POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Firmware development for an automotive powertrain control item

Supervisors

Prof. Massimo VIOLANTE

Ing. Jacopo SINI

Candidate

Stefano DE CARO

October 2020

Summary

Environmental concern, fuel efficiency and new market trends push automotive companies, governments and research institutes to explore environmentally-friendly, efficient and sustainable personal and public transportation solutions. Automotive industry is following this trend developing new eco-friendly, smart, and connected vehicles. In this scenario, powertrains are one of vehicle subsystems undergoing several changes in architecture and implementation in favor of electric and hybrid solutions. These innovations within the automotive domain are driven by embedded systems and software solutions. It can be observed that the costs for embedded solutions in vehicles are growing rapidly while mechanical engineering based solutions are stagnating in importance.

The motivations of this thesis work are inserted in context related to the new trends of the automotive industry. To be more precise, the purpose of this work was to develop the low-level software for an Electronic Control Unit (ECU) responsible to control an electric powertrain. First, a review of the theoretical aspects of the features to be implemented and required by the system was made to provide a background knowledge about the discussed topics. Then, before to start developing the firmware, a software architecture was carefully defined through multiple software modules, also called software components, with well-defined interfaces. This approach was needed to hide the implementation details of the low-level software and increase the software modularity. The development process, aiming to verify the system requirements, was carried on through a bare-metal approach without any real-time operating system and using the C programming language. Once a first version of the firmware was ready, the testing and verification process was performed using common laboratory instrumentation and debug and trace tools provided by Lauterbach to interact with the microcontroller. Test programs and procedures were developed for each software component to verify the required features trying to maintain the independence with respect to their specific software implementations. Then, the obtained results were analyzed showing correspondences with the expected system responses. As last activities, the integration of the application software, obtained through automatic code generation tools, was performed. Furthermore, a processor-in-the-loop simulation

of the motor control algorithm, implemented as a periodic task in the application software, was executed to analyze its worst case execution time and demonstrate the feasibility of its implementation on the target platform. In conclusion, final considerations and further improvements were pointed out.

Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this thesis, and without whom I would not have completed my engineering studies.

The DUAIN department of the Politecnico di Torino, especially to my supervisors Prof. M. Violante, whose allows me to participate to this extraordinary project. And special thanks to Filippo and Jacopo, whose constant support and availability allowed me to carry on my thesis even in this moment of health emergency due to Covid 19.

Special thanks to my family for all the support you have shown me through this years as engineering student without whom I could not have the possibility to study far away from home. And finally, but not least, my biggest thanks to my girlfriend Ilenia for her patience and the constant encouragement during these special years of our lives.

Stefano De Caro

Table of Contents

List of Tables IX

List of Figures X

1 Introduction 1
- 1.1 Electric Vehicles . . . 1
  - 1.1.1 Electric Powertrain . . . 4
- 1.2 Embedded systems role . . . 5
- 1.3 Firmware development . . . 6
- 1.4 Thesis outline . . . 7

2 Theoretical Background 8
- 2.1 ARM Architecture . . . 8
  - 2.1.1 ARM Cortex-M7 . . . 9
- 2.2 Communication Protocols . . . 16
  - 2.2.1 Serial Peripheral Interface . . . 16
  - 2.2.2 Controller Area Network . . . 18
- 2.3 Analog-to-Digital Conversion . . . 21
  - 2.3.1 Successive Approximation ADCs . . . 24
- 2.4 Pulse Width Modulation . . . 26
- 2.5 Power Electronics . . . 30
  - 2.5.1 Inverter . . . 31
- 2.6 Resolver . . . 33

3 Firmware Development 36
- 3.1 Clock Management . . . 38
  - 3.1.1 Requirements . . . 38
  - 3.1.2 Implementation . . . 39
  - 3.1.3 API . . . 42
- 3.2 Pin Management . . . 42
  - 3.2.1 Requirements . . . 43

3.2.2 Implementation . . . 43
3.2.3 API Layer . . . 44
3.3 ADC Management . . . 47
3.3.1 Requirements . . . 47
3.3.2 Implementations . . . 48
3.3.3 API layer . . . 52
3.4 PWM Management . . . 55
3.4.1 Requirements . . . 55
3.4.2 Implementations . . . 56
3.4.3 API layer . . . 59
3.5 SPI Communication . . . 63
3.5.1 Requirements . . . 63
3.5.2 Implementation . . . 65
3.5.3 API Layer . . . 68
3.6 Digital Signal Management . . . 71
3.6.1 Requirements . . . 71
3.6.2 Implementation . . . 71
3.6.3 API Layer . . . 73
3.7 CAN Communication . . . 75
3.7.1 Requirements . . . 75
3.7.2 Implementation . . . 76
3.7.3 API Layer . . . 81

4 Firmware Verification and Testing 84
4.1 Structure of Test Procedures . . . 86
4.1.1 Initialization and Startup of Software Components . . . 87
4.1.2 Test Program Flow and Target Platform Interaction . . . 88
4.2 Laboratory Instrumentation . . . 91
4.2.1 Lauterbach Debug and Trace . . . 92
4.2.2 Oscilloscope . . . 94
4.2.3 Peak PCAN-USB . . . 95
4.2.4 Signal Generator . . . 97
4.2.5 Power Supply . . . 98
4.3 Clock Management Testing . . . 100
4.3.1 Test Procedure . . . 100
4.3.2 Results . . . 102
4.4 PWM Management Testing . . . 105
4.4.1 Test procedure . . . 105
4.4.2 Results . . . 108
4.5 ADC Management Testing . . . 111
4.5.1 Test procedure . . . 112

4.5.2 Results . . . 115
4.6 SPI Communication Testing . . . 119
4.6.1 Test procedure . . . 120
4.6.2 Results . . . 124
4.7 Digital Signal Management Testing . . . 128
4.7.1 Test procedure . . . 129
4.7.2 Results . . . 131
4.8 CAN Communication Testing . . . 133
4.8.1 Test procedure . . . 136
4.8.2 Results . . . 139

5 Software Integration Process 143
5.1 Software Components Integration . . . 145
5.1.1 Initialization and Startup Sequence . . . 145
5.1.2 Software Components Integration Test . . . 146
5.2 Processor-in-the-Loop Simulation . . . 150
5.2.1 Processor-in-the-Loop Implementation . . . 151
5.2.2 Processor-in-the-loop Results . . . 153
5.3 Firmware and Application Integration . . . 155
5.3.1 Software Environment Initialization . . . 156
5.3.2 Software Component Scheduler . . . 158

6 Conclusions 164

Bibliography 166

List of Tables

3.1 Frequency limits of the main clock signals present in the microcontroller. . . . 39
3.2 Peripheral/Module clock input signals to enable and their corresponding maximum supported frequencies. . . . 40
3.3 Groups of analog signals to be acquired with their corresponding sampling rate and timing requirements. . . . 48
3.4 List of PWM signals with their corresponding switching frequency, duty cycle variable, and polarity. . . . 56
3.5 Timing requirement of SPI communication between the MCU and the resolver-to-digital converter. . . . 65
3.6 Parameters of the peripheral FlexCAN-FD used to configure the required time segmentation and bit rate of the CAN node represented by the MCU. . . . 78

4.1 Measurements of the MCU clock frequencies (test 1). . . . 103
4.2 Results and measurements of the PWM generation with fixed duty cycles (test 1). . . . 108
4.3 Measurements performed during constant analog signal acquisition (test 1). . . . 115
4.4 Acquisition timing measurements about the triple of signals in the BCTU conversion list (test 3). . . . 118
4.5 Timing requirement and measurements of SPI communication between the MCU and the resolver-to-digital converter. . . . 126

List of Figures

1.1 Generic architecture of an electric vehicle. 2
1.2 Architecture of an electric vehicle. 3

2.1 ARM Cortex-M7 processor core architecture. 10
2.2 Structure of the core register set of ARM Cortex-M7 processor. . . 11
2.3 Addressable memory space of ARM Cortex-M7 processor. 13
2.4 Vector table of ARM Cortex-M7 processor. 15
2.5 Block diagram of master (on the left) and slave (on the right) connection in SPI communication. 17
2.6 Example of SPI peripheral implementation using circular buffers. . 18
2.7 Representation of CAN-bus network with nodes. The components of a node can be seen in Node 1. 18
2.8 Electric signals in the CAN-bus are represented in red and green with their corresponding logical and single-ended signals. 19
2.9 Example of a complete CAN standard data frame. 20
2.10 Structure of a bit in CAN-bus. 21
2.11 Sampling (on top) and quantization (on bottom) of an analog signal. 22
2.12 Scheme of a sampling and hold circuit. 23
2.13 Example of real and ideal ADC characteristics. 24
2.14 Block scheme of a successive approximation analog-to-digital converter. 25
2.15 Timing diagram of partial results of a Successive Approximation ADC. 26
2.16 Example of PWM signal with 30% of duty cycle. 27
2.17 Three PWM signals with the same duty cycles and switching frequency - in blue leading edge aligned, in red trailing edge aligned, and in green centered aligned. 28
2.18 PWM generation with a sinusoidal modulating wave. 29
2.19 Different carrier waveforms - on top a triangular carrier, in the middle a trailing sawtooth carrier, and on bottom leading sawtooth carrier. 30
2.20 Two complementary PWM signals with deadtime insertion. 31
2.21 Basic structure of a three phase inverter. 32

2.22 Example of a three phase sinusoidal PWM generation. 33
2.23 Output signal of a resolver. The excitation wave on the top, sine feedback in the middle, and cosine feedback on the bottom. 34

3.1 UML class diagram of an example of software component. 37
3.2 UML class diagram of the software component Clock Management. 38
3.3 Clock signals obtained using PLL as clock source. 40
3.4 Clock signals obtained using FIRC as clock source with its maximum working frequency. 41
3.5 UML class diagram of the software component Pin Management. . . 43
3.6 UML class diagram of the software component ADC Management. . 47
3.7 Timing constraints of the phase currents acquisition. 49
3.8 Logic diagram of the multiple parallel conversions list operations implemented by the BCTU. 50
3.9 UML class diagram of the software component PWM Management. 55
3.10 Time evolution of the internal counter of an eMIOS channel configured in MCB mode. 57
3.11 Time evolution of internal counter of an eMIOS channel configured in OPWMCB mode with lead deadtime insertion. 58
3.12 UML class diagram of the software component SPI Communication. 63
3.13 Timing diagram of the serial input sequence to configure the resolver-to-digital converter. 64
3.14 Timing diagram of the serial output sequence to acquire data from the resolver-to-digital converter. 64
3.15 Structure of the data transmitted through SPI by the resolver-to-digital converter. 66
3.16 Structure of the configuration frame transmitted through SPI to the resolver-to-digital converter. 67
3.17 UML class diagram of the software component Digital Signal Management. 71
3.18 UML class diagram of the software component CAN Communication. 75
3.19 Logic diagram of a FlexCAN-FD peripheral connected to CAN bus. 76
3.20 Structure of the register CTRL1 of the peripheral FlexCAN-FD. . . 77
3.21 Structure of a Message Buffer for the peripheral FlexCAN-FD. . . . 79

4.1 Sequence diagram of the structure of test procedures. 86
4.2 Example of laboratory instrumentation setup adopted during the tests of the software component CanMgm. 91
4.3 Complete structure of the Lauterbach hardware tools with the corresponding connection to the target platform [7]. 92

4.4 Lauterbach Debug and Trace hardware tools structure used for firmware testing [7]. 93
4.5 The oscilloscope Yokogawa DLM3054. 95
4.6 The PCAN-USB FD. 96
4.7 An example of the bus load window of the software PCAN-View. . 97
4.8 The signal generator Tektronix AFG1022. 98
4.9 The power supply GWInstex GPS4303. 99
4.10 Structure of the multiplex and divider of the MC_CGM module connected to the signal CLKOUT_RUN. 101
4.11 Frequency measurement of the MCU clock signal HSE_IPG_CLK. 103
4.12 Configuration and status registers of a peripheral with enabled clock gating. 104
4.13 Screenshot of the acquisition of two PWM complementary signals with switching period measurement. 109
4.14 Screenshot of the acquisition of two PWM complementary signals with dead-time measurement. 109
4.15 Screenshot of the acquisition of the enabling of two PWM complementary signals. 110
4.16 Spectrum analysis of the PWM signals when the duty cycles are set by the V/f control algorithm. 111
4.17 Statistical figure of the time distance between two consecutive interrupts of the BCTU peripheral due to the end of the conversion list. 116
4.18 Digital raw values acquired from an analog input connected to a sinusoidal waveform observed with Trace32. 117
4.19 Digital raw values acquired from an analog input connected to a triangular waveform observed with Trace32. 117
4.20 Setup used to test the software component SPI Management when the target platform is connected to the resolver and the R2D converter. 120
4.21 Screenshot of the oscilloscope reporting a generic frame sent by the MCU. 125
4.22 Correspondence between the frame sent through the MOSI line and the one acquired back by the MCU itself. 126
4.23 SPI frame sent by the MCU to configure the R2D converter at startup. 127
4.24 Time evolution of the variable R2dSpiData acquired through the trace feature using the Lauterbach debugging tools. 128
4.25 Screenshot of the acquisition of a digital output signal with negative polarity when the corresponding virtual button is set to 1. 132
4.26 Screenshot of the acquisition of the digital output signal with negative polarity when the corresponding virtual button is set to 0. 133

4.27 Screenshot of the acquisition of the digital input signal when the corresponding voltage level is set to 1. 134
4.28 Screenshot of the acquisition of the digital input signal when the corresponding voltage level is set to 0. 134
4.29 Time evolution of the variable R2dSpiData and R2dDigData acquired through the trace feature using the Lauterbach debugging tools. . . 135
4.30 Values of the variables R2dSpiData and R2dDigData acquired respectively through the serial and digital ports of the resolver-to-digital converter. 136
4.31 Screenshot representing the correspondence between the messages sent by the MCU node and the ones acquired by the PCAN-USB node. 139
4.32 Screenshot representing the correspondence between the messages sent by the PCAN-USB node and the ones acquired by the MCU node. 140
4.33 Screenshot representing the correspondence between the messages exchanged between the PCAN-USB node and the MCU node. . . . 141
4.34 Measurement of the bus load in conditions as close as possible to the nominal operating ones. 141
4.35 Screenshot of the acquisition of the TX message 02 sent by the MCU node. 142

5.1 Block scheme of the processor-in-the-loop simulation of the task representing the motor control algorithm of the application software implemented on the target platform. 150
5.2 Lauterbach Trace32 environment used to perform the PIL simulation. In the figure, the output data structure of the application software, corresponding to data structure Control_Y, is called LlcOutputStruct. 153
5.3 Time evolution of the controlled variable of the plant obtained by PIL simulation. 154
5.4 Time evolution of the controlled variable of the plant obtained by SIL simulation. 155
5.5 Statistical figure about the execution time of the control motor task of the application software. 155
5.6 Software architecture including the device drivers provided by the silicon-vendor, the firmware constituted by the developed software components, and the application software obtained through automatic code generation process. 156
5.7 Graphical representation of the time instants related to the software application execution and firmware acquisitions. 159

5.8 Timeline of the scheduling algorithm implemented through the software component Scheduler. 160

Chapter 1

Introduction

Environmental concern, fuel efficiency and new market trends push automotive companies, governments and research institutes to explore environmentally-friendly, efficient and sustainable personal and public transportation solutions. Automotive industry is following this trend developing new eco-friendly, smart and connected vehicles. In this scenario, powertrains are one of vehicle subsystems undergoing several changes in architecture and implementation in favour of electric and hybrid solutions. The motivations of this thesis are the result of these trends in transportation industry.

This thesis was carried out with the collaboration of the Department of Automation and Computer Science (DAUIN) of the Polytechnic of Turin. The purpose of this thesis is to project and develop an embedded system firmware responsible to manage and control a proof of concept of a full electric powertrain. Some implementation details and technological aspects of the project are protected by a non disclosure agreement and have been neglected. An overview of embedded systems for automotive purpose and electric vehicles will be provided in this chapter to better understand the application of this thesis.

1.1 Electric Vehicles

An electric vehicle (EV) is a vehicle based on one or multiple motor (electric or traction) to ensure propulsion. The degree of electrification varies from one to another through a scale from zero (internal combustion engine vehicles) to one (all electric vehicles). In the context of this thesis, any kind of vehicles powered by fossil fuels have been neglected and only full EVs have been considered. Almost every EV is Battery Electric Vehicle (BEV). BEVs make use of a high capacity battery to store energy. So, they derive all the power from their batteries pack and have no internal combustion engine (ICE), neither fuel cell, nor fuel tank. The

only way to recharge its batteries is by plugging in the vehicle to a charging point.

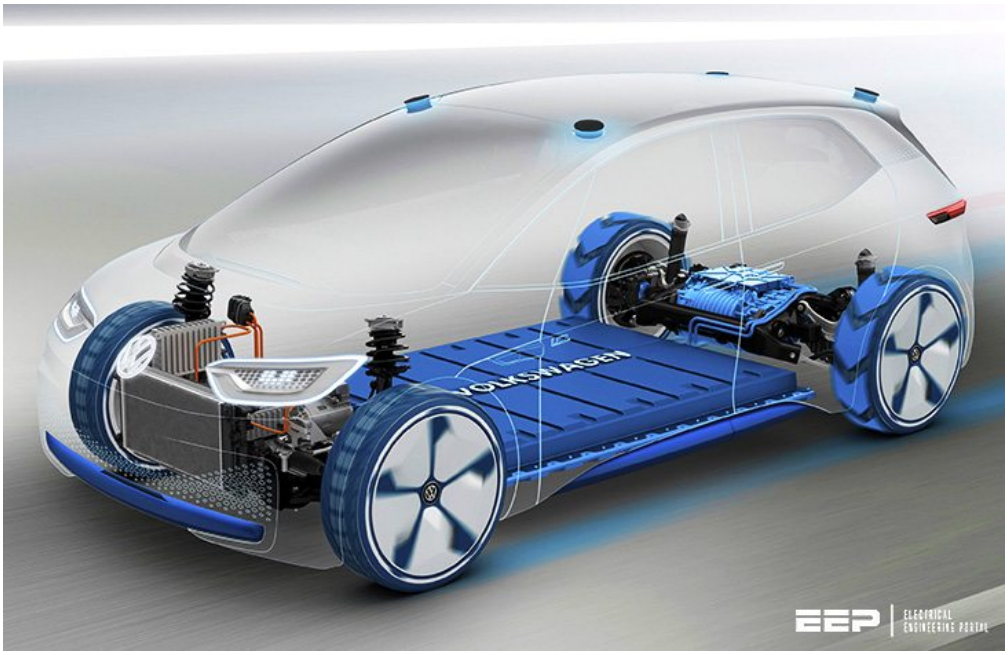

Figure 1.1: Generic architecture of an electric vehicle.

The composition of an EV is shown in Fig. 1.2. The architecture consists in three major subsystem - electric propulsion, energy source, and auxiliary. The electric propulsion subsystem comprises the electronic controller, power converter, electric motor, mechanical transmission, and driving wheels. The energy source subsystem involves the energy source, the energy management unit, and the energy refuelling unit. The auxiliary subsystem consists of the power steering unit, temperature control unit, and auxiliary power supply. In Fig. 1.2, a mechanical link is represented by a double line, an electrical link by a thick line, and a control link by a thin line. The arrow of each line denotes the direction of electrical power flow or control information communication. Based on the control inputs from the brake and accelerator pedals, the electronic controller provides proper control signals to switch on or off the power devices of the power convert which function is to regulate power flow between the electric motor and the energy source. The backward power flow is due to regenerative breaking of the EV and this regenerative energy can be stored provided that energy source is receptive. The most available EV batteries readily accept regenerative energy. The energy management unit cooperates with the electronic controller to control regenerative braking and its energy recovery. It also works with the energy refuelling and to monitor usability of the energy source. The auxiliary power supply provides the necessary power with different voltage levels for all EV auxiliaries, especially the temperature control and the steering

units.

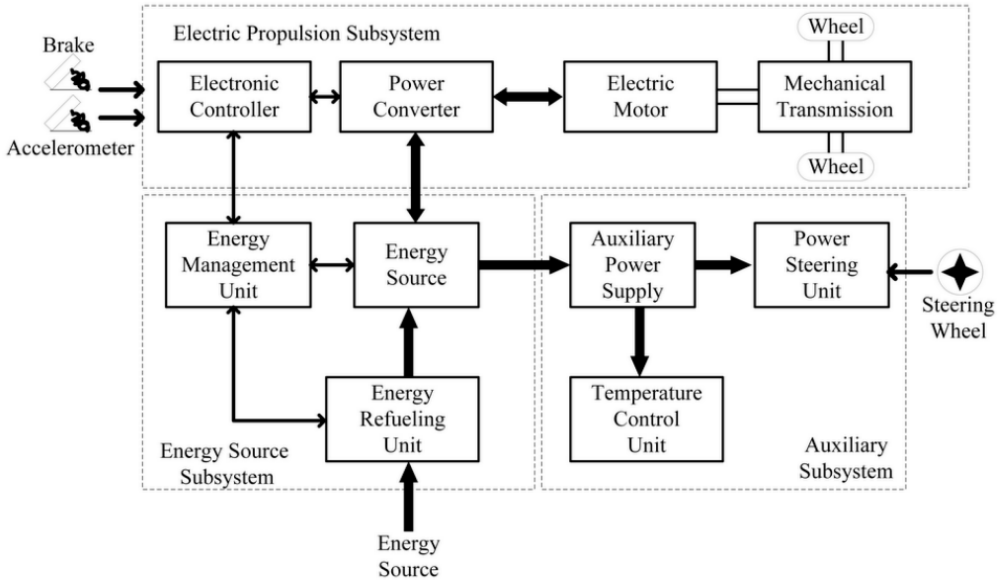

Figure 1.2: Architecture of an electric vehicle.

On the basis of this architecture, some considerations about the design flexibility of EVs can be discussed. First, the energy flow is mainly electrical and implemented via flexible electrical wires rather than bolted flanges or rigid shafts as in internal combustion engine vehicles (ICEV). Then, the minor encumbrance of electric motors allows different positioning inside the vehicle chassis. Hence, distributed subsystems in the EV are really achievable allowing different propulsion arrangements such as independent four wheels and in wheel drives [1]. Research communities and automotive companies are exploiting this flexibility to implement different powertrain configurations in an EV to improve as much as possible efficiency and performances reducing costs. Succeed in developing the right configuration for its own production can lead to significant advantages in relation to competitors. For these reasons, this field is becoming even more populated of studies, researches and investments that propose innovative solutions suitable for the different kinds of applications. Nowadays, there are two approaches through which it is possible to improve the performances and the efficiency of an EV. The first one is to make the most of the already available hardware developing different control and system management strategies; the second one is related to technology research as it is often done in the field of battery. In general, the latter approach can be more expensive and time consuming with respect the first one and it is the reason why searching for new configurations, algorithms and control strategies is often done to maximize performances and efficiency of EVs. Efficiency and performance of an

EV are mainly related to its powertrain, and for this reason, the structure of an EV powertrain and functional characteristics of its constitutive components have been analyzed in detail in the following section.

1.1.1 Electric Powertrain

In an automotive vehicle, the powertrain comprises the main components that generate power and deliver that power to the road surface. This includes the motor/engine, transmission, drive shafts, differentials, and the drive wheels. All EVs eliminate the engine altogether, relying solely on electric motors for propulsion [2]. The main components of an electric powertrain can be identified from the Fig. 1.2. More precisely, the components included in the subsystem are an electronic controller, a power converter, one or multiple electric motors, and sometimes a mechanical transmission.

Electric Motor

An electric motor is an electric machine that converts electric energy into mechanical energy. Most electric motors operate through the interaction between the the magnetic field of the motor and electric current in a wire winding to generate force in the form of torque applied on the motor shaft. Electric motors can be powered by direct current (DC) sources, such as from batteries, motor vehicles or rectifiers, or by alternating current (AC) sources, such as a power grid, inverters or electrical generators.

Electronic Controller

An Electronic Controller or Electronic Control Unit (ECU) is an embedded system that controls electric subsystems in a transport vehicle. In other words, the ECU is the brain of the system. In general, it receives measurements of significant quantities of the vehicle by means of sensors, compares these measurements with the desired behaviour and determines the appropriate control actions on the base of the control algorithm acting on the available actuators. In the context of an Electric Powertrain, the Electronic Controller is responsible to drive the Power Converter to manage the electric power flow from the energy source to the electric motor and vice versa in case of regenerative breaking.

Power Converter

A Power Converter is an electronic device that controls the flow of electric energy by supplying voltages or current in a form that is optimal suited for the load. A Power Converter is necessary in a electric vehicle to transform the DC energy

of batteries into current waveforms suitable to control the motor. The most used Power Converter in Electric Powertrain is the inverter. Inverters are power electronic devices that transform direct current into alternating current with specific amplitude and frequency. More about inverters is discussed in section 2.5.1.

Mechanical Transmission

A Mechanical Transmission is a machine in a power transmission system, which provides controlled application of the power. Mechanical Transmissions are often called gearboxes in automotive applications. In general, gearboxes have a fixed ratio in Electric Powertrains allowing much more simpler transmission systems with respect to ICEVs.

1.2 Embedded systems role

The design of electric vehicles requires a complete paradigm shift in terms of embedded systems architectures and software design techniques that are followed within the conventional automotive systems domain. It is increasingly being realized that the evolutionary approach of replacing the engine of a car by an electric motor will not be able to address issues like acceptable vehicle range, battery lifetime performance, battery management techniques, costs and weight, which are the core issues for the success of electric vehicles. While battery technology has crucial importance in the domain of electric vehicles, how these batteries are used and managed pose new problems in the area of embedded systems architecture and software for electric vehicles. At the same time, the communication and computation design challenges in electric vehicles also have to be addressed appropriately.

Nowadays most innovations within the automotive domain are driven by embedded systems and software solutions. Many of these innovations like anti-lock braking systems, electronic stability control, or emergency brake assistants significantly reduce vehicle accidents and increase safety. On the other hand, embedded systems increase the driving comfort with driver assistance functions like adaptive cruise control. Furthermore, infotainment systems and smart devices increase the user acceptance and contribute to the value of modern cars. It can be observed that the costs for embedded solutions in vehicles are growing rapidly while mechanical engineering based solutions are stagnating in importance. It is projected that the importance and costs of embedded systems and software in electric vehicles will grow much further.

The introduction of electric vehicle is speed up this process. It is widely understood that the approach of replacing the engine of a conventional car by an electric motor is only an intermediate solution on the way to a fully customized electric vehicle. Therefore, the embedded systems and software challenges in electric

vehicles go beyond the engine and energy control. A redesign of the communication and computation architecture in electric vehicles entails several opportunities, but also bears many challenges.

For computation support, novel hardware and programming paradigms have to be considered. While the automotive industry already started considering multi-core systems, alternative solutions might be graphic processors or reconfigurable hardware to cope with the computational demands of active safety functions. The electric powertrain also has to be designed and controlled to achieve the highest possible efficiency. To achieve significant energy savings, a distributed embedded control approach becomes necessary to control the power management of the entire vehicle. This is a challenging task because multiple energy source like the batteries, solar panels, or regenerative braking have to be taken into account. For this purpose, control strategies for these batteries will become necessary that take into account different driving patterns.

1.3 Firmware development

In computing, firmware is a specific class of computer software that provides the low-level control for a device specific hardware. Firmware can either provide a standardized operating environment for more complex device software (allowing more hardware-independence), or, for less complex devices, act as the complete operating system of the device, performing all controls, monitoring and data manipulation functions. Typical examples of devices containing firmware are embedded systems, consumer appliances, computers, computer peripherals, and others. Almost all electronic devices beyond the simplest contain some firmware.

Firmware is held in non-volatile memory devices such as ROM, EPROM, or flash memory. Changing the firmware of a device was rarely or never done during its lifetime in the past, but is nowadays a common procedure; some firmware memory devices are permanently installed and cannot be changed after manufacture. Common reasons for updating firmware include fixing bugs or adding features to the device. This requires ROM integrated circuits to be physically replaced, or EPROM or flash memory to be reprogrammed through a special procedure. Firmware such as the ROM BIOS of a personal computer may contain only elementary basic functions of a device and may only provide services to higher-level software. Firmware such as the program of an embedded system may be the only program that will run on the system and provide all of its functions.

In the proof of concept object of this thesis, the control software of the electric powertrain was developed as an embedded firmware with the absence of an operating systems. Therefore, there wasn't strict distinction between the specific device drivers and the application software represented by the control algorithms. However, the

specific device drivers and the application software were developed and integrated together manually after being developed and tested separately - the devices drivers were developed with handwritten software development processes, instead the code of application software was automatic generated by means of specific toolboxes in Matlab environment.

1.4 Thesis outline

The main objective of this master thesis is to develop the firmware for an electronic control board responsible to implement the control algorithms for an electric powertrain proof of concept. However, the multidisciplinary topic of electric powertrains led to consider aspects which aren't strictly related to the firmware development, and to interact with different engineering domains in the field of electric motor, power converters, digital electronics, and embedded real-time systems. The work developed for this thesis will be discussed according to the following structure:

- **Theoretical Background**
  In chapter 2, a brief overview of the theoretical aspects of the topics discussed in this thesis is provided to the reader to better understand the design choices taken during the firmware development process.

- **Firmware Development**
  Chapter 3 is the central work of this thesis and aims to describe the firmware development process. Features, implementation mechanisms, and application issues of each software components are discussed.

- **Firmware Validation and Testing**
  The testing procedures used to validate and verify the firmware are provided in chapter 4 with the corresponding test results and changes made to fulfill the requirements.

- **Application Integration**
  After being developed and tested in Matlab environment, the application software was generated by mean of code generation using specific toolboxes. The application integration process and the used scheduling approach is discussed in chapter 5.

- **Conclusions**
  The conclusive considerations and possible future improvements are discussed in chapter 6.

Chapter 2

Theoretical Background

In this chapter, some theoretical aspects about electric motors, power electronics, microcontrollers, analog-to-digital conversion, and digital communication protocols will be discussed. The theoretical description about these topics will be restricted to some significant aspects useful for the purposes of this thesis. For this reason, this chapter has to be seen as a miscellaneous of contents that provides a theoretical background aiming to understand the description of the system structure and its functional requirements, and their corresponding implementation in the firmware development process. All the development aspects will be analyzed in next chapters and, at that time, it will be clear why some topics were discussed respect to others.

2.1 ARM Architecture

The ARM architecture is based on 32-bit reduced instruction set computer (RISC) processor cores developed by ARM Holdings. RISC processors are characterized by the use of small, highly-optimized instructions with explicit memory access (i.e. load/store instructions) which is always aligned. Moreover, the instruction set is orthogonal with fixed length and single cycle execution time for all instructions. These features are finalized to easy the decode process, allow the pipeline technique, and simplify the hardware implementation and testing. ARM processors can be classified in classic, embedded and application on the base of their purposes. Embedded and application processors are constituted by the ARM Cortex family which is subdivided as following:

- **Cortex-A**
  The Cortex-A category of processors is dedicated to Linux and Android devices. Any devices – starting from smartwatches and tablets and continuing with networking equipment – can be supported by Cortex-A processors.

- **Cortex-R**
  Cortex-R processors primarily target real-time solutions. They find application in controllers, networking equipment, media players, and other similar devices. Furthermore, this type of ARM processor provides great support for the automotive industry. Cortex-R processors have a lot in common with high-end microcontrollers, but at the same time have the ability to fulfill more scalable tasks.

- **Cortex-M**
  The point of interest of Cortex-M processors is the MCU market. Cortex-M is known as an industry standard and find their implementation in FPGA, integrated memories, clocks, etc. Different members of the set have different improved features: some of them demonstrate higher performance, others are more energy efficient. Of course, each of the designed controllers is tailored to a particular segment of the market.

In this thesis, a microcontroller containing a Cortex-M7 processor core was utilized for the firmware development and the controller implementation of the proof of concept. For this reason, the Cortex-M7 processor core will be discussed in detail in the following section.

2.1.1 ARM Cortex-M7

The Cortex-M7 processor is built on a high-performance processor core, with a 6-stage pipeline Harvard architecture. The Cortex-M7 processor implements a version of the Thumb instruction set based on Thumb-2 technology, ensuring high code density and reduced program memory requirements. The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb-2 technology is a major enhancement to the Thumb instruction set. It adds 32-bit instructions that can be freely intermixed with 16-bit instructions in a program. As can be seen in Fig. 2.1, the Cortex-M7 processor core provides some in-core peripherals that are used for real-time application, interrupt handling and fast memory access management:

- **Nested Vector Interrupt Controller**
  The NVIC is an embedded interrupt controller that supports low latency interrupt processing.

- **System Control Block**
  The System Control Block (SCB) is the programmers model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.

