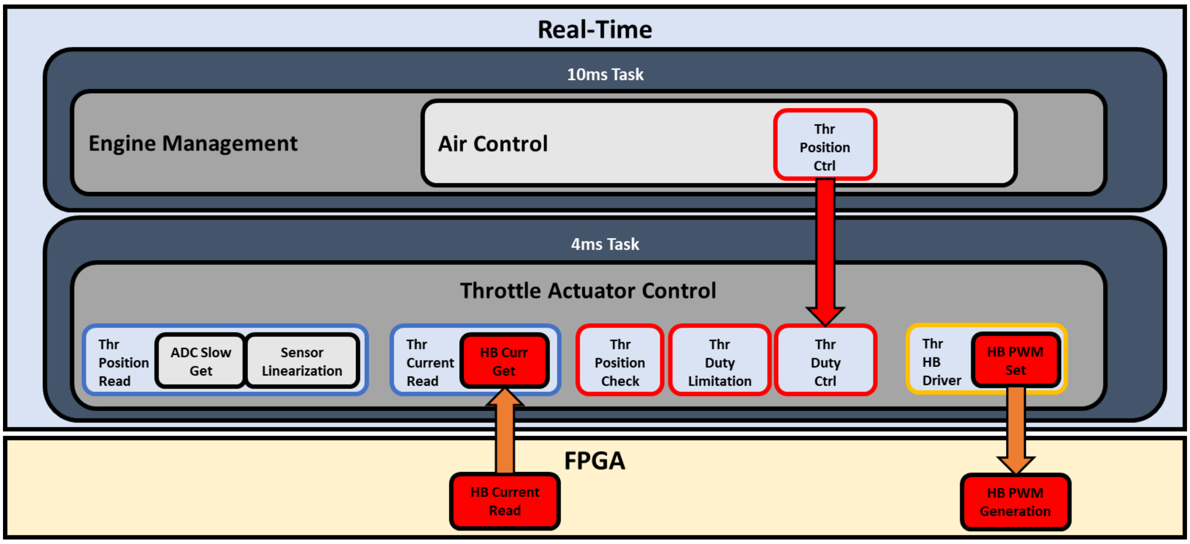


Figure 86: Real-Time and FPGA Throttle Control functional scheme

The **Throttle Position Read** gets from the ADC inputs (in this example the slow inputs of Spark ECU) the raw electrical value form the two TPS sensors and it linearizes them into a 0-100% position. Error checking on the signals is provided accordingly to 3.2.12.

Also for the reading section, the **Throttle Current Read** gets the actual HB current provided by the electrical driver. This information is available from the accessory libraries of the ECU (3.2.14).

Then in the **Throttle Position Check** function the coherence between the two TPS signal is verified within a certain tolerance and the unique throttle position variable, the one used for throttle control, is updated. In case of error of one of the sensors or coherence missing error flags are raised and recovery strategies are activated to choose as reference the TPS that displays the correct value. In case of failure of both TPS a critical error is reported.

In the **Throttle Duty Limitation** functions a set of checks and strategies are implemented to ensure the safety of the driver and the throttle actuator. This function applies additional limitations to the calibration values of the maximum and minimum duty cycle in case of overcurrent (coherently with the manufacturer prescriptions) and critical error of the throttle position reading (duty cycle set to 0% and consequently throttle in limp-home position).

Then in the **Throttle Duty Control** (Figure 87) is contained the actual PWM control of the throttle. A static gain from duty cycle to torque produced has been considered, neglecting the electrical dynamics compared to the overall dynamics of the actuator. Form the control point of view simple PID control cannot reach nor the precision neither the responsiveness required by the application, so a fine-tuned open loop model is crucial to achieve a good performance on position control. For this reason, the control model implements spring and frictions compensation plus a PID controller with gain scheduling inspired by previous publications [5] [6].

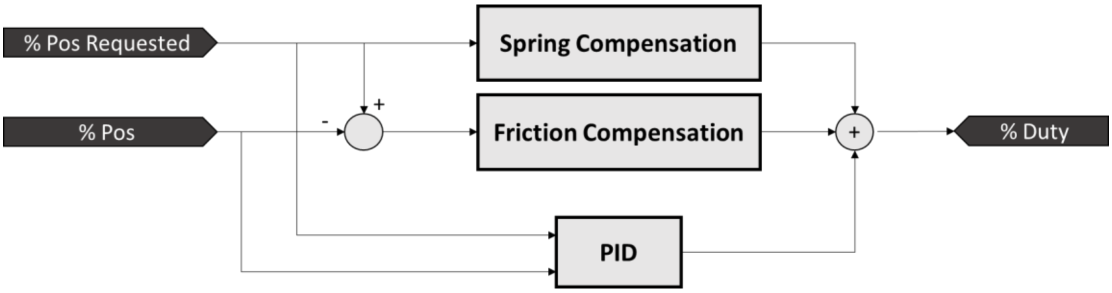


Figure 87: Throttle Duty Cycle controller calculation scheme

The **Spring Compensation** (Figure 88) contains the open loop duty cycle contribution due to the double-spring dead zone: the throttle actuator has a rest position different from zero when not excited, called limp-home position. The mathematic implementation of the compensation is described below in Equation 1.

Equation 1: Throttle Spring Compensation

$$u_{spring} = \begin{cases} {u_{lh}}^{+} + k^{+}(pos - {pos_{lh}}^{+}) & if\ pos > {pos_{lh}}^{+} \\ \dfrac{{u_{lh}}^{+}(pos - pos_{lh})}{({pos_{lh}}^{+} - pos_{lh})} & if\ pos_{lh} < pos < {pos_{lh}}^{+} \\ \dfrac{{u_{lh}}^{-}(pos_{lh} - pos)}{(pos_{lh} - {pos_{lh}}^{-})} & if\ {pos_{lh}}^{-} < pos < pos_{lh} \\ {u_{lh}}^{-} + k^{-}({pos_{lh}}^{-} - pos) & if\ pos < {pos_{lh}}^{-} \end{cases}$$

$$pos = actual\ throttle\ position$$

$$pos_{lh} = limp - home\ rest\ position$$

$${pos_{lh}}^{+} = superior\ limp - home\ position$$

$${pos_{lh}}^{-} = inferior\ limp - home\ position$$

$$k^{+} = superior\ spring\ eleastic\ constant$$

$$k^{-} = inferior\ spring\ eleastic\ constant$$

$${u_{lh}}^{+} = duty\ cycle\ at\ superior\ limp - home\ position$$

$${u_{lh}}^{-} = duty\ cycle\ at\ inferior\ limp - home\ position$$

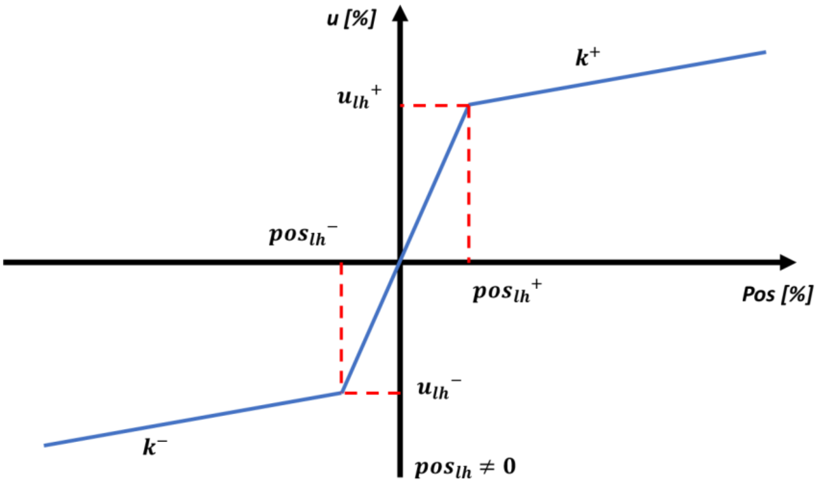


Figure 88: Throttle Spring Compensation graph

The **Friction Compensation** (Figure 89) contains the duty cycle contribution necessary to unlock the throttle from its position. The friction model is based on a simple Coulombian representation of the static friction with the addition of a dead zone and a transition zone to ensure the stability of the control around the target.

Equation 2: Throttle Friction Compensation

$$u_{friction} = \begin{cases} 0 & if\ |err| < tol \\ \tilde{u}_{fr} * sgn(err) * \dfrac{|err| - tol}{tr} & if\ tol < |err| < tol + tr \\ \tilde{u}_{fr} * sgn(err) & if\ |err| > tol + tr \end{cases}$$

$$\begin{cases} \tilde{u}_{fr} = \tilde{u}_{fr}{}^{+} & if\ pos > pos_{lh} \\ \tilde{u}_{fr} = \tilde{u}_{fr}{}^{-} & if\ pos < pos_{lh} \end{cases}$$

$$err = requested\ pos\ minus\ actual\ position$$

$$\tilde{u}_{fr}{}^{+} = friction\ duty\ compensation\ above\ limp$$

$$-\ home\ position.\ It\ is\ the\ real\ u_{fr}\ multiplied\ by\ 1.01\ to\ win\ the\ friction$$

$$\tilde{u}_{fr}{}^{-} = friction\ duty\ compensation\ below\ limp$$

$$-\ home\ position.\ It\ is\ the\ real\ u_{fr}\ multiplied\ by\ 1.01\ to\ win\ the\ friction$$

$$tol = tolerance\ on\ error$$

$$tr = error\ transition\ zone$$

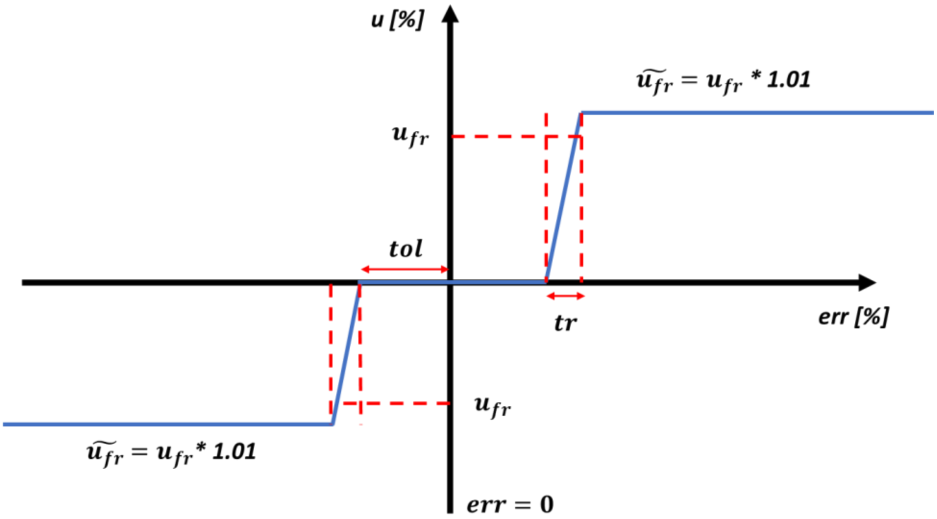

u [%]
$\widetilde{u_{fr}} = u_{fr}$ * 1.01
u_{fr}
tol
tr
err [%]
u_{fr}
$\widetilde{u_{fr}} = u_{fr}$* 1.01
err = 0

Figure 89: Throttle Friction Compensation graph

To avoid excessive integral charging and slip-stick effects around the target, an integral gain scheduling on the integral gain has been implemented (Figure 90). An overall good result has been reached in terms of stability of the control around the target position and precision through large and small steps of target. The gain scheduling curve is completely configurable and in Figure 90 is presented the implementation chosen for the device under test. Additionally, and integral reset is arranged if form one iteration to another the error on the requested position exceeds 0.5% and a multiplicative compensation of the gains is implemented taking in account the actual voltage battery compared to the calibration.

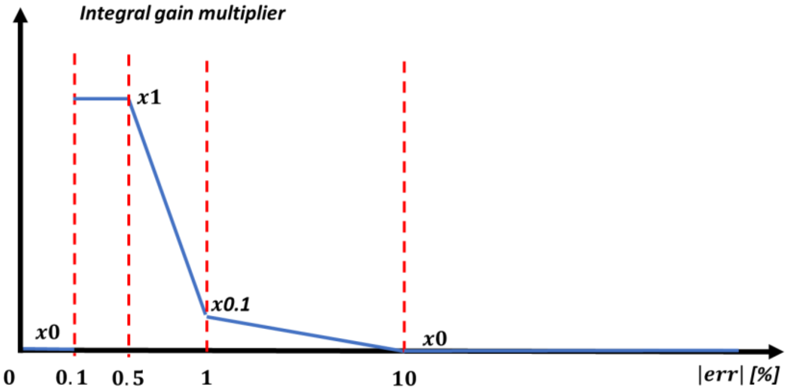
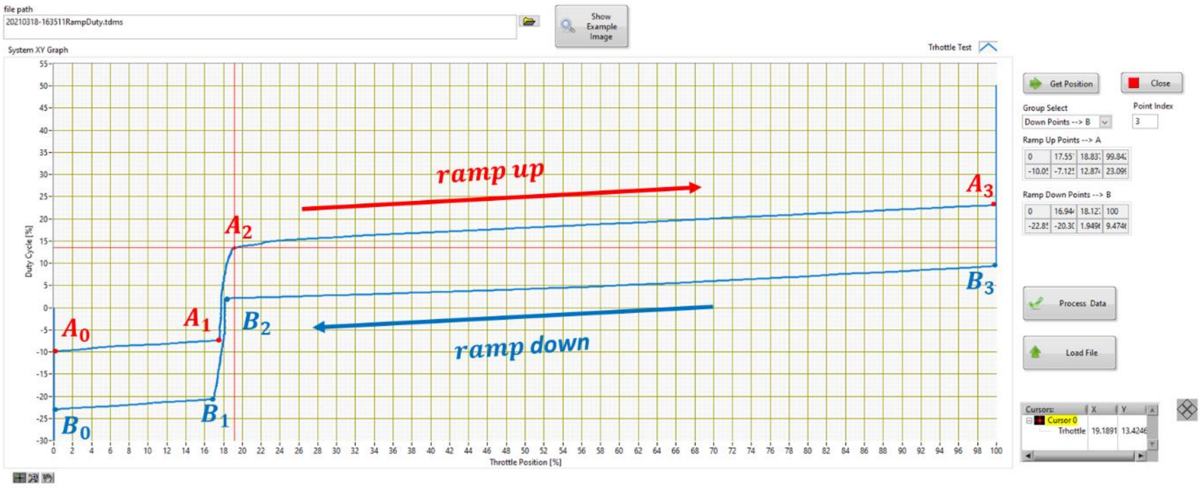


Figure 90: Throttle Integral Gain scheduling example

A semi-automatic tuning tool has been developed to speed up the calibration procedure: a specific throttle control mode in the RT software is demanded to acquire duty cycle and position data from a standard duty cycle ramp test with the device connected and a host software tool (Figure 91) process them to produce the calibration parameters of the spring and friction compensations.


file path: 20210318-163511RampDuty.tdms

Group Select: Down Points --> B

Point Index: 3

Ramp Up Points --> A

0	17.55	18.83	99.84
-10.0	-7.12	12.87	23.09

Ramp Down Points --> B

0	16.94	18.12	100
-22.8	-20.3	1.949	9.474

Cursors:	X	Y
Cursor 0		
Trhottle	19.1891	13.4246

Figure 91: Throttle Calibration tool

The automated procedure calculates the friction and spring compensation parameters through the identification of 4 points for the ascending part (A0, A1, A2, A3) and 4 for the descending part (B0, B1, B2, B3) of the duty-position ramp [5] [6]. Then, finally, the **Throttle HB Driver** sends the PWM instructions to the FPGA through the H-Bridge library (3.2.6).

Equation 3: Spring Compensation Parameters

$$pos_{lh} = \frac{A_{1pos} + A_{2pos} + B_{1pos} + B_{2pos}}{4}$$

$$pos_{lh}{}^{+} = \frac{A_{2pos} + B_{2pos}}{2}$$

$$pos_{lh}{}^{-} = \frac{A_{1pos} + B_{1pos}}{2}$$

$$u_{lh}{}^{+} = \frac{A_{2duty} + B_{2duty}}{2}$$

$$u_{lh}{}^{-} = \frac{A_{1duty} + B_{1duty}}{2}$$

$$k^{+} = \frac{A_{3duty} - A_{2duty}}{A_{3pos} - A_{2pos}}$$

$$k^{-} = \frac{A_{1duty} - A_{0duty}}{A_{1pos} - A_{0pos}}$$

Equation 4: Friction Compensation Parameters

$$u_{fr}{}^{+} = \frac{A_{0duty} - B_{0duty}}{2}$$

$$u_{fr}{}^{-} = \frac{A_{3duty} - B_{3duty}}{2}$$

In order to validate the position control model a set of tests with a reference throttle body (disconnected form the engine) has been accomplished after the calibration of the friction and spring parameters and the tuning of the PID coefficients. The aim of this experimental activity has been the verification of the performance during a wide range of transient and stationary position requests: the validation tests in the figures below submit small and large steps in the throttle position request as well as a free request. Together with setpoint and actual position of the throttle also open loop and total duty cycle control values are presented. To enhance the performance of the controller also a low-pass filter on the setpoint value has been applied: in the examples presented the cut-off frequency has been set to 50Hz.