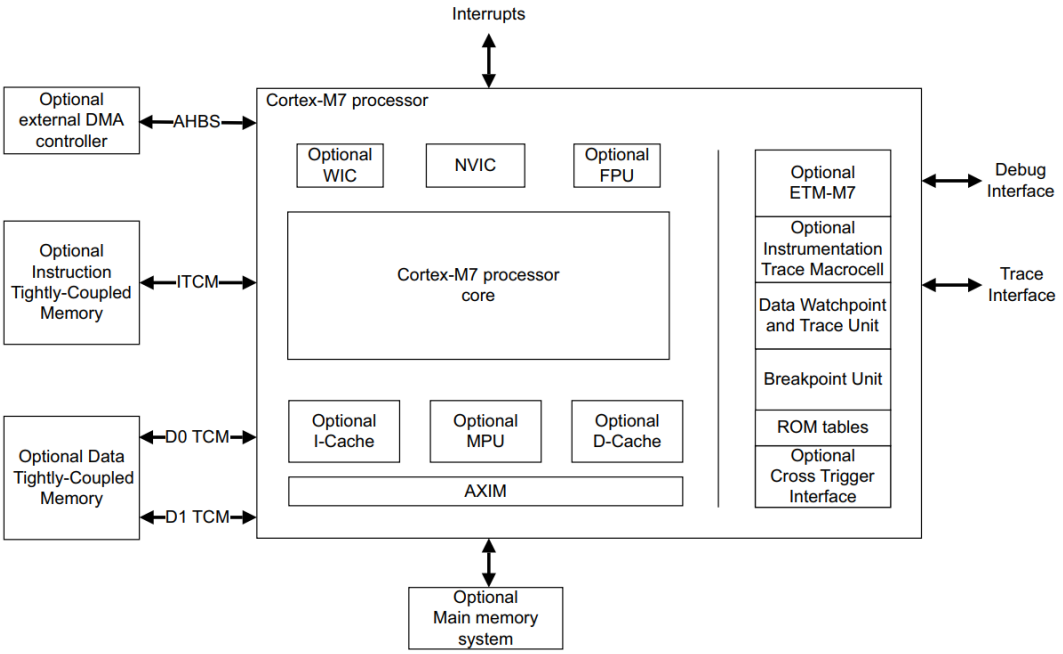

Figure 2.1: ARM Cortex-M7 processor core architecture.

- **System Timer**
  The system timer, often called SysTick, is a 24-bit count-down timer. It is used for Real Time Operating System (RTOS) tick timer or as a simple counter.

- **Integrated Instruction and Data Caches (optional)**
  The instruction and data caches provide fast access to frequently accessed data and instructions, and can increase average performance when system based memory is used.

- **Memory Protection Unit (optional)**
  The Memory Protection Unit (MPU) improves system reliability by defining the memory attributes for different memory regions. Depending on vendor-specific implementation, it can provide up to 8 or 16 different regions, and an optional predefined background region.

- **Floating-point unit (optional)**
  The FPU provides IEEE754-compliant operations on 32-bit single-precision and 64-bit double-precision floating-point values.

Programmer's Model

The programmer's model of a processor contains a concise description of its internal registers and their functions. Only the program visible registers (i.e. directly accessible by applications) are included in the programmer's model.

From a functional point of view, the Cortex-M7 can operates in two different modes called Thread mode and Handler mode, which are respectively used to execute application software or handles exceptions. Also two privileged levels are available to have an unlimited or limited access to all the CPU resources during software execution. The structure of the processor core registers are reported in

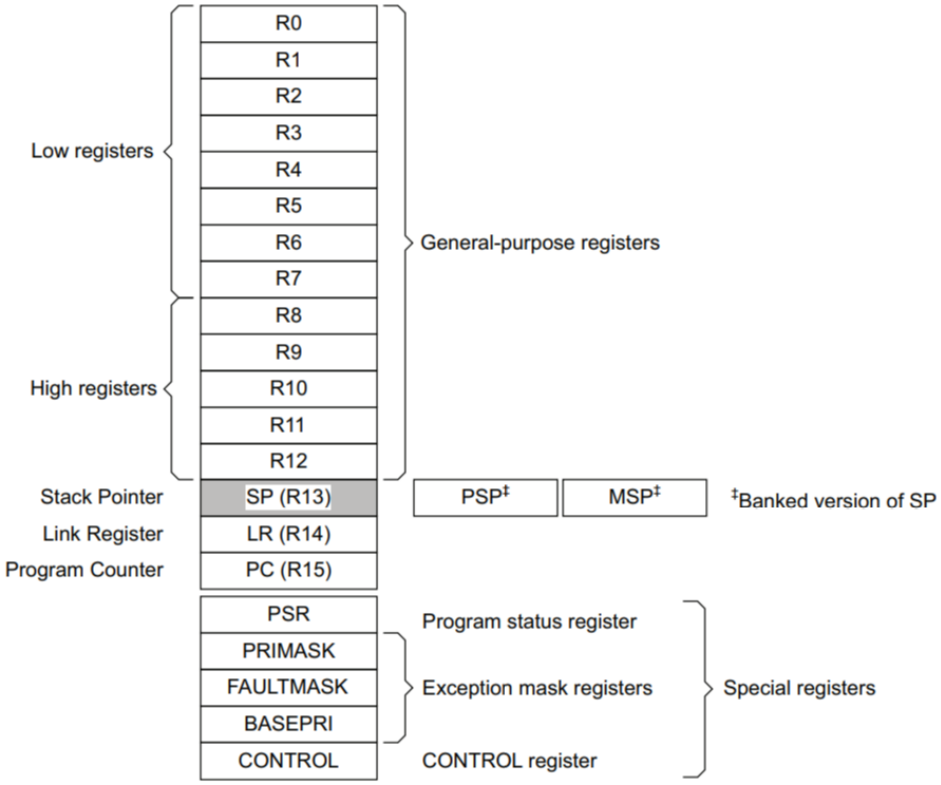

Figure 2.2: Structure of the core register set of ARM Cortex-M7 processor.

Fig. 2.2. There are available thirteen general purpose registers (R0-R12) for data operations, five special registers to manage the processor status and configuration, and four other registers for specific purposes. Two registers, called Main Stack Pointer and Process Stack Pointer (MSP and PSP), are mutually used to store the stack pointer according to the bit[1] of Control register. The Link Register (LR) and the Program Counter (PC) store respectively the return information for subroutines and the next instruction to be fetched. The five special registers are Processor Status Register (PSR), Priority Mask Register (PRIMASK), Fault Mask

Register (FAULTMASK), Base Priority Mask Register (BASEPRI), and Control Register (CONTROL). PSR combines the Application PSR (APSR), Interrupt PSR (IPSR), and Execution PSR (EPSR) as three mutually exclusive bit fields in its 32 bits. These three registers contains respectively the current status of condition flags, exception type number of the current Interrupt Service Routine, and execution state bits. When they are set, the PRIMASK and FAULTMASK are used to prevent respectively the activation of all exceptions with configurable priority and the activation all exceptions except for Non-Maskable Interrupt (NMI). The BASEPRI is used to define the minimum priority for exception processing. The CONTROL defines which register is selected for the stack pointer, the privilege level for software execution when the processor is in Thread mode, and if implemented, indicates whether the FPU core peripheral is active.

The access to processor core registers and core peripherals can be done using the Cortex Microcontroller Software Interface Standard (CMSIS) provided by ARM. CMSIS allows a standardized and device-independent way to interact with the available resources provided by the processor core.

Memory Model

The memory model describes processor memory map and the behavior of memory accesses. ARM Cortex-M7 processor divides its 4GB of addressable space in various memory regions with different purposes. The memory map of the processor is reported in Fig. 2.3. The low memory addresses are used for code storing, SRAM and peripheral mapping. The code region is often implemented with NOR-Flash memories which are able to retain the non-volatile data such as software to be executed at startup. The peripheral memory region is used to map in-chip peripherals in MCU architectures. External memories or devices have their own memory regions which are mapped respectively in External RAM and External Devices regions. Instead, the private peripheral bus is used to map some processor core peripherals - NVIC, System Timer, and System Control Block. Also a vendor-specific memory region is available and mapped in higher memory addresses.

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word. In byte-invariant big-endian format, the processor stores the most significant byte of a word at the lowest-numbered byte, and the least significant byte at the highest-numbered byte. In little-endian format, the processor stores the least significant byte of a word at the lowest-numbered byte, and the most significant byte at the highest-numbered byte. The Cortex-M7 architecture supports both memory endianness, but in general, only one of them is supported in specific-vendor chip implementations.

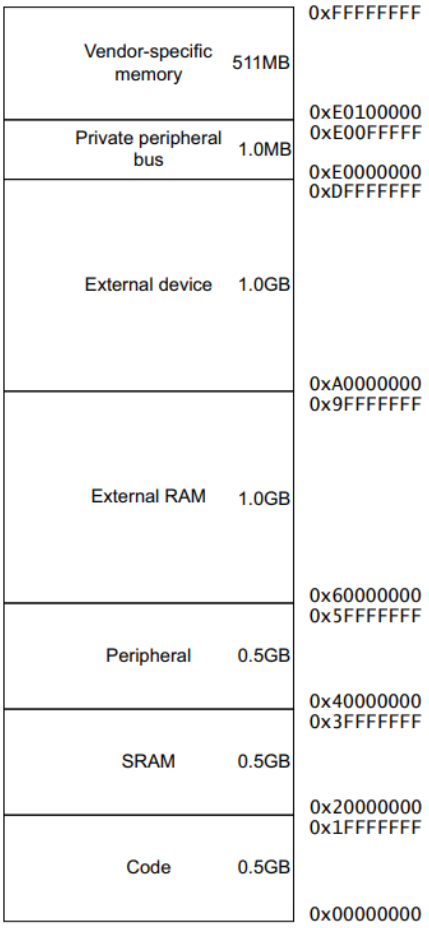

Figure 2.3: Addressable memory space of ARM Cortex-M7 processor.

Exception Model

The Exception Model describes how the processor handles and manages exceptions and interrupts. In Cortex-M7 processor, four different states are provided to manage an exception - inactive when it is not active and not pending, pending when it is waiting to be serviced by the processor, active when it is serviced by the processor and it is not yet completed, and active and pending when it is active and another exception of the same source is pending. In Cortex-M7 processor, exceptions are classified in different types as follow:

- **Reset**
  Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. It is permanently enabled and has a fixed priority of -3.

- **NMI**
  A Non-Maskable Interrupt (NMI) can be signaled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2.
- **HardFault**
  A HardFault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. HardFaults have a fixed priority of -1, meaning they have higher priority than any exception that has a configurable priority.
- **MemManage**
  A MemManage fault is an exception that occurs because of a memory protection related fault. The fixed memory protection constraints, or the MPU if implemented, determines this fault, for both instruction and data memory transactions.
- **BusFault**
  A BusFault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.
- **UsageFault**
  A UsageFault is an exception that occurs because of a fault related to instruction execution.
- **SVCall**
  A Supervisor Call (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.
- **PendSV**
  PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.
- **SysTick**
  A SysTick exception is an exception generated when the SysTick reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.
- **Interrupt (IRQ)**
  A interrupt, or IRQ, is an exception signaled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

The Fig. 2.4 represents the so called vector table which contains the memory locations where the first instruction of the different exception and fault handlers, and interrupt service routines are placed. More precisely, the first word of the vector table is the default stack pointer which is loaded by the processor as first operation. Then, the address to the first instruction of the Reset Handler is placed. The table continues with the same logic with all exceptions/interrupts first instruction addresses in increasing priority order. In this way, every time an exception is serviced, the program counter is loaded with the corresponding address present in the vector table. All exceptions have an associated priority. Lower priority value

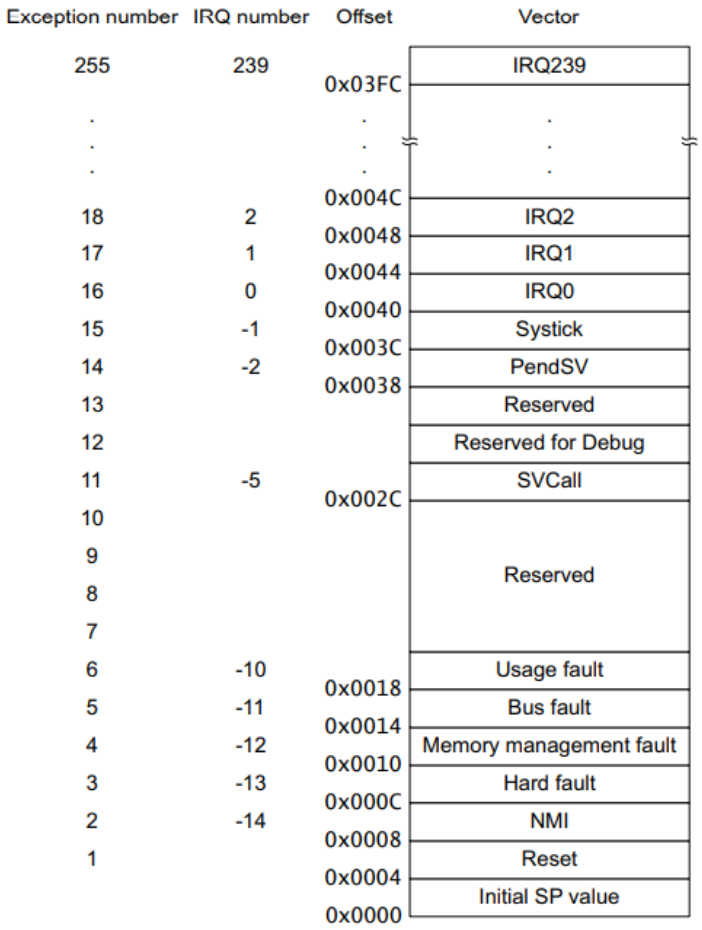

Figure 2.4: Vector table of ARM Cortex-M7 processor.

indicates higher priority. Priorities are configurable except for Reset, NMI, and HardFault. Preemption between different exception handlers occurs only when a higher priority exception occurs. If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the

exception number. However, the status of the new interrupt changes to pending. To increase priority control in systems with interrupts, the NVIC supports priority grouping. This divides each interrupt priority register in an upper field that defines the group priority and a lower field that defines a subpriority within the group. Only the group priority determines preemption of interrupt exceptions. When the processor is executing an interrupt exception handler, another interrupt with the same group priority as the interrupt being handled does not preempt the handler. If multiple pending interrupts have the same group priority, the subpriority field determines the order in which they are processed. If multiple pending interrupts have the same group priority and subpriority, the interrupt with the lowest IRQ number is processed first.

2.2 Communication Protocols

In this section, the main features of the communication protocols used in the project will be described. The communication will be implemented in the MCU using dedicated in-chip peripherals. In the proof of concept, two protocols were mainly used: the SPI (Serial Peripheral Interface) for in-board communication and the CAN-bus (Controller Area Network) for the communication between the system and an external workbench.

2.2.1 Serial Peripheral Interface

The Serial Peripheral Interface is a synchronous serial communication interface specification used for short-distance communications. The two main features of SPI are the full-duplex mode and master-slave architecture. Full-duplex mode means that the devices interested in the transmission receive and send data contemporary. Master-slave architecture is an architecture where there are a master, which controls the data transmission, and one or more slaves that answer to the master. SPI supports up to four slaves. An example of SPI connection between a master and a slave is represented in Fig. 2.5. This type of connection is often called four-wire serial bus due to the necessity of four signals to implement the interface:

- **Chip Select (CS)**
  The CS is an enable signal and it is activated by the master for all the duration of the transmission to select the slave to communicate with.

- **Serial Clock (SCLK)**
  The SCLK signal is the clock signal provided by the master to synchronize the transmission.

- **Master Output/Slave Input (MOSI)**
  The MOSI signal is the serial data transmitted from the master to the slave.

- **Master Input/Slave Output (MISO)**
  The MISO signal is the serial data transmitted from the slave to the master.

In SPI communication, the master selects the slave activating the corresponding CS when a transmission is required. SCLK is driven with a frequency supported by both devices to synchronize the transmission. Sometimes, a waiting period is inserted between the activation of the CS and the SCLK driving. A bit is transmitted for each SCLK pulse in a full-duplex mode through MOSI and MISO lines. When all bits have been transmitted, the SCLK is stopped and the CS deactivated.

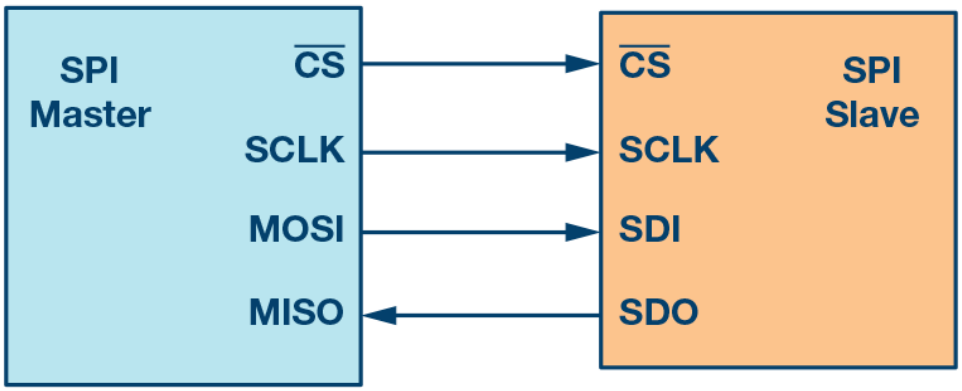

Figure 2.5: Block diagram of master (on the left) and slave (on the right) connection in SPI communication.

From the hardware point of view, SPI peripherals are often provided with a shift register of a given word-size as represented in Fig. 2.6. This type of structure is usually referred to the ring topology ones. The shift registers of the master and slave are loaded before to start the transmission. At each clock pulse, the two devices shift out the bit to send at a specific clock edge and sample the received bit at the other edge. This procedure continues until all the bits are sent or if one of the two shift register is full or empty. The event to full or empty the shift register of a SPI peripheral can be used to set flags associated to an interrupt request which can be used to react quickly to the event. Another relevant setting of the communication is the clock polarity and phase with respect to the data. The clock polarity indicates if the leading and the trailing edges are respectively rising and falling edges of the SCLK or vice versa. The clock phase determines when bits are changed and captured in MOSI and MISO lines with respect to the clock pulses.

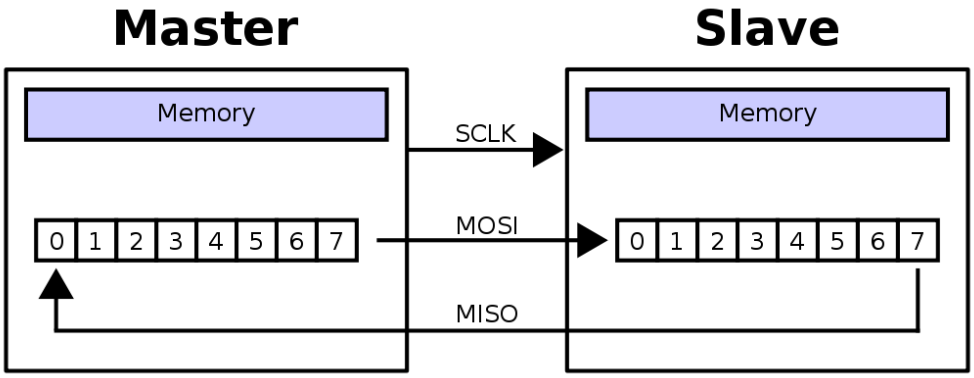

Figure 2.6: Example of SPI peripheral implementation using circular buffers.

2.2.2 Controller Area Network

The Controller Area Network is a bus communication protocol designed for automotive applications. CAN network has become a standard in-vehicle communication thanks to its high integrity, low implementation costs and light weight wiring. In the proof of concept, the CAN network was strongly suggested by the application itself, since powertrains are subsystems of vehicles where this kind of communication is available. Trying to implement a CAN communication between the proof of concept and an external workbench was a way to test a feature which will be almost surely needed in the final product.

CAN is an asynchronous communication protocol with a two-wire bus. Each device connected to the bus is called node. The bus structure and node components are represented in Fig. 2.7. More precisely, a node is made of a microcontroller, a

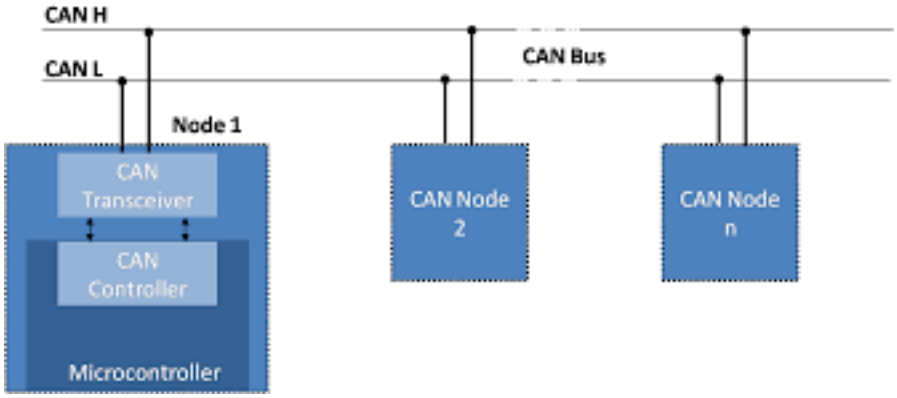

Figure 2.7: Representation of CAN-bus network with nodes. The components of a node can be seen in Node 1.

CAN controller and a CAN transceiver. The CAN controller manages the message reception and sending over the network. The CAN transceiver is used to adapt signals provided by the CAN controller to compliant signals with respect to bus electric characteristics and vice versa. Indeed, the transceiver can be seen as an electric interface between the controller and the bus. An example of the same signal from the bus and controller perspectives is represented in Fig. 2.8. As can be seen in the figure, the driver logic of the controller is single-ended, instead the signals are transmitted over the bus network as differential ones. Two signals, CAN

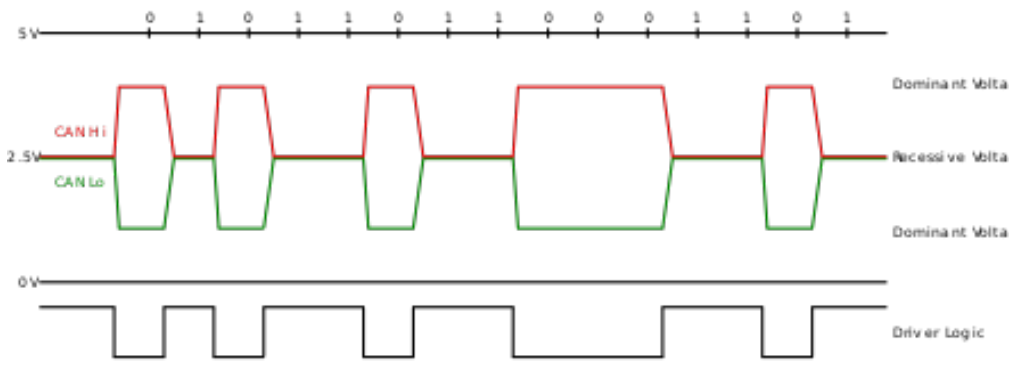

Figure 2.8: Electric signals in the CAN-bus are represented in red and green with their corresponding logical and single-ended signals.

high (CANH) and CAN low (CANL) are either driven to a *dominant* state with CANH > CANL, or not driven and pulled by passive resistors to a *recessive* state with CANH ≤ CANL. A 0 data bit encodes a dominant state, while a 1 data bit encodes a recessive state. The CAN network is wired-AND which means that, in case of multiple contemporary sending over the network, the resulting bus state is the AND of the transmitted signals. This is why the 0 data bit encodes the so called dominant state.

CAN-bus is a message-based protocol where frames are received by all devices, including the transmitting one. The protocol provides four types of frame:

- **Data Frame**
  Data Frames are frames containing a data payload to be transmitted.
- **Remote Frame**
  Remote Frames are frames requesting the transmission of a specific message.
- **Error Frame**
  Error Frames are frames transmitted when a transmission error is detected.
- **Overload Frame** Overload Frame are frames used to inject a delay between data or remote frames.

Each frame can be structure in base (standard) or extended format. The difference between them is the length of the identifier field - the identifier of the base frame format is composed by 11 bits, the extend frame format one by 29 bits. In this thesis, only standard data frames will be analyzed (reported in Fig. 2.9). A data frame is constituted by multiples fields, each of them encodes a specific information. Since the CAN-bus is an asynchronous communication protocol, the first bit of the frame is the Start of Frame. Then, the Arbitration Field is transmitted. It is composed by a unique identifier of the message which represents also its priority and by the Remote Transmission Request (RTR) bit. This bit says if the frame is a remote one or not, and it must be dominant (0) in case of data frame. The

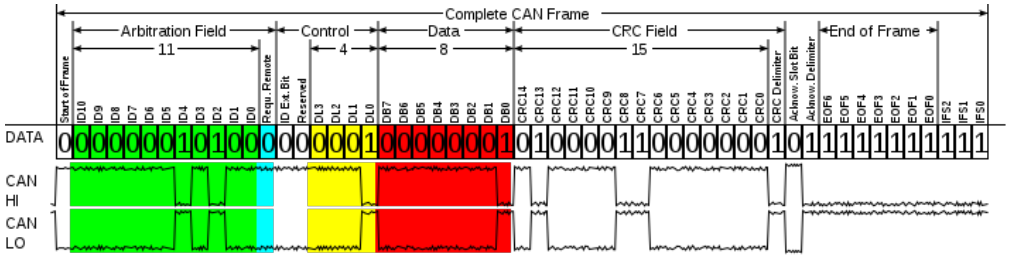

Figure 2.9: Example of a complete CAN standard data frame.

Control Field is composed by the Identifier Extension Bit (IDE), a Reserved bit, and 4 bits for the Data Length Code (DLC). The IDE bit must be dominant (0) for a standard frame as well as the Reserved bit. The DLC represents the data payload length in number of byte and must be between 0 and 8. The Data Field can range between 0 and 64 bits in length depending on the declared DLC and it is the data chunk to be sent. The CRC Field is composed by 15 bits of Cyclic Redundancy Check followed by a CRC delimiter that must be dominant. Then, an Acknowledge (ACK) slot is inserted and sent by the receivers to signal with a recessive (1) bit if the frame was correctly received. The ACK slot is followed by a recessive ACK delimiter. At the end of the message, there are 7 recessive (1) bits to signal the End of Frame.

A predefined nominal bit rate and a mechanism for the transmission synchronization in CAN-bus are required due to the asynchronicity of the communication. First, a hard synchronization occurs on the first recessive to dominant transition after a period of bus idle (Start of Frame bit). Then, resynchronization occurs on every recessive to dominant transition during the frame transmission. Even if all the nodes implement the nominal bit rate, the presence of noises, phase shifts, oscillator tolerances and oscillator drifts can lead to different node bit rates with respect the nominal one. For this reason, a mechanism of bit rate adjustment must be taken into account. Each time a bit is transmitted, the CAN controller expects the transition to occur at a multiple of the nominal bit time. If the transition

does not occur at the exact time the controller expects it, the controller adjusts the nominal bit time accordingly. The bit rate adjustment is done dividing each bit as it is represented in Fig. 2.10. The bit is divided into four segments: syn-

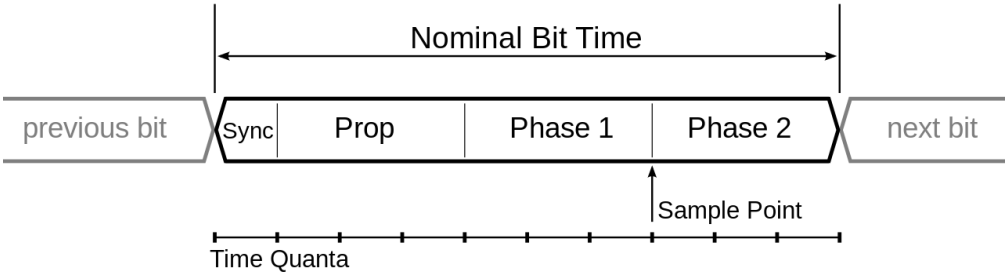

Figure 2.10: Structure of a bit in CAN-bus.

chronization segment, propagation segment, phase segment 1 and phase segment 2. Each segment is made by a well defined number of quanta which is proper of each CAN controller. When a transition occurs before or after it is expected, the CAN controllers which are receiving the message adjust their timings according to the transmitting one. In this way, possibles variation in bit rates of different nodes are compensated.

2.3 Analog-to-Digital Conversion

Analog-to-Digital conversion is the process to converter analog signals into digital ones. As is known, digital electronics such as MCUs works with binary logic, instead physical systems are characterized by an analog behaviour with continuous-time and continuous-amplitude signals. So, Analog-to-Digital converters (and also Digital-to-Analog converters in the opposite way) are devices that allow these two domains to interact. In the proof of concept, ADCs were used to acquire system measurements such as motor phase currents, temperatures, and voltages.

In general, amplitude and time discretization, which are called respectively sampling and quantization, are performed to convert an analog signal into a digital one (represented in Fig. 2.11). The conversion can occur at a fixed rate or sporadically in specific time instances depending on the application. In both cases, each time a conversion starts, a sampling phase is followed by a quantization one. First, a sample of the input signal is acquired using dedicated circuitry. The most used one is the sample-and-hold circuit which block diagram is represented in Fig. 2.12. From a functional point of view, a control signal opens the switch $S1$ for a time interval sufficient to charge the capacitor C_H to a voltage level equivalent to V_{IN}. Then, the switch is closed and the charge to the capacitor maintained

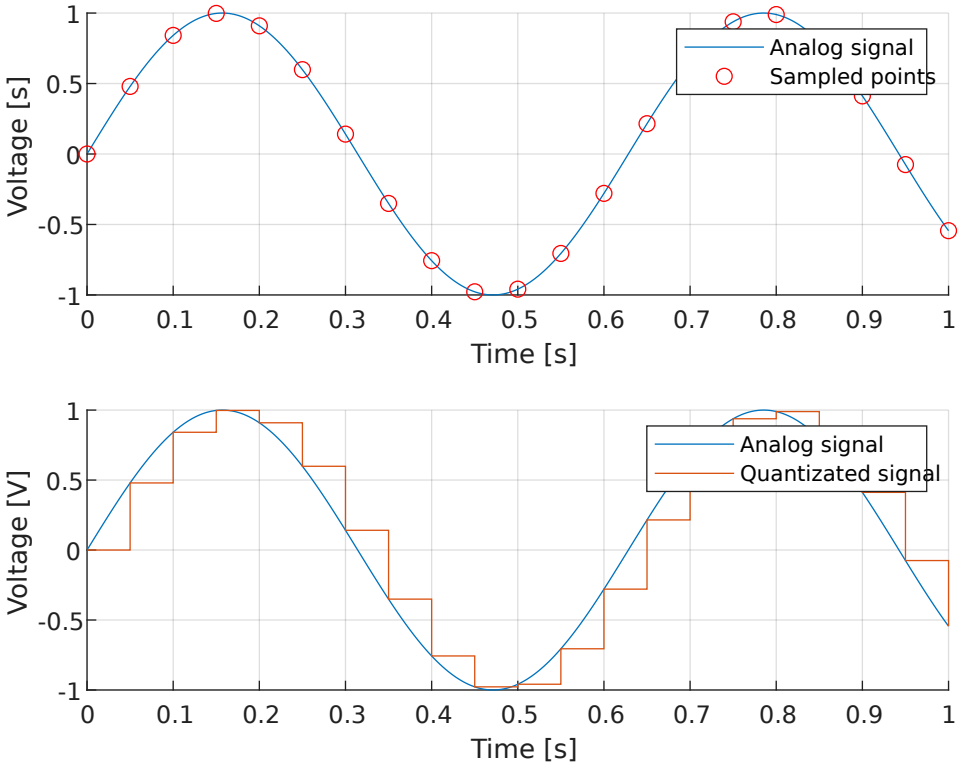

Figure 2.11: Sampling (on top) and quantization (on bottom) of an analog signal.

thanks to the high impedance of the buffer. The buffer output V_{out} represents the sampled signal and it is used for the quantization phase. The sampling rate, and its corresponding sampling frequency, is one of the most relevant characteristic of an ADC. The maximum sampling frequency constitutes the ADC bandwidth. In order to correctly acquire an analog signal, its spectrum and the ADC bandwidth can't be neglected. Indeed, according to Nyquist-Shannon sampling theorem, the sampling frequency of an analog signal must be at least double of its higher frequency in its spectrum. This is why, ADC bandwidth has to be taken into account in the choice of the most suitable converter.