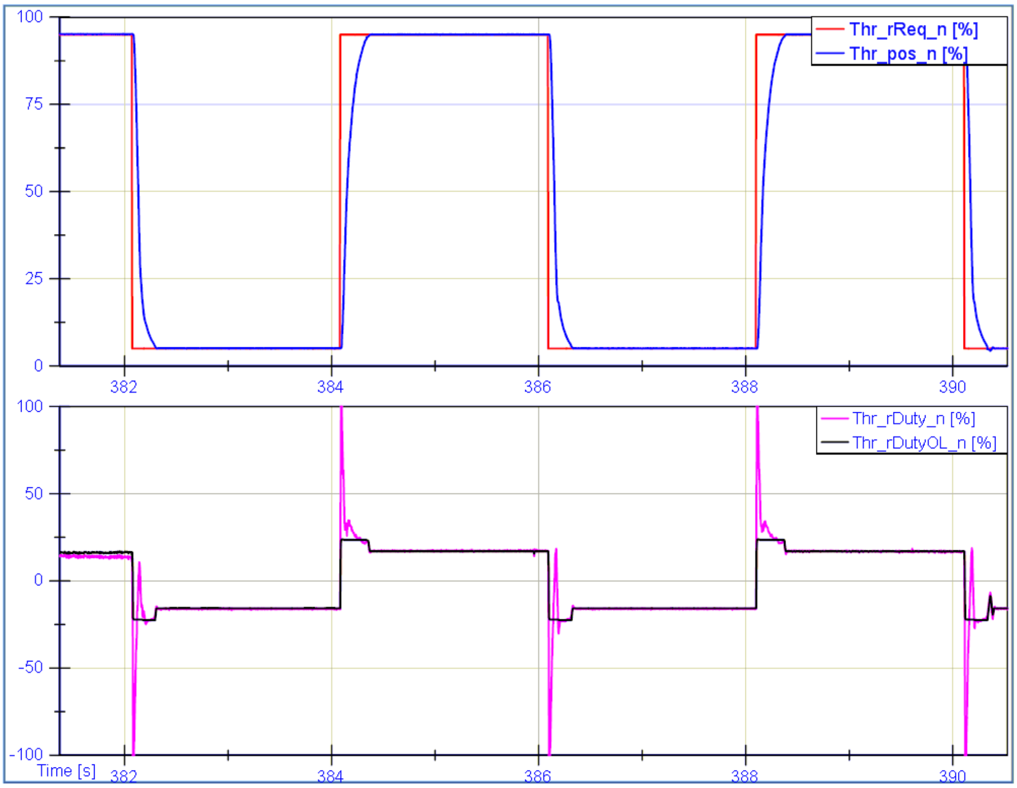


Figure 92: Throttle 5-95 % Steps

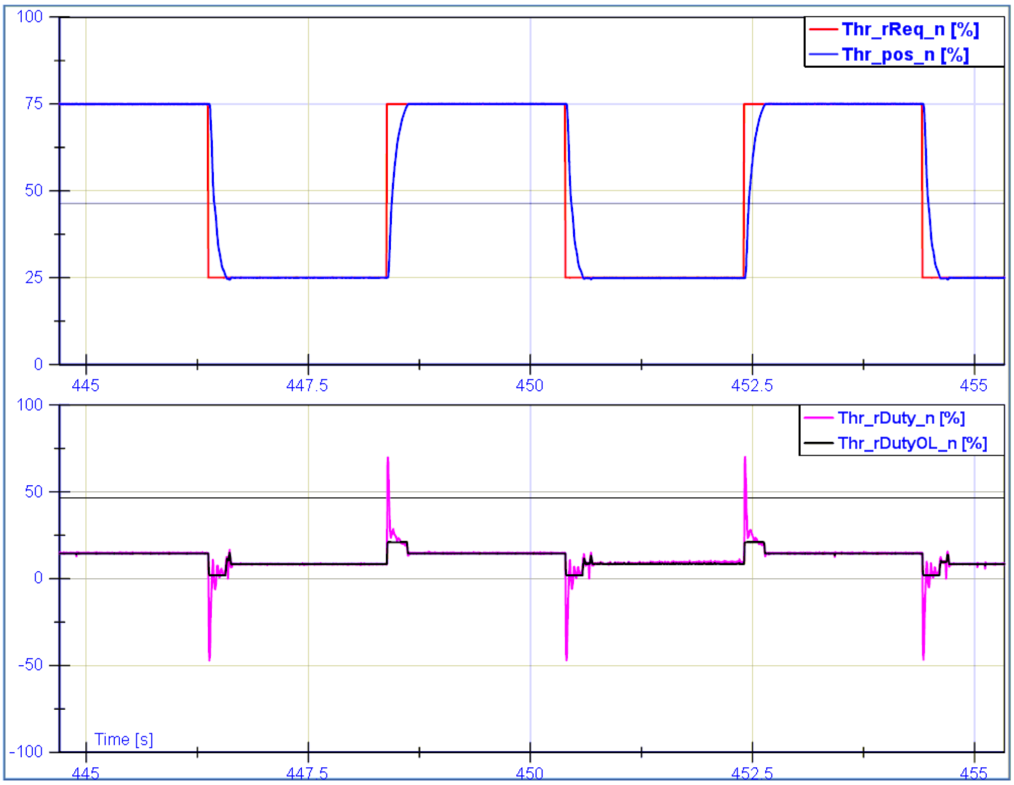


Figure 93: Throttle 25-75 % Steps

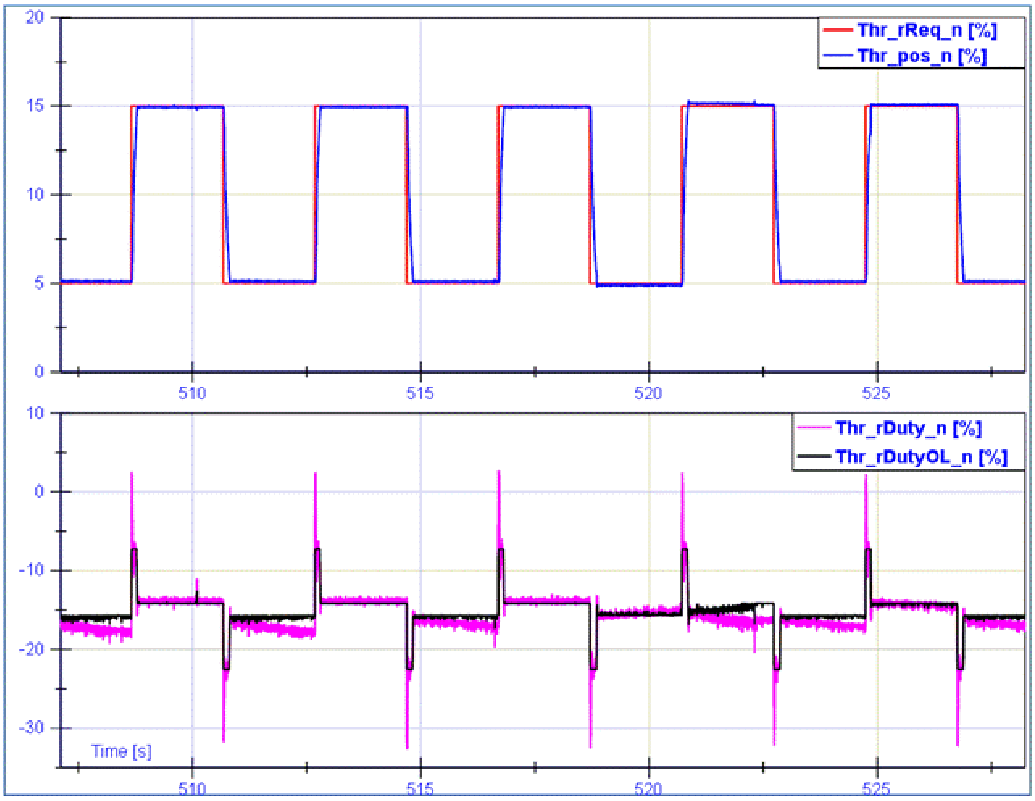


Figure 94: 5-15 % Steps

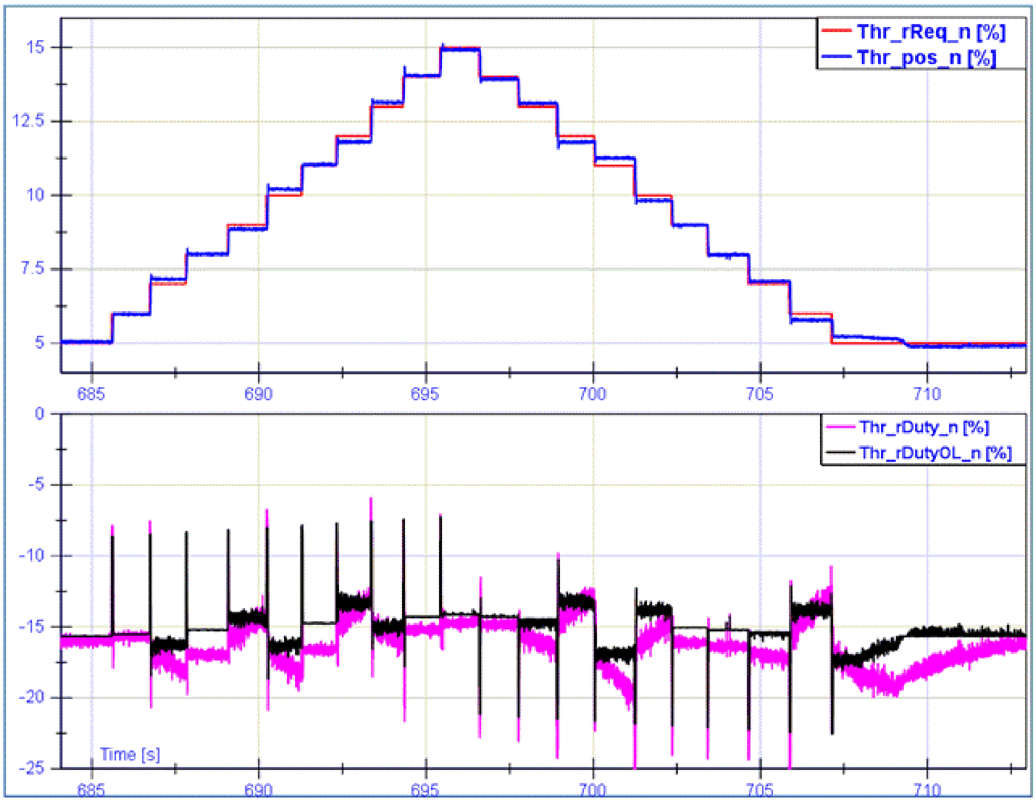


Figure 95: 1% steps 5-15%

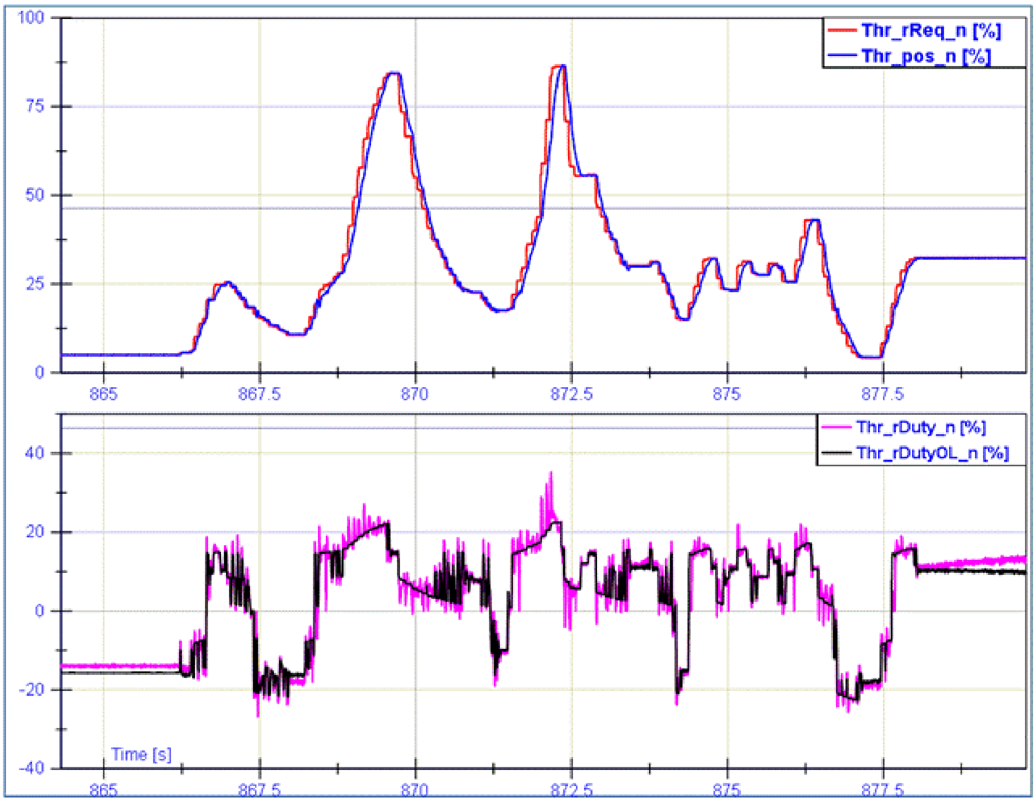


Figure 96: free setpoint test

4.1.3 *GDI Pump Controller*

DI gasoline engines require a high-pressure pump, typically a volumetric pump driven by the engine camshaft. The pump is usually controlled in the useful displacement by means of a phased actuated recirculation valve: the position of the valve closing determines the delivery angle and consequently the useful displacement of the pump (Figure 97). The model under test was a 3 lobes cam pump, so 3 actuations per revolution were required with a specific current profile and timing: a specific software module has been developed, exploiting both FPGA and Real-Time environment (Figure 98).

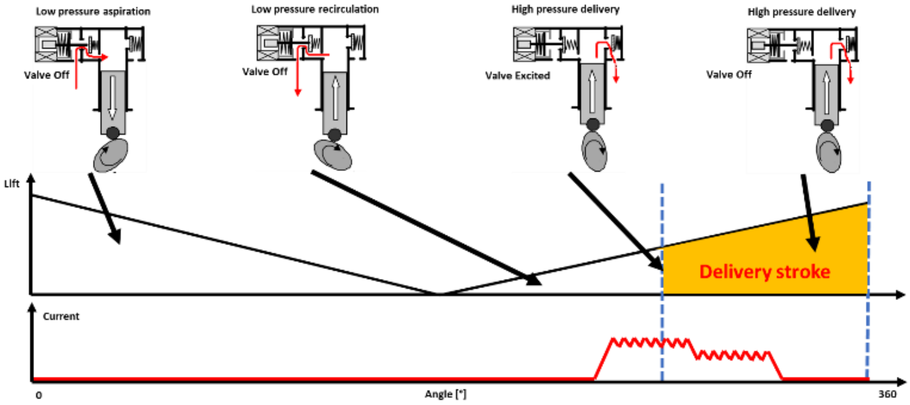


Figure 97: GDI pump working cycle and actuation

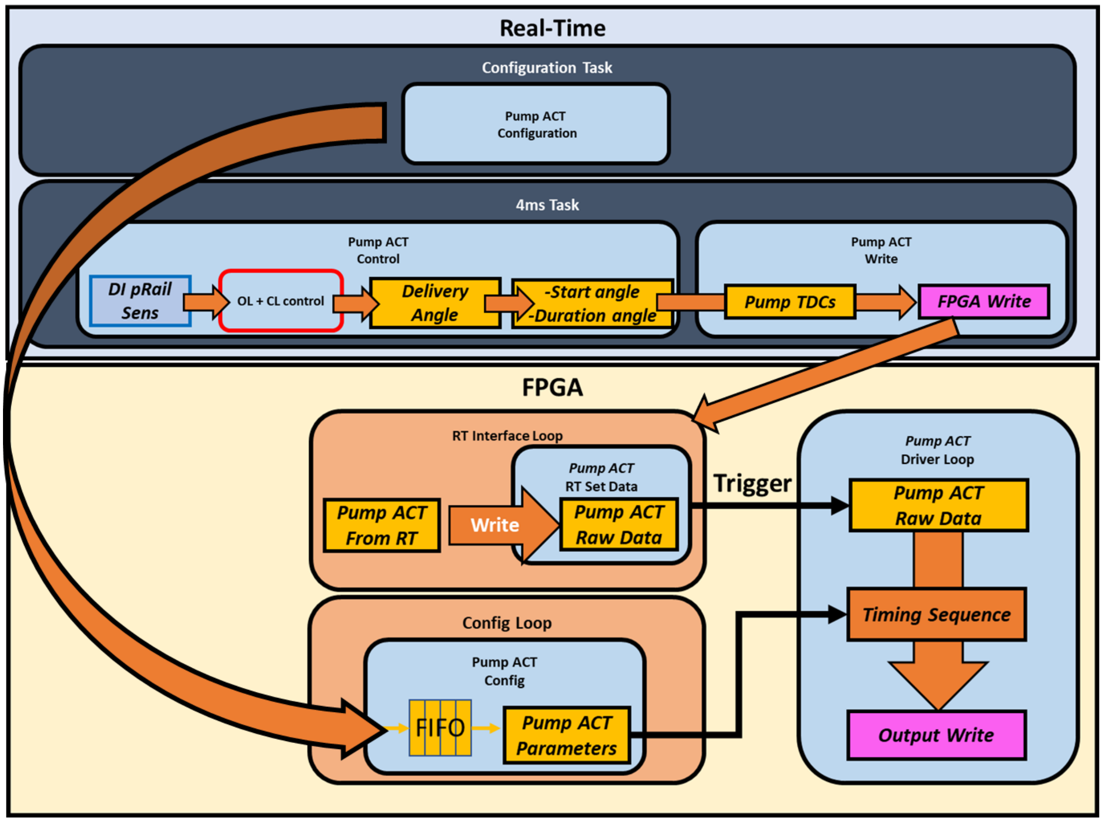


Figure 98: Real-Time and FPGA GDI Pump Control functional scheme

On the **FPGA** side the current profile control has been implemented (Figure 99): since the required battery voltage was not compatible with the P&H (3.2.7), already configured for the high-voltage DI injectors, an alternative current profile control has been realized. Following the pump manufacturer indications, a three-phase PWM open loop current control has been implemented by means of a simple LS driver (3.2.6), while exploiting the existing angle-based phasing system (3.2.2) assured the precise control of position of the 3 actuations per revolution referred to the pump TDCs. A triggering algorithm inherited from the injection and ignition modules (3.2.7, 3.2.8) ensures that the actuation parameters, calculated from RT, are the most recent available before firing the actuation. Since each of the 3 actuations needs to be defined in the delivery angle the parameters transmitted to the FPGA are:

- The actuation **Start Angle**, defined on the pump lobes TDCs
- The **Angle Duration** of the actuation: the 3 PWM phased are automatically generated though an injector-like sequence. The duration needed to ensure the proper opening of the control valve can be shorter than the total delivery angle

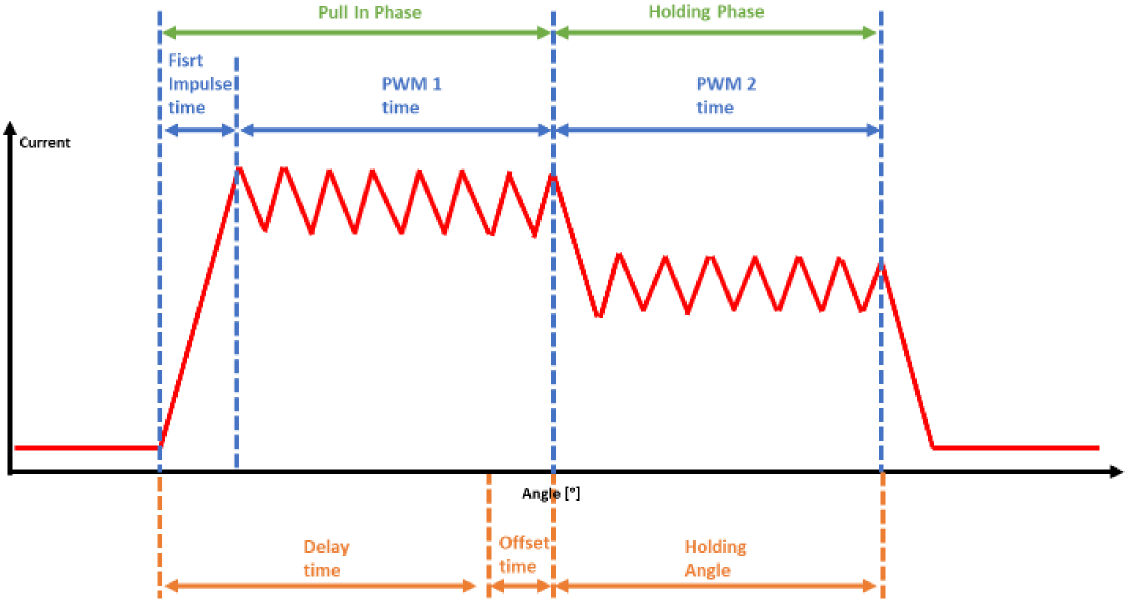


Figure 99: GDI Pump actuation current profile definition

On the **Real-Time** side the high-level control has been implemented calculating the delivery angle with a combination of open loop maps and closed loop with the pressure feedback. Moreover, PWM levels, timing and the angles of the actuations (Figure 99) are determined with calibrated look-up tables and producer-defined strategies as a function of the required delivery angle and for example the battery voltage. A closed loop only control has been used in the example in Figure 100, demonstrating, after the calibration, good stationary results (±5 bar over the target) and an acceptable step transient response for a typical test bench installation.

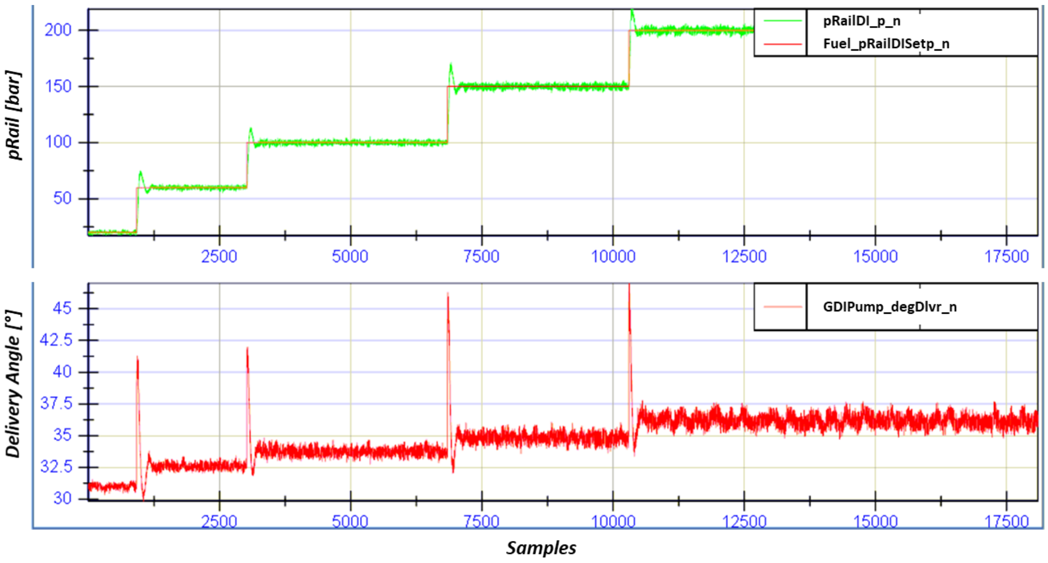


Figure 100: GDI Pump steps ramp control results sampled at 250Hz. Above: in red the pressure setpoint and in green the measured one. Below: the delivery angle requested by the controller

4.1.4 ***MultiAir Control***

When facing the installation of the two-cylinder engine (4.1) one of the main challenges during the development of the full bypass RCP has been the implementation of the control module for the MultiAir system, installed on the intake valves of the engine. This system form Schaeffler [UniAir [3] [4]] provides an enormous flexibility on the intake: timing, duration and also lift profiles are influenced by the command that drives the hydraulic valves of the system. In function of the starting and ending of the excitation of the actuator coil, the valve lift profile can be strongly modified unlocking the potential of strategies to control the trapped fresh air mass, the turbulence of the charge, the internal exhaust gases recirculation and simple cylinder deactivation. For these reasons the system can be of interest not only for "traditional " consumption and emissions reduction, but can also enhance advanced combustion strategies, such as HCCI, that may gain stability from the precise guidance of the charge motion and internal EGR dosing [4] [14] [17].On the counterpart the actuator requires a specific control command with precise angle references in the form of a Peak&Hold current profile.

On the **FPGA** side, starting form the previously developed ignition and injector drivers (3.2.7, 3.2.8) an angle-based P&H driver has been implemented requiring as command instruction:

- Start Excitation Angle: the angle on which the P&H current profile is started.
- End Excitation Angle: the angle on which the current is set back to 0.

The current profile is automatically managed in the timing of its phases though the same configuration parameters and functions as for the injectors.

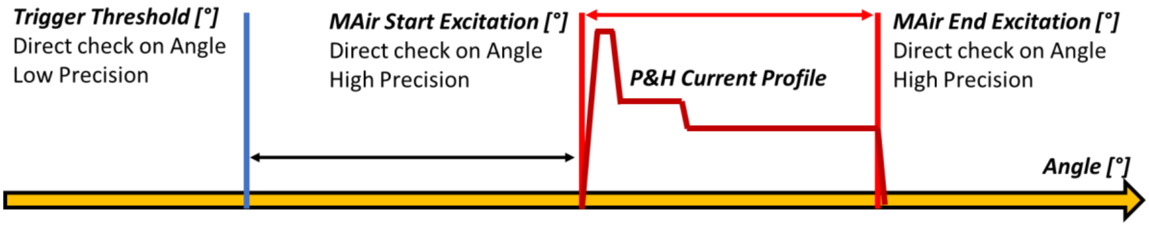


Figure 101: Angle based MultiAir command: Start + End actuation angles

The determination of the start and end excitation setpoints is placed on the **Real-Time** side. The strategy implemented on the basis software includes simple RPM and Load maps for the angle determination together with the standard manual bypass control for testing purpose. As it can be seen from Figure 102, the start and end excitation angles have a direct impact on the volumetric efficiency: for example, at low load and speed, with a reduced total duration of the excitation and so of the intake valve opening, the mass trapped in the cylinder is reduced and as a consequence the throttle can be left more open and reducing pumping losses.

Volumetric efficiency has been mapped together with the MAir maps during the tests since Load index and manifold pressure have a direct relation. The first attempt MAir control maps, presented in Figure 102, have been filled with the values that granted the same air mass flow through as for the original control system at the same manifold pressure and engine speed conditions.

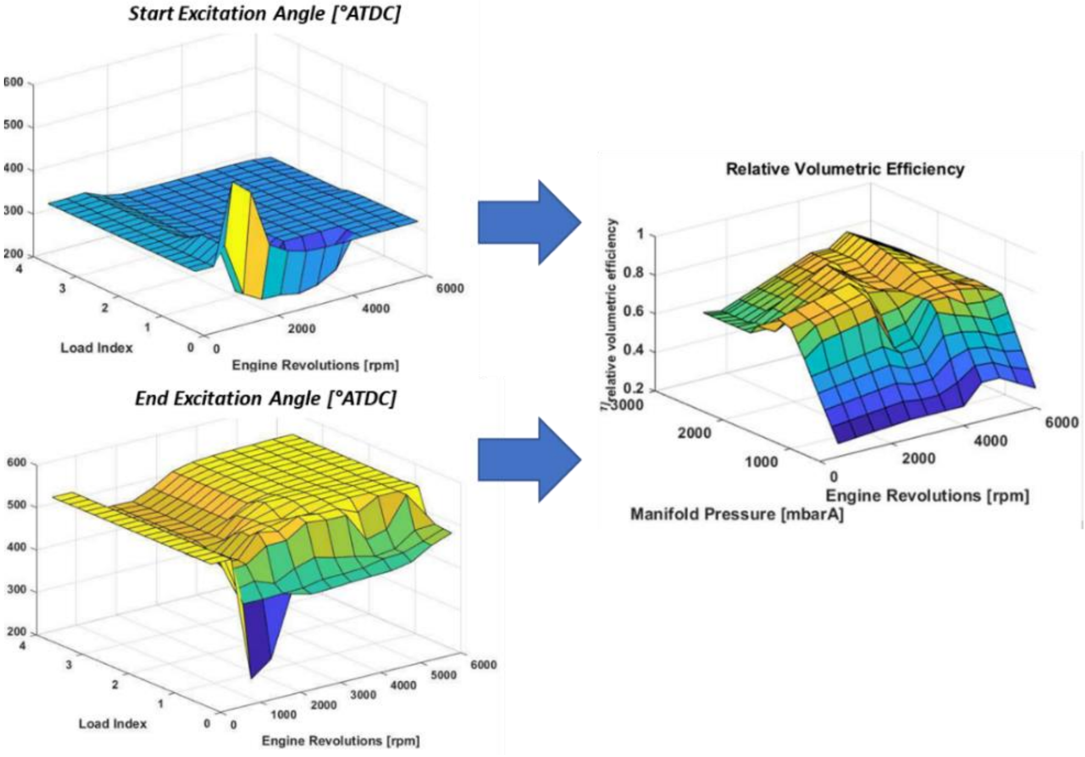


Figure 102: MultiAir test control maps and direct correlation with the volumetric efficiency of the engine

4.1.5 *VVT Control*

Since air path control is widely used in modern engines to accomplish efficiency and pollution targets, additionally to the previously developed MultiAir, a VVT control module has been developed (Figure 104). A hydraulic valve is electrically actuated in order to fill the volumes of the VVT actuator to change the relative position of the camshaft and so the intake / exhaust valves opening and closing (Figure 103).

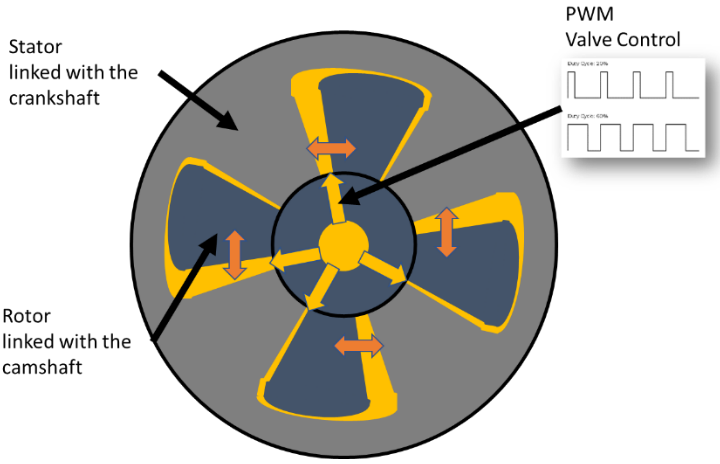


Figure 103: physical scheme of a hydraulic VVT actuator

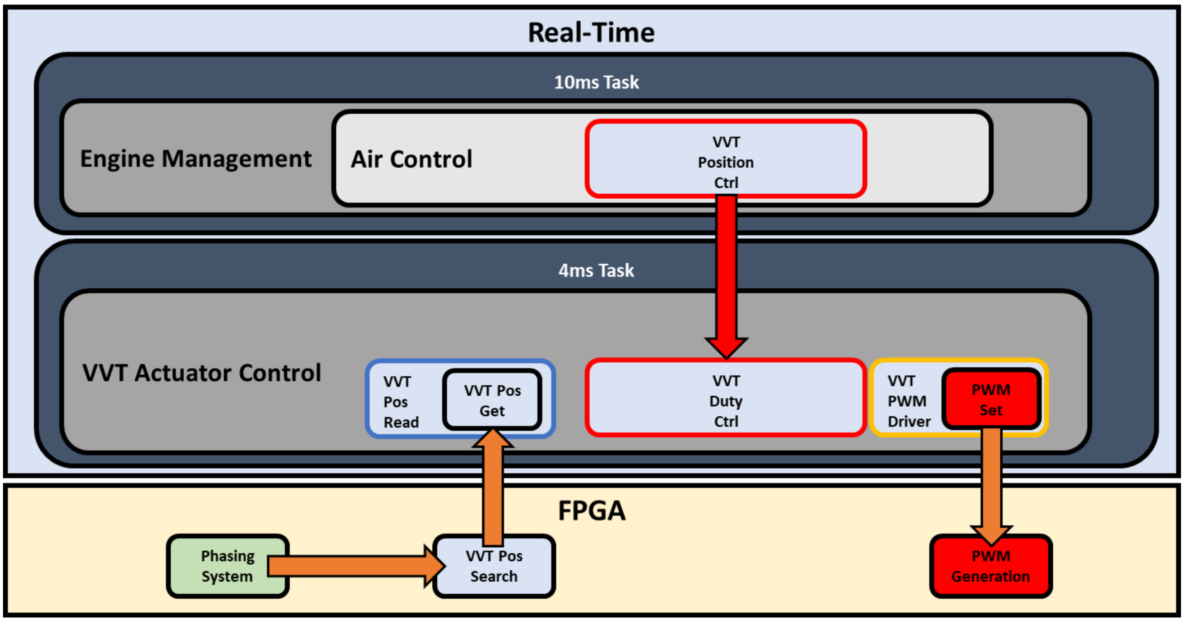


Figure 104: Real-Time and FPGA VVT Control functional scheme

The software identifies the IVO (Intake Valve Opening) and EVC (Exhaust Valve Closing) based on CRANK and SCAMs signals and the first function necessary to the system control is the identification of the relative position of the camshaft compared to the absolute reference system. This has been implemented naturally in the **FPGA** environment exploiting the fast acquisition and computations and the already present phasing system (3.2.2). For each camshaft, up to 4, the absolute angle from the CRANK signal provides the 0° reference while the SCAM signals rising / falling edges are used to identify the relative position of the camshaft: a start search angle is chosen for the detection of the next falling / rising edge (configurable) that identifies the relative position of the camshaft (Figure 105).

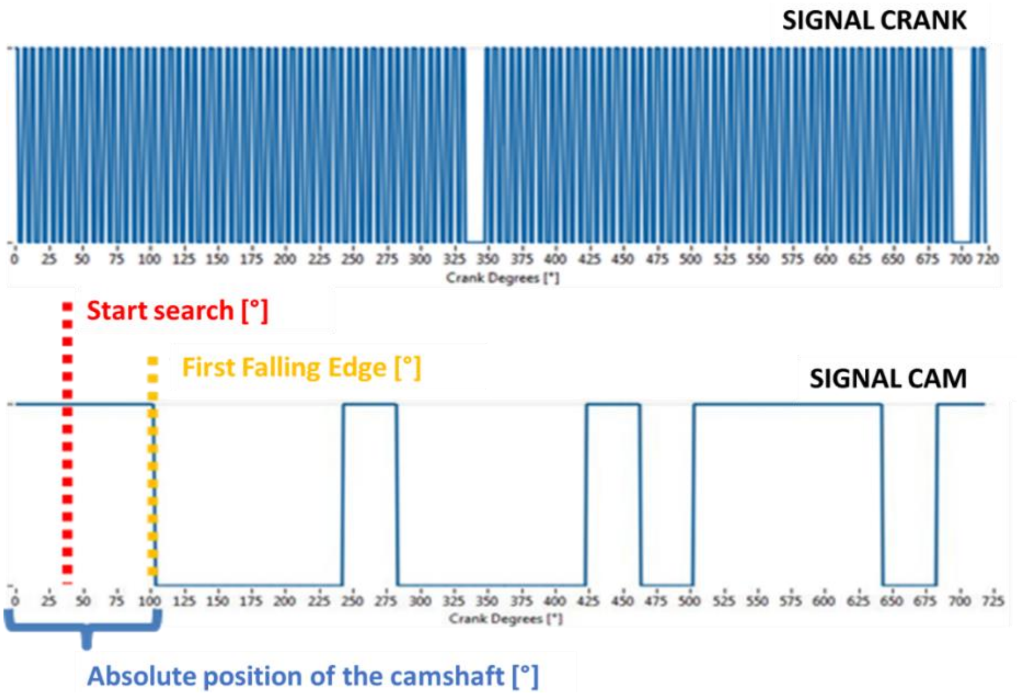


Figure 105: Crank/cam sensors signals referred to the VVT positioning logic

These absolute angles are then made available to the **Real-Time** environment where are translated respectively into IVO and EVC angle applying an offset calibratable in function of the geometric characteristics of the engine. Once IVO and EVC values are affordably estimated they are used as feedback into for closed loop plus open loop control strategy on the PWM actuation (3.2.6) for the hydraulic valve that allows the oil flow into the VVT actuator. A control example of the described system is shown in Figure 106 for an exhaust camshaft.