Another relevant characteristic of an ADC is the number of bits used to express converted results which constitutes its resolution. Higher resolutions means more precise results. Given the operational voltage range of operation of the ADC, the resolution establishes the width of each voltage step in the discretization. More precisely, the following relation is hold $V_Q = V_{ref}/2^M$, where M is the resolution of the ADC. It is evident that higher resolution means higher precision of the conversion due to smallest voltage steps in quantization process, but high precision

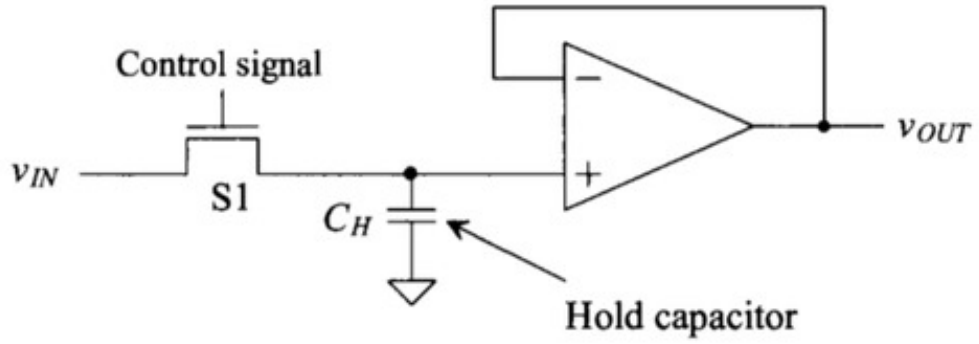

Figure 2.12: Scheme of a sampling and hold circuit.

doesn't mean high accuracy. Indeed, the real characteristic of an ADC (i.e. the curve describing the relation between its input and its output) can be affected by many causes such as the tolerance of the components used to implement the conversion circuitry, gain errors, offsets and non-linearities. For this reason, many ADCs provides dedicated self-calibration procedures to mitigate these non-idealities. An example of the difference between ideal and real characteristic of an ADC is reported in Fig. 2.13.

From amplitude point of view, the sampled signal can be quantizated using different strategies and circuitries. The mechanism utilized to perform the conversion defines the type of the ADC. The most common types are the following:

- **Flash Converter**
  Flash ADCs are the fastest ones. They make use of one comparator per voltage level and a string of resistors. The outputs of the comparators are connected to a logic network that determines the result of the conversion. This type of ADCs are low energy efficient and more expensive with respect to other ones. Moreover, the accuracy of the conversion is strictly related to the tolerance of the used string of resistors.

- **Sigma-Delta Converters**
  Sigma-Delta ADCs digitize an analog signal with a very low resolution (1 bit) ADC at very high sampling rate. By using oversampling techniques in conjunction with noise shaping and digital filtering, the effectiveness resolution increase. Decimation is then used to reduce the effective sampling rate at the ADC output. The high accuracy of Sigma-Delta ADCs is related to the lack of resistors which can have different value due to tolerance.

- **Successive Approximation (SAR) Converters**
  SAR ADCs makes use of a comparator, a DAC, and a logic network that

implement a binary search through all possible quantization levels to converge to the result of the conversion. They are less accurate, but faster than Sigma-Delta converters and at the same time slower, and more accurate than Flash converters.

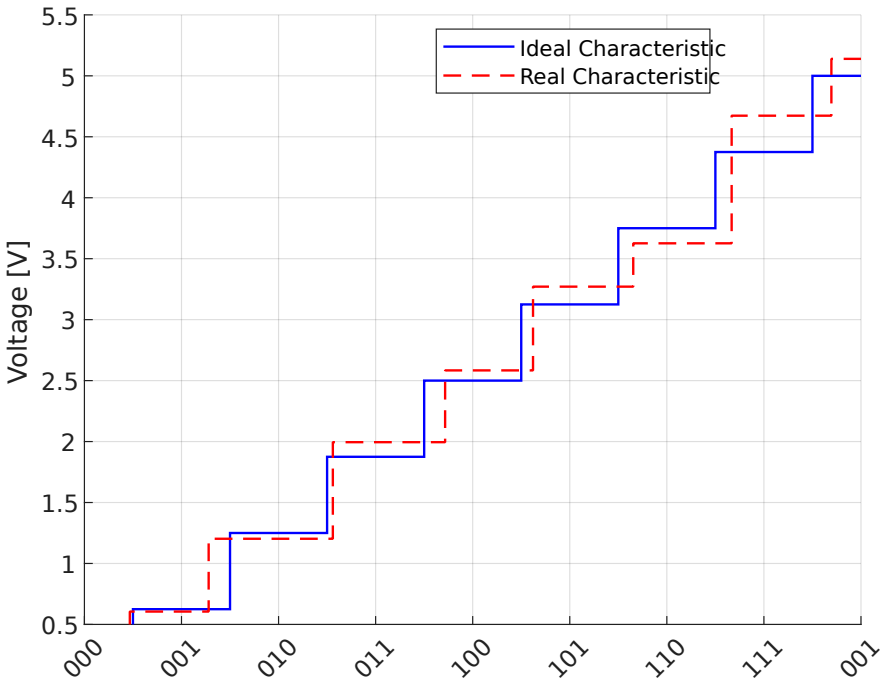

Figure 2.13: Example of real and ideal ADC characteristics.

In summary, Flash and Sigma-Delta converters are respectively the best solution in terms speed and accuracy, and SAR ADCs are the best trade-off between the two performances. In the project of this thesis, SAR ADCs converters were used because yet available as in-chip peripheral in the MCU. For this reason, they will be discussed more deeply in the following section.

2.3.1 Successive Approximation ADCs

In general, SAR ADCs are a type of analog-to-digital converter that converts a continuous analog waveform into a digital values via a binary search through all possible quantization levels before finally converging upon a digital output for each conversion. The structure of a SAR ADC can be seen in Fig. 2.14. The converter is made by a comparator, a sample-and-hold circuit for sampling the analog input signal, a digital to analog converter to convert the partial result in an

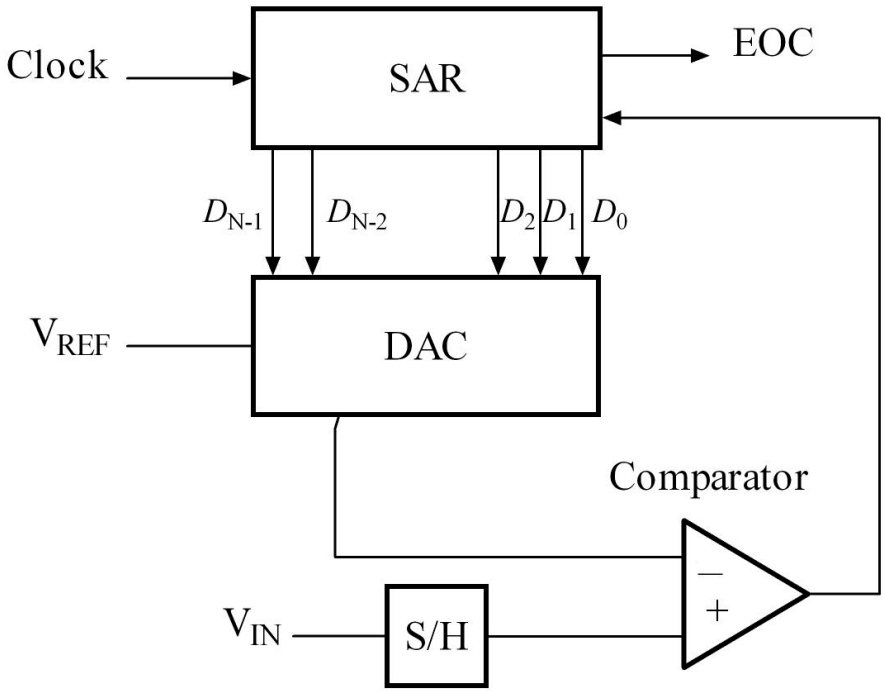

Figure 2.14: Block scheme of a successive approximation analog-to-digital converter.

analog value to be compared with V_{in}, and a circuitry that implements the binary search algorithm. A graphical representation of the algorithm used to perform the conversion is reported in Fig. 2.15 and described from a functional perspective by the following steps:

1. First, the successive approximation register (i.e. the register that stores the result of DAC conversion) is initialized so that only the most significant bit (MSB) is equal to a digital 1 and this code is fed into the DAC converter which provides in output $V_{ref}/2$;

2. At the beginning of the conversion, the input voltage is sampled by a sample-and-hold circuit;

3. The sampled input voltage and the voltage output of the DAC are compared by means of the comparator, and if the analog voltage of the partial result exceeds V_{in}, the comparator causes the reset of this bit, otherwise the bit is left 1;

4. The algorithm is repeated from point 3 for each bit until the LSB is determined.

5. The final converted value stored in the successive approximation register is connected to the output, meanwhile the End Of Conversion (EOC) signal is asserted to indicate that the conversion terminated.

By means of this algorithm, the sampled analog input is converted using M steps where M is the resolution of the converter. Other features about ADCs could be treated more in detail, but this brief overview is explaining of the key concepts necessary to understand the topics discussed in this thesis.

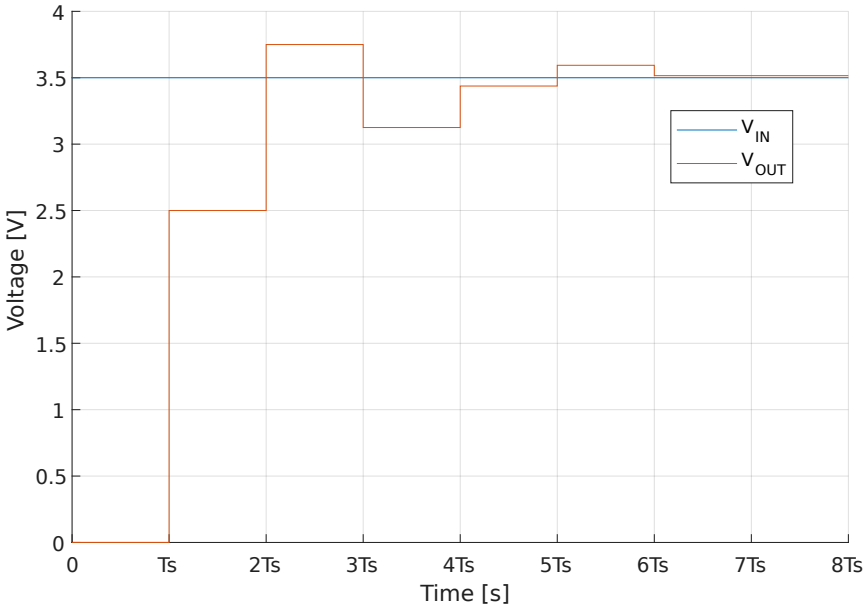

Figure 2.15: Timing diagram of partial results of a Successive Approximation ADC.

2.4 Pulse Width Modulation

Pulse width modulation (PWM) is a method of reducing the average delivered by a signal by effectively chopping it up into discrete parts. The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power supplied to the load. PWM is particularly suited for running inertial loads such as motors, which are not as easily affected by this discrete switching due to the slow inertial reaction. So, the PWM switching frequency has to be high enough not to affect the load, which means that the

resultant waveform perceived by the load must be as smooth as possible. In the application of this thesis, the PWM will be used to drive the inverter switches to control the motor supply.

The two characteristics that defines a PWM signal are the switching period and duty cycle. The switching period is the time that elapses between two positive edge of the signal, and in general, is maintained constant. The duty cycle is the percentage of time that the signal is on with respect to its period, and in general, is changed according to the control strategy. In Fig. 2.16 is represented a PWM signal showing a complete cycle with its period, and T_{ON} and T_{OFF} time intervals.

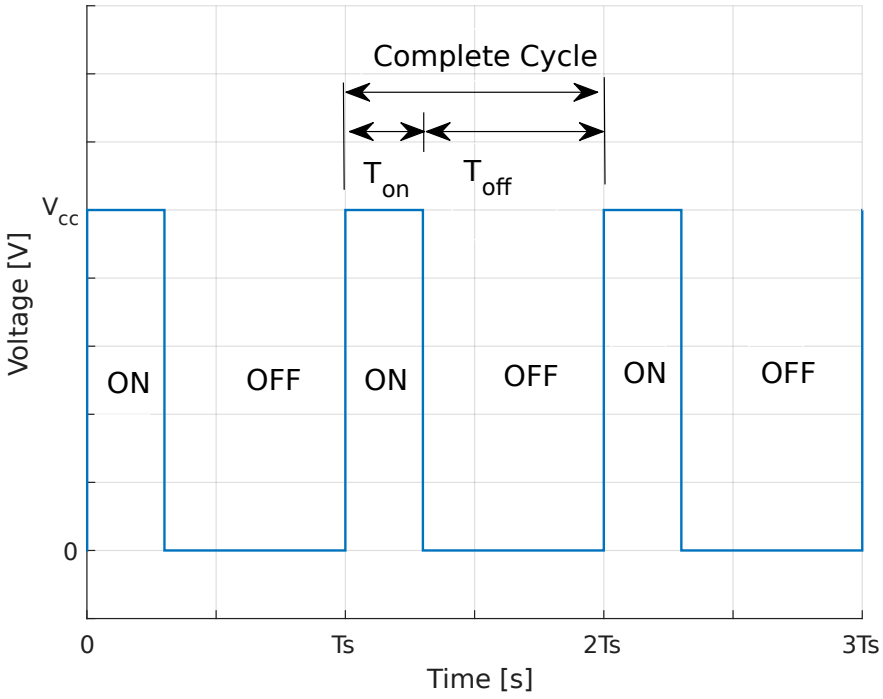

Figure 2.16: Example of PWM signal with 30% of duty cycle.

PWM can be classified in edge aligned and centered aligned pulse modulation. In edge aligned PWM, the T_{ON} period starts from the left or right edge of the switching period. In leading edge modulation, the T_{ON} period is placed at the beginning of the switching period and expands to the right with the increase of the duty cycle, instead trailing edge modulation implements the opposite behaviour. In centered modulation, the T_{ON} period is placed at the center of switching period and grows symmetrically to both edges when the duty cycle increases. This is why it is often called symmetrical PWM. An example of center aligned and edge aligned PWM is represented in Fig. 2.17.

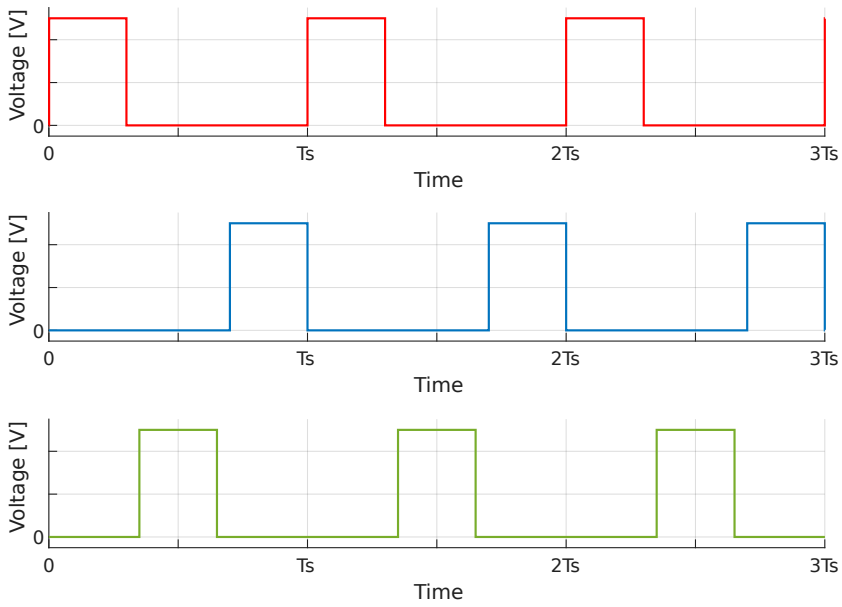

Figure 2.17: Three PWM signals with the same duty cycles and switching frequency - in blue leading edge aligned, in red trailing edge aligned, and in green centered aligned.

For the purpose of this thesis, it is useful to discuss how PWM signals are generated. The base concept of PWM generation is the comparison between the so called carrier wave and modulating wave. In general, the carrier signal is a sawtooth or triangular waveform with a frequency equal to the switching frequency. The modulating wave can be different depending on the application - sinusoidal waves or constant DC signals are often used. The constantly comparison between the carrier and modulating waveforms gives result to the so called modulated signal. The modulated wave has the characteristics seen in Fig. 2.16. Indeed, the comparison gives an output waveform which is high when the modulating wave is greater than the carrier one and vice versa. In such a way, the resulting duty cycle can be modified according to changes in the modulating wave. An example of this is reported in Fig. 2.18. Different carrier waveforms are used to generate different types of PWM signals - sawtooth carriers are used for edge aligned modulation, triangular ones for center aligned modulation. In leading edge modulation the carrier is a sawtooth leading edge with positive ramp followed by the step decay, instead in trailing edge modulation a sawtooth trailing edge with a vertical rise followed by a negative ramp is used. The triangular carrier can be also seen as a double edge sawtooth wave with a positive ramp followed by the negative one, and

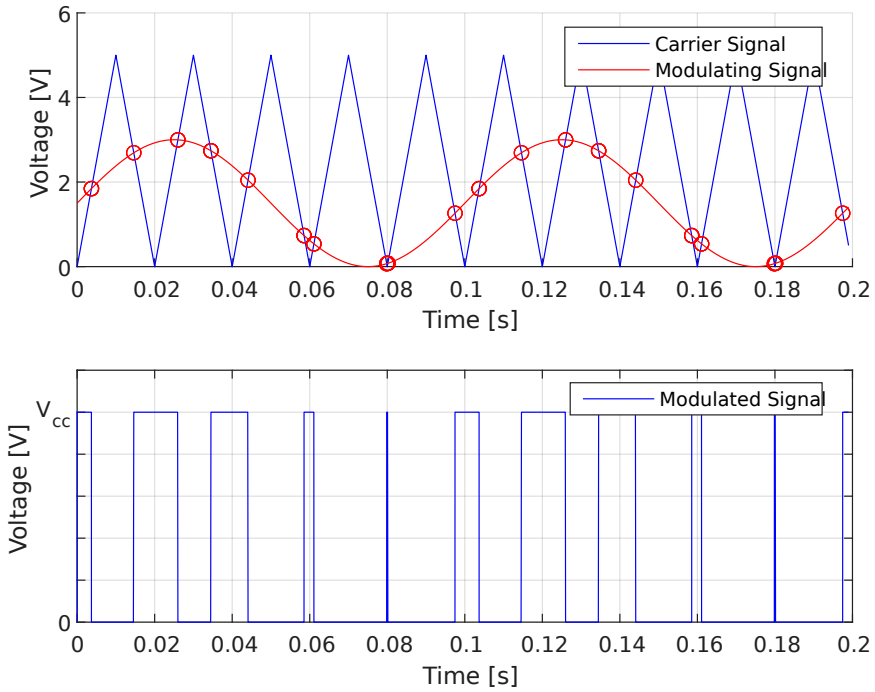

Figure 2.18: PWM generation with a sinusoidal modulating wave.

it used for centered modulation. The three different carrier waves are represented in Fig. 2.19.

An application of PWM related to this thesis is the driving of switches (often constitute by thyristors) in power electronics circuitries. As will be seen in section 2.5.1, the turning on and off of these switches is done by means of PWM signals. There are many cases where two switches must be driven in a complementary fashion to avoid short circuits. For this reasons, complementary PWM signals (represented in Fig. 2.20) are used for the purpose. Their main characteristic is that one signal is high when the other is low and vice versa. From a theoretical point of view, this is enough to avoid shorts due to switches that must be driven in a complementary fashion, but in practice turn-off and turn-on delays can't be neglected. To take into account delays of real components, a deadtime insertion could be required. In complementary PWM signals, the deadtime is the time interval that elapses from the turn off of one signal to the turn on of its complementary one. An example of deadtime insertion between two complementary PWM signals is reported in Fig. 2.20. The application and the necessity of complementary PWM and deadtime insertions will be more clear in section 2.5.1 where inverters are discussed. More about PWM could be explained, but the topics discussed in this section were

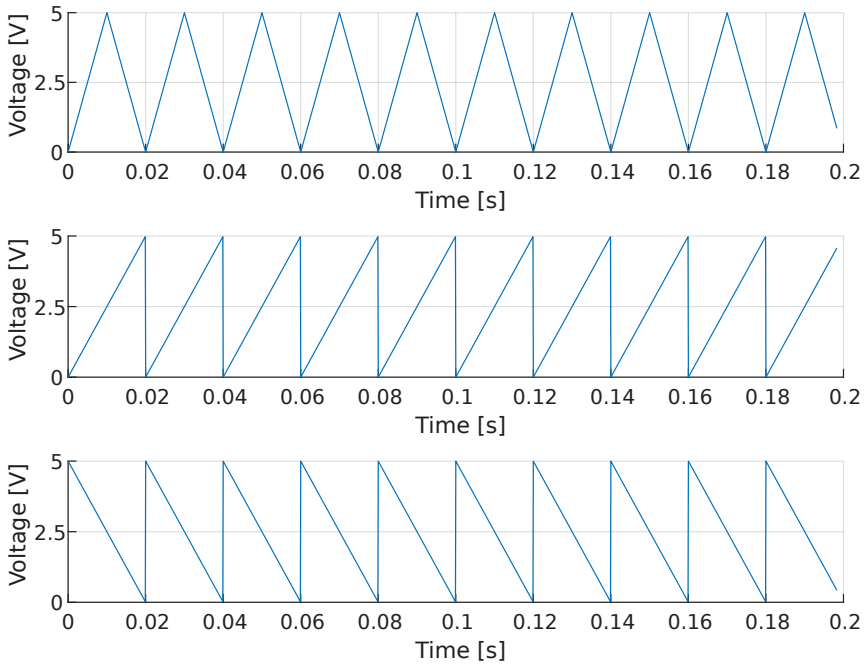

Figure 2.19: Different carrier waveforms - on top a triangular carrier, in the middle a trailing sawtooth carrier, and on bottom leading sawtooth carrier.

considered the most pertinent for the purpose of this thesis.

2.5 Power Electronics

In broad terms, the task of power electronics is to process and control the flow of electric energy by supplying voltages and currents in a form that is optimally suited for user loads. Power converters are autonomous systems. Therefore, their performance does not only depends on the hardware design but also on the control strategy used. In an electric powertrain system, the inverter is the power converter responsible to manage the power flow between the DC energy source, often constitutes by a battery, and the motor which almost always requires AC supply. For these reasons, an overview of inverters is required to understand some implication in the firmware development and peripheral configurations. Only three-phase pulse-width-modulated inverters will be treated and their structure, working principle, and related control techniques will be discussed in the following section. Instead, discussions about other power converters will be neglected because

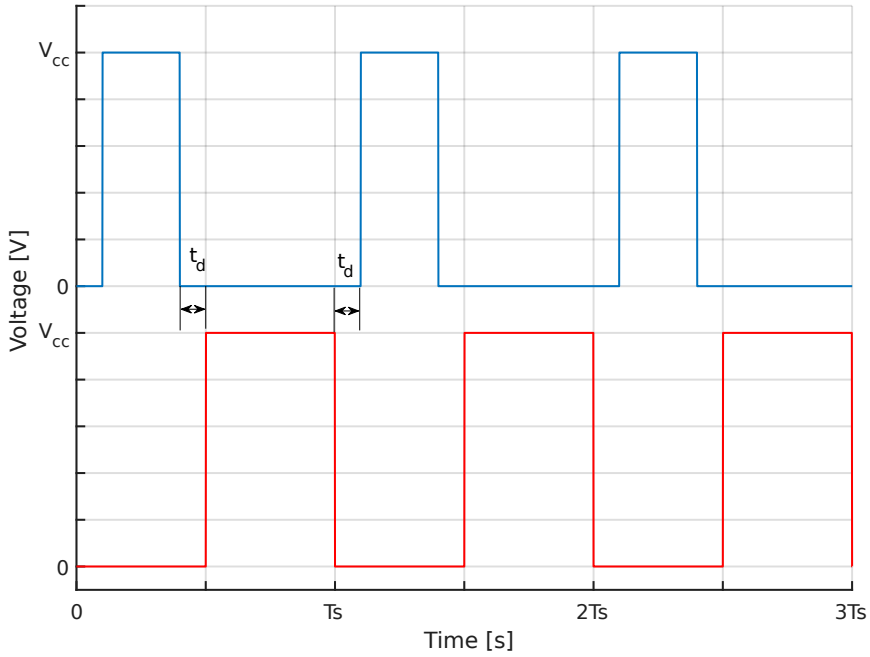

Figure 2.20: Two complementary PWM signals with deadtime insertion.

outside the purposes of this thesis.

2.5.1 Inverter

In general, an inverter is a power electronic device that converts direct electric energy (DC) in alternating electric energy (AC). Inverters are used in AC motor drives and in interruptible AC power supplies where the objective is to produce a sinusoidal AC output whose magnitude and frequency can both be controlled. This power flow through inverters is reversible. However, most of the time the power flow is from the DC side to the motor on the AC side, but the inverse condition can occur in case of regenerative breaking. The DC energy source can be obtained using batteries or rectifying and filtering line voltage. In this context, the power conversion stage used to obtain the DC energy source is neglected.

The general structure of a three-phase inverter circuit consists of three legs, one for each phase, as shown in Fig. 2.21. The supply is assumed to be a constant voltage source. Each leg is made of two transistors and two diodes. Transistors and diodes are both solid-state semiconductor devices – the first ones act exclusively as switches, conducting when the voltage between the gate and emitter exceeds the threshold value V_g of the device, and continuing to conduct until this voltage does

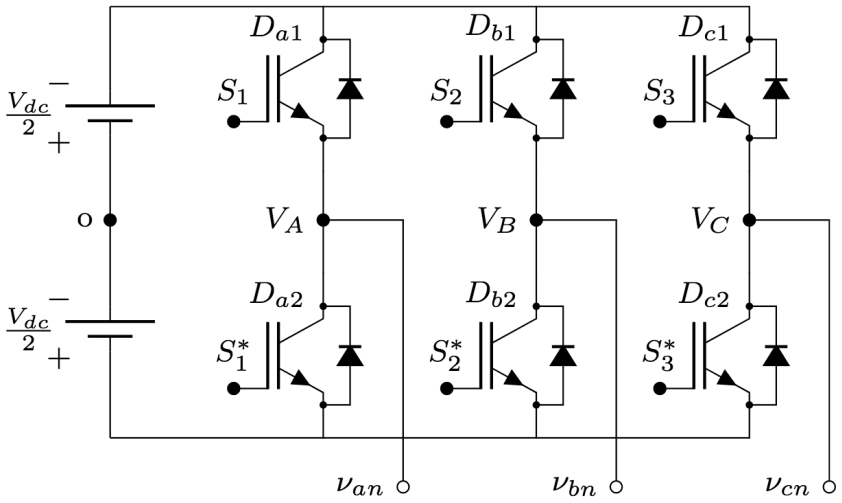

Figure 2.21: Basic structure of a three phase inverter.

not decrease under V_g, instead the latter ones conduct current primarily in one direction. As shown in Fig. 2.21, points A, B, and C can be connected to ground turning on the switch in low position and turning off the corresponding one placed in high, and the opposite can be done to connect the points to V_d. This means that the two switches must be driven in a complementary fashion to avoid short circuits. As highlighted in section 2.4, the switches are not ideal that means that the status of the two switches in an inverter leg can't be changed instantaneously from on to off and vice versa. This delay is often called blanking time and this is the motivation of the deadtime insertions described in section 2.4.

The AC energy on load size is obtained driving the thyristors with opportunely control techniques which objective is to shape and control the three-phase output voltages in magnitude and frequency using the constant input voltage V_d. Practically, controlling an inverter means to generate three different couple of complementary PWM signals which turn on and off the switches represented by the six thyristors. For example, the three-phase output voltages can be obtained using the same triangular carrier voltage waveform and comparing it with three sinusoidal modulating voltages that are 120° out of phase, as shown in Fig. 2.22. Deadtimes should be insert in the generated signals to avoid shot circuits due to blanking times. Of course, more advanced techniques can be used to generate the PWM signals to drive the inverter, but they will not be treated because outside of the scope of this thesis. In fact, the main objective of this section remains to analyze only the aspects that will be relevant for the next chapters. However, a complete discussion about inverters can be found in chapter 8 of reference [3].

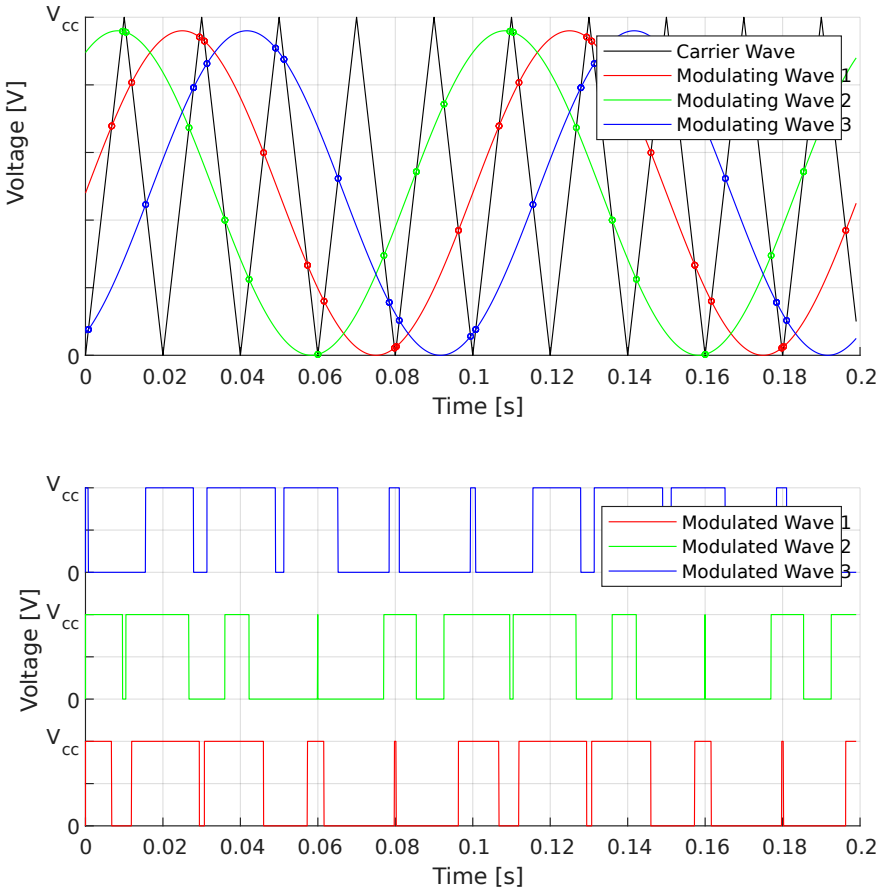

Figure 2.22: Example of a three phase sinusoidal PWM generation.

2.6 Resolver

A synchronous resolver is a type of rotary electrical transformer used for measuring degrees of rotation. In the application of this thesis, the resolver was used to measure the angular position of the motor rotor. It looks like as a small electrical motor with a stator and a rotor, but it is different for the internal wire windings. The stator houses three windings: an exciter winding, and two-phase windings. The rotor houses a winding which can be considered the secondary coil of a rotatory transformer, where the first one is constituted by the exciter winding. They are arranged in the axis of the resolver. In such a way, a current in the stator coil is inducted without limiting its rotation and no need for brushes. The two-phase windings in the stator are configured at 90 degrees from each other.

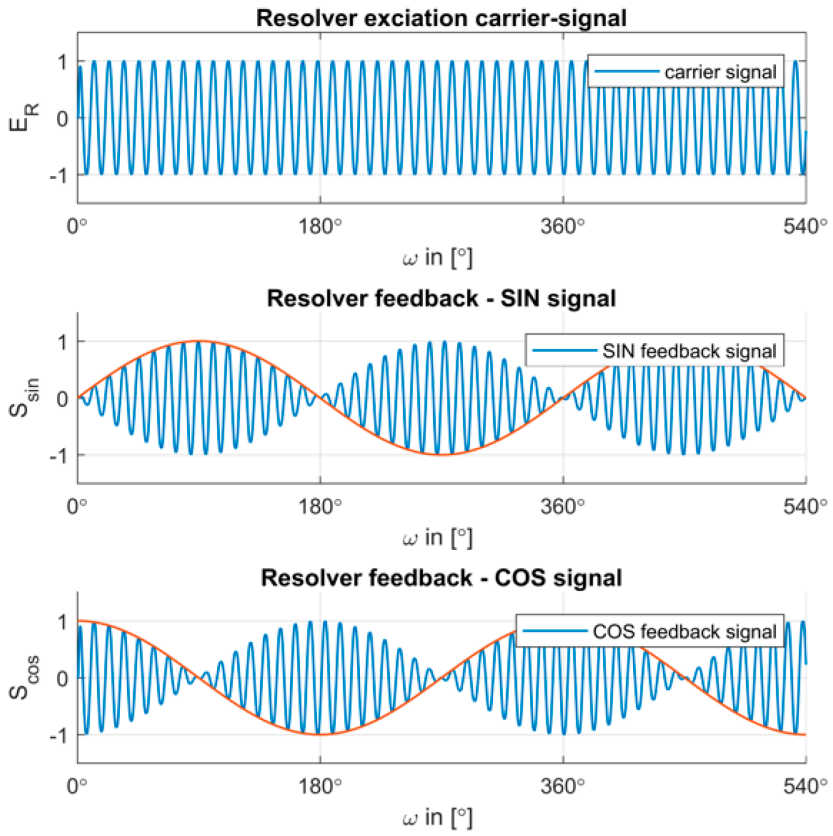

Figure 2.23: Output signal of a resolver. The excitation wave on the top, sine feedback in the middle, and cosine feedback on the bottom.

The working principle of the resolver is based in the excitation of the two two-phase windings on the stator by the rotor coil. More precisely, the primary winding of the transformer, fixed to the stator, is excited by a sinusoidal electric current, which by electromagnetic induction induces current in the rotor. The two two-phase windings, fixed at right (90°) angles to each other on the stator, produce a sine and cosine feedback current. As is shown in Fig. 2.23 The relative magnitudes of the two-phase voltages are measured and used to determine the angle of the rotor relative to the stator. Since no direct position acquisition can be done by analog-to-digital conversion, Resolver-to-Digital Converters are commonly used as interface between the resolver and other digital devices.

A Resolver-to-Digital converter converts the electrical information (analog signal) corresponding to a mechanical rotational angle of the Resolver to the corresponding digital data and output it. The most implemented technique to convert the analog signals coming from the resolver to digital signals is the Digital Tracking Method, which is not described because outside the purpose of this thesis. In the proof of

concept, an in-board chip Resolver-to-Digital Converter was used to provide the position of the rotor of the motor through SPI or digital parallel interface.

Chapter 3

Firmware Development

As stated in previous chapters, firmware is a specific class of computer software that provides the low-level control for the hardware of a specific device [4]. In the context of the proof of concept, its role was to provide a hardware abstraction layer (HAL) to be placed between the device and the application software. In such a way, it was possible to develop the application software independently using automatic code generation tools and without caring about the specific hardware implementation of the features required to suitable interact with the plant. The logical separation between firmware and application was taken into account during all the software development process, and for this reason, the interfaces between them were analyzed and defined a priori. In fact, defining the interfaces between the two layers allowed to develop the code separately, and when necessary, make changes in the implementation of one layer regardless to the other without affecting their interaction.

As is usual in firmware development, the programming language used to write the source code was the C language. C is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, while a static type system prevents unintended operations. By design, C provides constructs that map efficiently to typical machine instructions and has found lasting use in applications previously coded in assembly language [5].

For the electric powertrain prototype, no coding or international standards (i.e. MISRA C, ISO 26262, etc.) were used, but good practices were followed to write high-quality code and prevent bugs. The firmware was developed using the device drivers provided by the MCU silicon-vendor. The source code was organized in difference software components (SWC) - couples of a header files (i.e. SwcName.h) and a C source file (i.e. SwcName.c). Each software component manages and implements a specific set of related functions using one or more peripherals, and sometimes, other software components. In Fig. 3.1, a graphical representation of

the logical structure of a software component is reported. In the following sections, all software components are discussed in detail. Each section highlights the system specifications that shall be implemented, used peripherals and their settings, and features provided to the application software. More in detail, the sections describing software components provide brief overviews of their functions and applications, list of requirements that shall be implemented, specific firmware implementations, and API layer constituted by C functions and global accessible variables.

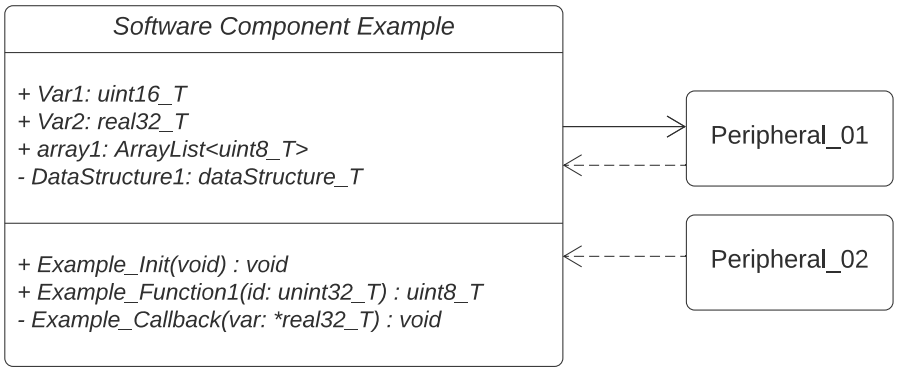

Figure 3.1: UML class diagram of an example of software component.

The source code was written following naming conventions, guidelines and policies provided by the customer for the proof of concept. The choice to follow a standardized approach was motivated by several reasons:

- Promote portability and avoid unexpected results;
- Avoid reliance on compiler or platform-specific constructs;
- Avoid errors due to ambiguity in language constructs;
- Measurably reduce program complexity;
- Reduce long-term support costs due to coding errors;
- Make software reuse easier;
- Improve code understanding and maintainability by different developers under the same organization.

In this thesis, the source code is not entirely reported because the focus is placed more on constructs, algorithms and procedures rather than the specific written code itself. Moreover, NDA policies do not allow to publish it.

3.1 Clock Management

The software component Clock Management, composed by the files ClkMgm.h and ClkMgm.c, is responsible to initialize the clock signals in the microcontroller. Its only purpose is to set the frequencies of the clock signals according to the requirements, and enable them to be provided to the used in-chip peripherals. It makes use of a Fast Internal RC Oscillator (FIRC), external fast crystal oscillator (FXOSC) which output signal is provided as input to a Phase-Locked Loop (PLL) in-chip circuitry, Clock Generation Module, and Mode Entry Module. These clock signal sources are both processed by dividers and selectors to generate all the device clock signals.

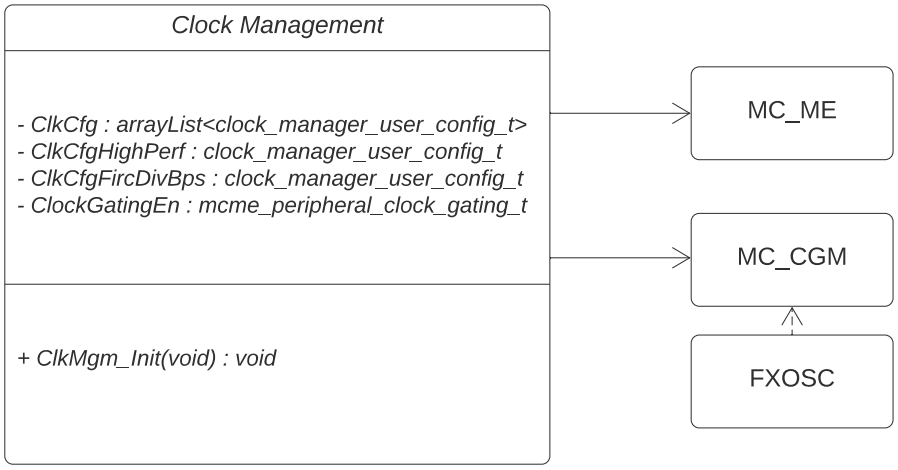

Figure 3.2: UML class diagram of the software component Clock Management.

3.1.1 Requirements

The requirements that Clock Management (ClkMgm) implements are related, of course, to the frequencies that shall be provided by the clock signals. First, a list of the required clock signals for the used in-chip peripherals was made. Then, the clock sources used to generate such signals were analyzed. In order to maximize the performance, the higher possible working clock frequencies shall be set for the processor core and peripherals. An in-chip clock source shall be used in the case of a hardware fault related to the external oscillator occurs. The clock source signals chosen for this purpose were the FIRC and PLL. The PLL can provide a frequency up to 1280MHz, instead the FIRC up to 48MHz. The main advantage of the FIRC with respect to PLL is that it doesn't need any external oscillator to work. For this reason, it is also the boot source clock signal chosen by the silicon-vendor. Therefore, the PLL clock generator was used as main clock source signal, and the

FIRC oscillator as recovery one. In the board designed for the MCU, the external oscillator connected to the device and used by the PLL is an external quartz crystal oscillator (FXOSC) with frequency of 16MHz. The upper limits of the frequencies of the each clock signal generated from the main clock source is reported in Tab. 3.1. More signals can be generated (mainly for communication peripherals) starting

Maximum Clock Signal Frequencies		
Device	**Clock Signal**	**Frequency**
CM7, CM7 Flash control port	M7_CORE_CLK	320MHz
CM33 cores, AXBS, Flash controller port	M33_CORE_CLK	160MHz
SRAM/XBAR/AIPS	AIPS_PLAT_CLK	80MHz
Peripherals	AIPS_SLOW_CLK	40MHz
HSE IPS Interface	HSE_IPG_CLK	80MHz
PLL Clock	PLL_CLK	320MHz
FIRC Clock	FIRC_CLK	48MHz

Table 3.1: Frequency limits of the main clock signals present in the microcontroller.

from the ones of Tab. 3.1, but they are neglected because not used in the project.

The frequency and enabling of each clock signal can be managed in hardware by means of the Clock Generation Module (MC_CGM). The MC_CGM implements software configurable clock dividers and multiplexers for selecting from the various clock sources guaranteeing glitch-less transitions. Therefore, the software component Clock Management shall configure the Clock Management Module to set the maximum possible frequency for each signal in Tab. 3.1 using the PLL output or FIRC as clock source signals. The peripheral clock signal inputs can be enabled or gated using the Mode Entry Module (MC_ME). Before to use the in-chip peripherals and modules, their corresponding clock input signals shall be enabled. This procedure shall be executed at startup and it is a task that the software component shall perform. The list of the peripheral clock signals to be enable by ClkMgm through the MC_ME module is reported in Tab. 3.2. In the table, it is also reported the clock signals used by the in-chip devices and their corresponding maximum working frequencies. The clock gating and clock frequencies shall not be changed at run-time, but only in the initialization phase.

3.1.2 Implementation

In order to configure the MC_CGM and MC_ME according to the described specifications, the corresponding device driver provided by the silicon-vendor was used. This choice was motivated by the way implemented in the driver to configure

Peripheral/Module Clock Input Signals		
Peripheral/Module	**Clock Input Signal**	**Max. Frequency**
Fast External Crystal Oscillator	-	16MHz
Phase Locked Loop	FXOSC	320MHz
eMIOS 0	M33_CORE_CLK	160MHz
Analog-to-Digital Converter 0	M33_CORE_CLK	80MHz
Analog-to-Digital Converter 1	M33_CORE_CLK	80MHz
Analog-to-Digital Converter 2	M33_CORE_CLK	80MHz
Body Cross Triggering Unit	M33_CORE_CLK	160MHz
Programmable Interrupt Timer 0	AIPS_SLOW_CLK	40MHz
Programmable Interrupt Timer 1	AIPS_SLOW_CLK	40MHz
Low Power SPI 2	AIPS_SLOW_CLK	40MHz
FlexCAN 2	FIRC_CLK	48Mhz

Table 3.2: Peripheral/Module clock input signals to enable and their corresponding maximum supported frequencies.

the various clock signals - it allows to choose among five different configurations and two of them implement the described requirements using the PLL output or FIRC as clock source signals. The driver needs two configuration structure: the first one is used to configure the clock signal frequencies, the second one to disable the clock gating of the clock input signals for the peripherals that the user wants to use. The selected clock configurations are the *High Performance* and the *FIRC Divider Bypassed*.

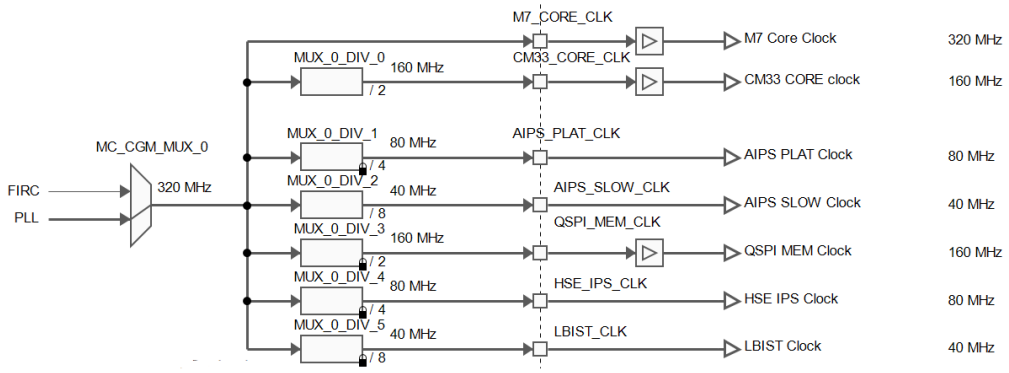

Figure 3.3: Clock signals obtained using PLL as clock source.

In High Performance mode, the MC_CGM is configured to have the PLL output as source clock for other clock signals with a frequency equal to 320MHz. As said

in the previous section, the input of the PLL circuitry is a 16MHz clock signal provided by the FXOSC. The other clock signals are obtained processing the PLL output by means of dividers to have the maximum allowed frequency for each of them. The corresponding configuration of the MC_CGM is show in Fig. 3.3.

The configuration *FIRC Divider Bypassed* implements the same mechanism using the FIRC as clock source. In this configuration, the FIRC isn't processed by any divider, obtaining its maximum working frequency of 48MHz. This means that the other clock signals can't be faster than 48MHz. In particular, the frequencies of the clock signals are set as shown in Fig. 3.4. Of course, this leads to lower performances, but allows the system to boot and work also in case of hardware fault about the external oscillator.

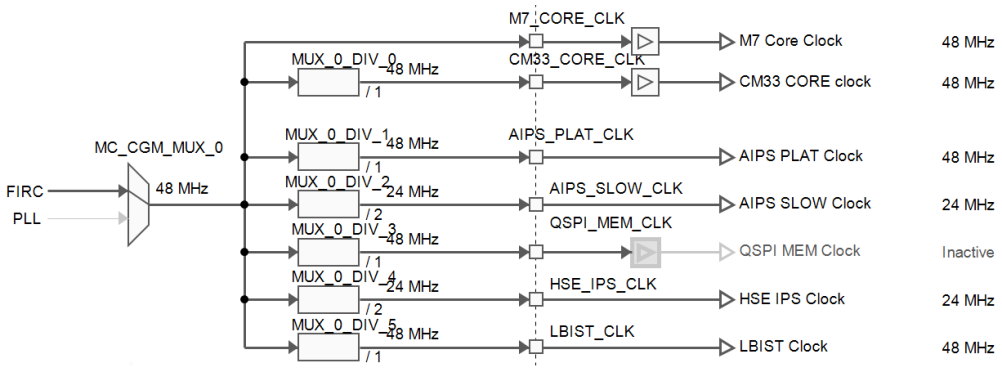

Figure 3.4: Clock signals obtained using FIRC as clock source with its maximum working frequency.

The peripheral clock gating, managed through the MC_ME, is configured providing to the driver an array with a list of mcme_peripheral_clock_gating_t variables which represents the peripheral to enable. Since the clock gating and clock frequencies shall not be changed at run-time, the configuration selection was done by means of macros. More precisely, the constant SYSTEM_CFG_CLK_HIGH_PERF shall be defined in case of the user wants to set *High Performance* configuration, instead SYSTEM_CFG_CLK_FIRC_DIV_BPS for *FIRC Divider Bypassed* one. Both configuration constants are defined in the header file System_cfg.h and one of them shall be commented depending on the selected configuration.

Listing 3.1: Clock Gating configuration array.

```
/*Array of peripherals to enable through the MCME module */
mcme_peripheral_clock_gating_t
    ClockGatingEn[CLK_GATE_EN_COUNT] =
```

```
{
    MCME_eMIOS_0,
    MCME_Analog_to_digital_converter_0,
    MCME_Analog_to_digital_converter_1,
    ...., // The list continues
};
```

3.1.3 API

The API layer of the software component Clock Management is constituted of only by an initialization function and a function used to provide one clock signal to an output pin. The initialization function is called at startup as first before to initialize other peripherals. Instead, the use of the function ClkMgm_ClkOut() is described in the chapter related to testing in section 4.3.1 and was not inserted in the API of the software component.

Functions

ClkMgm_Init()

status_T ClkMgm_Init(void)

This function configures the MC_CGM and MC_ME modules to obtain the *High Performance* or *FIRC Divider Bypassed* clock configuration according the macro defined in System_cfg.h file.

Parameters

None.

Returns

status_T - Status return code.

status_success	The clock configuration is set correctly.
status_clkInitError	An error occurs in initializing one of the clock signals.

Notes

This function shall be called before to initialize any peripheral due to clock gating issues.

3.2 Pin Management

The software component Pin Management (PinMgm), composed by the files PinMgm.h and PinMgm.c, is responsible to set the correct pin configuration for each input or output signal. It makes use of the module SIUL2 to select the functions

and electrical characteristics that appear on the external pins of the device. The functions implemented by Pin Management aren't used by the application software, instead they are called in the initialization functions of the various software components before to configure the corresponding peripherals.

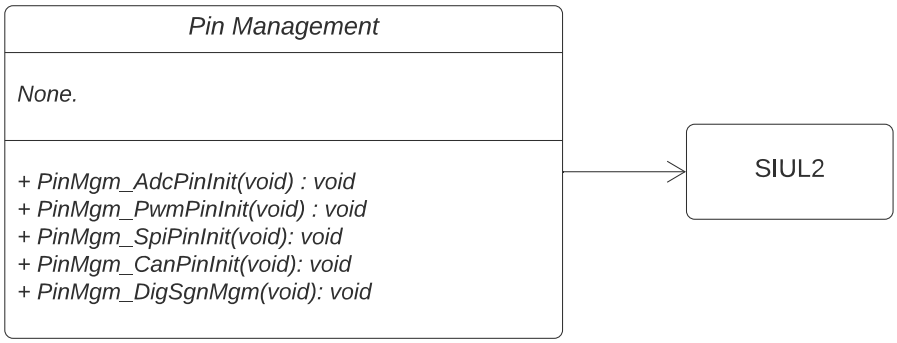

Figure 3.5: UML class diagram of the software component Pin Management.

3.2.1 Requirements

The software component Pin Management shall implement the pin-out defined for the proof of concept. A list of signals with their corresponding MCU pins was provided on the base of board that hosts the MCU.

3.2.2 Implementation

The module SIUL2 is managed writing and reading directly in the peripheral configuration registers because dedicated drivers are not provided by the silicon-vendor. The Multiplexed Signal Configuration Registers (MSCR) are used to select which source signal is connected to the register's associated destination, which is a chip pin that is or can be configured as an output. The Input Multiplexed Signal Configuration Registers (IMCR) is used to select which source signal is connected to the register's associated destination, which is an internal module port that is, or can be configured as, an input. For our configuration purposes, the MSCR were predominantly used.

The pin alt mode (i.e. the connection between the physical chip pin and the in-chip peripherals) is selected writing a specific value, reported in the datasheet, in the Source Signal Select bit field (SSS_0, SSS_1, SS_2, and SSS_3 bits) placed inside the MSCR of the corresponding pin. The Output Buffer Enable (OBE) bit, and Input Buffer Enable (IBE) bit are set respectively in case of output or input signals. The Invert (INV) bit was used to invert the polarity of the corresponding

pin directly in hardware. Other settings are left as default ones. An example of the code used to configure a MCU pin is shown in the following:

Listing 3.2: Example of pin configuration

```
/* Source Signal Select_0: 0x00u. */
SIUL2->MSCR49 &= ~SIUL2_MSCR48_SSS_0_MASK;
/* Source Signal Select_1: 0x01u. */
SIUL2->MSCR49 |= SIUL2_MSCR48_SSS_1_MASK;
/* Source Signal Select_2: 0x00u. */
SIUL2->MSCR49 &= ~SIUL2_MSCR48_SSS_2_MASK;
/* Source Signal Select_3: 0x00u. */
SIUL2->MSCR49 &= ~SIUL2_MSCR48_SSS_3_MASK;
/* GPIO Output Buffer Enable: 0x01u. */
SIUL2->MSCR49 |= SIUL2_MSCR48_OBE_MASK;
/* GPIO Input Buffer Disable: 0x00u. */
SIUL2->MSCR49 &= ~SIUL2_MSCR48_IBE_MASK;
/* GPIO polarity: not inverted */
SIUL2->MSCR49 &= ~SIUL2_MSCR48_INV_MASK;
```

3.2.3 API Layer

The API Layer of Pin Management provides an initialization function for each software components. These functions are called before to configure the peripheral during the initializations of the software components done at startup.

Functions

PinMgm_AdcPinInit()

void PinMgm_AdcPinInit(void)

This function initializes the pin alt mode and electrical characteristics for acquisition of the analog signals.
Parameters
None.
Returns
None.
Notes
This function is called by the initialization function of the software component ADC Management.

PinMgm_PwmPinInit()

void PinMgm_PwmPinInit(void)

This function initializes the pin alt mode and electrical characteristics for PWM generation.

Parameters

None.

Returns

None.

Notes

This function is called by the initialization function of the software component PWM Management.

PinMgm_SpiPinInit()

void PinMgm_SpiPinInit(void)

This function initializes the pin alt mode and electrical characteristics for SPI communication.

Parameters

None.

Returns

None.

Notes

This function is called by the initialization function of the software component SPI Communication.

PinMgm_CanPinInit()

void PinMgm_CanPinInit(void)

This function initializes the pin alt mode and electrical characteristics for CAN communication.

Parameters

None.

Returns

None.

Notes

This function is called by the initialization function of the software component CAN Communication.

PinMgm_DigSgnPinInit()

void PinMgm_DigSgnPinInit(void)

This function initializes the pin alt mode and electrical characteristics for digital signal acquisition and driving.

Parameters
None.
Returns
None.
Notes
This function is called by the initialization function of the software component Digital Signal Management.

3.3 ADC Management

The software component ADC Management (AdcMgm) is composed by the files AdcMgm.c and AdcMgm.h, and it is responsible to manage the acquisition of analog signal according to the system requirements. ADC Management makes use of 3 SAR-ADCs and the peripheral BCTU present in the MCU. The BCTU is utilized to trigger analog-to-digital conversions of specific ADC channels listed with a precise sequence, and the use of more ADCs is motivated by the possibility to implement multiple parallel conversions.

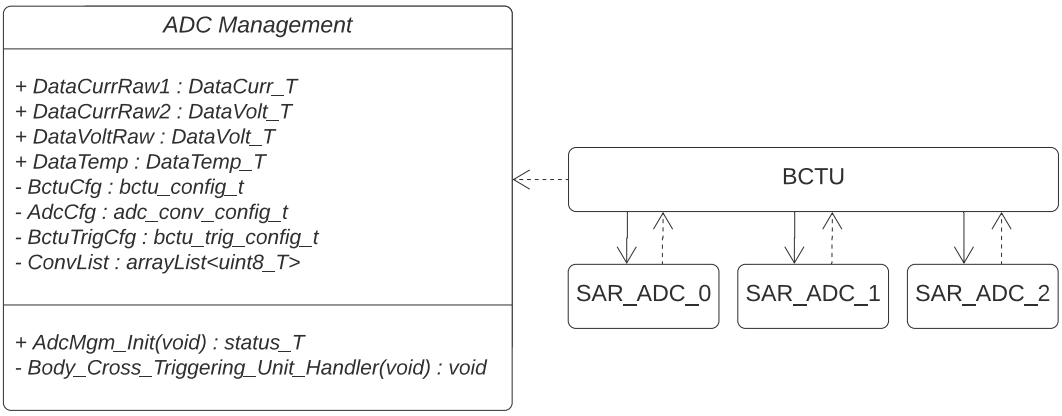

Figure 3.6: UML class diagram of the software component ADC Management.

3.3.1 Requirements

The specifications that were implemented through AdcMgm are related to analog signal acquisition - a list of analog signals shall be acquired in specific time instants with well-defined sample rates and, and sometimes, in a synchronous manner. In Table 3.3, the list of time requirements of each analog signal that has to be acquired is reported. The table shows three groups of signals which can be classified as critical or not. Critical signals are the ones that shall be acquired with a specific sampling frequency because are relevant from the control point of view. They are constituted by all the system electrical quantities (i.e. voltages and currents) and their required sampling frequency was not reported for confidentiality. In particular, in all this thesis, the sampling frequency of the critical signals was set to a generic value of 16kHz which is a perfect divider of the clock signal M33_CORE_CLK (160 MHz). Not critical signals are mainly system temperatures. They shall be monitored, but their dynamics are relatively slow with respect to the electric quantities of the system, which means that they can be acquired with a lower sampling rate without affecting the control actuation.

Signal Group	Analog Acquisition Requirements		
	Rate	**Timing**	**Description**
Phase currents	16kHz	To be acquired simultaneously at the center of the PWM switching period.	The phase currents represent the main physical quantities used to control the electric motor.
Supply voltages	16kHz	To be acquired simultaneously at the center of the PWM switching period.	The supply voltages represent the main physical quantities to be monitored to be sure that the system is correctly supplied.
System temperatures	1kHz	No specific requirements	The system temperatures shall be monitored for safety reasons and to correct temperature dependent parameters in the system model.

Table 3.3: Groups of analog signals to be acquired with their corresponding sampling rate and timing requirements.

As can be noticed (see section 3.4), the sampling frequency of critical signals is the same of the switching frequency (16kHz) of the PWM. Since the main control actuation corresponds to the change in duty cycle which is updated every switching period, to simplify the implementation, sampling electrical signals every switching period allows to monitor the system variations at each control step. The three phase currents were acquired simultaneously to provide a coherent acquisition. In general, all electric signals were acquired at the center of the PWM switching period because it was the only event available in hardware to trigger the conversion. The list of all the analog signals to be acquired is not reported for NDA policies.

3.3.2 Implementations

The described requirements were implemented starting to analyze the timing constraints on the acquisition of the phase currents because more critical with respect to other groups of signals from the control point of view. In Fig. 3.7, their timing constraints are reported graphically. As is shown in the figure, the center of

the T_{ON} period is a constraint about the time instant when to start the conversion due to the available hardware features. Since the switching frequency was set equal to 16kHz to simplify the implementation, these time instants were chosen to acquire the signals. This time instant corresponds to the event of reaching the highest value of the carrier waveform used as timebase for the PWM generation. For this reason, this event is utilized to signal to the BCTU when to start the conversion of the first elements of the conversion list which are constituted by the 3 phase currents.

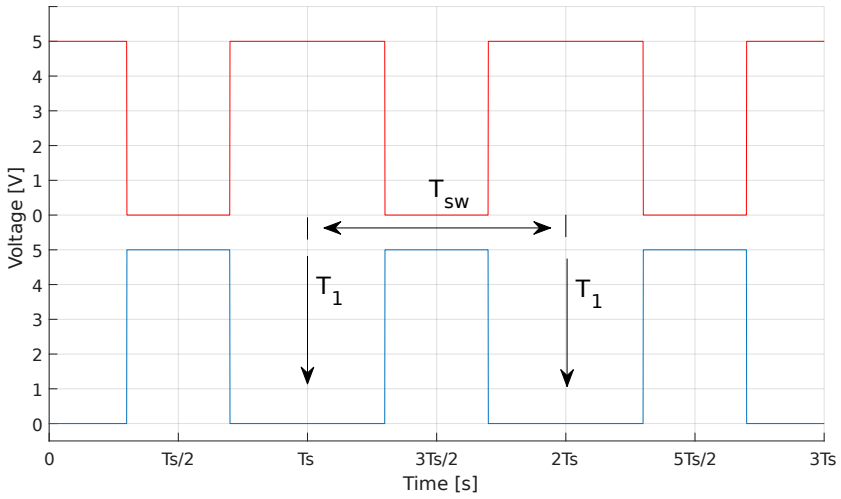

Figure 3.7: Timing constraints of the phase currents acquisition.

In order to acquire the three phase currents simultaneously, all the 3 SAR-ADCs hardware instances were used - one ADC for each signal. The ADCs were configured to be triggered by the BCTU, and the BCTU was configured to trigger the ADCs using a conversion list. The use of a conversion list allows multiple parallel conversions which are required for simultaneous acquisitions. The list was implemented using specific configuration registers that set which channels and ADCs has to be triggered for the conversion. The ADC selection is done by means of the register TRGCFG that contains 1 for each ADC to be set or cleared depending on the need to trigger on not the corresponding ADC. As is shown in Fig. 3.8, channels in the conversion list are grouped together in 2 or 3 elements if more than one ADC are selected. The registers LISTCHR_i allow the definition of the elements in the conversion list - each register can contain two elements and provides two bits per element to set if the element is the last in the list and to insert a waiting for another trigger source event to continue the conversions. Therefore, all analog signals of the groups of Table 3.3 were listed according to their priorities and requirements.

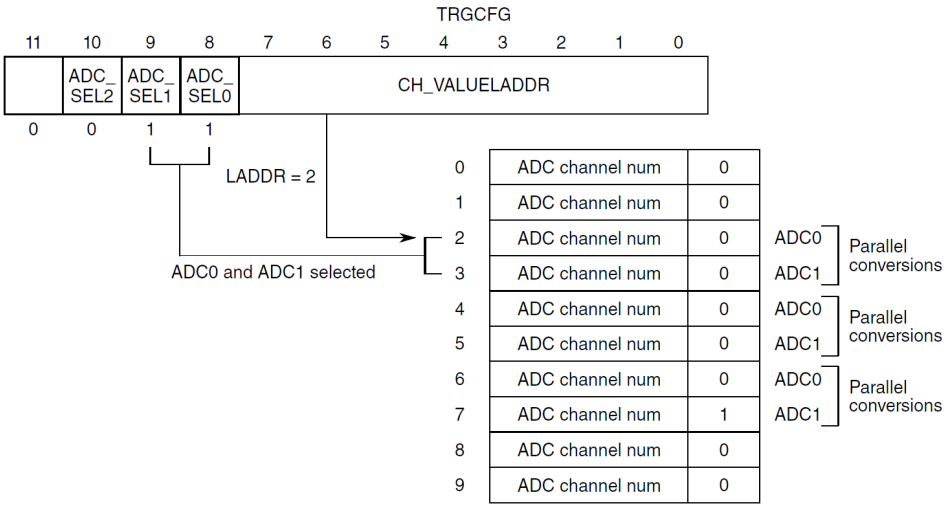

Figure 3.8: Logic diagram of the multiple parallel conversions list operations implemented by the BCTU.

The raw converted data are read by means of interrupts. More precisely, an interrupt, associated to the BCTU peripheral, is set in correspondence of the end of conversion of the last element in the conversion list. The corresponding interrupt handler is responsible to read the raw converted data and scale them from uint16_T (16 bit unsigned integers) to real32_T (float) variables with the correct measurement unit. Since the BCTU trigger event the instant T_1 in Fig. 3.7, the conversion of the elements in the list is performed every time the signals with the highest sample rate have to be acquired. This causes that all signals are converted with a sampling frequency of 16kHz which is higher than the required for the system temperatures signals. For this reason, even if their real sample rate is 16kHz, they are read in the BCTU interrupt handler in such a way to guarantee their nominal sampling rate and reduce the computational effort of the BCTU interrupt handler.

The reading of the raw converted results in the BCTU interrupt handler introduces a delay between the moment when the analog signals are converted and the moment when they are effectively read. Since the control algorithm in the application software needs these data to be executed, they shall be read before the beginning of each PWM switching period. Therefore, the maximum possible delay for the selected example frequency is a switching period that is equal to

$$d^{max} = \frac{1}{f_{sw}} = \frac{1}{16\text{kHz}} = 62.5\mu\text{s} \; . \tag{3.1}$$

For this reason, to verify the feasibility of this approach, the introduced delay was

analyzed and estimated according to the expressions reported in the MCU reference manual. The conversion time of the i-th triple of channels (three signals at time are converted due to the multiple parallel conversions feature) is given by the following expressions:

$$t_{conv}^{(i)} = \left[\Big(\mathrm{ST} + \mathrm{PST} + \mathrm{CT} + \mathrm{DP}\Big) + \mathrm{TPT}\right] \times \mathrm{TAD_{clk}}\,, \tag{3.2}$$

where the terms in the expression are:

- **ADC Clock Cycles (TAD_clk)**
  The TAD_clk is the time period of an AD_clk cycle and corresponds to the inverse of the clock frequency provided to the ADC peripherals. Since the clock signal provided to ADC peripherals has a frequency of 80Mhz, the value of TAD_clk is 12.5ns.

- **Sample Phase Time (ST)**
  The sample time is controlled by writing the the bit field of the configuration register INPSAMP[7:0] with an 8 bits value which represents units of AD_clk cycles. The minimum value of sample time is eight. If the value programmed is less than 8 then it has no effect on sample time duration and in this case sample time will be 8 AD_clk cycles. In this context, the ST was chosen by trial and error procedure and set to 100.

- **Pre-Sample Phase Time (PST)**
  The pre-sample phase time is equal to sample time with one AD_clk cycle delay for phase transition from pre-sample phase to sample phase.

- **Compare Phase Time (CT)**
  The compare phase time is affected by the evaluation time of a single bit and number of bits to be converted. For N bit conversions the value of CT will be N multiplied by the evaluation time of a single bit in terms of AD_clk cycles. The evaluation time of a single bit is estimated to 4 AD_clk in the MCU datasheet. Since the resolution of the SAR-ADCs is 15 bits, the CT was estimated to be equal to 60 ADC_clk cycles.