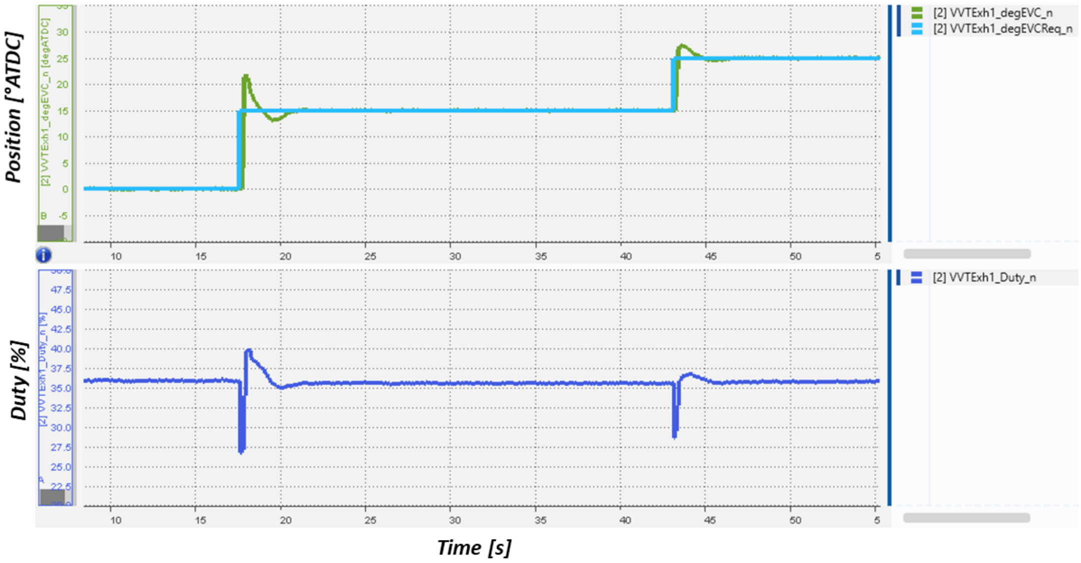


Figure 106: VVT Control Example. Above EVC position request (light blue) and actual EVC (green) and below the duty cycle PWM control (blue)

4.2 Advanced Combustion Engine Management

An area on which Rapid Control Prototyping is almost mandatory is the research on advanced combustions: often prototypal engines and system are used in the tests and custom control solutions are needed to support reaching of the objectives. Starting from the same base framework some RCPs have been realized, with control software structures very similar and in part inherited from the spark ignition engine management (4.1.1).

4.2.1 ***Single Cylinder Control for Innovative Combustion Research***

The objective of the research activity was to test the capabilities of RCCI combustions with gasoline injected both with PFI and DI injectors. The stability of these combustion can be difficult especially when testing with extreme parameters and sometime the combustion is not even capable to sustain the engine motion. For these reasons, in this case study an automotive 4-cylinder diesel engine have been used converting one cylinder as "laboratory unit" for RCCI [7] [15] [16] [18], while the other 3 had the purpose to simply ensure the stability and the controllability of the test conditions such as engine speed, boost pressure, EGR production etc.

Starting from a previous torque-based diesel control structure, the RCCI managing has been placed inside the fuel system block, where the torque request is converted in total fuel quantity (mg/cycle) for cyl1 independently from the other three cylinders load (Figure 107). Then RCCI total quantity is split in diesel and gasoline accordingly to the split ratio (rpm-torque map) expressed as energy: fuels densities and lower heating values (LHV) are used to calculate the final quantities in mm3/cycle for PFI and DI injectors. Diesel multiple injection is possible both for standard (up to 5) and for RCCI mode (up to 2), so a further RCCI diesel quantity split between two injections is actuated. Eventually, volume quantities are converted in injection times through the cyl1 diesel injector map and the PFI map (or the linearization as gain and offset time): both are dedicated maps for the single injector for maximum quantity precision. In the end RCCI injection times, PFI time (µs) and DI times array (µs) are sent to the respective injector with the dedicated phasing in terms of EOI (End Of Injection) for PFI and SOI (Start Of Injection) for DI.

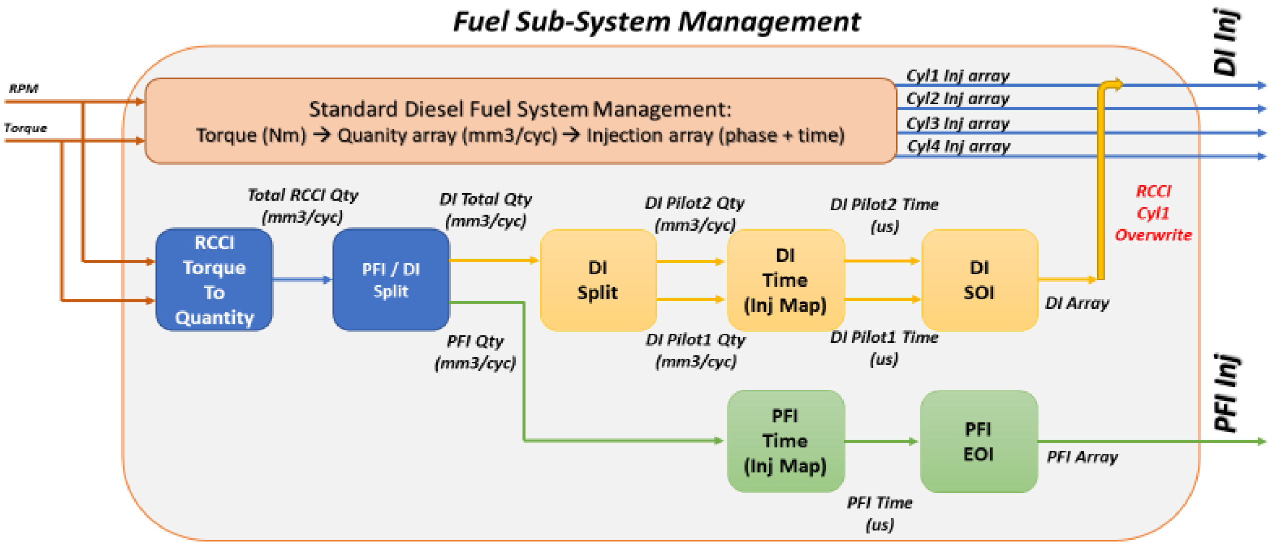


Figure 107: Fuel-Subsystem management for single cylinder RCCI combustion

Example of successful control on the independent RCCI cylinder is shown in Figure 108 compared to the other three “standard” diesel cylinders at 2000 rpm and same fuel quantity injected: the different nature of the combustion is noticeable by the extremely different profile in the pressure signal evolution during the combustion.

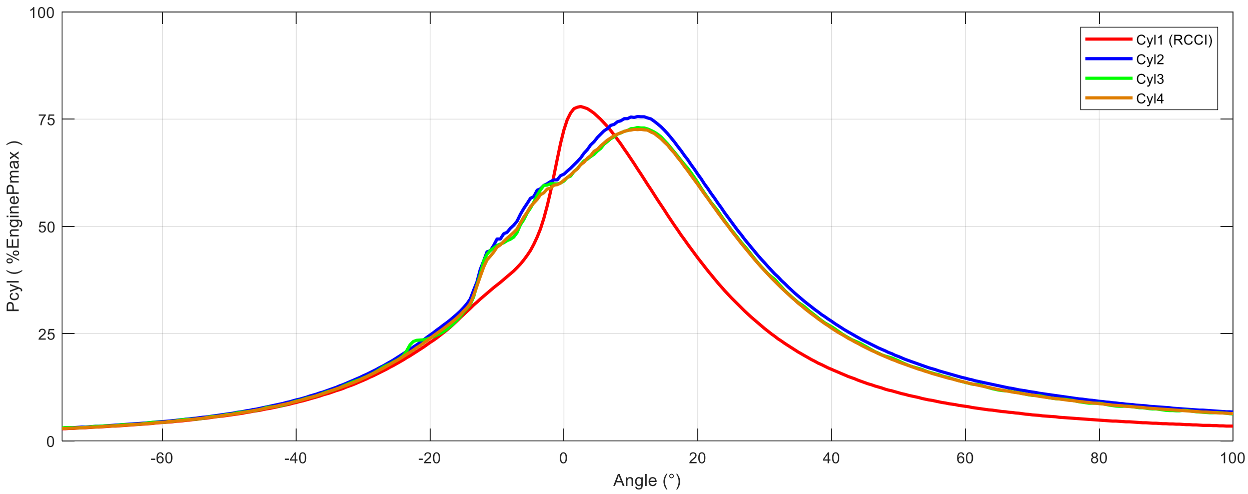


Figure 108: RCCI combustion example, the cyl1 has a complete independent injection control

4.2.2 *Closed Loop Combustion Control*

In order to support strategies such as knock protection or combustion phase control, data coming from and indicating system must be integrated in the engine management (Figure 109).

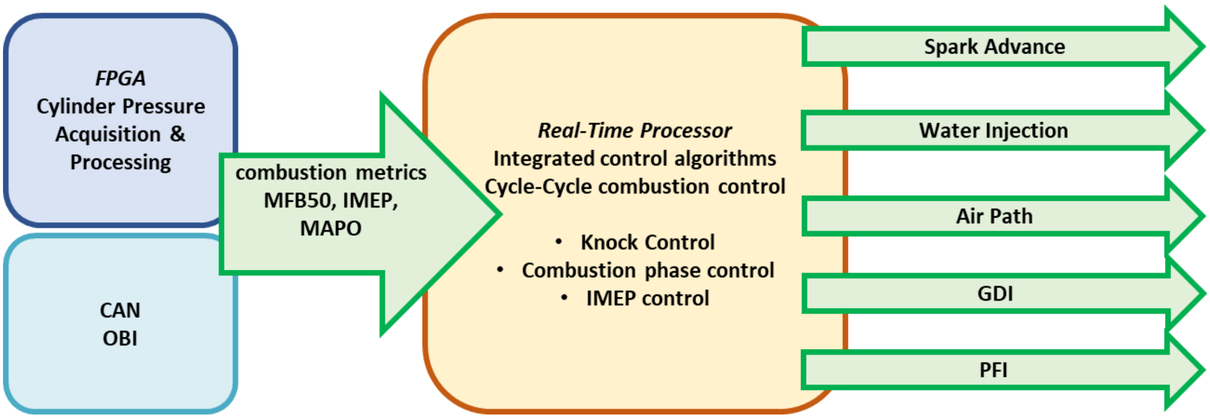


Figure 109: Closed Loop Combustion Control overview example for Spark Ignition engines

This is almost necessary for research activities such as the one presented in 4.2.3. From the developed framework point of view two options have been implemented.

The first possibility is to read the combustion indexes from an external combustion analysis hardware through CAN. This operation can be done through the CAN module (3.2.9). Another option is integrating the combustion analysis on the same control ECU as a software module (Figure 110). This allows an all-in-one analysis and control ECU where the combustion metrics are made available to the control software through the shared memory mechanism. A previously developed indicating system have been made part of the framework as an independent module, divided by the FPGA and Real-Time environments:

- On **FPGA** the cylinders pressure signals, up to 4, are acquired through fast AI (3.2.3) and then processed. The combustion metrics are directly obtained in the FPGA and sent through FIFO to the Real-Time controller on cycle basis, exploiting an IRQ mechanism like the one presented in 3.2.13
- The data are then read form the FIFO in the **RT** environment and converted form raw FPGA unit to physical values. At this point the combustion indexes are stored into a specific DVR that can be easily accessed from any part in the software and can be used for control purpose or simple monitoring through XCP communication for example.

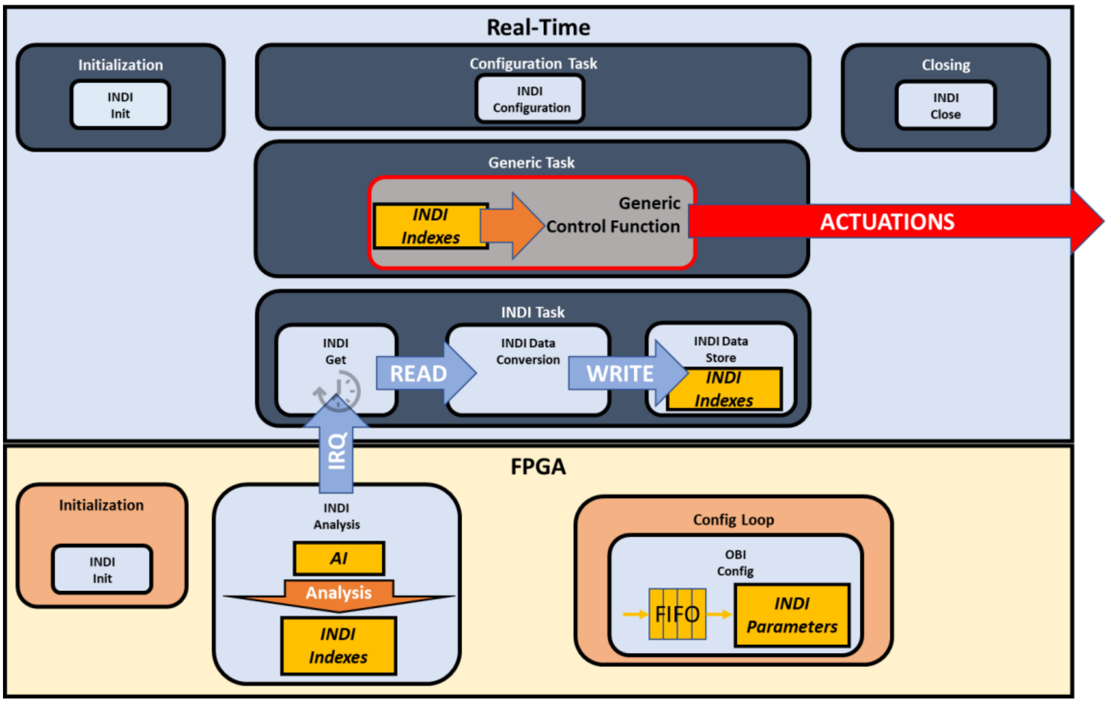


Figure 110: Real-Time and FPGA Indicating Integrated module functional scheme

4.2.3 *Full Engine Control for Innovative Combustion Research*

Coming from previous experience on single cylinder testing and research activity [14] [17] [19] a PhD colleague has exploited the developed framework to implement on Spark ECU its own control system. The need was to build a full control system for a small diesel engine to be fully converted to gasoline injection for research purpose:

- Testing innovative combustion methodology and strategies
- Additional actuators and sensors needed for the purpose
- Cylinder per cylinder control with combustion closed loop

The choice of using the developed framework together with the ECU hardware was driven by the replacement of an older and more complex testing system made of a calibratable ECU plus several external RCPs based on NI platform including an indicating system. In order to centralize the control strategies and make possible an easy implementation of custom algorithms without recurring at functional bypass and external systems, a full bypass engine control management have been realized:

- ○ Replacing the complex previous RCP that combined development OEM ECU + cRIO +OBI.

- Limitations and time-consuming controls development and testing.
- High flexibility requested on sensors and actuators control.
- High frequency control strategies modification for test purpose.

Several actuators and systems have been integrated in the control system and are managed by the control system such as:

- External Roots compressor for boost generation with the engine in cranking condition
- Additional throttle valve for switching from Roots compressor to turbocharger
- External air preheater for intake temperature increment
- High precision sonic fuel meter
- Indicating system

Actuations are modified from one cycle to another, in order to achieve targets such as IMEP (adjusting injection time) or combustion phase (adjusting SOI) interfacing the engine management with the data coming from the combustion analysis. This approach is mainly necessary for granting the controllability of combustions that are often not very stable.