- **Data Processing Time (DP)**
  The data processing time is 2 cycles of AD_clk which are necessary to correct raw converted data from offset, gain, capacitor mismatch, etc.

- **Trigger Processing Time (TPT)**
  The trigger processing time consists in preparing the channel and calculating the initial gain value, and BCTU triggering time. It corresponds to about 1 AD_clk cycle and depends on the BCTU configuration.

According to these considerations, the conversion time of each triple of signals was estimated to be equal to

$$t_{conv}^{(i)} = \left[\Big(100 + 101 + 60 + 2\Big) + 1\right] \times 12.5\text{ns} = 3.3\mu\text{s}\,. \tag{3.3}$$

To compute the total conversion time, the structure of the conversion list has to be analyzed. As described before, the BCTU triggers the start of conversion of the channels in the conversion list. Since ADC0, ADC1, and ADC2 are used, the list elements are grouped by 3, assigning one element for each converter. Same spare channels were inserted in the conversion list because some signals (mainly not critical ones) were assigned to be converted by ADC0 even if ADC1 and ADC2 were free and available to be used. In those cases, a spare channel was inserted to maintain the alignment of the multiple parallel conversions of 3 channels at time. Of course, converted values of these channels aren't read in the dedicated interrupt handler because meaningless. This isn't the best solution for the project, but it was necessary to avoid the on-run reconfiguration of the BCTU peripheral. Since conversion list is structured to perform 9 multiple parallel conversions, the total conversion time was estimated to be equal to:

$$t_{conv} = n_{list} \times t_{conv}^{(i)} = 9 \times 3.3\mu\text{s} = 29.7\mu\text{s}\,, \tag{3.4}$$

with n_{list} the number of the multiple parallel conversions.

The acquired values shall be provided to the application software. For this reason, the interrupt handler, that is run at the end of the conversion of the last list element, is responsible to move the results of the conversions in well-organized and global accessible data structures which field are mapped from uint16_T to real32_T data when read by the application software. These data structures represents the data exchange interface between the AdcMgm software component and the application software and they are described in the following section.

3.3.3 API layer

The API layer of the AdcMgm software component is constituted by an initialization function called to set up correctly the involved pins and peripherals, and by a set of data structures utilized to store the mapped raw converted values.

Global Data Structure

DataCurrRaw

DataCurr_T DataCurrRaw

This data structure contains the measured currents at the center of the T_{ON}

time interval (instant T_1 in Fig.3.7) of the PWM switching period. The corresponding data fields are constituted by the 3 phase currents of the electric motor and are acquired synchronously through ADCs.

DataVoltRaw

DataVolt_T DataVoltRaw

This data structure contains the measured voltages at the center of the T_{ON} time interval (instant T_1 in Fig.3.7) of the PWM switching period. The corresponding data fields are constituted by supply voltages which are constantly monitored by the application software and acquired through ADCs.

DataTempRaw

DataTemp_T DataTempRaw

This data structure contains the temperatures measured with a delay with respect to the center of the T_{OFF} time interval (instant T_1 in Fig.3.7) of the PWM switching period. The corresponding data fields are constituted by system temperatures which are constantly monitored by the application software and acquired through ADCs.

Functions

AdcMgm_Init()

status_T AdcMgm_Init(void)

This function initializes the SAR-ADCs, BCTU, and if not yet initialized, the eMIOS timebase for generating the source events to trigger conversions.

Parameters

None.

Returns

status_T - Status return code.

status_success	The clock configuration is set correctly.
status_clkInitError	An error occurs in initializing one of the clock signals.

Notes

The operations performed during the initialization are the following:

1. Initialize the pins connected to the analog signals to be used by SAR-ADCs.
2. Configure the ADC0, ADC1, and ADC2 in BCTU triggered conversion mode.
3. Enable all the required channel related to the analog to digital signals to be converted.

4. Perform the self-calibration procedure of ADC0, ADC1, ADC2.
5. If not yet done, set the eMIOS time base (see section 3.4);
6. Reset the BCTU and set it in Configuration Mode;
7. Set the conversion list in the corresponding BCTU configuration registers;
8. Set the source trigger for the conversion list of BCTU;
9. Set the BCTU in Normal Mode to start performing multiple parallel conversions.
10. Enable the interrupt related to the end of conversion of the last element of the conversion list.

This function shall be called after the initialization of the software component PWM Management to avoid issues in configuring the eMIOS timebase.

3.4 PWM Management

The software component PWM Management (PwmMgm) is made by the files PwmMgm.c and PwmMgm.h, and it is responsible to manage PWM generation to drive the inverter board. PWM Management makes use of 1 eMIOS peripheral - 1 channel is used to generate the timebase representing the carrier waveform and the source events for analog-to-digital conversion, and other 6 channels to generate a PWM signal for each inverter switch.

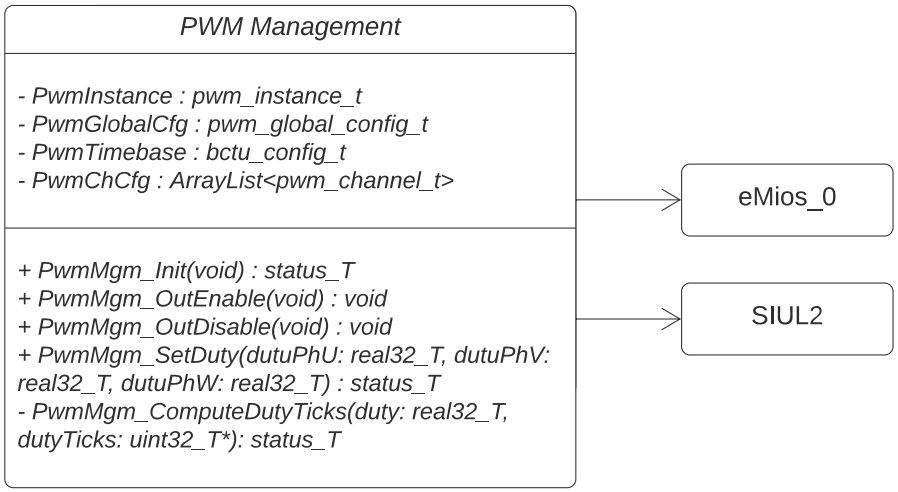

Figure 3.9: UML class diagram of the software component PWM Management.

3.4.1 Requirements

The requirements that were implemented with PwmMgm are related to the PWM generation. The PWM signals shall generated coupled by 2 in a complementary manner, and with a constant dead-time insertion. The PWM alignment shall be centered in order to simplify the implementation with the used hardware. In Table 3.4, the specifications of the six PWM signals are reported. Of course, all the PWM signals shall have the same switching frequency that, as already explained in section 3.3.1, was chosen equal to 16kHz as example because the used switching frequency was not reported due to nondisclosure policies. In the same way, the value of the considered dead time was set to 3μs only as example. Each couple of signals drives an inverter leg, that is why they shall work in the complementary manner. From empirical considerations, the dead-time insertion was required to take into account the blanking time of the inverter switches. The duty cycle of each couple of switches is computed by the motor control algorithm of the application software with the same rate of the PWM switching period which, in this case was

Signal	PWM Generation Requirements				
	Switching Frequency	**Duty Cycle Variable**	**Polarity**	**Deadtime**	**Alignment**
PwmUH	16kHz	PhUHduty	Positive	3μs	Centered
PwmUL	16kHz	PhUHduty	Negative	3μs	Centered
PwmVH	16kHz	PhVHduty	Positive	3μs	Centered
PwmVL	16kHz	PhVHduty	Negative	3μs	Centered
PwmWH	16kHz	PhWHduty	Positive	3μs	Centered
PwmWL	16kHz	PhWHduty	Negative	3μs	Centered

Table 3.4: List of PWM signals with their corresponding switching frequency, duty cycle variable, and polarity.

set to 16kHz as example. Moreover, the firmware shall provide the possibility to drive all the signals to low in case of faults disabling the PWM generation.

3.4.2 Implementations

The hardware used to implement PWM generation is constituted by the instance 0 of the in-chip peripherals eMIOS. The eMIOS clock source is the M33_CORE_CLK signal. Its internal counter is made by a 16 bits registers and it is updated every clock cycle. The peripheral allows to define a clock prescaler using the bit field GPRE in the configuration register MCR, but it was disabled in this context. The channel 0 was used to provide the time base to other channels which, of course, need a carrier waveform to generate the PWM output signals. The choice of channel 0 as channel to generate the time base was constrained because it is the only channel that can provide the time base to the internal shared bus to which all the required channels are connected.

Considering the system requirements described in the previous section, it is evident that a triangular carrier waveform is required to implement the centered aligned PWM signals (see section 2.4 for more details). Therefore, the channel 0 of the peripheral eMIOS was set in Modulus Counter Buffer (MCB) Up/Down Mode to act as an up/down counter providing a discrete triangular carrier waveform by its internal counter state. The channel operation mode is selected writing the appropriate value in the configuration register MODE corresponding to the channel. In general, when this mode is selected, the channel internal counter starts counting from 0x01 and is increased until the stored value in register A1 is reached. Then, the counter is decreased, and only when 0x01 is again reached, it restarts counting up. This mechanism is graphically explained in Fig. 3.10.

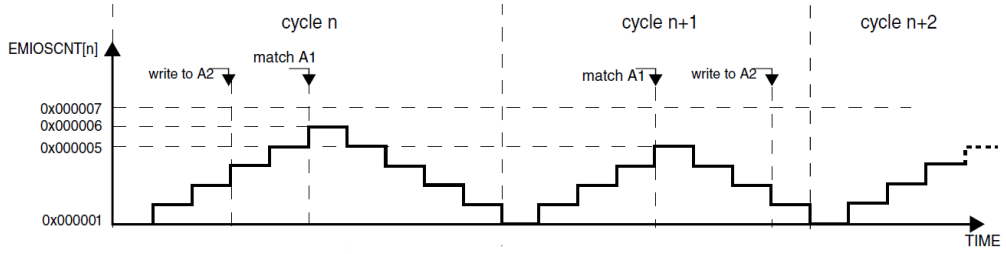

Figure 3.10: Time evolution of the internal counter of an eMIOS channel configured in MCB mode.

The frequency of the discrete triangular waveform, realized with the up/down counter, was set writing the register A1. As described in the MCU datasheet, its value can be computed with the folloing expression:

$$\frac{\text{PRECLK}}{f_{clk}} = \frac{2\big(\text{A1} - 1\big)}{2f_{sw}} \Rightarrow \text{A1} = \frac{f_{clk}}{2f_{sw}} - 1\,, \tag{3.5}$$

where PRECLK is the prescaler value, f_{clk} the frequency provided to the peripheral (M33_CORE_CLK), and f_{sw} the PWM switching frequency. Therefore, it was calculated to be equal to:

$$\text{A1} = \frac{f_{clk}}{2f_{sw}} - 1 = \frac{160\text{MHz}}{2 \times 16\text{kHz}} - 1 = 4999\,. \tag{3.6}$$

Since the register A1 can't be directly written, its shadow register A2 has to be used to copy the value in A1. As can be seen in Fig. 3.10, the register A1 is updated with the value contained in A2 whenever the counter reaches the value 0x01 (i.e. starting of switching period). The device driver provided by the MCU silicon-vendor allows to define the time base frequency in number of clock ticks (i.e. clock cycles) without the necessity to compute the exact value of A1. However, these considerations were done to verify the correct hardware configurations during the corresponding test procedure.

In order to generate the PWM signals with the requirements described in the previous section, the Center Aligned Output Pulse Width Modulation with Dead Time Insertion Buffered (OPWMCB) Mode was used for all the required channels with the exception of channel 0 that, as said, provides the time base. As the name suggests, this operation mode generates a center aligned PWM with dead time insertion in the leading or trailing edge depending on the channel configuration. In general, the time base selected for a channel configured in OPWMCB mode should be a channel configured in MCB Up/Down mode for the reasons already discussed in section 2.4. When operating with leading edge dead time insertion, the first A1

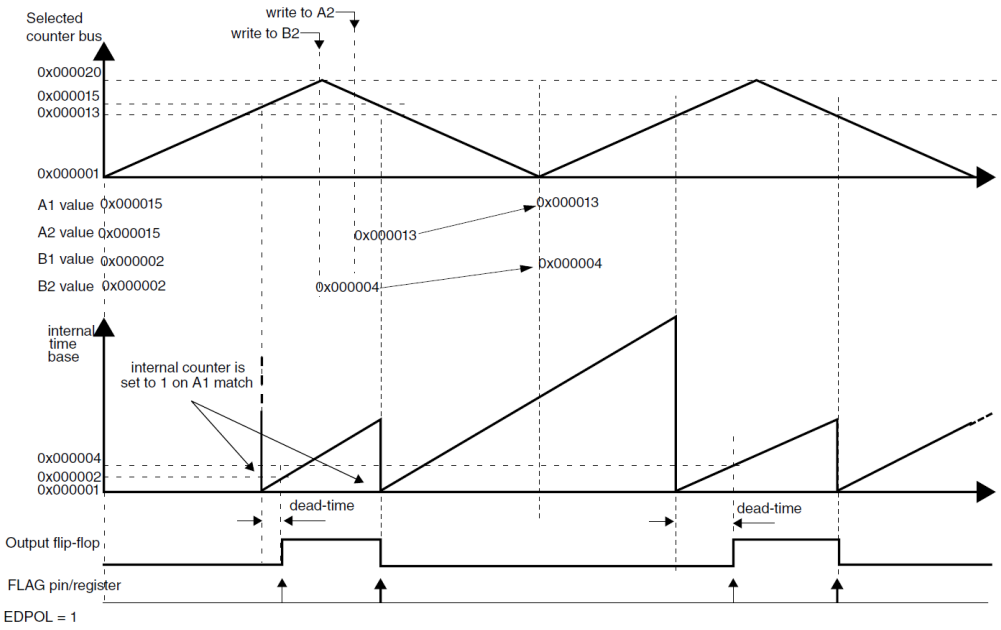

Figure 3.11: Time evolution of internal counter of an eMIOS channel configured in OPWMCB mode with lead deadtime insertion.

match sets the channel internal counter to 0x1. When a match occurs between register B1 and the internal time base, the output flip-flop is set to the value of the EDPOL bit. Instead, when operating with trailing edge dead time insertion, the first match between A1 and the selected time base sets the output flip-flop to the value of the EDPOL bit and sets the internal counter to 0x01. In the second match between register A1 and the selected time base, the internal counter is set to 0x01 and B1 matches are enabled. When the match between register B1 and the selected time base occurs the output flip-flop is set to the complement of the EDPOL bit.

It is evident that the value of B1 sets the deadtime duration of the PWM signal expressed in number of clock cycles. As done for the value of the register A1, its value was computed for verification purposes. It was computed in the following:

$$\mathrm{B1} = \mathrm{T}_{dt} \times f_{clk} = 3\mu\mathrm{s} \times 160\mathrm{MHz} = 480\,, \tag{3.7}$$

where T_{dt} is the deadtime time interval expressed in second. The register B1 is shadowed as well as the register A1, therefore the same consideration can be done about its update mechanism - both registers are buffered, that means that their values are changed writing their corresponding shadow register A2 and B2. A1 and B1 are updated with the value stored in A2 and B2 at the beginning of each switching period (i.e. when the provided time base is 0x01).

When the OPWMCB mode is selected, the output PWM duty cycle is equal to the difference between register A1 and register B1 for a leading edge dead time insertion, and to the sum of register A1 and register B1 for a trailing edge one. The polarity of the PWM signals is configured setting the bit EDPOL (contained in the eMIOS channel control register) to 1 in case of negative polarity or to 0 vice versa. Even in this case, the system call interface of the device driver hides these implementation details, but as alredy said, they were analyzed to verify the correctness of the hardware configurations during the verification phases.

Therefore, in order to implement the described system requirements, the PWM signals PwmUH, PwmVH, and PwmWH (channels 6, 2, 5) are set with leading edge deadtime insertion and positive polarities, instead PwmUL, PwmVL, and PwmWL (channels 4, 3, 1) with trailing edge deadtime insertions and negative polarities. The duty cycle of each couple of signals that controls an inverter leg must be the same to guarantee the complementary behaviour. Inserting the dead-time in trailing or leading edge depending on the signal polarity leads to the signal characteristics reported in Fig. 2.20. This mechanism resolves the issue of the blanking time of the inverter switches. The deadtime was selected a priori, so the register B1 was set only during initialization.

According to the requirements, features to disable or enable the outputs of PWM signals shall be provided. The eMIOS does not implement mechanisms to disable its outputs without reconfigure the peripheral itself, therefore a workaround was developed for these features. Indeed, the GPIOs connected to the corresponding pins of the PWM outputs were configured and set to low values. When the application software requires to disable the PWM outputs, the corresponding pin alt mode is changed such that the GPIO mode is selected instead of the eMIOS one. When the application software needs to enable the PWM outputs, the opposite is done. Of course, this is not the best solution, but it was necessary because other approaches couldn't be developed due the constraints about the pin-out of the board designed to host the MCU.

3.4.3 API layer

The API layer of the software component PwmMgm is composed by an initialization function to configure correctly the required channels of peripheral eMIOS0, two functions to enable or disable the outputs of the PWM signals, and a function to set the three duty cycles.

Functions

PwmMgm_Init()

status_T PwmMgm_Init(void)

This function initializes the channels from 1 to 6 of eMIOS peripherals in OPWMCB mode, and if not yet initialized, the eMIOS channel 0 in MCB mode to provide the time base to other channels.
Parameters
None.
Returns
status_T - Status return code.

status_success	The channels of the peripheral eMIOS0 are set correctly for PWM generation.
status_pwmInitError	An error occurs in initializing one of the channels of the peripheral eMIOS0.

Notes
The operations performed during the initialization are the following:

1. Initialize the pins related to PWM as output pins connected to eMIOS0.
2. If not yet done, configure the channel 0 in MCB mode to generate the time base with the correct frequency.
3. Configure the channels from 1 to 6 in OPWMCB mode to allow PWM generation.
4. Call the function PwmMgm_OutDisable to avoid that the PWM signals are provided to the output pins with unknown duty cycles.

The function shall be called before to initialize the software component ADC Management and use the PWM Management functions.

PwmMgm_OutEnable()

void PwmMgm_OutEnable(void)

This function enables the output pins of the generated PWM signals. When the PWM outputs are enabled, the output signals are driven to generate PWM waveforms according to the last set duty cycle, polarity and deadtime.
Parameters
None.
Returns
None.
Notes
The output enabling mechanism is implemented with the workaround described in the previous section.

PwmMgm_OutDisable()

void PwmMgm_OutDisable(void)

This function disables the output pins of the generated PWM signals. When the PWM outputs are disabled, the output signals are driven to low values.
Parameters
None.
Returns
None.
Notes
The output disabling mechanism is implemented with the workaround described in the previous section.

PwmMgm_SetDuty()

status_T PwmMgm_SetDuty(real32_T dutyPhU, real32_T dutyPhV, real32_T dutyPhW)

This function sets the duty cycle of the three inverter legs.
Parameters

in	dutyPhU	Duty cycle of phase U
in	dutyPhV	Duty cycle of phase V
in	dutyPhW	Duty cycle of phase W

Returns
status_T - Status return code.

status_success	The duty cycles are set correctly.
status_pwmDutyError	One of the duty cycle is outside the expected range and it is saturated to lowest or highest limits.

Notes
The input parameters shall be comprised between DUTY_MIN_VALUE and DUTY_MAX_VALUE - these values are macro defined for the specific application. For this reason, a check is done to saturate the duty cycles. This was implemented by the following source code:

Listing 3.3: Duty cycle conversion from percentage to ticks

```
    /* Verify that the duty cycle is in the correct
       range and saturate it if outside the expected
       limits */
    if(duty > DUTY_MAX_VALUE)
    {
        duty = DUTY_MAX_VALUE;
        return status_pwmDutyError;
    }else if(duty < DUTY_MIN_VALUE)
```

```
    {
        duty = DUTY_MIN_VALUE;
        return status_pwmDutyError;
    }

    /* Compute the duty cycle in number of ticks */
    *dutyTicks =
        (uint32_T)((real32_T)(PWM_TICK_PERIOD) * duty);
    /* Return Status_success */
    return status_success;
```

Moreover, the function arguments are converted to uint32_T (unsigned integers of 32 bit) variables corresponding correspond to the T_{ON} period time lengths expressed in number of clock cycles. This conversion is required because the function of the device driver used to set the duty cycles needs their values expressed in ticks.

3.5 SPI Communication

The software components SPI Communication (SpiCom), composed by the files SpiCom.h and SpiCom.c, is responsible to manage SPI communication with the resolver-to-digital converter connected to the motor resolver. It makes use of an LPSPI peripheral that is the master of the communication. SPI Communication implements features related to the resolver-to-digital converter configuration and data acquisition. SPI communication is necessary for the device configuration, instead the acquisition of the converted data through SPI is a redundant mechanism due to the presence of a digital parallel output port.

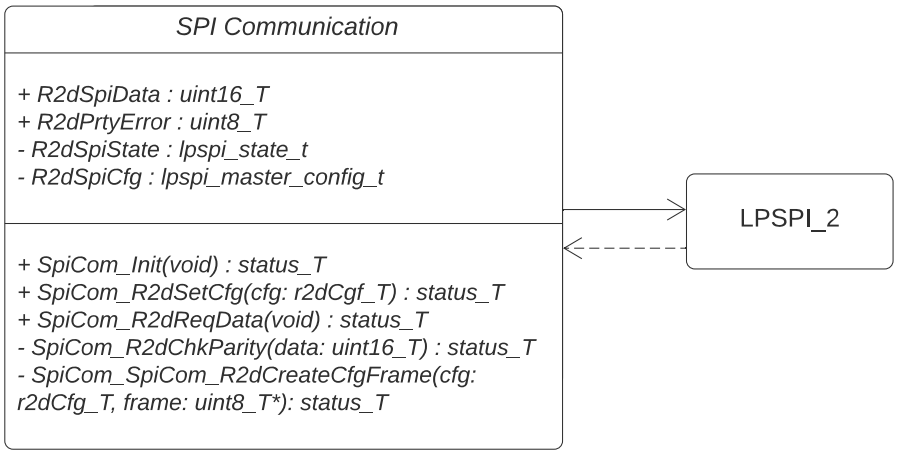

Figure 3.12: UML class diagram of the software component SPI Communication.

3.5.1 Requirements

The software component SPI Communication shall satisfy specification related to the configuration mechanism and data acquisition of the resolver-to-digital (R2D) converter done through its serial interface. As described in section 2.2.1, four signals are required to implement the SPI communication protocol with specific timing characteristics. For this reason, the timing diagrams of the serial input and output sequences were studied to define explicitly timing specifications.

The serial input sequence (i.e. input from the point of view of the resolver-to-digital converter and output for the MCU), used for device configuration, is reported in Fig. 3.13. As shown in the figure, the clock polarity is negative, and its phase positive. This means that the clock signal (SCK) is first high, and data (SSDT) bits are sampled by the R2D on the falling edge of each clock pulse. Each clock cycle shall last no less than 200ns that corresponds to a maximum bandwidth of 5MHz in the transmission. The chip select signal (SSCS) is active low and shall

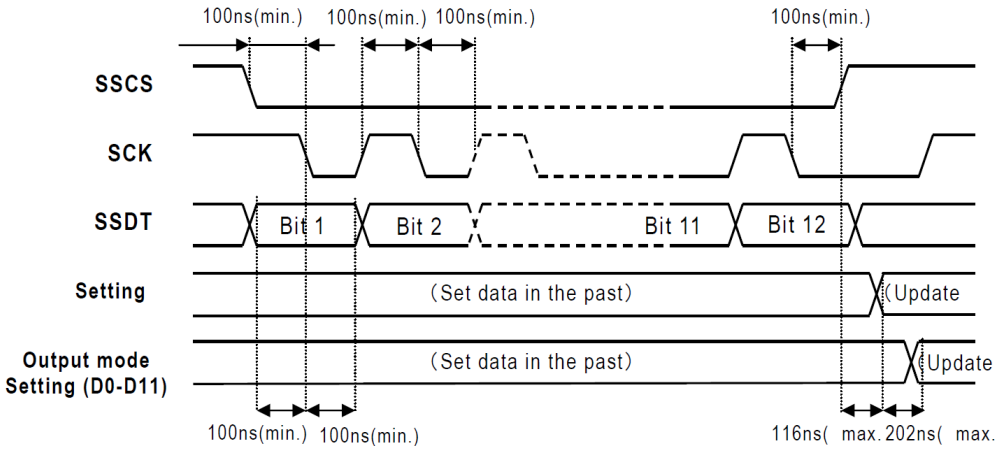

Figure 3.13: Timing diagram of the serial input sequence to configure the resolver-to-digital converter.

be activated at least 100ns before the falling edge of the first clock cycle. The configuration frame is constituted by 12 bits which are sent to the R2D starting from the LSB. The frame is divided in 7 fields summarized in Fig. 3.16. After last bit is sent, the SSCS signal shall be deactivated (i.e. driven high) and the clock signal shall remain low for at least 318ns. This time is necessary to let the device to update its settings.

The serial output sequence (i.e. output from the point of view of the resolver-to-digital converter and input for the MCU) is reported in Fig. 3.14. As can be seen

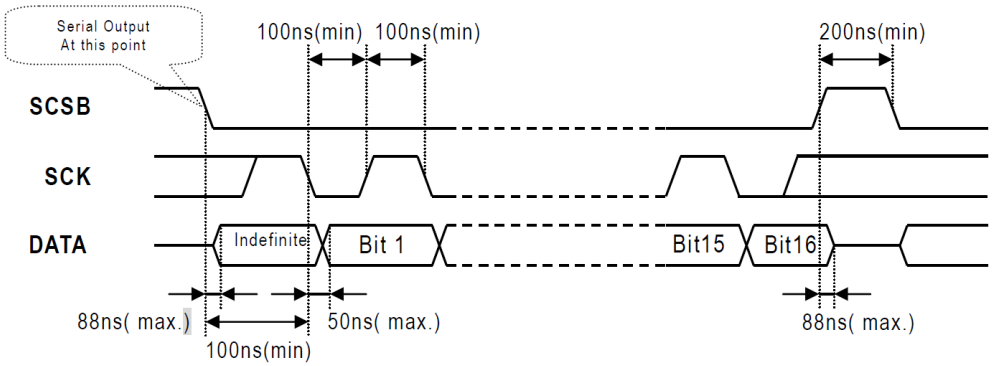

Figure 3.14: Timing diagram of the serial output sequence to acquire data from the resolver-to-digital converter.

in the figure, the characteristics of the clock signal are like the serial input sequence ones. In this case, the clock characteristics are the opposite - the clock polarity is negative, and its phase is positive that means that the clock signal (SCK) is first

high, and data (DATA) is sampled on the rising edge of each clock pulse by the LPSPI peripheral. The communication bandwidth is always 5MHz. The chip select signal (SCSB) is active low and shall be activated at least 100ns before the falling edge of the first clock cycle as in the previous case. The data frame is constituted by 16 bits divided in different fields depending on the selected mode for the R2D converter. The data are sent starting from the LSB. The SCSB signal shall be disactivated after the last clock cycle and a time period of at least 200ns shall last between two different activations. The overall timing specifications are listed in Tab. 3.5.

SPI Communication Timing Requirements			
Symbol	**Parameter**	**Min.**	**Max.**
t_{hSCK}	SPI clock high time	100ns	
t_{lSCK}	SPI clock low time	100ns	
$t_{setSCSB}$	Data Chip select setup time	100ns	
$t_{holSCSB}$	Data Chip select hold time	100ns	
$t_{disSCSB}$	Data Chip deselect time	200ns	
$t_{setSSCS}$	Setting Chip select setup time	100ns	
$t_{holSSCS}$	Setting Chip select hold time	100ns	
$t_{setSSDT}$	Setting input setup time	100ns	
$t_{holSSDT}$	Setting input hold time	100ns	

Table 3.5: Timing requirement of SPI communication between the MCU and the resolver-to-digital converter.

3.5.2 Implementation

The only hardware needed to implement SPI communication is constituted by a hardware instance of the peripherals LPSPI. The LPSPI2 was configured by means of the device drivers provided by the silicon-vendor. First, the peripheral was configured as master in the communication. Its bit-rate was set to 5 Mbits/s which corresponds to the maximum supported bandwidth of the R2D converter. Two different chip selects were used for the SSCS and SCSB signals which correspond respectively to the selection of configuration and data acquisition transmissions. The clock polarities and phases of the peripheral were set according to specifications of Tab. 3.5, and the data were acquired or sent starting from the LSB. The LPSPI was set to generate an interrupt every times a transmission ends. In practice, a data request or configuration sending is started by the application software. Then, the corresponding interrupt handler is executed every time the interrupt is asserted at

the end of the transmission. The interrupt handler runs a service routine necessary to manage the acquired data after an output sequence, and is skipped after a input sequence. Indeed, the interrupt is called the same in case of configuration transmission, but the no specific instructions are executed.

The structure of the data frame sent by the converter is reported in Fig. 3.15. As shown in the figure, the data frame presents different structures depending on the mode setting. For the proof of concept, the required mode is the Absolute Output Mode. When this mode is used, the R2D converter sends a frame constituted by 12

Serial output Mode setting [Bit 4,3]	"DATA" output Bit NO.															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Absolute output mode setting [00]	LSB ϕ12	ϕ11	ϕ10	ϕ9	ϕ8	ϕ7	ϕ6	ϕ5	ϕ4	ϕ3	ϕ2	MSB ϕ1	PRTY	0	0	PRTY2
Pulse output mode setting [01]	Encorder equivalent Pulse						–	ERR	ERR HLD	ERR CD1	ERR CD2	ERR CD3	PRTY	1	0	PRTY2
	A	B	Z	U	V	W										
Serial callback setting [10]	Serial setting register contents												PRTY	0	1	PRTY2
	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit10	Bit11	Bit12				
BIST result setting [11]	Default setting		BIST CD1	BIST CD2	BIST CD3	BIST CD4	BIST 実行中	VMD	ERR HLD	ERR CD1	ERR CD2	ERR CD3	PRTY	1	1	PRTY2
	Bit 1	Bit 2														
Absolute out 16Bit mode [—](Special)	LSB ϕ16	ϕ15	ϕ14	ϕ13	ϕ12	ϕ11	ϕ10	ϕ9	ϕ8	ϕ7	ϕ6	ϕ5	ϕ4	ϕ3	ϕ2	MSB ϕ1

Figure 3.15: Structure of the data transmitted through SPI by the resolver-to-digital converter.

bits representing the converted data, the bit PRTY, two bits that reports the mode setting bit code, and the bit PRTY2. The bits PRTY and PRTY2 are even parity bits which means that they are 1 in case of the number of 1s of the considered frame is odd, and 0 vice versa. The bit PRTY expresses the parity of the serial data bits Bit1∼12, and PRTY2 the serial data bits Bit1∼15. Therefore, a parity check is done at firmware-level every time a new value is acquired from the R2D through SPI interface. Parity check errors are signaled to the application software using a global variable.

The structure of the configuration frame is reported in Fig. 3.16. As can be seen in the figure, the frame don't require the insertion of parity or other redundant bits. The settings selected for the proof of concept are highlighted in yellow and they are not required to change at run-time. Indeed, the pre-selected settings are sent to the R2D during its initialization and they are not changed when the application software is running.

In order to estimate the delay between the request of a data from the R2D and the corresponding end of data acquisition was computed easily according to the

Bit NO.	Item	設		
1	Output Mode Setting (D0～D11)	[0] : Absolute-value (φ1～φ12) Parallel Angle Data [1] : Equivalent Encorder Pulse (A,B,Z,U,V,W)		
2	Selection of Exciting clock	[0] : Internal Oscillator [1] : External Clock Input		
3 4	Serial Output Mode Setting [Bit 4,3]	[00] : Absolute-value (φ1～φ12) Angle data [01] : Equivalent Encorder Pulse (A,B,Z,U,V,W) [10] : Serial Callback (Setting Register Confirmation) [11] : Failure Detection／BIST result		
5 6	Loop gain setting [Bit 6,5]		Loop gain setting group A [Bit 11]=[0]	Loop gain setting group B [Bit 11]=[1]
		[00]	Fixed value① (Bandwidth 800Hz(typ.))	Fixed value⑤ (Bandwidth 1,000Hz(typ.))
		[01]	Fixed value② (Bandwidth 2,000Hz(typ.))	Fixed value⑥ (Bandwidth 500Hz(typ.))
		[10]	Fixed value③ (Bandwidth 2,500Hz(typ.))	Fixed value⑦ (Bandwidth 200Hz(typ.))
		[11]	Fixed value④ (Bandwidth 1,500Hz(typ.))	Loop gain Auto tuning (Bandwidth 220Hz～460KHz (typ.) automatic adjustment)
7 8 9 10	BIST (Built In Self Test) Setting & Special mode setting [Bit 10,9,8,7]	[0000] : BISTVLD (Input) Invalid [0001] : Reserved (Do not use) [0010] : Reserved (Do not use) [0011] : Reserved (Do not use) [0100] : Reserved (Do not use) [0101] : Angle convert BIST:Angle-1 (0°) [0110] : Angle convert BIST:Angle-2 (45°) [0111] : Angle convert BIST:Angle-3 (270°) [1000] : Reserved (Do not use) [1001] : Failure detection BIST:resolver signal abnormality BIST [1010] : Failure detection BIST:Signal break BIST (COS side) [1011] : Failure detection BIST:Signal break BIST (SIN side) [1100] : Failure detection BIST:conversion abnormality BIST [1101] : System reset (Re-boot) [1110] : Serial Absolute-value Output 16 BIT Mode [1111] : Reserved (Do not use)		
11	Loop gain setting Group selection	[0] : Group A (＊Refer to Loop gain setting [Bit6,5]) [1] : Group B (＊Refer to Loop gain setting [Bit6,5])		
12	Threshhold selection for signal abnormality	[0] : 0.1×VCC [Vp-p] [1] : 0.14×VCC [Vp-p]		

Figure 3.16: Structure of the configuration frame transmitted through SPI to the resolver-to-digital converter.

following expression:

$$\begin{aligned}\Delta \mathrm{T}_{\mathrm{SPI}} &= \mathrm{n}_{\mathrm{bit}} \times f_{clk} + (t_{selSCSB} + t_{holdSCSB}) \\ &= 16 \times 200\mathrm{ns} + (100\mathrm{ns} + 88\mathrm{ns}) = 3.388\mu\mathrm{s}\ .\end{aligned}$$

This is negligible with respect to the application software execution rate. The corresponding calculation to estimate the delay to send the configuration frame

required by the R2D converter was not performed because no relevant timing constraints are present in the initialization phase.

3.5.3 API Layer

The API layer of the software component CanCom is composed by an initialization function used to initialize the peripheral LPSPI2, two functions to send the pre-selected configuration to the R2D converter and request the converted data by the device, and two global accessible variables to store the result of the data acquisition and signal any errors related to parity checks.

Global Data Structure

R2dSpiData

uint16_T R2dSpiData

This variable contains the value acquired from the R2D converter through SPI. This data is requested synchronously with respect to the analog acquisition of the motor phase currents.

R2dSpiPrtyError

uint8_T R2dSpiPrtyError

This variable is set to signal to the application software if an error occurs in computing the parity error of the acquired data through the SPI interface of the resolver-to-digital converter.

Functions

SpiCom_Init()

status_T SpiCom_Init(void)

This function initializes the LPSPI peripherals for the SPI communication with the the resolver-to-digital converter.
Parameters
None.
Returns
status_T - Status return code.

status_success	The LPSPI peripheral is set correctly.
status_spiInitError	An error occurs in initializing the LPSPI peripheral.

Notes
The operations performed during the initialization are the following:

1. Initialize the pins related to SPI communication protocol.
2. Configure the LPSPI peripheral as master in the communication according to the discussed requirements.

This function shall be called before to interact with the R2D converter.

SpiCom_R2dSetCfg()

status_T SpiCom_R2dSetCfg(r2dCfg_T cfg)

This function sends to the resolver-to-digital converter a configuration frame through SPI.
Parameters

in	r2dCfg	Resolver-to-digital converter configuration data structure

Returns
status_T - Status return code.

status_success	The configuration is sent correctly.
status_spiBusy	The configuration can't be sent at the moment because the LPSPI peripheral is busy.
status_spiFrameError	An error occurs in creating the configuration frame to be sent through LPSPI peripheral.
status_spiTxError	An error occurs in starting the LPSPI transmission.

Notes
This function shall be call before to start the application software and after the initialization of the software component SpiCom. A pre-selected configuration is defined as global static variable in SpiCom.c, but any allowed configuration can be send using this function.

SpiCom_R2dReqData()

status_T SpiCom_R2dReqData(void)

This function starts an SPI transmission to acquire a data from the resolver-to-digital converter.
Parameters
None.
Returns
status_T - Status return code.

status_success	The configuration is sent correctly.
status_spiBusy	The configuration can't be sent at the moment because the LPSPI peripheral is busy.
status_spiTxError	An error occurs in starting the LPSPI transmission.

Notes

After being received, the acquired data is moved in the global variable R2dSpiData only if the parity check is passed. This function shall be called synchronously with respect to the analog acquisition of the motor phase currents.

3.6 Digital Signal Management

The software components Digital Signal Management (DigSgnMgm), composed by the files DigSgnMgm.h and DigSgnMgm.c, is responsible to manage digital input and output signals. It makes use of the module SIUL2 to acquire digital input signals and drive digital output ones. Digital signals were required by the project to handle fault, reset, and enabling signals, and a digital parallel interface connected to the Resolver-to-Digital converter. This digital parallel interface was implemented with 13 digital input signals.

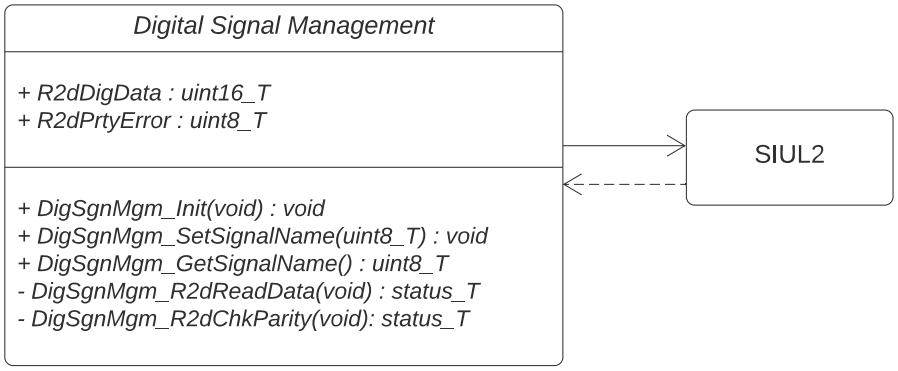

Figure 3.17: UML class diagram of the software component Digital Signal Management.

3.6.1 Requirements

The requirements of the software component Digital Signal Management are related and limited to the acquisition of the digital input signals and the driving of the digital output ones. Timing requirements are not present in the management of digital signals except for the digital parallel interface. The data coming from the resolver-to-digital converter through the parallel interface shall be acquired periodically and asynchronously with respect to the application tasks. They shall be sampled as near as possible to the time instants when the phase currents are sample by ADCs.

3.6.2 Implementation

The SIUL2 peripheral is managed writing and reading directly in the peripheral configuration registers because dedicated drivers are not provided by the silicon-vendor. The digital signals required by the application software are classified as Digital Input (Di) or Digital Output (Do) signals. The GPIO Pad Data Output

(GPDO) registers are used to set the desired output values of Do signals. The GPIO Pad Data Input (GPDI) registers are used to read the input values of Di signals.

The Di signals are read through polling mechanism when their values are required by the application software. Instead, the polling of the digital input signals that constitute the digital parallel interface is performed periodically and synchronously with respect to the analog acquisition of the motor phase currents.

The application software recognizes only active (1) and deactive (0) values for digital signals, so their electric polarity is managed in hardware as described in section 3.2. Indeed, the application values correspond to the digital signal states for positive polarities (active when high), and to their opposite for negative polarities (active when low). For example, a digital output signal with negative polarity shall be driven low when set as active by the application software. These signal polarities are taken into account in the corresponding setting and reading functions which are implemented as static inline functions (defined in DigSgnMgm.h) to increase the code efficiency and avoid a function call for only one statement. Their implementation is the following:

Listing 3.4: Example of setter and getter static inline functions used for digital signals.

```
/* Static inline function for setting a Do signal */
static inline void DigSgnMgm_SetSignalName(uint8_T value)
{
    SIUL2->GPDO193 = (uint32_T)value;
}

/* Static inline function for getting a Di signal */
static inline uint8_T DigSgnMgm_GetSignalName()
{
    return (uint8_T)(SIUL2->GPDI229);
}
```

The digital parallel interface, connected to the Resolver-to-Digital converter, is constituted by 12 data bit and 1 parity bit. Before to read the data coming from the Resolver-to-Digital Converter, two signal shall be asserted with the correct timings. More precisely, the chip select and the read pin of the R2D converter are asserted one after the other to freeze the digital state of the parallel output port. At that time, each signal of the interface is read and a their value are stored in an uint16_T variable. Then, a parity check is performed to verify the correctness of the acquisition, and if passed, the acquired value is moved to a global variable

accessible by the application software.

3.6.3 API Layer

The API layer is constituted by a variable which stores the data acquired from the resolver-to-digital converter through the digital parallel interface, a variable to signal if a parity error occurred in its acquisition, and the get and set functions for the digital input and output signals. The function used to acquire the data from the parallel interface of the resolver-to-digital converter was not inserted in the API of the software component because it is not called directly by the application software. Indeed, it is called by the firmware itself to acquire periodically the data converted by the R2D that the application software can read through the global variable R2dDigData used as interface.

Global Data Structure

R2dDigData

uint16_T R2dDigData_rad

This variable contains the data acquired through the digital parallel interface of the resolver-to-digital converter.

R2dPrtyError

uint8_T R2dPrtyError

This variable is set to signal to the application software if an error occurs in computing the parity error of the acquired data through the digital parallel interface of the resolver-to-digital converter.

Functions

DigSgnMgm_Init()

void DigSgnMgm_Init(void)

This function initializes the GPIO pins for the digital signal acquisition and driving.
Parameters
None.
Returns
None.
Notes
This function shall be called before to read or drive digital pins. Only the pin initialization is performed to initialize the software component.

DigSgnMgm_SetSignalName()

static inline void DigSgnMgm_SetSignalName(uint8_T value)

This function sets the output value of a digital output pin considering the configured signal polarity.

Parameters

in	value	Value to be set to the digital output pin.

Returns

None.

Notes

This function is only an example of set function. In the software component, a function for each digital output signal was developed, but they are not reported for NDA policies.

DigSgnMgm_GetSignalName()

static inline uint8_T DigSgnMgm_GetSignalName(void)

This function reads the input value of a digital output pin considering the configured signal polarity.

Parameters

None.

Returns

uint8_T - Value of the input pin considering the configured signal polarity.

Notes

This function is only an example of get function. In the software component, a function for each digital input signal was developed, but they are not reported for NDA policies.

3.7 CAN Communication

The software component CAN Communication (CanCom), composed by the files CanCom.h and CanCom.c, is responsible to manage the CAN bus node constituted by the MCU. It utilizes a peripheral FlexCAN-FD to send and receive messages with other nodes in the network, and a Periodic Interrupt Timer (PIT) to send messages periodically. The CAN communication was required as communication protocol to interface to the MCU externally from the test bench which is the only other CAN node connected to the bus besides the microcontroller.

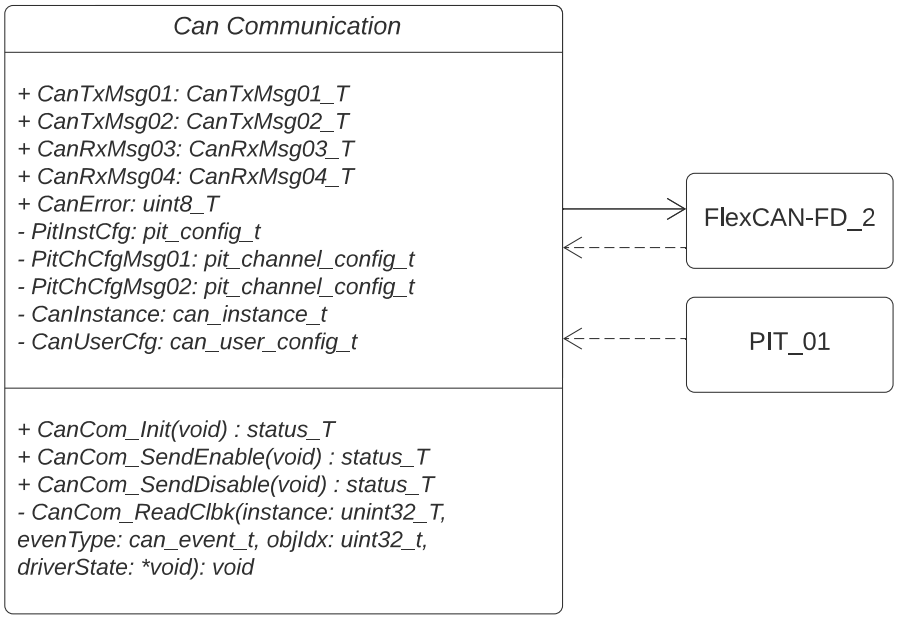

Figure 3.18: UML class diagram of the software component CAN Communication.

3.7.1 Requirements

The CAN Communication software component shall be developed to receive 2 messages transmitted by the test bench through CAN bus. These messages contain information about the operation that the system shall perform. In such a way, an external operator can interact with the plant placed at a safe distance without acting directly on the system components. On the other side, the MCU shall send periodically messages containing information about its status to the test bench through CAN bus to allow a correct plant monitoring. The CAN database was defined on the needs of the project. The characteristics of the CAN messages refer to the CAN 2.0A message frame standard with 11 bits for the identifier field (more

details about CAN are discussed in section 2.2.2). For the proof of concept, no error messages or remote frames were considered.

The firmware shall provide a message of error (CanError) when no messages are received from the workbench for a time interval bigger than a given threshold. This is required because physical faults in the transmission lines can occur due to vibrations or accidents, and the system shall be able to detect the interruption in the transmission. Since the peripheral FlexCAN-FD provides digital logic values to its outputs, an external transceiver shall be connected to the peripheral output pins as described in section 2.2.2.

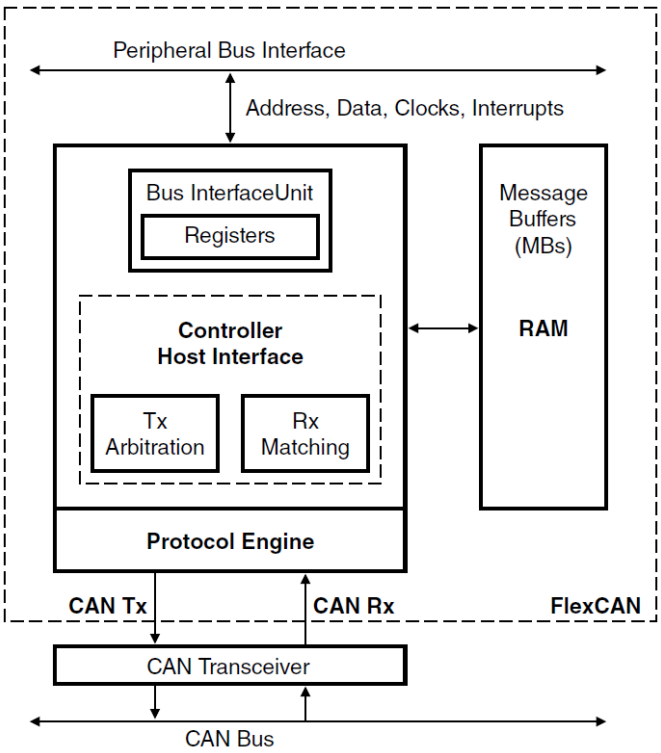

Figure 3.19: Logic diagram of a FlexCAN-FD peripheral connected to CAN bus.

3.7.2 Implementation

The device drivers provided by the silicon-vendor were used to configure the CAN peripheral according to system specifications. The peripheral FleCAN-FD is set in normal mode. When this mode is selected, the module operates receiving and/or transmitting message frames, managing errors normally, and enabling all features provided by the CAN protocol. The peripheral provides also a feature to implement flexible bit-rate, but this one is neglected for the project purposes.

In order to set the required bit rate and the corresponding bit segmentation, the register CTRL1 of the peripheral FlexCAN-FD was analyzed. Its structure is reported in Fig. 3.20 and the descriptions of the bit fields set for the purpose are listed in the following:

- **PRESDIV**
  The Prescaler Division Factor is an 8-bit field and defines the ratio between the peripheral clock frequency and the inverse of the period of a time quantum of the CAN protocol.

- **RJW**
  The Resync Jump Width is a 2-bit field and defines the maximum number of time quanta plus one that a bit time can be changed by one resynchronization. The valid programmable values are between 0 and 3.

- **PROPSEG**
  The Propagation Segment is a 3-bit field and defines the length of the propagation segment in the bit time. The valid programmable values are between 0 and 7.

- **PSEG1**
  The Phase Segment 1 is a 3-bit field and defines the length of phase segment 1 (number of time quanta plus one) in the bit time. The valid programmable values are between 0 and 7.

- **PSEG2**
  The Phase Segment 2 is a 3-bit field and defines the length of phase segment 2 (number of time quanta plus one) in the bit time. The valid programmable values are between 0 and 7.

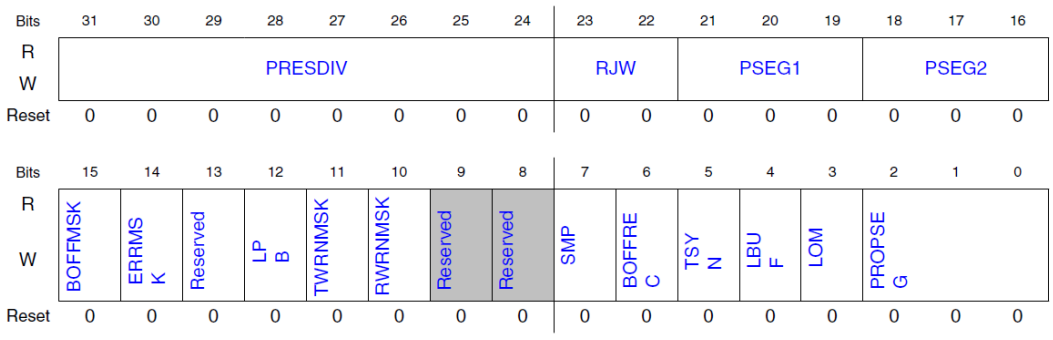

Figure 3.20: Structure of the register CTRL1 of the peripheral FlexCAN-FD.

CAN Bit Segmentation settings		
Bit Field	**Description**	**Value**
PRESDIV	Prescaler Division Factor	2+1
PROPSEG	Propagation Segment	7+1
PSEG1	Phase Segment 1	3+1
PSEG2	Phase Segment 1	2+1
RJW	Resync Jump Width	1+1

Table 3.6: Parameters of the peripheral FlexCAN-FD used to configure the required time segmentation and bit rate of the CAN node represented by the MCU.

The desired bit rate (1Mbps) was obtained using 16 time quanta with a duration of $6.25\mu s$. This was implemented setting the values reported in Tab. 3.6. The sample point placed between the two phase segments, was set at the 83,33% of the bit.

Since the MCU shall send periodically CAN messages to the external test bench, as described in the previous section, two channel of the PIT1 hardware instance are used to trigger periodically interrupts. The corresponding interrupt handler associated to the timer peripheral is used to create a data frame for the data to be transmitted and send the messages with specific rates. The use of two channels is necessary to implement two different rates of 0.01s and 0.05s for the two messages.

The peripheral FlexCAN-FD makes use of Message Buffers (MB) to implement the message acquisition and transmission. The structure of a Message Buffers is reported in Fig. 3.21, and it is the same for RX and TX buffers. Each MB provides a register to configure or store message characteristics, a register to store the message ID and local priority, and 16 registers to store up to 64 bytes to be sent or received. Some of these parameters have a correspondence with the bits in the message frame. The description of the bit fields in the registers of Fig. 3.21 and their corresponding settings is described in the following:

- **Extended Data Length - EDL**
  EDL is a configuration bit to distinguish between CAN standard format and CAN FD format frames, and it is set to 0 to disable CAN FD format frames.

- **Bit Rate Switch - BRS**
  BRS is a configuration bit to define whether the bit rate is switched inside a CAN FD format frame. Since EDL is set to 0, this bit is ignored.

- **Error State Indicator - ESI**
  ESI is a status bit to indicate if the transmitting node is error active or error passive.

	31	30	29	28	27	24	23	22	21	20	19	18	17	16	15	8	7	0
0x0	EDL	BRS	ESI		CODE			SRR	IDE	RTR	DLC				TIME STAMP			
0x4	PRIO				ID (standard/extended)										ID (extended)			
0x8	Data byte 0						Data byte 1								Data byte 2		Data byte 3	
0xC	Data byte 4						Data byte 5								Data byte 6		Data byte 7	
0x10	Data byte 8						Data byte 9								Data byte 10		Data byte 11	
0x14	Data byte 12						Data byte 13								Data byte 14		Data byte 15	
0x18	Data byte 16						Data byte 17								Data byte 18		Data byte 19	
0x1C	Data byte 20						Data byte 21								Data byte 22		Data byte 23	
0x20	Data byte 24						Data byte 25								Data byte 26		Data byte 27	
0x24	Data byte 28						Data byte 29								Data byte 30		Data byte 31	
0x28	Data byte 32						Data byte 33								Data byte 34		Data byte 35	
0x2C	Data byte 36						Data byte 37								Data byte 38		Data byte 39	
0x30	Data byte 40						Data byte 41								Data byte 42		Data byte 43	
0x34	Data byte 44						Data byte 45								Data byte 46		Data byte 47	
0x38	Data byte 48						Data byte 49								Data byte 50		Data byte 51	
0x3C	Data byte 52						Data byte 53								Data byte 54		Data byte 55	
0x40	Data byte 56						Data byte 57								Data byte 58		Data byte 59	
0x44	Data byte 60						Data byte 61								Data byte 62		Data byte 63	
	= Unimplemented or reserved																	

Figure 3.21: Structure of a Message Buffer for the peripheral FlexCAN-FD.

- **Message Buffer Code - CODE**
  CODE bit field encodes the buffer status in case of RX buffers or message settings in case of TX buffers. For RX buffers, it indicates if the buffer is inactive, empty, full, busy, if an overrun occurred, or if a remote request has to be serviced. In case of TX buffers, it indicates if is inactive, if the transmitted frame is a data, remote request, remote response frame, or if the transmission is aborted.

- **Substitute Remote Request - SRR**
  SSR bit is used only in extended format, therefore it is ignore for this application.

- **ID Extended Bit - IDE**
  This field identifies whether the frame format is standard or extended, and it is set to 0 (frame is standard) in this application. This field was used to store the message ID of the message provided in the CAN database for the project.

- **Remote Transmission Request - RTR**
  RTR bit affects the behavior of remote frames and is part of the reception filter. It is set to 1 if remote frame has to be transmitted by a TX buffer or it is stored in a RX Buffer. Otherwise, it is set to 0. In the MCU configuration, it is always set to 0.

- **Length of Data in Bytes - DLC**
  DLC bit field represents the length (in bytes) of the RX or TX data. This is set according to the lengths of the message of the CAN database.

- **Free-Running Counter Time Stamp - TIME STAMP**
  This 16-bit field is a copy of the Free-Running Timer, captured for TX and RX frames at the time when the beginning of the ID field appears on the CAN bus. It is not used for this application.

- **Local priority - PRIO**
  PRIO bit field is used only when the TX buffer priority is enable, and it defines which TX buffer has to send its message first in case of concurrent message sending.

- **Frame Identifier - ID**
  The ID bit field store the message ID of TX and RX frames. In standard frame format, only the 11 most significant bits (28 to 18) are used for frame identification in both receiving and transmitting cases.

- **Data Field - DATA BYTE 0 to 63**
  Up to sixty four bytes can be used for a data frame, depending on the size of payload selected for the message buffers. They are used to read the data received in RX transmissions, and to set the data to be sent in TX transmissions.

A TX Buffer is configured for each message that shall be sent. The CAN messages to be sent are declared with appropriate data structures in the source code. In such a way, their IDs and lengths can be set a priori according to CAN database, and only message data frames needs to be set before to start the frame transmission.

RX Buffers are configured with a message ID filter. Therefore, they store only the frames of the recognized IDs. A RX buffer is configured for each message to be received present in the CAN database. A callback for each RX buffer is installed. They are executed in the interrupt handler asserted at the end of a RX transmission. These callbacks are used to transfer the data from the RX buffers which have received the message to the global variables used as interface with the application software.

The corresponding channels of the hardware timer PIT1, used to implement the periodic message sending, are also used to implement the signal CanError. They are used to decrease variables which act as count-down timers to set the signal CanError in case of their values reach 0. Therefore, in case of RX message reception, the corresponding counter variables is reset to its initial value. Otherwise, an error is reported through the variable CanError for the absence of the expected RX

message. In such a way, interruptions in the CAN transmission line are signaled to the application software.

3.7.3 API Layer

The API layer between the application software and software component CAN Communication is constituted by the variables that stores the data sent and received through CAN bus, and by functions to initialize, start, and stop the transmission over CAN bus.

Global Data Structures

- **CanError**

  uint8_T CanError

  This variable contains the CAN error signal which is asserted if no CAN messages have been received for a time interval greater than the double of the expected one.

- **CanTxDataMsg01**

  CanTxDataMsg01_T CanTxDataMsg01

  This data structure contains the variables that shall be sent by the MCU to the external test bench through CAN bus with the message 01. This data structure is updated by the application software with a rate of 0.001s and read by the software component CanCom before to send the corresponding message. The field of the data structure are represented by real32_T (float) variables and are not reported for NDA policies. They are opportunely converted in uint16_T before to be sent over CAN bus. The data contained in CanTxDataMsg01 are sent by MCU to the external test bench every 0.05s.

- **CanTxDataMsg02**

  CanTxDataMsg02_T CanTxDataMsg02

  This data structure contains the variables that shall be sent by the MCU to the external test bench through CAN bus with the message 02. This data structure is updated by the application software every 0.001s and read by the software component CanCom before to send the corresponding message. The field of the data structure are represented by real32_T (float) variables and are not reported for NDA policies. They are opportunely converted in uint16_T before to be sent over CAN bus. The data contained in CanTxDataMsg02 are sent by MCU to the external test bench every 0.01s.

- **CanRxDataMsg03**

  CanTxDataMsg03_T CanTxDataMsg03

  This data structure contains the variables that shall be received by the MCU from the external test bench through CAN bus with the message 03. This data structure is updated by the software component CanCom when a message 03 is received (nominal transmission rate is 0.01s) and read by the application software every 0.001s. The fields of the data structure are represented by real32_T (float) variables and are not reported for NDA policies. They are opportunely converted from uint16_T to real32_T or uint32_T data types before to be stored in the data structure.

- **CanRxDataMsg04**

  CanTxDataMsg04_T CanTxDataMsg04

  This data structure contains the variables that shall be received by the MCU from the external test bench through CAN bus with the message 04. This data structure is updated by the software component CanCom when a message 04 is received (nominal transmission rate is 0.05s) and read by the application software every 0.001s. The fields of the data structure are represented by real32_T (float) variables and are not reported for NDA policies. They are opportunely converted from uint16_T to real32_T or uint32_T data types before to be stored in the data structure.

Functions

CanCom_Init()

status_T CanCom_Init(void)

This function initializes the FlexCan-FD peripheral to allow CAN communication with the external test bench using the in-board CAN transceiver.
Parameters
None.
Returns
status_T - Status return code.

status_success	The FlexCAN-FD and PIT peripherals are set correctly.
status_canInitError	An error occurs in initializing the FlexCAN-FD or PIT peripherals.

Notes
The operations performed during the initialization are the following:

1. Initialize the pins related to CAN Management.

2. Configure the TX buffers for the CAN messages 01 and 02.
3. Configure the RX buffers for the CAN messages 03 and 04.
4. Install the callbacks for the reception of the RX messages.
5. Configure the channels of the PIT1 to periodically send TX messages.

After the initialization, the function CanCom_SendEnable() shall be called to start to send periodically TX messages according to the CAN database.

CanCom_SendEnable()

void CanCom_SendEnable(void)

This function starts the channels of PIT1 to send periodically TX messages according to the CAN database.
Parameters
None.
Returns
None.
Notes
This function shall be called after the initialization of the software component CAN Management to start sending messages.

CanCom_SendDisable()

void CanCom_SendDisable(void)

This function stops the channels of PIT1 which are used to send periodically TX messages according to the CAN database.
Parameters
None.
Returns
None.
Notes
This function is called at the end of initialization of the software component CAN Management to prevent sending messages before that the application software is started.

Chapter 4

Firmware Verification and Testing

In software project management, software testing, and software engineering, verification and validation (V&V) is the process of checking that a software system meets specifications and that it fulfills its intended purpose. It may also be referred to as software quality control [6]. As already discussed, the firmware can be considered as a specific kind of software that provides the low-level control for a device's specific hardware. Verification and validation are independent procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfills its intended purpose. More precisely, verification is the process to evaluate of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Differently, validation is the process to guarantee that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers [6].

In the context of this thesis, the firmware testing aimed mainly to verification procedures rather than validation ones. As already said, the final goal of the project of this thesis is to develop a proof of concept for an electric powertrain which, as the name suggests, doesn't need to be validate as product. In fact, the project goal is to proof that the realization of the system that has been designed is achievable. Therefore, the purposes of the tests discussed in this chapter are to verify that the firmware configures the embedded hardware to interact correctly to the other subsystems connected to the MCU, and provides the necessary features required by the application software. In this chapter, the tests, performed in the verification phase, are discussed in the following sections. Section 4.2 is dedicated to the description of the laboratory instrumentation used to perform the MCU debug and trace and the measurements of the involved electrical quantities. The

other ones provide the description of correlated tests performed on the specific software components or hardware peripherals. Each of these sections is structured to describe the test purposes and the requirements to be checked, the developed test programs to execute, the specific test procedure to be performed, the expected results, and the analysis of the results obtained by the test. It is important to underline that no specific tests were developed for the software component Pin Management because its functions are tested one at a time during the testing of other software components.

4.1 Structure of Test Procedures

The structure of the procedure performed for each test is described by the sequence diagram of Fig. 4.1. As can be seen in the figure, the laboratory instrumentation is set up before to interact with the MCU. Then, the test program is downloaded from a version control system (VCS), also know as revision control or source control system A VCS is a software utility that tracks and manages changes to firmware source code, and provides a secure backup in case of data loss or corruption. The test program is written in the MCU flash memory by means of the specific Lauterbach hardware and software tools (described in detail in section 4.2.1). After that, the test program is started, the electrical signals are measured with the laboratory instrumentation such as oscilloscopes, multimeters, or logic analyzers and the MCU state checked by means of JTAG or ETM interfaces. The acquired results are compared with the expected ones - if they are consistent with each

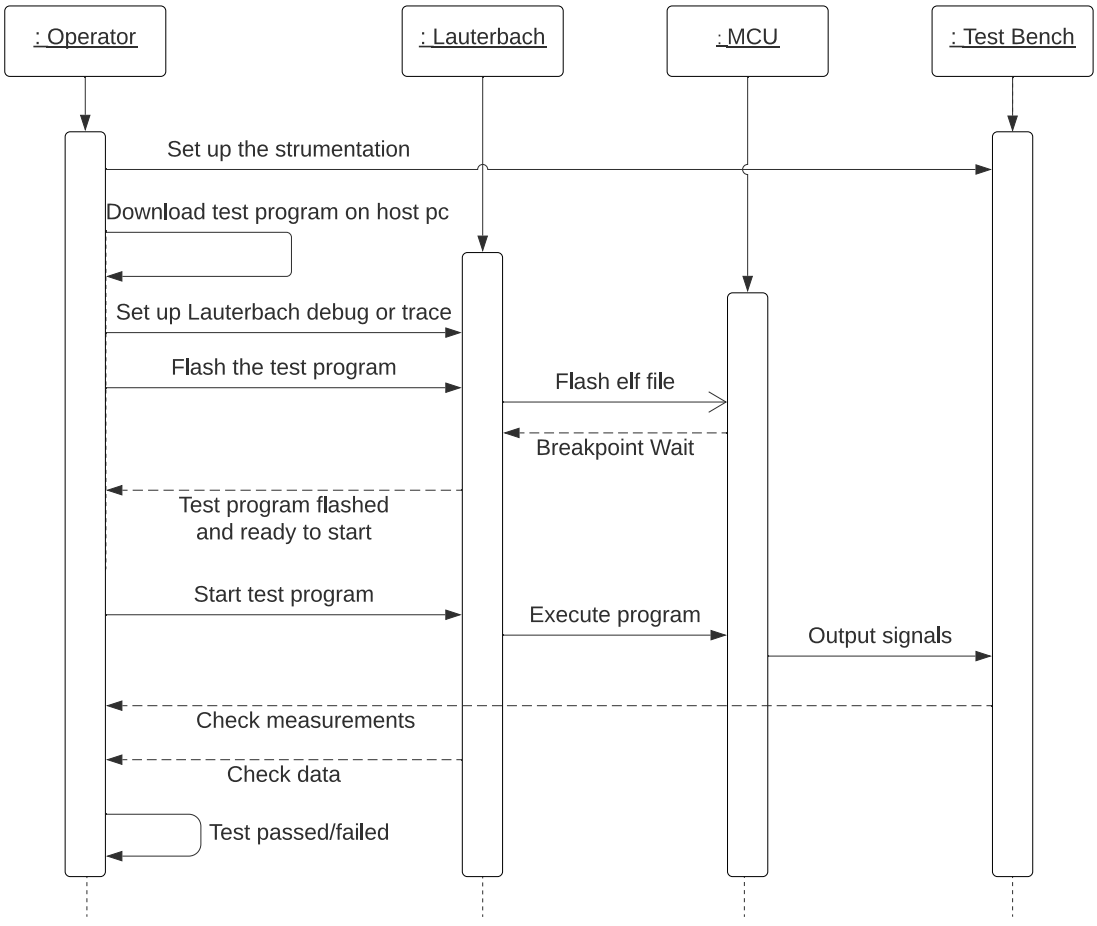

Figure 4.1: Sequence diagram of the structure of test procedures.

other, the test is passed, otherwise it is failed meaning that one or more bugs are present in the source code.