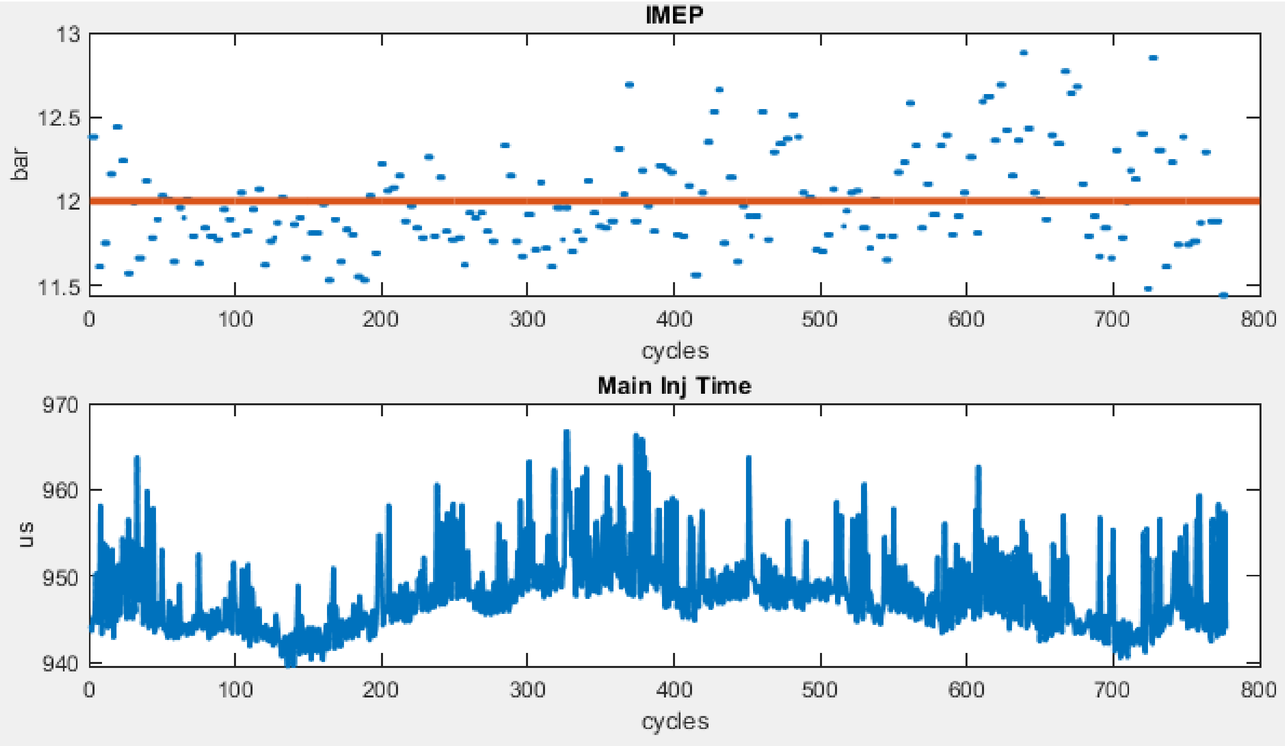


Figure 111: IMEP closed loop control adjusting main injection time

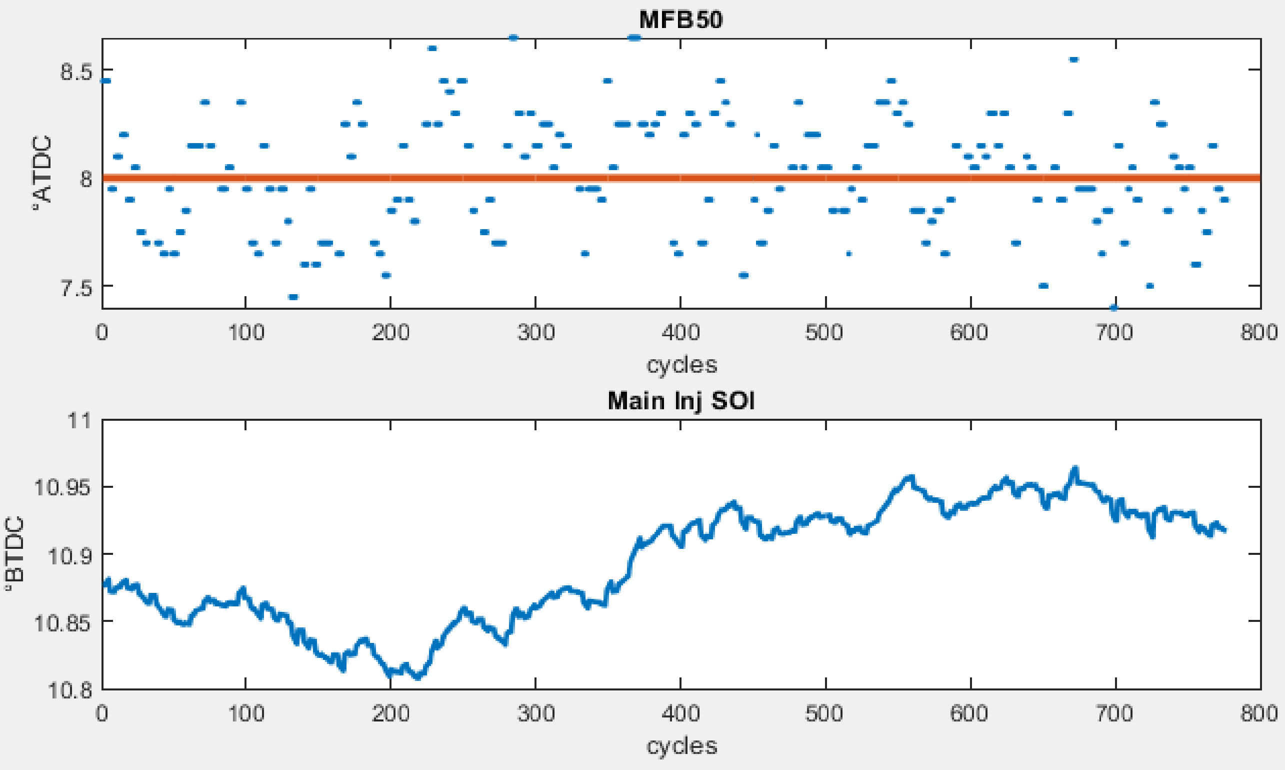


Figure 112: MFB50 closed loop control adjusting main injection SOI angle.

In Figure 111 and Figure 112 closed loop combustion control on Injection Time and on SOI are presented as application examples during a combustion stability test at 2000 rpm on the diesel engine fully converted to gasoline injection on all the 4 cylinders. The results presented have been acquired through the test bench indicating system and are relative to cylinder 1.

4.3 ADDED FUNCTIONALITIES RCP

4.3.1 *Water Injection*

Water Injection nowadays is often taken in account as a solution for excessive exhaust temperatures and heavy knocking operation, which are, especially in downsized and boosted engines, cause of limitations in terms of achievable efficiency levels [24] [30]. An example of flexibility and reuse of the code has been the development of a simple water injection “added functionality” RCP: this simple software allows the user to control a water pump, the phase and the amount of water to be injected. Exploiting the base software architecture, the implementation was easy because all the main drivers and base functions have been already developed and modularized, making it easy to concentrate only on the custom control for the RCP (Figure 113). The need of this RCP was driven by the fact that the engine control unit was calibratable in the parameters but could not be changed in the strategies and actuators implementation.

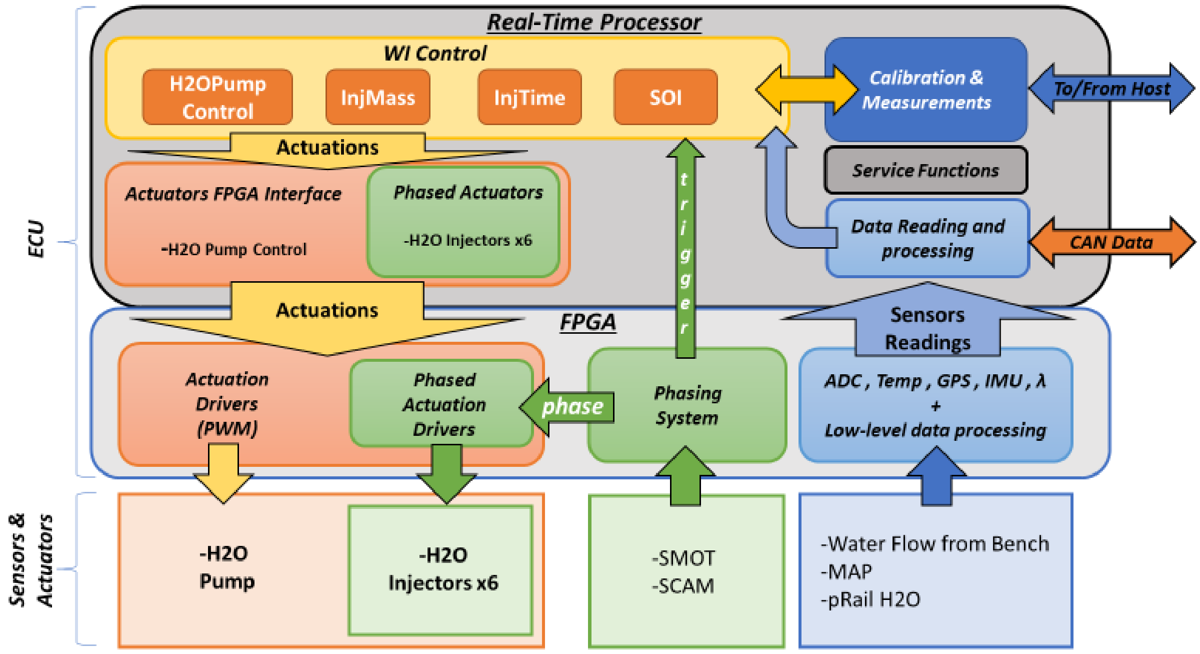


Figure 113: Water Injection functional overview: developed on the Base Architecture

On the **FPGA** side, no modifications have been required compared to the base framework, while on the **Real-Time** the software application has been built to control up to 6 water injectors, each one phased with a specific TDC (Figure 114).

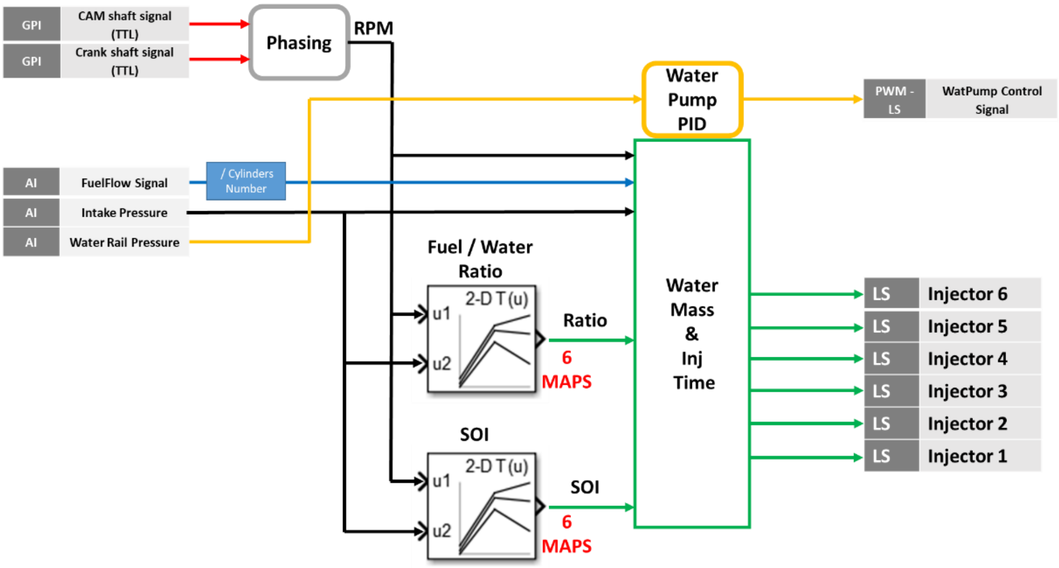


Figure 114: Water Injection Real-Time control scheme

Up to 6 calculation per cycle are possible: up to 6 angular triggers configurable. The Water Injection control application consists in 4 main modules:

- **Water Mass Calculation:** uses the fuel flow signal (analog) to calculate the water mass to be injected cylinder per cylinder. Based on rpm/MAP maps. Included in the cycle sync task.
- **Injection Time Calculation:** converts the mass to be injected in injection time. Included in the cycle-sync task.
- **Start of Injection:** Based on rpm/MAP maps. Included in the cycle sync task.
- **Water Pump Controller:** controls the pump PWM with a PID. The task time is 5ms.

The water injection control modules and variables have been also easily included as an independent SW module in the SIEM control, adding a new functionality with low-cost code maintenance.

4.3.2 ***Variable Valve Lift***

As for the water injection RCP, the need of extra actuations and strategies for prototyping tests on a variable valve lift system were required. Variable Valve Lift systems, similarly to VVT and MultiAir systems, have as main objectives the charge control under in terms of quantity and motion, the exhaust gases recirculation and in general the increase of efficiency, especially at partial load [31]. The software purpose was controlling the valve lift of a 2banks-4camshafts V6 engine through a cam-shifting mechanism based on barrel cams and pin actuators. Three Low-Side solenoids per cylinder and per side (intake/exhaust) were actuated for controlling the lift of the valves: when a solenoid is excited the relative pin acts on the relative barrel cam and therefore the valve cam profile is shifted from one to another [31]. The total number of the solenoids was 36 = 6 Cyl x 2 Sides x 3 Solenoids that control the 3 states of each cylinder:

- **Full Lift**: maximum cam profile
- **Mini Lift**: reduced cam profile
- **Zero Lift**: for cylinder deactivation

The actuation of the solenoids can be driven both in manual mode, selecting and firing manually the actuators, or in automatic mode based on the VVL state request and the actual estimated state of the VVL. All the service standard modules form the framework have been exploited allowing a quick realization of the control system, whose functionalities are shown in Figure 115.

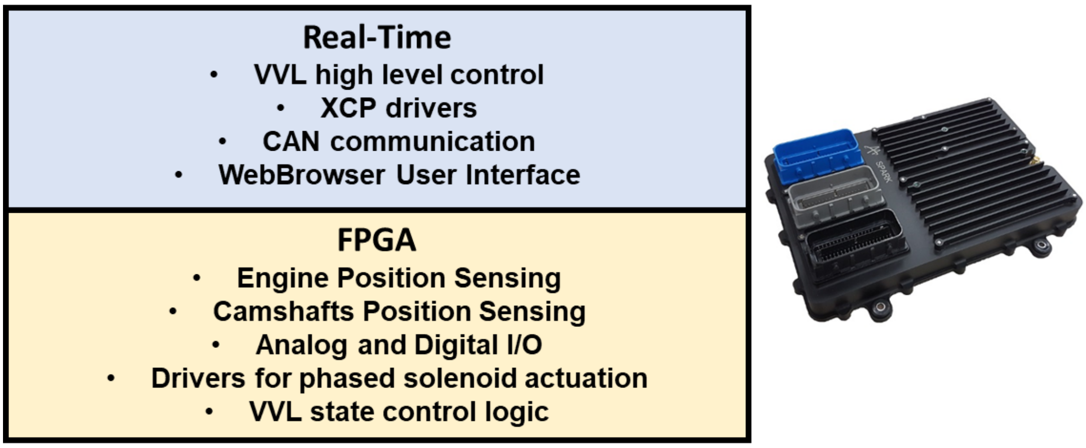
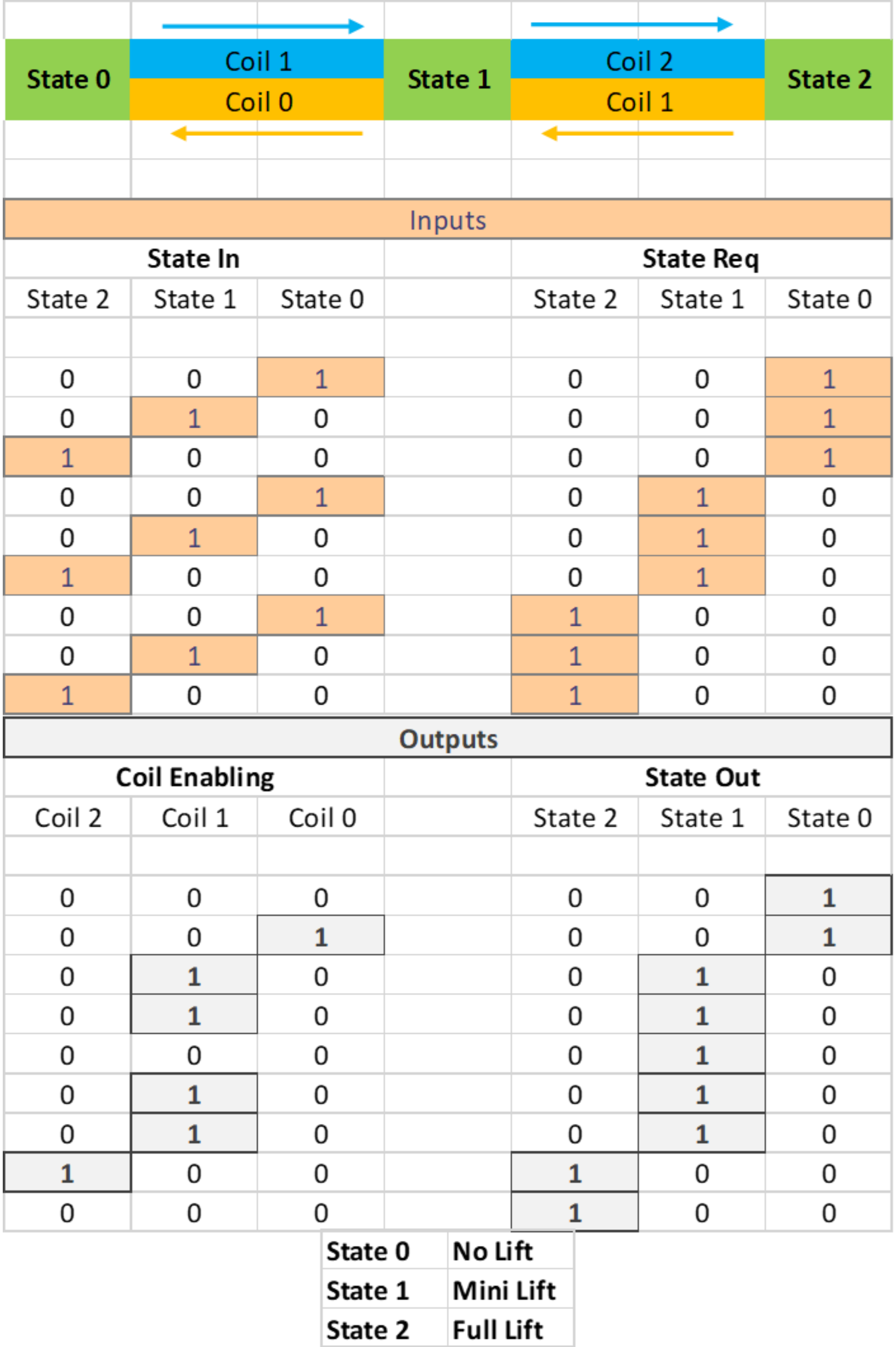

Real-Time
• VVL high level control
• XCP drivers
• CAN communication
• WebBrowser User Interface
FPGA
• Engine Position Sensing
• Camshafts Position Sensing
• Analog and Digital I/O
• Drivers for phased solenoid actuation
• VVL state control logic

Figure 115: VVL RCP functions collocation between RT and FPGA

The main part of the application is placed in the **FPGA** because of the need of phased actuations with precise angle to angle on camshaft reference. A modified phasing system together with the VVT position determination (4.1.5) provide the camshafts angle estimation on which the actuation is based to trigger the mechanical VVL system to change position. Since the actuation must be performed not only at precise angles, but also as single events in response at a state request change a truth table system implements the state machine logic of activation of the right solenoid for each cylinder. The lookup tables below represent the coil enabling for each combination of Input State – Required State and have been implemented in the solenoids control in the FPGA. Based on the decision made by these tables (Figure 116), the coils are fired as soon as possible, with the proper timing and sequence cylinder per cylinder for reaching the requested lift state. Since no feedback was provided for the actual position of the lift system, the state machine for the estimation of the current state (State Out), based on the previous state and actuations, has been included in the truth tables (Figure 116).

On the **Real-Time** side the high-level control manages the user request of the state of the lift and specific function blocks provide encoding and decoding of the requested and actual state to and from the FPGA. Moreover, the RT provides also all the calculations for the determination of the correct angle of actuation cylinder per cylinder.



State 0	Coil 1 Coil 0	State 1	Coil 2 Coil 1	State 2

Inputs						
State In				State Req		
State 2	State 1	State 0		State 2	State 1	State 0
0	0	1		0	0	1
0	1	0		0	0	1
1	0	0		0	0	1
0	0	1		0	1	0
0	1	0		0	1	0
1	0	0		0	1	0
0	0	1		1	0	0
0	1	0		1	0	0
1	0	0		1	0	0

Outputs						
Coil Enabling				State Out		
Coil 2	Coil 1	Coil 0		State 2	State 1	State 0
0	0	0		0	0	1
0	0	1		0	0	1
0	1	0		0	1	0
0	1	0		0	1	0
0	0	0		0	1	0
0	1	0		0	1	0
0	1	0		0	1	0
1	0	0		1	0	0
0	0	0		1	0	0

State 0	**No Lift**
State 1	**Mini Lift**
State 2	**Full Lift**

Figure 116: VVL state diagram implemented in FPGA

In Figure 117 an example of request and control action on a single bank (3 cylinders) is shown: the request of the bank state is rapidly changed within few engine cycles (camshaft angle is shown in “gold color” in the first picture) and the single cylinders change state as soon as possible, accordingly to the actuation sequence of the solenoids.

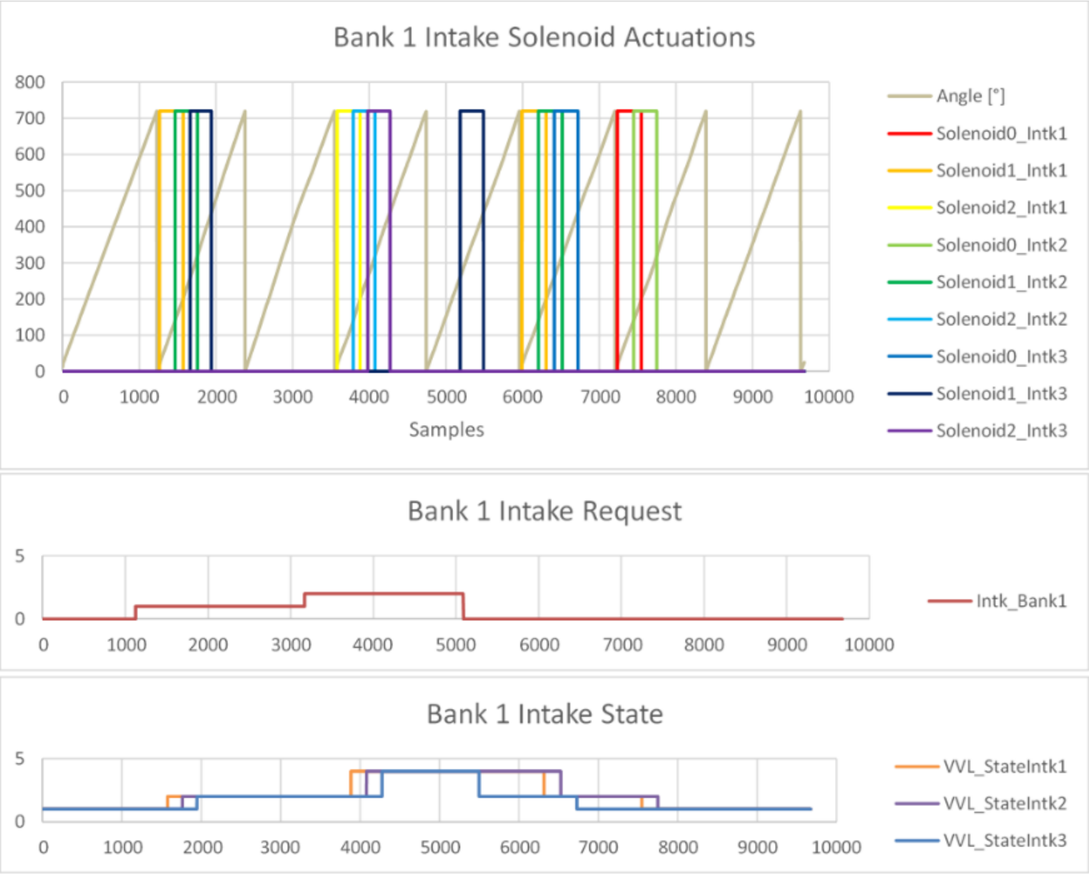


Figure 117: Example of state request (for 1 bank with 3 cylinders) and control action

4.4 HYDRAULIC BRAKE CONTROL

Facing the installation and testing of a new hydraulic brake at the test bench involved the challenge of controlling the brake torque. This kind of machines are characterized by high achievable brake torque, compared to the dimensions, but slow dynamics due to the filling and emptying mechanism that determines the water trapped in the rotor and, therefore, the available torque. The challenge of a good control system is the ability to grant a small oscillation around target speed, and sufficient performance during transients in terms of speed and overshoots. Due to the lack of data and experience on the control of this type of plant, a set of tests have been planned on a previously working installation and the data, together with literature information [20] [21], have been used to produce a Simulink representation of the physical behaviour of the brake [22]. The modelling of high dynamic inlet pump and an outlet valve, together with a simple representation of the engine torque behaviour let us exploit the potential of simulation, validating and pre-calibrating a prototype of the control system.

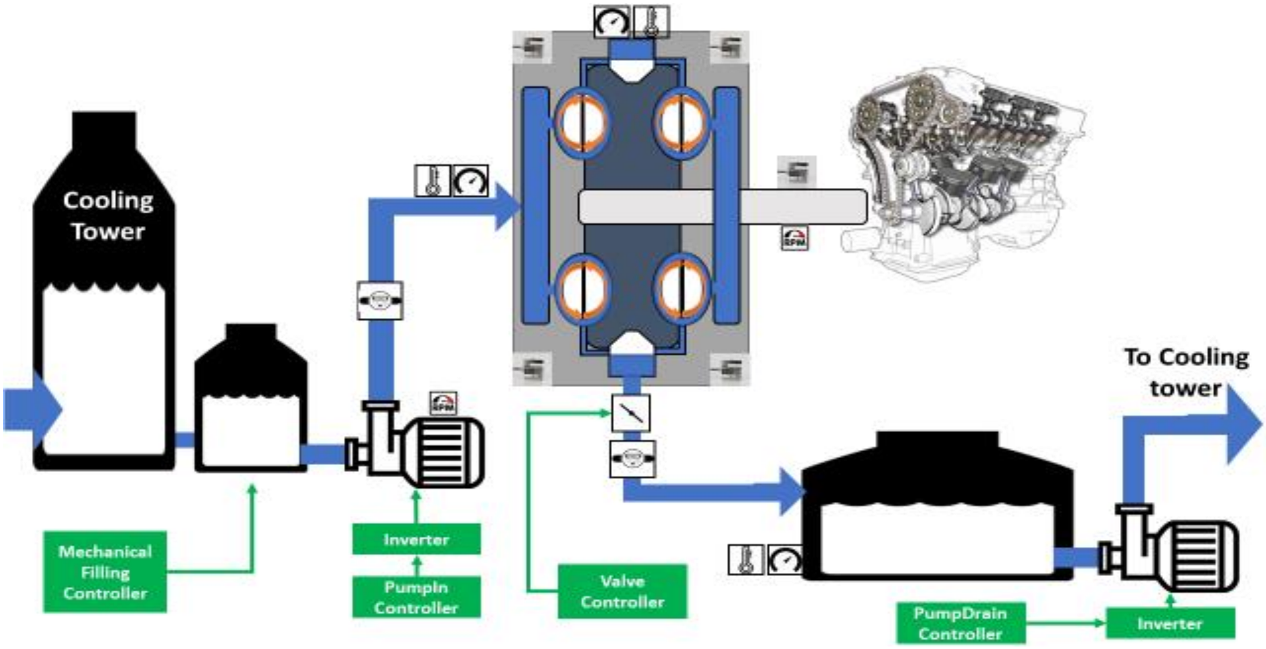


Figure 118: Hydraulic brake plant overview scheme

The physical installation of the brake and the engine (a 2.0 liters spark ignition turbocharged engine equipped with MultiAir) also involved the inlet pump, the outlet valve, the drain pump and the tanks as shown in Figure 118. A wide range of sensors have been displaced in the plant such as temperature and pressure sensors in different points of the circuit, rpm sensor for control and monitoring, and 5 load cells (1 for torque measurement and 4 for the estimation of the water mass filling the brake).

The next step has been the implementation of an RCP system to actually testing the proposed control solution for the brake and managing all the accessories needed by the installation at the test bench. Miracle[2] by Alma-Automotive has been selected as main controller: the computation core of the controller is the same sbRIO 9651 used previously with Spark, and the only differences are a more general purpose I/O and the lack of a power stage for driving actuation (not necessary). A quick adaptation of the low-level drivers let us reuse the previously developed software architecture with all its basic functions and concentrating on the specific RCP implementation. The FPGA implements all the low-level drivers for AI, AO and GPIO while on the RT side the control strategies are executed together with all the base modules of the framework. Interaction and integration with the test bench system is granted through CAN and XCP calibration protocol. Three working modes have been programmed into the controller:

- RPM setpoint: the actuators are controlled in feedback to follow the speed setpoint

- Torque setpoint: the actuators are controlled in feedback to follow the torque setpoint
- Manual Inlet Pump RPM setpoint

All the three modes can be selected both via XCP or directly via CAN from the test bench and while the respective setpoints can be configured also via analog input.

The main actuators to be controlled are three and have their own control SW module in a 4ms task. The main actor for the emptying and filling dynamics and consequently on the braking torque is the **inlet pump.** The pump is controlled by an external inverter whose speed setpoint is sent over CAN by the RCP controller: since the brake control needs to be capable of operating without any information of the engine connected, a simple but smart PID control with gain scheduling has been chosen as the first attempt implementation. For example, when running in RPM setpoint mode the inlet pump is accelerated and decelerated to adjust the inlet flow and maintain the engine speed at target. The **outlet valve**, due to its slow dynamics is controlled mainly as stationary in different positions to produce the same amount of torque with different flow rates and therefore control the outlet water temperature. The valve setpoint is sent over AO and an industrial motor controller maintains the actuator in position. Another actuation demanded to Miracle^2 is the **drain pump** that has the role of keeping within the proper level the tank where the outlet flow is disposed. The level is evaluated through a pressure sensor and used as input to control the pump with hysteresis cycles: the pump has its own inverter and the rpm setpoint is sent through an AO.

From the acquisition point of view several sensors are acquired directly from the controller through standard libraries:

- AI inputs: pressure on the lines, the drain tank level and the throttle valve feedback position
- GPIO inputs: the speed of the brake and of the inlet pump

On the other hand, a cRIO with a wide range of inputs capabilities has been used as high-frequency recorder for analysis and as sensor extension of the Miracle^2 for those sensors that needed specific electrical conditioning, for example load cells for torque measurement and thermocouples. A CAN line has been laid between the controller, the cRIO and the test bench system and provides data exchanging for both control and information purpose.

An example of RPM control with steps in the request is presented in Figure 119, showing an encouraging controllability both in stationary and transient conditions. The example shows two steps from of 200 RPMs and one of 100 RPMs at a constant engine pedal request of 30% and outlet valve at 80%. In correspondence of the RPM request steps are noticeable the drops in the pump speed as action of the control to reduce the inlet water flow, and therefore in the braking torque. It is also evident the lower pump speed required to keep the engine in stationary conditions at higher RPMs because of the descendant torque map with the engine speed. Tests on the brake and calibration of the control are currently ongoing and future step is going to be the replacement of the engine development ECU with the Spark-based engine management system (4.1), gaining advantage of higher control flexibility for advanced testing for example on piezoelectric washers and combustion feedback control.

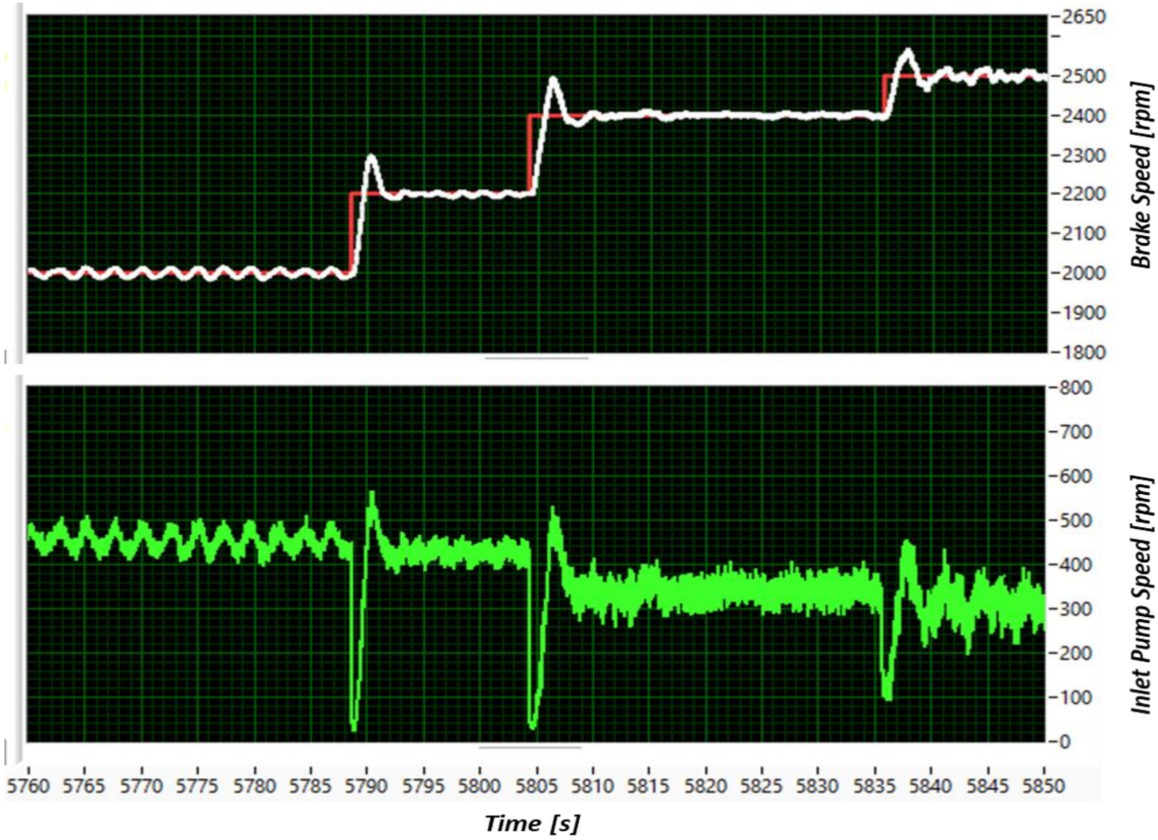


Figure 119: Example of brake control: brake speed (white) setpoint (red) steps and control of inlet pump speed (green). The engine pedal request is constant at 30% and the outlet valve is closed at 80% during the steps.

5. CONCLUSIONS

A high degree of flexibility and independence in the implementation of control and testing solutions for automotive engines has been reached and the basis of a modular/multiuse RCP for automotive control and testing has been developed.

The entire work has been conducted concentrating the efforts to obtain a centralized and programmable control system easily adaptable to the specific needs of various research activities. The final result is open to the future challenges in the automotive world, and not only: it can be an added value wherever a complex and innovative system undergoes to test and verification especially in the early prototypal stage, where both hardware and control algorithms may vary considerably and prototypal solutions must be found both for the system itself and the auxiliary facilities. The natural field of application are innovative control strategies for internal combustion engines, systems for test benches, as well as complex hybrid and electrics powertrains and their management. The advantages introduced by the kind of framework proposed is the capability of speeding up the development and testing circle, making available in a single concept flexibility, extensible integration and ease of use, which are hard to be found in closed RCP environments.

The developed framework has been used for various applications, spacing from full bypass to added functionality RCP, demonstrating an encouraging flexibility. As an indicator of the effective modular structure can be remarked the independent usage by a PhD colleague of the proposed architecture for the realization of research activities on advanced combustions.

Briefly summarizing, the main research goals achieved during the PhD have been:

- The development of a reusable framework for rapid control prototyping implementation, co
- The development of a complete Spark Ignition Engine Management for engine testing, integrating specific software modules for specific actuators control such as throttle, gasoline high pressure pump, VVT and MultiAir.