4.1.1 Initialization and Startup of Software Components

The test programs, described in the following sections, were developed using the same sequence of statements to initialize the target platform and the software component to be tested. The initialization sequence is described by the following steps:

1. Disable the global interrupts with the function INT_SYS_DisableIRQGlobal() provided by the MCU driver;
2. Configure the MCU clock signals and peripheral clock gating with the function ClkMgm_Init() of Clock Management;
3. Initialize the software component to be tested with its initialization function;
4. Enable the global interrupts the the function INT_SYS_EnableIRQGlobal() provided by the MCU driver;
5. Perform the startup sequences for the target platform correlated to the software component to be tested;
6. Configure other peripherals or execute other functions required for the specific test program.

As can be noticed, the interrupts are disable before to start the initialization sequence. Then, the clock signals and the peripheral clock gating are configured, and the software component to be tested initialized. Before to execute any startup sequence, the interrupts are re-enabled. An example of the code used to implement the described initialization sequence is the following:

Listing 4.1: Structure of the initialization and startup sequence of a generic test program.

```
int main(void){

    status_T l_err = status_success;

    /* Disable global interrupts */
    INT_SYS_DisableIRQGlobal();
```

```
    /* Clock initialization */
    l_err |= ClkMgm_Init();

    /* SWC initialization */
    l_err |= SwcName_Init();

    /* Initialization check */
    while(l_err != status_success){}

    /* Enable global interrupts */
    INT_SYS_EnableIRQGlobal();

    /* Startup procedures (Optional) */
    SwcName_Startup();

    /* Other operations required by the test program */
    ..................
}
```

This initialization sequence allows to correctly set up the target platform before to start the test program and test the initialization functions of the software component to be tested. In detail, the MCU is blocked inside an infinite loop in case of any error occurs during the initialization. Indeed, the test program is not started in case of initialization failure.

As already explained, since the clock signals and clock gating configuration shall be performed before to initialize any software component, Clock Management features were the first to be tested.

4.1.2 Test Program Flow and Target Platform Interaction

During the development of the tests, it was necessary to find a mechanism to allow the user to interact with the execution of the test programs and with the target platform. The solution found relies in the use of global variables which can be modified through the Lauterbach debugging tools. More precisely, the execution flow of the tests programs was controlled using these global variables as conditional expressions in conditional instructions. An example is reported in the following:

Listing 4.2: Example of execution flow control through a global variable.

```
    switch (CtrlFlow)
    {
```

```
        case CtrlFlow_Value1:
            /* Do something */
            break;
        case CtrlFlow_Value1:
            /* Do something */
            break;
        /* Other cases */
        default:
            break;
    }
```

As can be seen in the example, the program execution flow is affected by the value of the variable CrtlFlow. Since the variable is defined as global, the user is capable to modify it with the software Trace32 (described in section 4.2.1) and control the program flow without stopping its execution. This mechanism allows to implement more than one test for each test program.

In some cases, the global variables used to control the execution flow of the test programs work as "virtual buttons". Indeed, as commonly known, a button is a physical device that can assume ON or OFF states and is used by the user to interact with a system and and affect its behaviour. In the same way, a virtual button allows the user to interact with the system. An example is reported in the following:

Listing 4.3: Example of virtual button implementation.

```
    if (isEnabled)
    {
        /* Do something */
    }
    else
    {
        /* Do something else */
    }
```

As can be seen in the reported example, the variable isEnable can assume two significant values affecting the execution flow - the first block of code is executed if it assumes any values different from zero, the second one otherwise. Indeed, the behaviour implemented is the same of the pushing or releasing of a physical button.

In some test programs, global variables were used to interact with the target platform behaviour. In such cases, these global variables were used as arguments for the API functions called by the test programs. These variables work as the

ones used by the application software to interact with the firmware. An example is the setting of the PWM duty cycle with the corresponding function (described in detail in section 4.4.1). Since the variables used to define the duty cycles to be set are defined as global, the user is capable to interact with the target platform to perform the required test.

In order to test all the features implemented in the firmware, the approach used was to develop one test program for each software component implementing all the required tests. Then, global variables were inserted to control the program execution and interact with the target platform as described in this section.

4.2 Laboratory Instrumentation

The test procedures designed to verify the correct behaviour of the features implemented in the firmware required mainly five electronics laboratory instruments: a debug and trace hardware tool, a power supply, an oscilloscope, a signal generator, and a CAN to USB adapter.

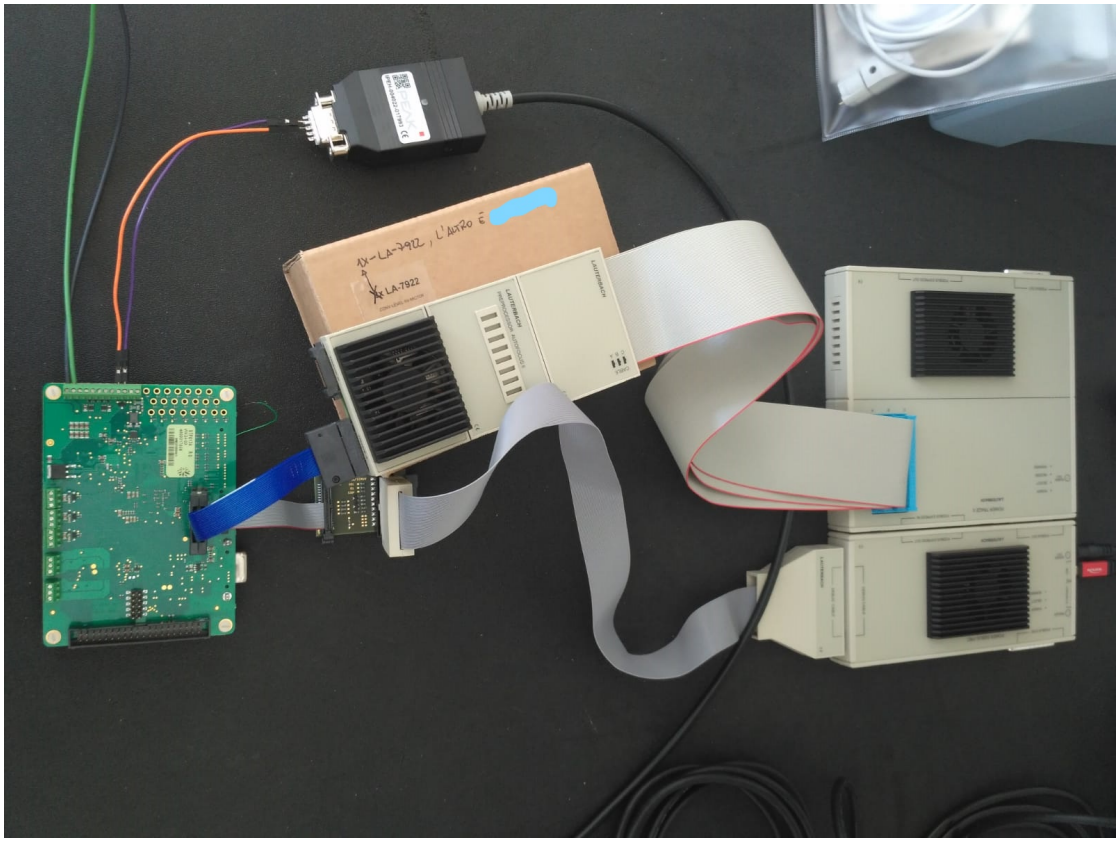

Figure 4.2: Example of laboratory instrumentation setup adopted during the tests of the software component CanMgm.

The power supply was used to supply the MCU. The debug and trace hardware tool (described in section 4.2.1) used during the firmware development was provided by Lauterbach company. It was utilized to download the test programs into the MCU flash memory and observe the MCU internal state and execution timings. The Lauterbach practice scripts used to properly pre-configures the MCU and download the test programs in its non-volatile memory will not be discussed in this thesis due to non-disclosure policies. The oscilloscope was an fundamental to observe the MCU analog and digital output signals. Moreover, the used oscilloscope (described in section 4.2.2) is provided with serial decoding features which where used to

analyze SPI and CAN data frames sent over their communication buses. The signal generator was used to stimulate properly the MCU input pins providing signals to be converted, acquired, and processed by the MCU. The used CAN interface for USB device was the Peak PCAN-USB (described in section 4.2.3). It was fundamental to analyze the messages sent over the CAN network and simulate the behaviour of the test bench to which the MCU will communicate during the tests of the overall proof of concept tests. An example of the setup used in laboratory is represented in Fig. 4.2.

4.2.1 Lauterbach Debug and Trace

The Lauterbach Trace32 tools are designed around common universal and architecture independent hardware modules. The tools play a very important role in the development of embedded systems and are used in the following ways:

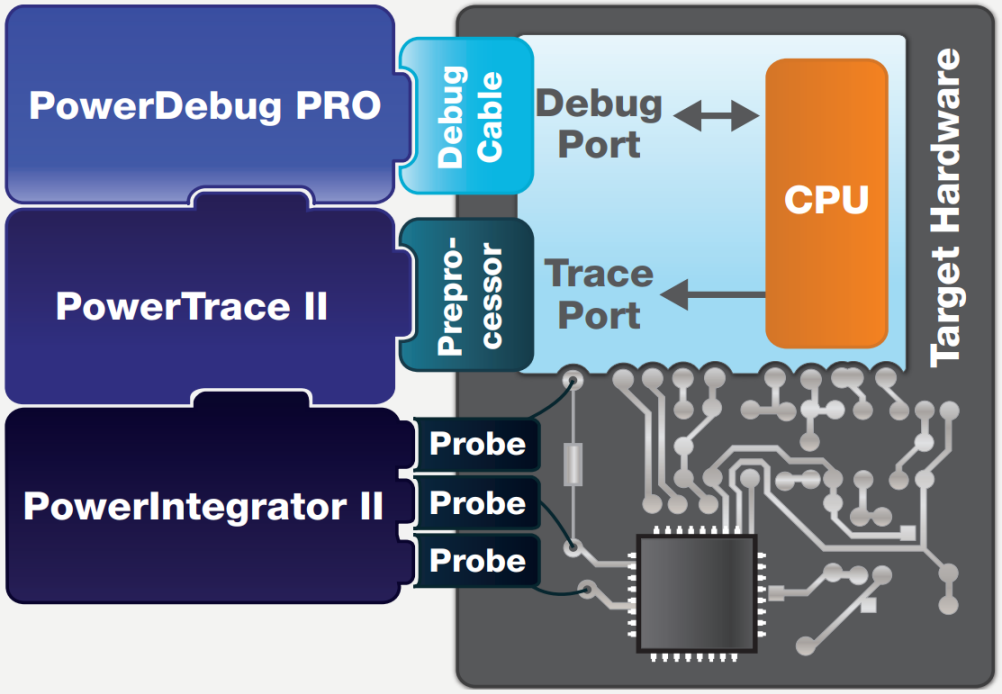

Figure 4.3: Complete structure of the Lauterbach hardware tools with the corresponding connection to the target platform [7].

- **Debug**
  Most cores for the embedded market provide access to on-chip debug features via a debug port. Trace32 tools connect to this to control the core, access the data being processed by the core and provide developers with debugging over

the embedded device: start, stop, step control; reading and writing memory and registers; setting breakpoints; tracking values of variables and so on. This means developers can diagnose software failures and memory corruption issues and correct the system to make it perform as expected.

- **Debug and Trace**
  In many applications it is no longer enough to run a simple test on your code. In markets such as automotive, medical, aerospace and defence, it is increasingly necessary to prove how the code behaved under all possible conditions in real-time. This requires the tools to record the program flow information from the core via the integrated trace port of the processor. Both long-term and high-speed trace options are supported.

- **Debug, Trace and Logic Analyzer**
  A debug system can be extended by adding a trace module or a logic analyzer. In some cases, both can be added to provide a very capable hard and software debug solution. Such a system can provide signal trace for logic analysis and protocol analysis as well as correlating power usage to the code operation.

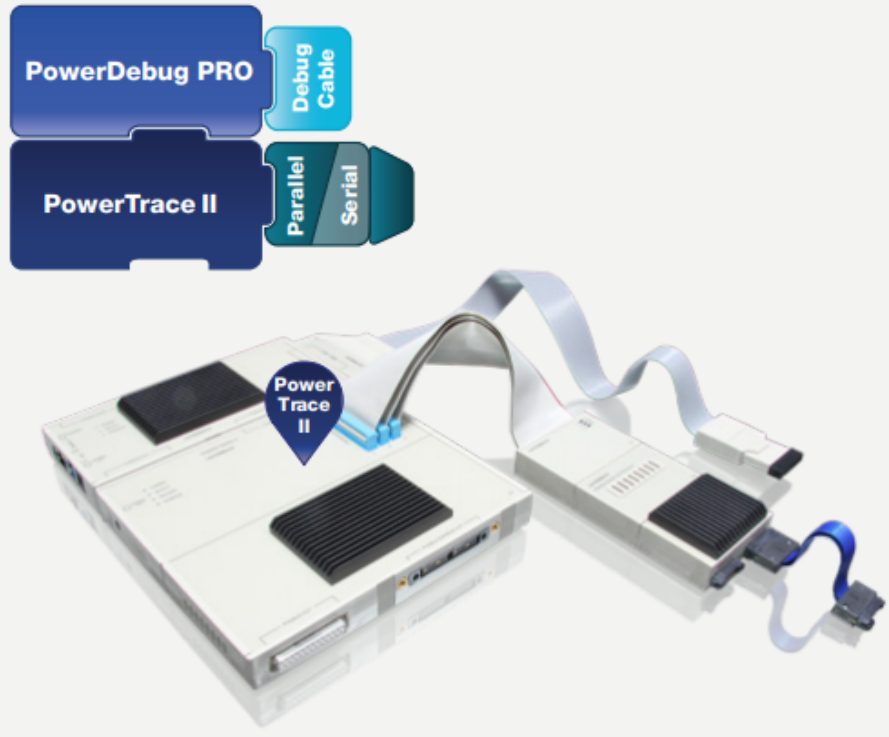

Figure 4.4: Lauterbach Debug and Trace hardware tools structure used for firmware testing [7].

For the purposes of the firmware developed for the project of this thesis, the Debug and Trace solution represented in Fig. 4.4 was adopted.

As debug information is provided by the on-chip debug interface, the PowerDebug hardware module makes possible to test and analyze every aspect of the target operation including the bootstrap code, the target initialization, the interrupts, the drivers and the kernel. The Debug Cable represents the physical architecture-specific interface between the target debug port ad the PowerDebug module. On the other hand, the PowerTrace hardware module Real-time trace provides fast and systematic troubleshooting capabilities to detect complex errors that only occur under run-time conditions. The recorded and time-stamped program/data flow allows an overall analysis of the system performance as well as quality assurance features such as code coverage and cache analysis. Fast trace evaluation and analysis are guaranteed through advanced compression technologies and speed-optimized system software. With up to 4GB trace memory and the possibility of streaming to the host computer, a large amount of program and data flow information can be traced. The Preprocessor adapter connects the real time trace module to the standard trace port connector as defined by the silicon manufacturer. The AutoFocus feature allows a preprocessor's self-calibrating hardware to ensure signal integrity up to 600 Mbps per channel for parallel trace ports.

4.2.2 Oscilloscope

As commonly known, an oscilloscope is a laboratory instrument used to display and analyze the waveform of electronic signals by means of a graph of the instantaneous signal voltage as function of time. The oscilloscope utilized in laboratory during testing phase was the Yokogawa DLM3054 represented in Fig. 4.5.

The Yokogawa DLM3054 is a mixed signal oscilloscope designed for automotive, mechatronics, and power supply application like hybrid and electric vehicles, in-vehicle serial buses, motors and drives, and office appliances. The key features of the oscilloscope are the following:

- Bandwidth: 500MHz;
- Number of Analog Channels: 4;
- Maximum Sampling Rate: 2.5 GS/s;
- Maximum Record Length: 125 Mpoints;
- Max. Input Voltage: $300V_{rms}/400V_{peak}$ (1MΩ) or $5V_{rms}/10V_{peak}$ (50Ω);
- Voltage Axis Sensitivity: 500μV/div to 10V/div (1MΩ) or 500μV/div to 1V/div (50Ω);

- A/D Resolution: 8 bit (25 LSB/div) Max. 12 bit (in High Resolution mode);
- Display: 8.4-inch TFT color LCD, 1024 × 768;
- SSD Hard Disk: 60 GB;
- Internal Storage R/W Speed: 50 MB/s.

Figure 4.5: The oscilloscope Yokogawa DLM3054.

The described oscilloscope integrates a lot of features and functions to match the modern needs in today's mechatronics applications. Among all of them, the serial decoding was extensively used during the tests related to SPI and CAN communication. Developing and verifying these kinds of communication requires analog physical-layer verification of waveform quality, noise, and simultaneous measurement of logical-layer on the communication bus. These serial bus decode functions can display decoded bus data and physical layer waveforms simultaneously to verify that communication works as expected according to system requirements.

4.2.3 Peak PCAN-USB

The CAN FD adapter PCAN-USB FD allows the connection of CAN FD and CAN networks to a computer via USB. A galvanic isolation of up to 500 Volts decouples

the PC from the CAN bus. A picture of the device is reported in Fig. 4.6. Some of its key features are listed in the following:

- CAN bus connection via D-Sub, 9-pin (in accordance with CiA 303-1);
- Complies with CAN specifications 2.0 A/B and FD;
- CAN bit rates from 25 kbit/s up to 1 Mbit/s;
- Time stamp resolution of 1 μs;
- Galvanic isolation up to 500 V;
- CAN termination can be activated through a solder jumper;
- Measurement of bus load including error frames and overload frames on the physical bus;
- Voltage supply via USB.

Figure 4.6: The PCAN-USB FD.

The monitor software PCAN-View was used to interface with the PCAN-USB FD module via PC. More precisely, the software was used to analyze the correct behaviour of the nodes connected to the CAN bus and simulate the role of the test bench which shall periodically send and receive messages. Moreover, when nominal operating conditions were simulated, the PCAN-USB FD allowed also to measure the bus load.

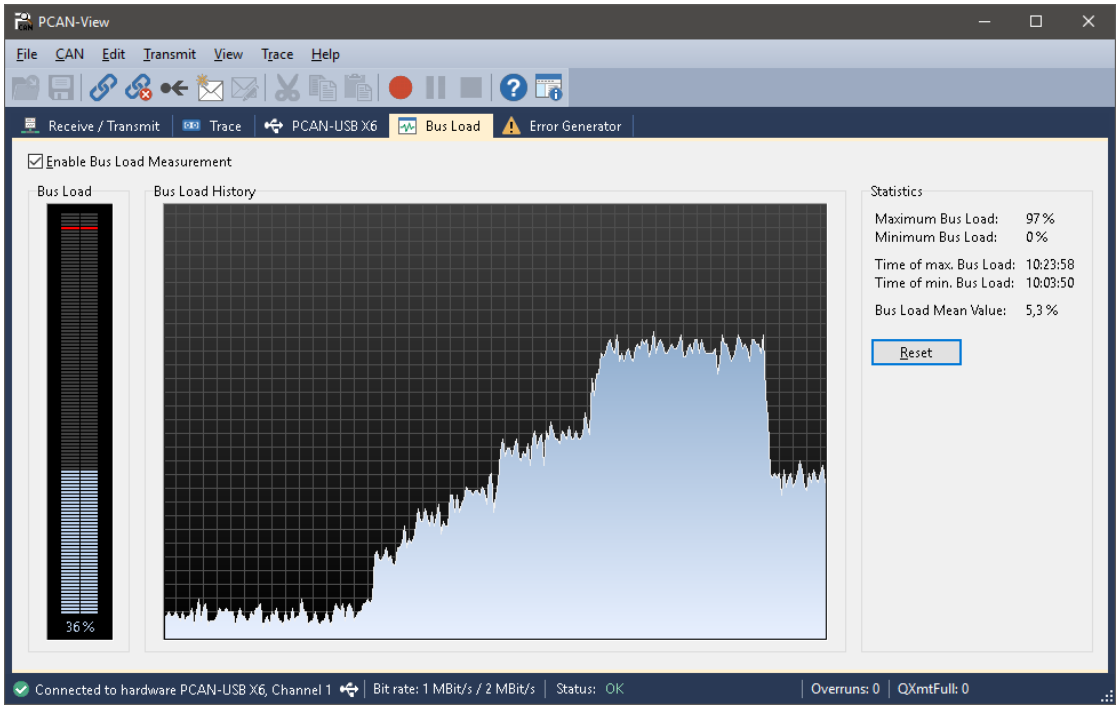

Figure 4.7: An example of the bus load window of the software PCAN-View.

4.2.4 Signal Generator

A signal generator is piece of test equipment that produces an electrical signal in the form of a wave with set properties of amplitude, frequency, and shape. This is used as a stimulus for the item being tested. The signal generator utilized in laboratory to perform the designed test procedures is the Tektronix AFG1022 represented in Fig. 4.8.

The signal generator Tektronix AFG1022 includes two channels, up to 60MHz bandwidth and up to $10V_{pp}$ output amplitude. The 3.95" TFT LCD, short-cut buttons, USB interface and PC software are provided to configure the instrument. The key features of the signal generator are the following:

- Two channels;
- Bandwidth: 25MHz sine waveforms, 12.5MHz square waveforms;
- Resolution: 14 bits, 125MS/s arbitrary waveforms;
- Record Length: 8k points;
- Amplitude: $1mV_{pp}$ to $10V_{pp}$ into 50Ω loads;

- Display: LCD TFT 3.95”.

Figure 4.8: The signal generator Tektronix AFG1022.

The described signal generator is designed to be used in application related to electric and electronics experiments, communications experiments, and sensor simulation. In the project of this thesis, it was used to stimulate the target platform testing the analog and digital acquisition.

4.2.5 Power Supply

A power supply is an electronics laboratory instruments used to supply in a controllable manner the target electronic equipment. The power supply utilized in laboratory to supply the target platform is the GWInstex GPS4303 represented in Fig. 4.9.

The signal generator GWInstex GPS4303 has 4 channels and can provide up to 200W output of linear DC power supplies. It includes overload and reverse polarity protection, and an output on/off switch keep their load safe from unexpected conditions. The key features of the signal generator are the following:

- 4 independent isolated outputs;
- 0.01% load and line regulation with low ripple and noise;
- Output ON/OFF switch;
- Supply voltage: up to 30V (60V in tracking series voltage);

- Supply current: up to 3A (6A in tracking parallel current);
- Over load and reverse polarity protection.

Figure 4.9: The power supply GWInstex GPS4303.

Furthermore, a high regulation (0.01%+3mV) and low ripple/noise ($< 1\text{mV}_{\text{rms}}$, 5Hz 1MHz) are maintained for channel 1 and 2 in constant voltage mode. Automated cooling fan speed control minimizes fan noise according to load conditions, ensuring quiet operation. As already said, the power supply was used as power source for the target platform.

4.3 Clock Management Testing

The software component Clock Management was tested to verify the correct configuration of the MCU clock signals. As explained in section 3.1, their configuration is performed by means of the function ClkMgm_Init(), which is also responsible to manage the peripheral clock gating used for the implementation of the features required by the application software. Therefore, the presence of bugs in its implementation could cause hard fault exceptions during the access to in-chip peripherals which clock gating is not disable, or to wrong timing behaviour due to unexpected clock frequency settings. For this reasons, the software component Clock Management was tested as first. In order to verify the requirements described in section 3.1.1, the following tests were performed:

1. **Clock frequency measurements**
   The purpose of this test is to physically measure the frequencies of the MCU clock signals by means of the oscilloscope. In order to make the test possible, a MCU pin was configured to carry out the desired clock source.

2. **Clock gating verification**
   This test aims to verify the correct clock gating configuration. In order to be sure that the clock gating of the used peripherals were disabled, a memory access to their registers was done by means of the hardware debugging tools.

The listed tests were considered sufficient to completely test the functional behaviour of the software component Clock Management. They were executed before to test the other software components because, as explained in section 4.1.1, the correct clock configuration is a prerequisite for other tests.

4.3.1 Test Procedure

In order to perform the tests described in the previous section, a function was added to the software component Clock Management. Its prototype is reported in the following:

```
void ClkMgm_ClkOut(clock_names_t clkOut).
```

The role of this function is to configure the connection between the desired clock signal source to be measured and a specific MCU pin (PTD14). In detail, the function ClkMgm_ClkOut() configures the multiplexer and divider of the MC_CGM module connected to the so called CLKOUT_RUN signal (represented in Fig. 4.10). The procedure performed by the function is the following:

1. Configure the pin PTD14 as output and connect it to the so called CLKOUT_RUN signal - it corresponds to the output of the MUX_11_DIV_6 of the module MC_CGM;

2. Set the value of the divider MUX_11_DIV_6 to 128 - it is its maximum allowed value;
3. Enable the divider;
4. Configure the multiplexer MC_CGM_MUX_11 to select the clock signal source to be connected to the signal CLKOUT_RUN - it corresponds to the value of the function parameter clkOut;
5. Enable the multiplexer;
6. Disable the clock gating of the signal CLKOUT_RUN.

As can be noticed by the listed procedure, the divider applied to the selected clock signal source is set to its maximum value to avoid bandwidth issues during the frequency measurements performed through the oscilloscope.

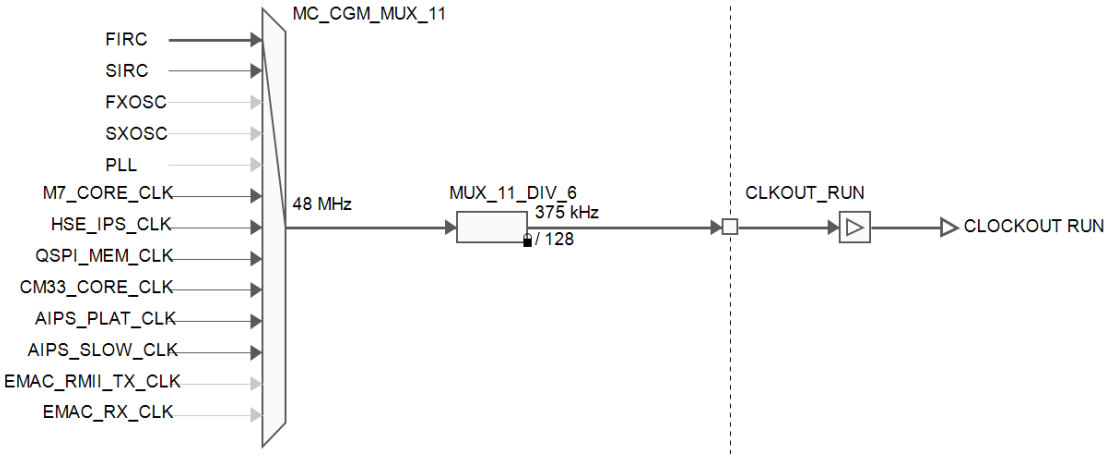

Figure 4.10: Structure of the multiplex and divider of the MC_CGM module connected to the signal CLKOUT_RUN.

Before to start the test, the probe of the oscilloscope shall be connected to measure the voltage waveform of the pin PTD14 which is where the signal CLKOUT_RUN is present. Once the instrumentation is set correctly, the test program can be downloaded in the MCU flash memory. The program structure is reported in the following:

Listing 4.4: Test program for the software component Clock Management

```
int main(void)
{
```

```
    /* Clock Management Initialization */
    ClkMgm_Init();

    /* Set M7_CORE_CLOCK as output clock source */
    ClkMgm_ClkOut(M7_CORE_CLOCK);

    /* Set M33_CORE_CLOCK as output clock source */
    ClkMgm_ClkOut(M33_CORE_CLOCK);

    /* Repeat for all interested clock sources */
    ..........

    /* Infinite loop */
    while(1){}
}
```

As can be seen in the source code, the program was written to test all the clock signal sources. Therefore, the multiplexer MC_CGM_MUX_11 is configured to change its source with the execution of the program. For this reason, the test program shall be executed step by step measuring the frequency of each clock signal source. The test is considered passed if all the frequencies of the considered clock signals are equal to the expected ones, otherwise the test fails and a bug in the software is present.

The clock gating verification test was performed using the software Trace32 to observe and modify some registers of the peripherals that are used by the other software components. More precisely, the possibility to read and write the configuration and status registers of a peripheral demonstrates a correct clock gating configuration. Therefore, in case an error occurs in disabling the clock gating of the desired peripheral through the MC_ME module, Trace32 signals that its registers are not accessible. The test is considered passed if all the registers of the peripherals that shall be enabled are accessible.

4.3.2 Results

In order to verify the requirements about the software component Clock Management listed in section 3.1.1, the previously described tests were executed. The measurements of the frequencies of the MCU clock signals are reported in Tab. 4.1. The values reported in the table are the expected frequency (Nominal Frequency), the frequency measured through the oscilloscope affected by the divider (Measured Frequency), and the actual frequency computed on the base of the measured one

MCU Clock Signal Frequencies Measurements			
Clock Signal	**Nominal Frequency**	**Measured Frequency**	**Computed Frequency**
M7_CORE_CLK	320.00MHz	2.4950MHz	319.36MHz
M33_CORE_CLK	160.00MHz	1.2453MHz	159.39MHz
FIRC_CLK	48.000MHz	377.21kHz	48.284MHz
AIPS_PLAT_CLK	80.000MHz	623.82kHz	79.848MHz
AIPS_SLOW_CLK	40.000MHz	312.69kHz	40.002MHz
HSE_IPG_CLK	80.000MHz	621.69kHz	79.576MHz

Table 4.1: Measurements of the MCU clock frequencies (test 1).

(Computed frequency). As can be noticed in the table, the measured clock frequencies show a correspondence with the expected nominal ones. An example of clock signal measurement through the oscilloscope is reported in Fig. 4.11.

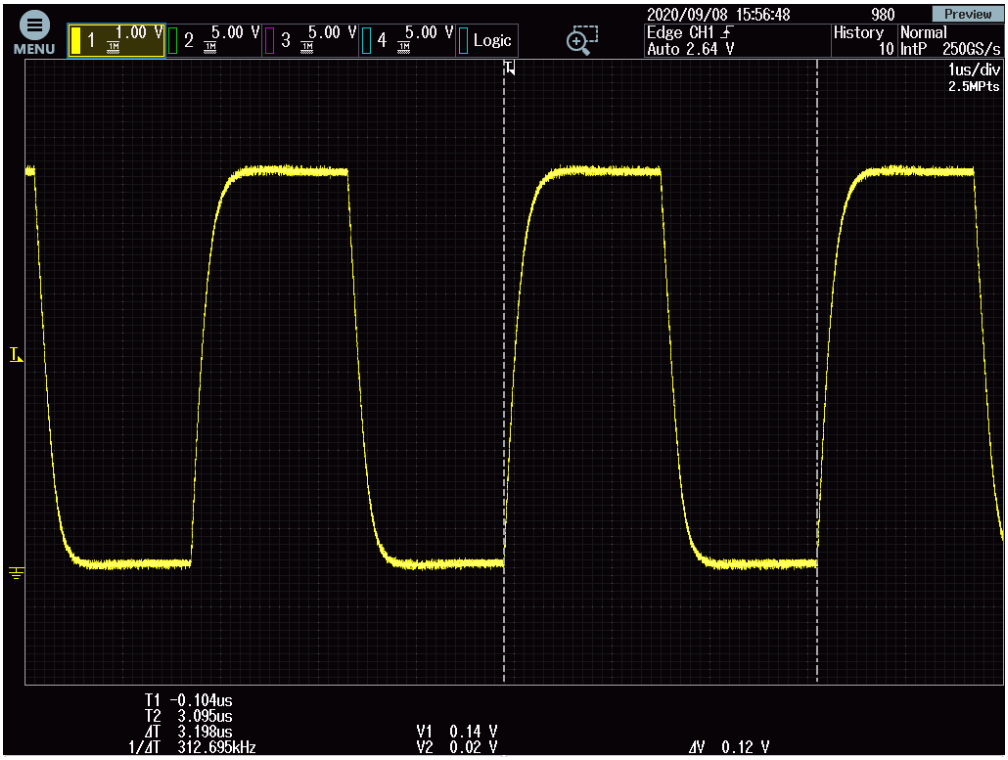

Figure 4.11: Frequency measurement of the MCU clock signal HSE_IPG_CLK.