- The development of SW personalities for advanced combustion control and the integration of combustion analysis functionalities. Reuse by other colleagues of the architecture for the implementation of full engine control systems for specific research tasks.
- The development of SW personalities for the implementation of added functionalities RCP such as the control of water injectors or a variable valve lift system
- The development of SW a personality for hydraulic brake control

Moreover, as presented in Appendix A: hardware in the loop for hybrid powertrains a first attempt implementation of an HIL rig for hybrid vehicle ECUs testing (abroad research period).

APPENDIX A: HARDWARE IN THE LOOP FOR HYBRID POWERTRAINS

In the context of a complex control system the need is to speed up the vehicle development implementing the plant simulation on an HIL to test and verify the control strategies and the interaction of the full control system.

An abroad research period has been attended at Aston Martin Lagonda Ltd. facilities in Wellesbourne, England (UK) and regarded the first attempt development and implementation of a hardware in the loop system to test a hybrid powertrain control system on Speedgoat Simulink Real-Time hardware. The main aim of the system was to implement the plant simulation on the HIL, to test and verify complex control strategies and the interaction of the full control system. Because of the complexity of the control system, based on several ECUs, and the strategies, such as the torque split, an HIL rig is a fundamental facility to speed up the vehicle development avoiding several tests (and issues to be solved) on the real car.

The final target of this HIL rig is to emulate all the sensors and subsystems expected from the control units to make the whole system behave as if it was on the real car running on track or road. The replication of driving profiles and condition is essential to test and validate the control algorithms, as well as the integration of the singles subsystems, without having the real car, still in development, running. The hybrid powertrain simulation involved 5 subsystems. The models of the physical subsystems were already developed or, as in the case of the ICE, rebuilt from data acquisition during previous tests.

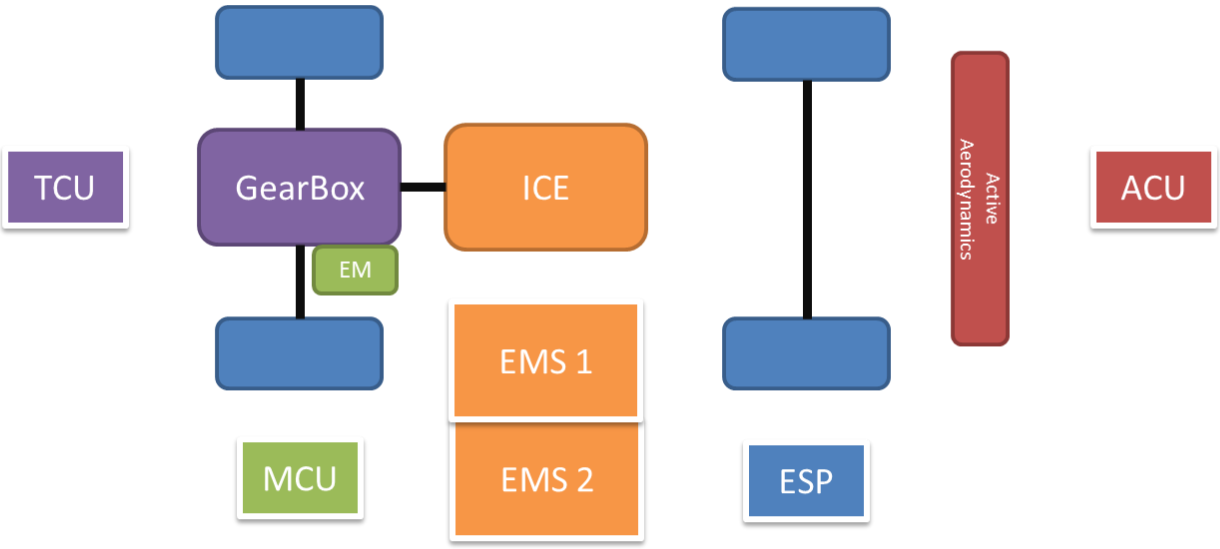


Figure 120: HIL vehicle functional diagram

The internal combustion engine with its control units (EMS)

A two ECUs (one per bench) engine management system is the core of the powertrain control: one acting as master and the other as slave. The main tasks of the EMS are the torque request coordination between combustion and electrical motor and the direct management of the ICE actuators. Starting from the pedal request of the driver EM and ICE torque setpoint are determined through a complex management algorithm for "optimal" split taking in account a long list of vehicle parameters including the battery state of charge. Then the ICE torque request is converted into throttles position, tumble flaps positions, VVTs positions, ignition SA, lambda target etc setpoints. Apart from throttles and knock sensors, directly connected to the ECUs in the HIL, all the other sensors have been emulated. A large dataset from test bench has been used to build a model based on lookup tables in function of the working point (RPM , pedal) that provides all the required sensors physical outputs: pressure, temperatures, VVTs and tumble flaps positions. Since the interest of the HIL was verifying the ECUs integration and strategies, the torque produced by the engine has been simulated simply applying a transfer function to the requested torque: the engine model simply outputs the expected sensor values for the EMS.

The electric motor with its control unit (MCU)

The permanent magnets electric motor on the gearbox is driven by an inverter and its control unit receiving the torque setpoint from the EMS via CAN bus. Motor speed is generated from gearbox speed information and motor temperatures are generated through calibratable parameters. As for

ICE the simulated produced torque is simply the application of a transfer function to the required one.

The vehicle and gearbox with its control unit (TCU)

The gearbox is controlled by an independent transmission control unit talking to the other vehicle ECUs and in particular with the EMS through CAN bus receiving the information for taking decision on how to manage the shifts. A vehicle longitudinal dynamics together with an electro-mechanical model of the gearbox from the manufacturer have been introduced in the HIL software. Torque requests (ICE and EM) from the EMS and PWM actuations from the TCU are the inputs system, while shaft speeds, actuators feedback positions, pressures and temperatures of the gearbox hydraulic system are the outputs to be provided to the TCU.

The wheels with the ESP control unit

The stability control unit is connected on CAN bus interacting with the EMS and influencing the torque demand. From the HIL point of view, all the vehicle dynamics is simulated in the vehicle/gearbox model and what needs to be done is the generation of the wheel-speeds and the brake pedal signals for the ESP.

The active aerodynamics with its control unit (ACU)

The vehicle had active aerodynamics to be controlled by means of a dedicated ACU. A preliminary study of the inputs and outputs necessaries to the simulation have been done, but it has not been implemented in the software because it was not a priority and of lack of information about the behaviour of the real system.

SOFTWARE IMPLEMENTATION

The implementation work involved the choice of the HW modules for the I/O necessary and the SW development of inputs reading from ECUs, physical simulation and output generation for the ECUs. Each HIL simulation subsystem can be synthetically represented by 5 software layers:

- Inputs read from hardware drivers
- Conversion of the inputs read from electrical values to engineering units (if needed)
- Simulation model that generates the values of the sensors to be given as outputs to the ECUs

- Conversion of the outputs from engineering unit to electric units (if needed)
- Outputs write to hardware drivers

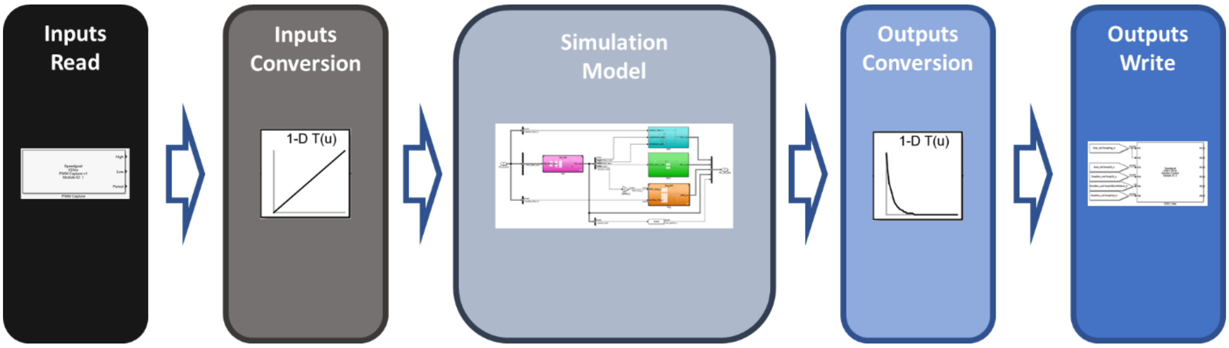


Figure 121: from inputs to outputs scheme

Analysing the list of the required inputs and outputs from the different ECUs involved in the project, a list of the necessaries I/O variables and hardware interface has been listed.

The **driver torque request** is calculated by a longitudinal driver function that follow the rpm data recorded on a track lap and replayed in the model. These operations are purely software and do not involve any I/O.

Position sensors such as tumble flaps feedbacks and pedal position request have been implemented as **analog outputs**, while for example clutch and gearbox barrels position feedbacks were requested from the TCU as **PWM outputs**. Other position feedbacks as gearbox forks proximity sensors, the brake and the reverse gear engaged signal have been implemented as simple **digital outputs**.

All the **pressure sensors** have been implemented as **analog outputs** to feed the respective acquisition ECU: manifold pressure, fuel pressure, oil pressures, gearbox hydraulic circuit pressures. Also the hydraulic fluid **level sensor** of the gearbox has been predisposed to feed the TCU with an **analog out**.

A wide range (in number and type) of **temperature sensor** were required, but only based on thermistor technology. For this reason, **programmable resistors** modules have been selected to reproduce the sensors behaviour and temperature to resistance LUTs have been predisposed for each sensor, allowing a simple configuration for NTCs or PTCs thermistors.

The **crank and the two camshaft signals**, necessary for the phasing of the EMS and the VVT position acquisition have been generated as **digital outputs** through a precompiled **FPGA module**, capable of being configured in the number and the pattern of the teeth and the angular offset between the channels. The VVT position variations are emulated adding an angular shift in the cam signals pattern compared to the crank signal.

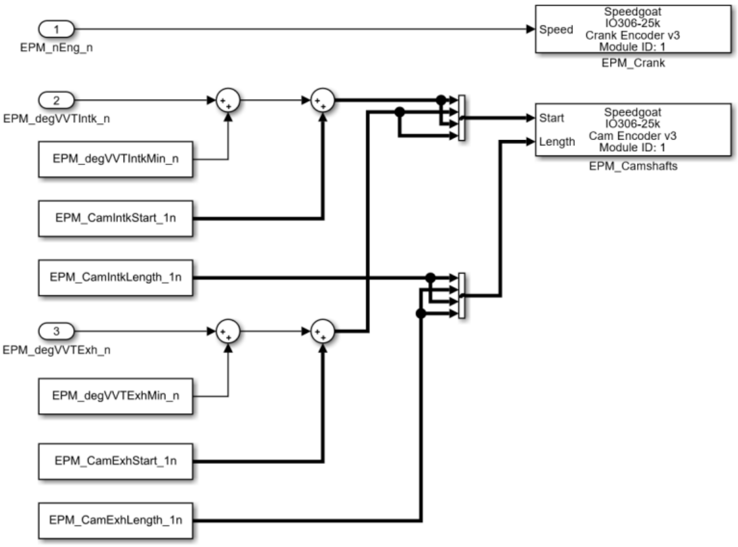


Figure 122: Speedgoat crank & cam signals generation for drivers

Wheel speed sensors to be emulated for the ESP required a current square-wave generation, so a specific module has been chosen for these outputs.

Since the gearbox model used had a very detailed simulation it required the actuation commands from the TCU to produce valid outputs such as the selected gear and so the shafts speed. The TCU **PWM actuation** are captured by **specific FPGA code** modules and reported in the software as simply percentages.

The **CAN communication** has been predisposed for reading signals form the EMS such as the requested torques and for writing the electric motor speed to the MCU

The preliminary study and implementation of the HIL rig software have not been tested integrally because of the immediate availability of hardware components (I/O modules and ECUs) and some missing information regarding the simulation. A reduced SW version has been realized, for EMS testing only, together with the hardware preparation and wiring, and the basic functionalities such as the phasing system, the main analog sensors emulations and CAN communication has been run real-time.

The HIL rig is still in development internally in Aston Martin Lagonda.

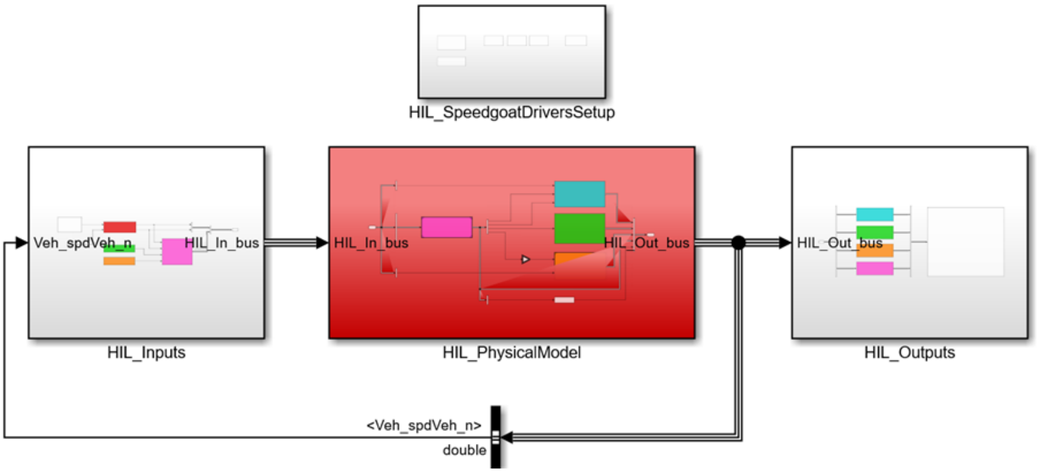


Figure 123: Simulink HIL code overview

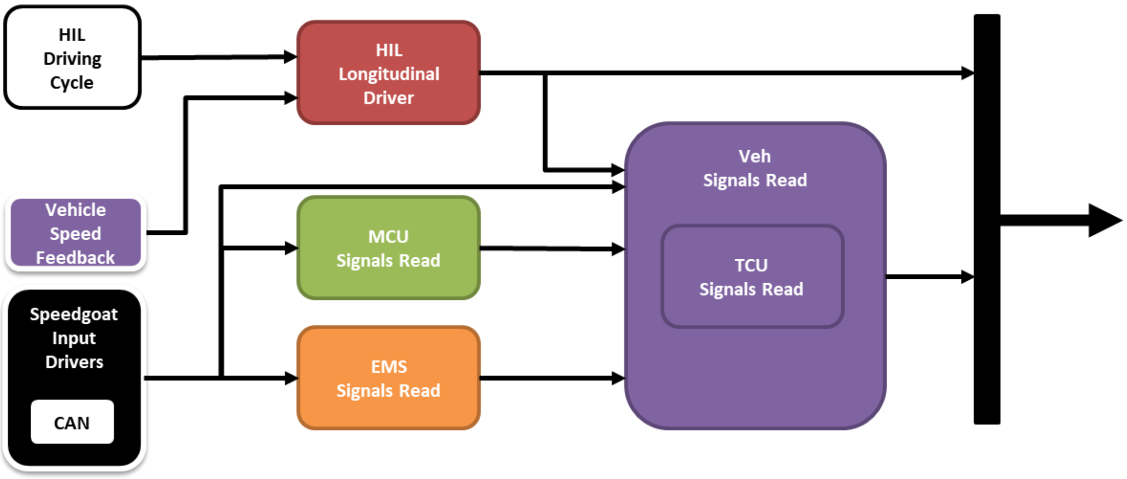


Figure 124: HIL Inputs scheme

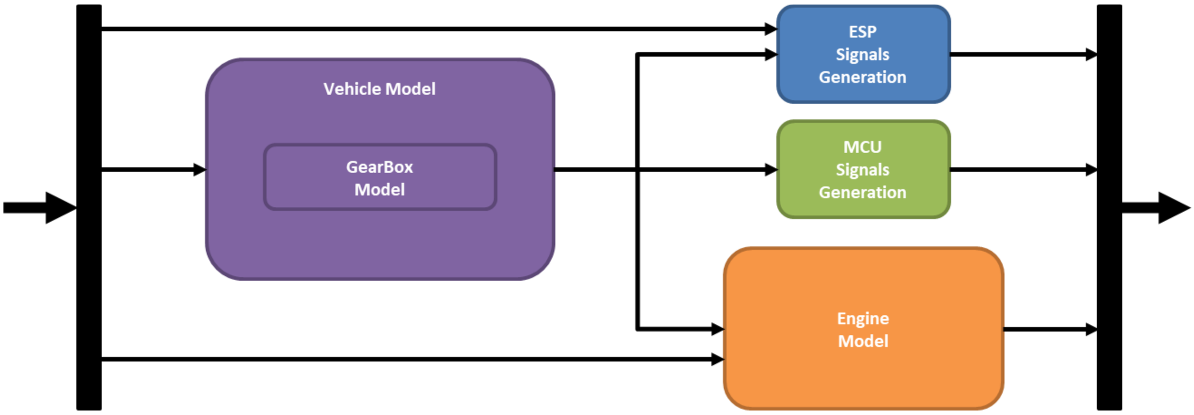


Figure 125: HIL Model scheme

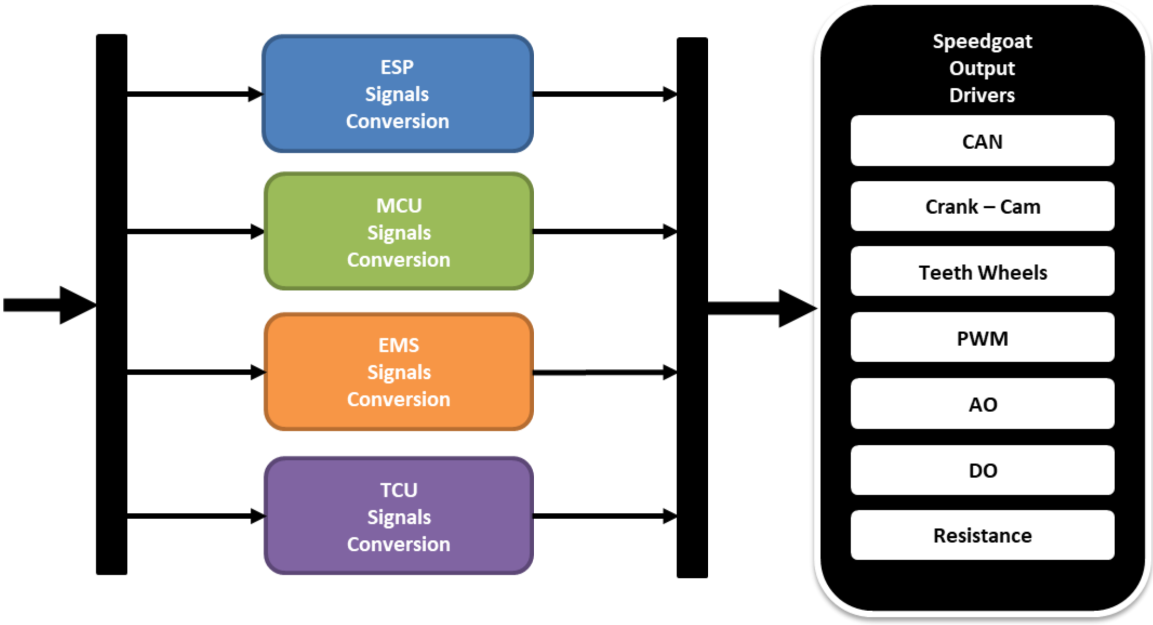


Figure 126: HIL Outputs scheme

APPENDIX B: STANDARD ECU OUTPUTS REVIEW

This appendix is intended to give a short high-level perspective on the standard ECUs outputs that have been used in particular in the context of this thesis work.

General Purpose Input/Output

Are the most basic digital input and output resources available for a generic microcontroller. An electrical voltage level, typically 0-5V or 0-3.3V, is translated in Boolean logic levels False-True. As suggested by the name, generally they can be configured both to produce or read a logic level. Additional logic provides also debouncing between high and low voltage states.

Low-Side and High-Side

LS and HS switches are controllable connection between ground and supply used to power on a device, generic load or an actuator. Typically for ECU actuators application the electric components that implement the switch are transistors. In the low-side configuration the load device is always connected to supply voltage and the switch interrupts the linking with the ground, while in the high side configuration the scheme is the opposite. The control unit (ECU) is demanded to drive the state of the switches to let the current flow through the load, that in the case of an automotive application can be for example a PFI injector, an Ignition Coil, waste-gate actuator or a generic valve.

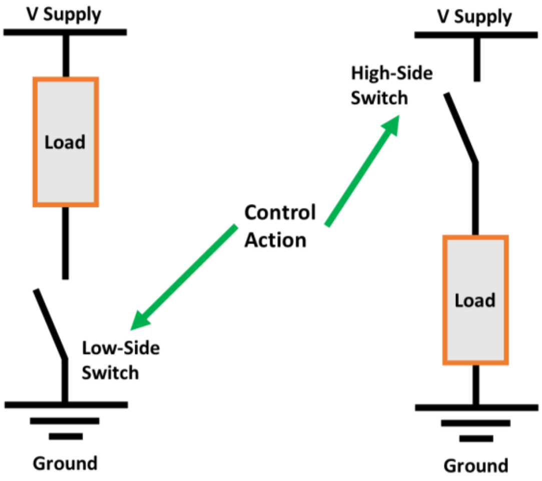


Figure 127: Low-Side and High-Side scheme

LS and HS drivers are often controlled by means of PWM (Pulse Width Modulation) control input for actuators that require a command modulation: the switch is changed alternatively from high to low state allowing to modulate the effect of the control in function of the “High Time” and “Low Time” (Figure 128). The switching frequency must be higher than electro-mechanical dynamics capabilities of the actuator in order to produce an output equivalent to a lower mean voltage. The value of the mean voltage equivalent to the PWM signal is represented by the “Duty Cycle” ratio multiplied by the effective supply voltage as defined in Equation 5.

Equation 5: PWM signal parameters definition

$$Period\ Time = \frac{1}{PWM\ Frequency} = High\ Time + Low\ Time$$

$$\boldsymbol{V\ Equivalent\ = V\ Supply * Duty\ Cycle}$$

$$Duty\ Cycle\ = \frac{High\ Time}{Period\ Time}$$

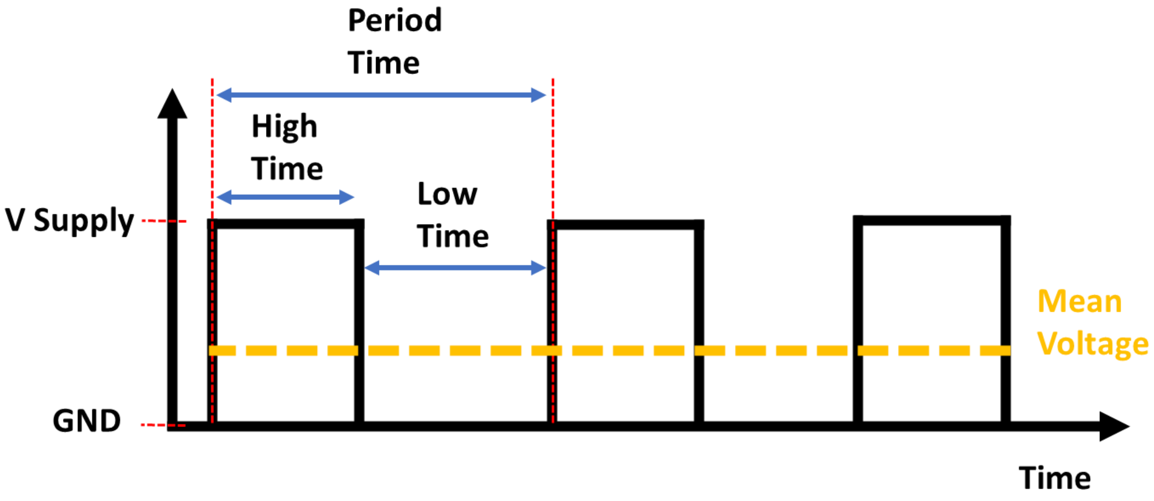


Figure 128: PWM Scheme

H-Bridge

An H-Bridge is an electronic circuit capable of commuting the polarity of a voltage supply applied to a load. Form a simple hardware point of view is a combination of 2 LS and 2 HS (Figure 129), while form the functional point of view it can be controlled in PWM mode (as a simple HS or LS) and,for example if used to drive an electric motor actuator, is also capable of reversing and braking the motion of the actuator (not possible with a simple LS or HS). For this reason is often used to drive actuators that require the possibility of a bidirectional control: main examples from automotive applications are electric throttle bodies or fully electric actuated waste gate valves.

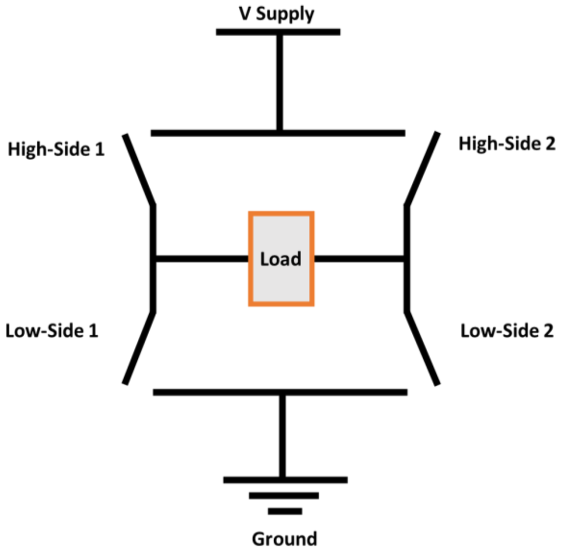


Figure 129: H-Bridge Scheme

The operation condition for a motor controlled with H-Bridge is determined by the Table 2

Table 2: H-Bridge Truth Table

High-Side 1	Low-Side 1	High-Side 2	Low-Side 2	**Operation State**
1	0	0	1	**Forward**
0	1	1	0	**Reverse**
0	0	0	0	**Coasting**
1	0	0	0	
0	1	0	0	
0	0	1	0	
0	0	0	1	
0	1	0	1	**Braking**
1	0	1	0	
X	X	1	1	**Short Circuit**
1	1	X	X	

Peak and Hold

Peak and Hold current control is a method of excitation for actuators based on the regulation of the current flowing in different steps. Typically, is the chosen method for driving Direct Injectors since it is possible to let flow a high current at first time (peak phase) to win the high-pressure contrast and open the injector quickly, and then set the current to a lower level just to maintain the nozzle opened (hold phase). Generally, these kinds of drivers are supplied with a boosted voltage, higher than the battery voltage, in order to permit faster current dynamics. The current level is generally driven in feedback by means of a PWM control.

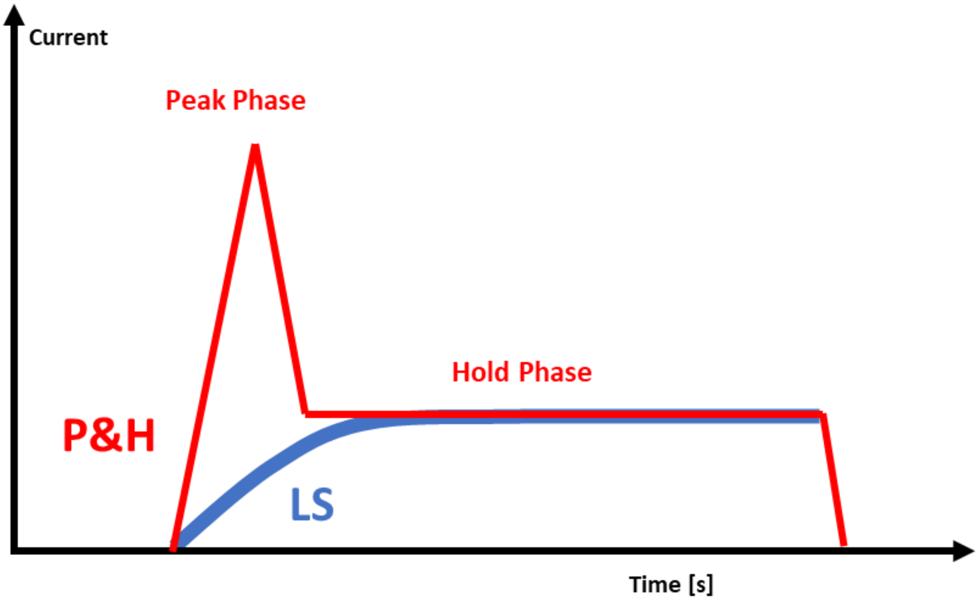


Figure 130: Peak and Hold actuation compared to a Low-Side actuation with the same final current level

FIGURES INDEX

REFERENCES

[1] National Instruments Corporation, “Building a Real-Time System With NI Hardware and Software,” [Online]. Available: https://www.ni.com/content/ni/locales/it-it/innovations/white-papers/06/building-a-real-time-system-with-ni-hardware-and-software.html.

[2] The MathWorks, Inc., “Rapid prototyping,” [Online]. Available: http://www.mathworks.com/rapid-prototyping/embedded-control-systems.html.

[3] Schaeffler ,Haas M., *UniAir – The first fully-variable, electro-hydraulic valve control system,* 2010.

[4] Schaeffler, Haas M., Piecyk T., *Get Ready for the Combustion Strategies of Tomorrow,* 2014.

[5] A. Thomasson e L. Eriksson, «Model-Based Throttle Control using Static Compensators and IMC based PID-Design,» in *IFAC Workshop on Engine and Powertrain Control, Simulation and Modeling Đaris (France)*, November 30 - December 2, 2009.

[6] A. Thomasson, *Modeling and Control of actuators co-surge in turbocharged engines,* Linköping studies in science and technology, 2014.

[7] V. Ravaglioli, F. Ponti, M. De Cesare, F. Stola e et al., «Combustion Indexes for Innovative Combustion Control,» *SAE Int. J. Engines 10(5):2371-2381,* 2017.

[8] M. De Cesare, V. Ravaglioli, F. Carra e F. Stola, «Review of Combustion Indexes Remote Sensing Applied to DifferentCombustion Types,» *SAE Technical Paper 2019-01-1132,* 2019.

[9] N. Cavina, E. Corti, L. Poggio e D. Zecchetti, «Development of a Multi-Spark Ignition System for Reducing Fuel Consumption and Exhaust Emissions of a High Performance GDI Engine,» *SAE Technical Paper 2011-01-1419,* 2011.

[10] E. Corti e C. Forte, «Real-Time Combustion Phase Optimization of a PFI Gasoline Engine,» *SAE International 2011,* 2011.

[11] K. P. Quillen, M. Viele e S. A. Ciatti, «Next-Cycle and Same-Cycle Cylinder Pressure Based Control of Internal Combustion Engines,» in *Proceedings of the ASME 2010 Internal Combustion Engine Division Fall Technical Conference ICEF2010 September 12-15, 2010, San Antonio, Texas, USA.*, 2010.

[12] E. Corti, M. Abbondanza, F. Ponti e L. Raggini, «The Use of Piezoelectric Washers for Feedback Combustion Control,» in *WCX SAE World Congress Experience*, 2020.

[13] M. Abbondanza, N. Cavina, D. Moro, F. Ponti e V. Ravaglioli, «Development of a Combustion Delay Model in the Control of Innovative Combustions,» in *75° National ATI Congress*, 2020.

[14] V. Ravaglioli, F. Ponti e De Cesare M., «Investigation of Gasoline Compression Ignition for Combustion Control,» *Journal of Engineering for Gas Turbines and Power,* 2020.

[15] V. Ravaglioli, F. Ponti, F. Carra e M. De Cesare, «Heat Release Experimental Analysis for RCCI Combustion Optimization,» in *ASME 2018 Internal Combustion Engine Division Fall Technical Conference*, 2018.

[16] V. Ravaglioli, F. Carra, D. Moro, M. De Cesare e F. Stola, «Remote Sensing Methodology for the Closed-Loop Control of RCCI Dual Fuel Combustion,» in *WCX World Congress Experience*, 2018.

[17] V. Ravaglioli, F. Ponti e M. De Cesare, «Investigation of Gasoline Compression Ignition for Combustion Control,» in *ASME 2019 Internal Combustion Engine Division Fall Technical Conference*, 2019.

[18] G. Silvagni, V. Ravaglioli e F. Ponti, «A review of remote-control strategies for reactivity controlled compression ignition combustion,» in *AIP Conference Proceedings 2191(1):020138*, 2019.

[19] F. Stola, M. De Cesare, V. Ravaglioli, G. Silvagni e F. Ponti, «Injection Pattern Investigation for Gasoline Partially Premixed Combustion Analysis,» *SAE International,* 2019.

[20] J. K. Raine e P. G. Hodgson, *Computer simulation of a variable fill hydraulic dynamometer. Part 1: torque absorption theory and the influence of working compartment geometry on performance,* vol. IMeche Vol.205, 1991.

[21] J. K. Raine e P. G. Hodgson, «Computer simulation of a variable fill hydraulic dynamometer. Part2: steady state and dynamic open loop performance,» *Journal of Mech. Eng. Science,* vol. Vol. 206 vols, 1992.

[22] E. Corti, N. Rojo, M. Abbondanza e L. Raggini, «Application of a Model for optimizing steady state and transient control of Hydraulic dynamometers,» in *74th Conference of the Italian Thermal Machines Engineering Association (ATI 2019)*, Modena, Italy, 2019.

[23] Altera Corporation, *Driving Flexibility into Automotive Electronics Design, white paper WP-01025-2.0,* 2020.

[24] N. Cavina, N. Rojo, A. Businaro, A. Brusa et al., «Investigation of Water Injection Effects on Combustion Characteristics of a GDI TC Engine,» *SAE Int. J. Engines 10(4):2017,* 2017.

[25] E. Corti, G. Cazzoli, M. Rinaldi e L. Solieri, «Fast Prototyping of a Racing Diesel Engine Control System,» *SAE Technical Paper 2008-01-2942,* 2008.

[26] S. Shanker, Enhancing Automotive Embedded Systems with FPGAs, 2016.

[27] Alma-Automotive s.r.l, «Spark, the open source ECU project,» [Online]. Available: http://www.alma-automotive.it.

[28] L. Eriksson e L. Nielsen, Modeling and Control of Engine and Drivelines, Wiley, 2014.

[29] L. Guzzella e C. H. Onder, Introduction to Modeling and Control of Internal Combustion Engine Systems, Springer, 2010.

[30] M. De Cesare, F. Ranuzzi, N. Cavina, G. Scocozza e A. Brusa, «Experimental Validation of a Model-Based Water Injection Combustion Control System for On-Board Application,» *SAE International,* 2019.

[31] Schaeffler, Nitz R., Elendt H., Ihlemann A., Nendel A., *INA cam shift ing system, Prepared for the future,* 2010.

A chi ha reso possibile questo percorso

A chi lo ha vissuto insieme a me

A chi ha condiviso anche solo un momento

A chi mi ha accolto

A chi ha speso una parola od un pensiero gentile

A chi è stato un buon amico

A chi un buon amico lo è diventato

A chi è stato un buon maestro

A chi c'è stato ed anche a chi non c'è stato

Ai primi ed agli ultimi

Ai ricordi che ci portiamo dentro ed a quelli che dobbiamo ancora costruire

All'entusiasmo ed alla passione

Alla stanchezza ed alla tristezza

Alla scienza ed all'arte

Alla ragione ed alle emozioni

A me, che ho speso tanto di me stesso ed ancora ho da spendere

A chi conserva gelosamente la scintilla di un'umanità vera e sincera

A chi non sarà mai abbastanza cinico da smettere di credere che il mondo possa essere migliore di com'è