After all the frequencies of the MCU clock signals were verified, the correct

clock gating configuration was analyzed. As already said, the debugging tools are able to access to the configuration and status registers of a peripheral only if its clock gating is disable. In Fig. 4.12 is reported a screenshot of Trace32 where it is not possible to access to the peripheral due to clock gating issues. All the used peripherals were checked to determine the correct clock gating configuration which is performed using the MC_ME module. Bugs were fixed until all the requirements described in section 3.1.1 were satisfied.

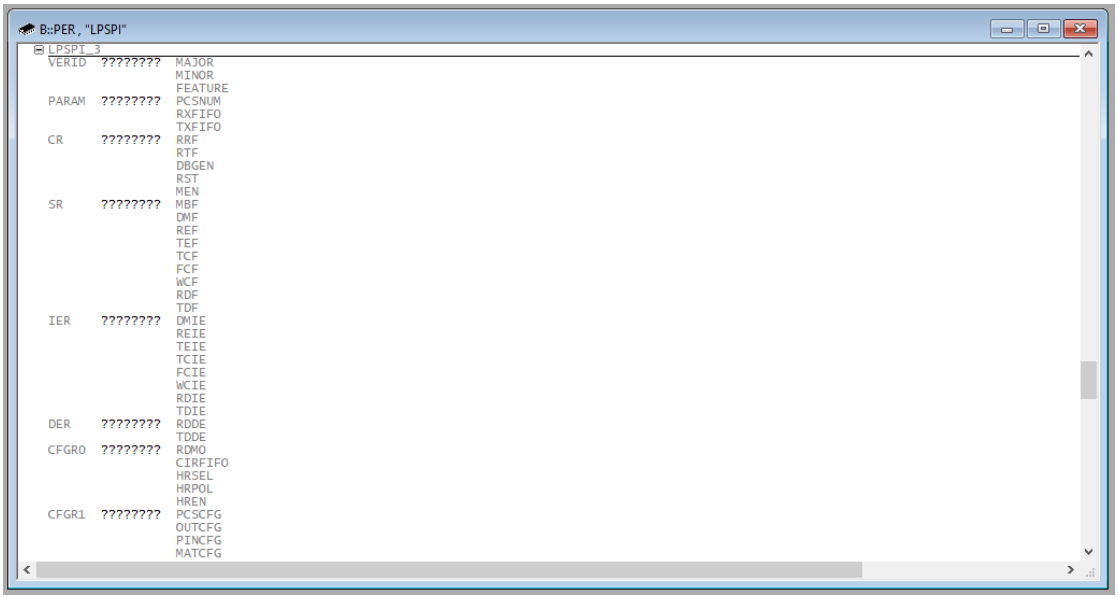

Figure 4.12: Configuration and status registers of a peripheral with enabled clock gating.

4.4 PWM Management Testing

The software component PWM Management was tested to verify that the three couple of complementary PWM signals are generated according to their requirements. All the features implemented by the API functions, described in section 3.4.3, were verified through a suitable test program. For the purpose, the following tests were executed:

1. **PWM generation with fixed duty cycles**
   This test aims to verify the PWM signal generation of the three couple of PWM signals when a fixed duty cycle is set. The generated output waveforms were acquired to verify that the signal generation is performed according to the requirements. More precisely, this test allows to measure the PWM switching frequency, the polarity of the complementary signals, any error in the duty cycles and the deadtime time interval. This test was performed more times with different values of duty cycles.

2. **PWM output enabling and disabling**
   After the PWM generation was tested and the corresponding system requirements verified through test 1, the disabling and enabling functions were tested. This test was performed introducing a so-called virtual button in the test program to enable or disable the PWM signal outputs.

3. **V/f control modulation**
   After all the implemented features were tested, an open-loop control algorithm was designed to generate at run-time the duty cycles for the PWM generation. The purpose of this test is to simulate an environment as close as possible to the actuation via PWM generation of the motor control algorithm implemented in the application software.

The listed testes were performed measuring through the oscilloscope a couple of PWM signals at time and using global variables in the test program to implement virtual buttons and controllable values. The use and implementation of virtual buttons is described in section 4.1.2 and was also extensively used to develop test programs for other software components.

4.4.1 Test procedure

Only one test program was developed to implement the tests listed in the previous section. A global variable was introduce to select the test that the user wants to execute. Since this variable is accessible through Trace32, it allows to control the test program execution changing its value.

As explained in the previous section, the test program was developed to reproduce conditions as close as possible to the nominal operating ones when the application software makes use of the PWM generation features. For this reason, a Periodic Interrupt Timer (PIT) was configured to generate an interrupt with a period of 62.5μs (corresponding to the chosen example frequency of 16kHz). This period corresponds to the period required by the motor control algorithm implemented in the application software. In the corresponding interrupt service routine (ISR), the V/f control algorithm or the constant duty cycle setting are executed depending on the control variable PwmGen. The global variable PwmGen can be modified by the user via Trace32 affecting the test program execution through a switch-case structure. This is shown by the following C code:

Listing 4.5: Interrupt service routine developed to test PWM Management.

```
void PIT_0_Handler(void){

    /* Enable/Disable PWM signal outputs */
    if (PwmEnabled)
    {
        PwmMgm_OutEnable();
    }
    else
    {
        PwmMgm_OutDisable();
    }

    /* Set duty cycle according to PwmGen */
    switch (PwmGen)
    {
        case PwmGen_const:
            PwmMgm_SetDuty(Duty_a, Duty_b, Duty_c);
            break;
        case PwmGen_CtrlVf:
            MotCtrl_PwmComputeVf(&Duty_a, &DutyB, &Duty_c);
            PwmMgm_SetDuty(Duty_a, Duty_b, Duty_c);
            break;
        default:
            break;
    }
}
```

The function PIT_0_Handler() is the ISR associated to the periodic interrupt generated by the PIT. The global variables Duty_a, Duty_b, and Duty_c are defined as single precision (32 bit) floating point numbers. They are used to set the duty cycles of the PWM signals - they are modified by the user in case of test 1 or the computed by the V/f control algorithm in case of test 3. As is shown in the source code reported above, the duty cycles are set each period independently from the variable PwmEnabled. Indeed, the variable PwmEnabled is used as virtual button to enable or disable the PWM signal outputs without affecting the duty cycles setting.

This test program allows to perform all the tests described in the previous section and verify all the features of the software component PWM Management. The procedure to follow to execute a complete test of the software component is described by the following steps:

1. Setup the oscilloscope to acquire a couple of complementary PWM signals and connect Lauterbach Debug and Trace to the target platform;

2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32;

3. Before to start the test program, assign the value PwmMgGen_const to the variable PwmGen to select test 1 and the duty cycle to be set to the considered PWM signals;

4. Start the test program;

5. Acquire the considered PWM signals and perform the required measurements;

6. Repeat point 5 with different values of duty cycles - duty cycles bigger that 1.0 and lower than 0.0 shall be also selected to test the behaviour of the saturation feature;

7. Change the value of the variable PwmEnable at run-time to perform test 2;

8. Assign the value PwmMgm_CtrlVf to the variable PwmGen to select test 3;

9. Acquire the considered PWM signals and perform the required measurements;

10. Change the value of the variable PwmEnable at run-time to repeat test 2;

11. Compare the acquired results to the expected ones and determine if the tests have been passed or failed.

4.4.2 Results

In order to verify the requirements about the PWM generation, the test procedure was performed on the target platform considering a couple of two complementary PWM signals at-time. The table 4.2 summarizes the obtained results of the PWM generation with fixed duty cycles. As can be seen in the table, the measured values

	PWM Generation Measurements					
	Switching Period		**Duty Cycle**		**Dead-time**	
Signal	**Nominal**	**Real**	**Nominal**	**Real**	**Nominal**	**Real**
PwmUH	16.000kHz	15.983kHz	0.666	0.629	3.00μs	2.99μs
PwmUL	16.000kHz	15.998kHz	0.333	0.286	3.00μs	3.01μs
PwmVH	16.000kHz	16.002kHz	0.500	0.452	3.00μs	2.98μs
PwmVL	16.000kHz	16.001kHz	0.500	0.453	3.00μs	2.99μs
PwmWH	16.000kHz	15.998kHz	0.333	0.286	3.00μs	3.01μs
PwmWL	16.000kHz	16.000kHz	0.666	0.630	3.00μs	2.99μs

Table 4.2: Results and measurements of the PWM generation with fixed duty cycles (test 1).

of the PWM switching frequency and dead-time time interval are really close to the nominal ones. The PWM signals acquired through the oscilloscope are shown in Fig. 4.13 and Fig. 4.14 - the measurements of the PWM switching frequency/period and the dead-time time interval are reported respectively in Fig. 4.13 and Fig. 4.14. The complementary behaviour of the two signals can be verified in both figures. Instead, the central alignment of the two signals can not be noticed graphically. This feature was verified observing the value of the configuration registers of the used eMIOS peripheral by means of Trace32.

An interesting consideration has to be done about the difference between the nominal duty cycles and the measured ones. The actual duty cycle of the PWM signals was computed measuring the T_{ON} period through the oscilloscope. In all the tested cases, the time difference between the actual T_{ON} time interval and the expected one is equal to about $3\mu s$. This value corresponds to the interval of the inserted the dead-time. From design point of view, it was decided to compensate this duty cycle error due to the dead-time insertion through the addition of a feedforward control action in the motor control algorithm of the application software. Therefore, no features about the compensation of the duty cycle error were implemented in PWM Management.

The enabling and disabling functions were verified acting on the virtual button PwmEnabled and observing the interested output signals through the oscilloscope. An example is reported in Fig. 4.15. The figure shows the time instant where the

Figure 4.13: Screenshot of the acquisition of two PWM complementary signals with switching period measurement.

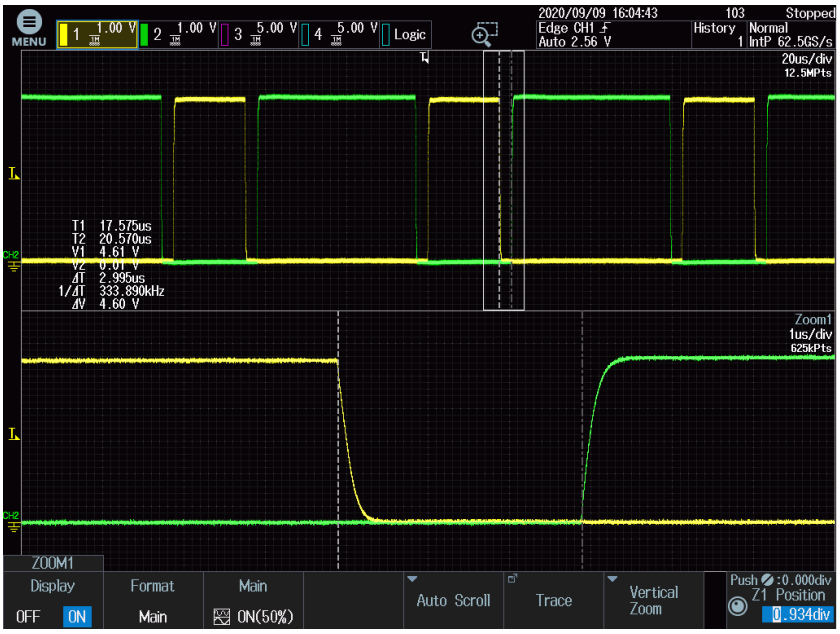

Figure 4.14: Screenshot of the acquisition of two PWM complementary signals with dead-time measurement.

variable PwmEnabled is changed from 0 to 1 corresponding to the enabling of the PMW output signals. The same was done to test the disabling feature obtaining, also in this case, the expected results. Therefore, test 2 was considered passed.

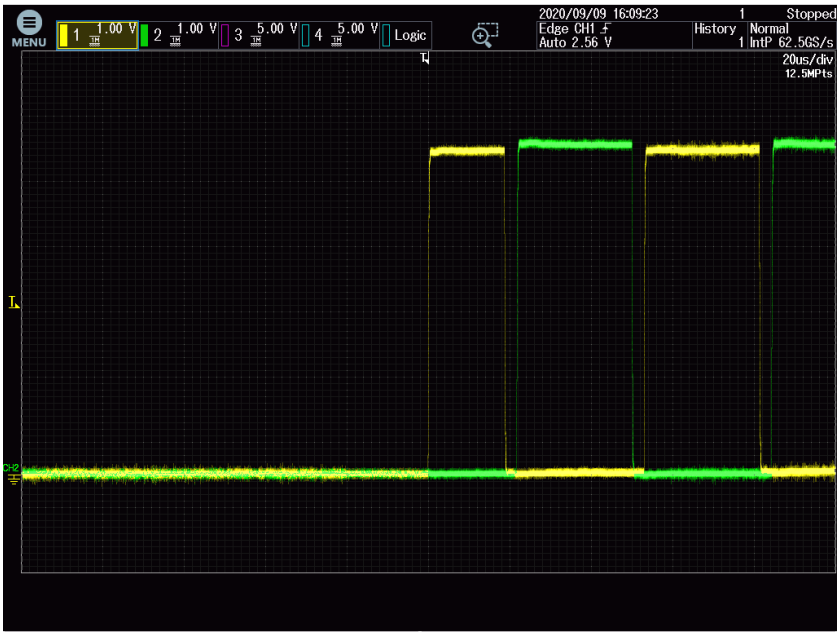

Figure 4.15: Screenshot of the acquisition of the enabling of two PWM complementary signals.

As described by the test procedure, the value PWMGen_CrtlVf was assigned to the variable PwmGen to start test 3. In order to verify the correct PWM generation, the oscilloscope was used to acquire the PWM signal outputs and perform a spectrum analysis on them. Since the control algorithm was designed to produce an output waveform with fundamental frequency of 50Hz, a peak around 50Hz is expected in the spectrum of the considered signals. The analysis was performed computing the Fast Fourier Transform (FFT) on one output signal by means of the oscilloscope. The result are reported in Fig. 4.16. As expected, a peak in correspondence of 50Hz can be observed in the figure. Furthermore, a second smaller peak was measured in correspondence of about 150Hz. This is justified by the third harmonic injection performed by the implement control routine. Therefore, the PWM generation with duty cycles assigned by the V/f control algorithm were successfully verified.

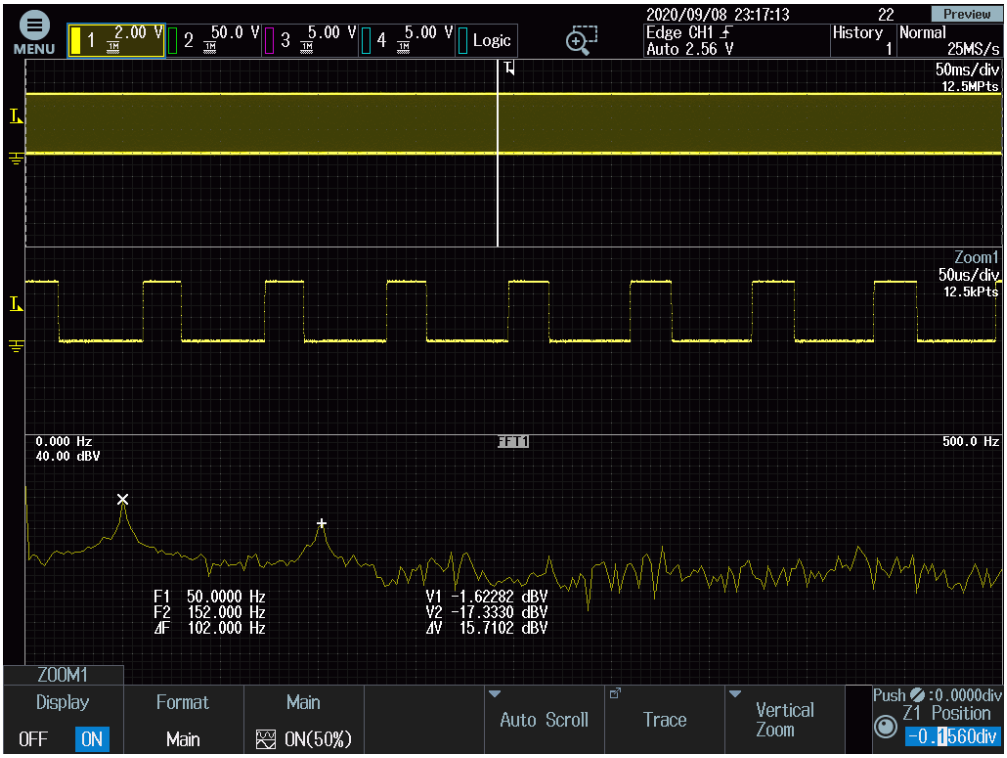

Figure 4.16: Spectrum analysis of the PWM signals when the duty cycles are set by the V/f control algorithm.

4.5 ADC Management Testing

The software component ADC Management was tested to verify that the analog signal acquisition behaves according to its requirements. The requirements, described in section 3.3.1, were verified using the signal generator to produce suitable constant and time-varying analog signals for the target platform and the Lauterbach trace features to observe the corresponding acquired values. The timings of the acquisitions of the analog signals were verified through a loop-back approach. More precisely, the software component PWM Management was used to generate suitable PWM signals to be acquired back to determine the time instants corresponding to the start of conversion of the acquisitions. The tests executed for the software component are listed in the following:

1. **Constant analog signal acquisitions**
   This test aims to verify that constant analog signals, generated by the signal generator, are acquired and converted properly by the target platform. The correspondence between the generated signals and the digital converted

values was checked to determine the test success. During this test, the correct acquisition frequency was verified through Trace32 analyzing the time distance between the occurrence of the interrupts associated to the end of conversion of the last channels in the conversion list (see section 3.3.2 for more implementation details).

2. **Time-varying analog signal acquisitions**
   This test aims to reproduce test 1 with time-varying analog signals generated by the signal generator. As described previously, the correspondence between the generated signals and the digital converted values was checked to determine the test success and the correct acquisition frequency was verified through Trace32. In this case, the digital converted raw values were observed using the feature Trace.Draw of Trace32.

3. **Acquisition timing measurements**
   The timing requirements about ADC sampling were verified through this test. As explained in section 3.3.1, the signal acquisition (especially of the phase currents) shall occur at the center of the T_{ON} period of PWM signals - this time instant corresponds to the center of the carrier waveform. The used approach was based on acquiring back a PWM signal with known duty cycle by means of the ADCs. Since the PWM generation is center aligned, the minimum duty cycle for which the MCU acquires a value of 5V indicates the time instant where the conversion of the acquired signal starts. For completeness, this test was done considering all the analog input signals, even if this characteristic is critical only for the acquisition of the phase currents and some voltages.

The listed tests were executed to verify the requirements described in section 3.3.1. Moreover, the acquisition timing measurement test allows to provide a figure about the nominal operating conditions to which the MCU will be subjected during the overall powertrain test for what concerns the analog signal acquisition.

4.5.1 Test procedure

In order to perform the tests described in the previous section, only one test program was developed. The test program makes use of the software component PWM Management to implement the acquisition timing measurements test.

The raw converted values were observed through Trace32 accessing to the data structures DataCurrRaw, DataVoltRaw, and DataTempRaw defined in ADC Management files. The values of the members of these data structures were observed plotting their time evolution in Trace32. The virtual button PwmEnabled was used to enable or disable the PWM generation, and consequentially, to start and stop the test 3 about acquisition timing measurements. The source code used for

PWM Management testing purposes was reused to generate the required PWM signals. More precisely, the software component initialization and the interrupt service routine (ISR), associated to the Periodic Interrupt Timer (PIT), are the same of the ones described in section 4.4.1 without the implementation of the V/f frequency control algorithm. Therefore, the PIT_0_Handler() was implemented as following:

Listing 4.6: Interrupt service routine for the test program of ADC Management.

```
void PIT_0_Handler(void){

    /* Peripheral interrupt flags check and clear */
    ..........................

    /* Enable/Disable PWM signal outputs for test 3 */
    if (PwmEnabled)
    {
        PwmMgm_OutEnable();
        PwmMgm_SetDuty(Duty_a, Duty_b, Duty_c);
    }
    else
    {
        PwmMgm_OutDisable();
    }
}
```

Of course, this implementation is based on the assumption that the software component PWM Management was already successfully tested. It is important to remember that the initialization of the software component PWM Management is necessary for the correct working of ADC Management independently of this specific test program implementation. Indeed, the initialization function of PWM Management configures the carrier waveform used both for center aligned PWM generation and as hardware trigger event to start the conversion of the first three elements in the conversion list (see section 3.3.2 for implementation details).

In order to perform all the tests listed in the previous section with the described test program, the test procedure represented by the following steps shall be followed:

1. Connect the signal generator output to an analog input signal of the target platform and the Lauterbach Debug and Trace to the debug and trace ports;
2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32

software;

3. Configure the signal generator to produce a constant analog signal between 0V and 5V and set the virtual button PwmEnabled to disable PWM signal outputs;

4. Start the test program;

5. Observe the digital converted value of the analog signal input connected to the signal generator using Trace32 and check the correspondence between the input signal amplitude and the raw converted value;

6. Stop the test program and repeat from step 3 with all the analog input signals of the target platform varying the amplitude of the input signal generated by the signal generator;

7. Observe the time distance between the occurrence of two different calls of the function Body_Crossing_Triggering_Unit_Handler() to verify the correct acquisition rate - this step concludes test 1;

8. Repeat from steps 3 to 7 for all the analog inputs using a time-varying waveform and observing the raw converted values through the feature Trace.Draw of Trace32 - these steps are related to the execution of test 2;

9. Set the value PwmEnable to enable the PWM signal outputs and connect one PWM output signal to one analog input of the target platform;

10. Restart the test program and decrease the duty cycle of the acquired PWM signal until the raw converted value still remains to its maximum value - the minimum value of duty cycle that guarantees to have a raw value corresponding to 5V allows to compute the time distance between the center of the carrier waveform and the real point where the analog acquisition occurs for the considered signal;

11. Repeat step 10 for all the analog input signals - these steps complete test 3;

12. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

It is important to underline that the test procedure can be executed entirely or one test at-time to easy the setup of the laboratory instrumentation.

4.5.2 Results

The test procedure described in the previous section about ADC Management was performed on the target platform. For simplicity, only one analog input signal is considered in this section to report the obtained results, but the procedure was executed considering all the analog inputs of the target platform.

The considered analog input signal is a motor phase current. The results of test 1 are reported in Tab. 4.3. The table reports the constant input voltage levels generated by the signal generator, the raw digital converted values, their corresponding values expressed in the voltage range from 0V to 5V, and the absolute error between the voltage input value and the acquired one. The conversion in the corresponding voltage range was done manually by means of the following expression:

$$V_{MEAS} = \frac{V_{MAX} - V_{MIN}}{RAW_{MAX}} \times RAW_{MEAS} = \frac{5\text{V} - 0\text{V}}{2^{15}} \times RAW_{MEAS}, \qquad (4.1)$$

where V_{MEAS} is the acquired voltage level, V_{MAX} and V_{MIN} respectively the maximum and minimum input voltage full-scales, RAW_{MAX} the digital value full-scale, and RAW_{MEAS} the raw digital converted value. As can be seen in the

Constant Analog Acquisition Results			
Input Voltage	**Digital Value**	**Voltage Value**	**Absolute Error**
0.000V	0	0.000V	0.000V
0.500V	3090	0.471V	0.029V
1.000V	6422	0.980V	0.020V
1.500V	9712	1.482V	0.018V
2.000V	12724	1.942V	0.058V
2.500V	15878	2.423V	0.077V
3.000V	19162	2.924V	0.076V
3.500V	22302	3.403V	0.097V
4.000V	25624	3.910V	0.090V
4.500V	28660	4.373V	0.127V
5.000V	32072	4.894V	0.106V

Table 4.3: Measurements performed during constant analog signal acquisition (test 1).

table, the error between the acquired values and the voltage input levels is always below than 110mV. It is important to notice that the absolute error, which is the difference between the nominal input voltage and the measured one, increases with the rise of the considered input values. This could be caused by a gain error in the

ADC characteristic. Anyway, the absolute error was considered sufficient small for the application of this thesis.

A statistical figure of the time distance between two consecutive interrupts of the peripheral BCTU due to the end of the conversion list is reported in Fig. 4.17. This statistical analysis was obtained with the feature Trace.STATistics.DIStance of Trace32. As can be seen in the figure, the result of the time distance analysis is

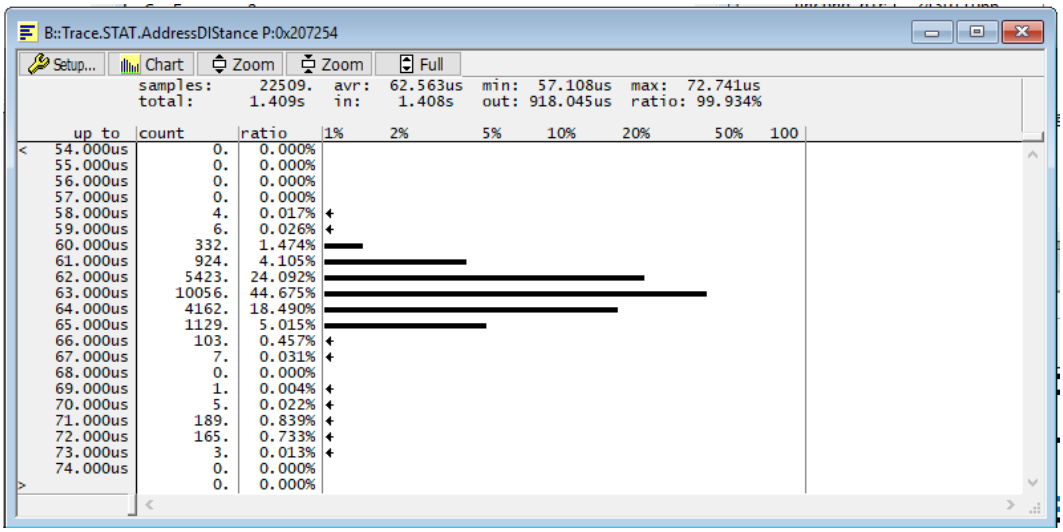

Figure 4.17: Statistical figure of the time distance between two consecutive interrupts of the BCTU peripheral due to the end of the conversion list.

a Gaussian distribution around the value of 62.5μs. This demonstrates that the analog acquisition occurs with the correct time rate according to the requirements described in section 3.3.1. It is important to underline that this result can be considered valid on the assumptions that no event interferes with the call of the ISR associated to the BCTU peripheral, and the conversion times of the ADCs are deterministic and always constant.

The time-varying signals used to perform test 2 are constituted by sinusoidal and triangular waveforms. The amplitude of the two signals was chosen to span between the minimum and maximum full-scales. Therefore, the first input signal chosen for the test was a sinusoidal waveform with a peak-to-peak voltage of 5V, offset of 2.5V, and frequency of 1.3kHz. The signal frequency was chosen according to the maximum frequency of the phase motor current. The signal acquisition was observed with the feature Trace.Draw of Trace32 and the results are represented in Fig. 4.18. As can be seen in the figure, no distortions occurred in the analog signal acquisition. The same was done using a triangular waveform as input signal with the same parameters of the previously used sinusoidal waveform. The results are reported in Fig. 4.19, and also in this case, no distortions occurred in the

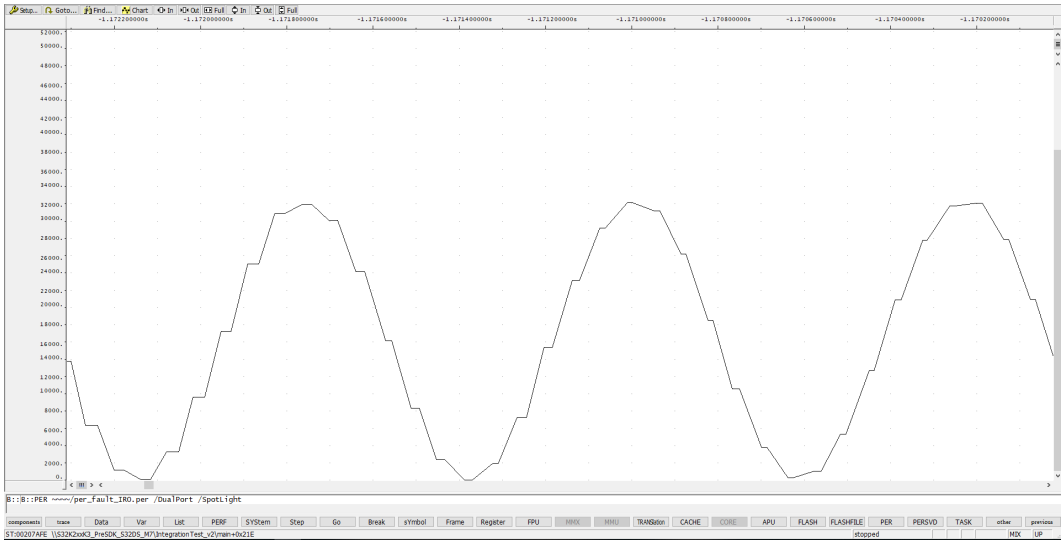

Figure 4.18: Digital raw values acquired from an analog input connected to a sinusoidal waveform observed with Trace32.

signal acquisition. In both figures, the number of samples for each period of the acquired waveform is 12-13. This results is another demonstration that the sampling frequency of the analog acquisition was correctly set to 16kHz.

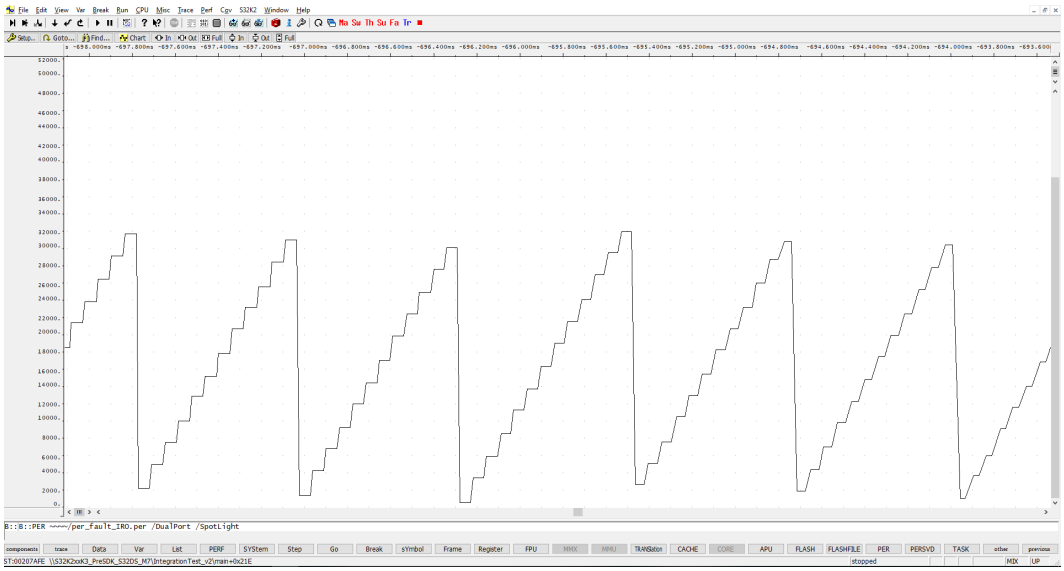

Figure 4.19: Digital raw values acquired from an analog input connected to a triangular waveform observed with Trace32.

In order to perform test 3, the PWM output signals were enabled and the loopback connection implemented in the target platform. As already explained, the PWM generated was acquired back through the considered analog input to measure the time distance between the center of the carrier waveform of the PWM signals and the real acquisition time instant. This time distance was obtained by the following expression:

$$T_d = \frac{T_{ON}}{2} = \frac{62.5\mu\text{s} \times D}{2}\,, \tag{4.2}$$

where T_d is the computed time delay, T_{ON} is the period where the PWM signal has a high voltage level in a switching period, and D is the set duty cycle. The time delays of each parallel conversion of triple of signals associated to the BCTU conversion list is reported in Table 4.4 As can be seen in the table, the time distance between the occurrence of the conversion of two different triple of signals is 3.4. Therefore, the time to convert all the signal in the conversion list can be estimated to be equal to 30.6μs. This result is very close to the theoretical one computed in section 3.3.2.

Acquisition Time Delay Measurements		
Signals	**Duty cycle**	**Computed time Delay**
Triple 1	10.88	3.4μs
Triple 2	21.76	6.8μs
Triple 3	32.64	10.2μs
Triple 4	43.52	13.6μs
Triple 5	54.40	17μs
Triple 6	65.28	20.4μs
Triple 7	76.16	23.8μs
Triple 8	87.68	27.4μs
Triple 9	97.92	30.6μs

Table 4.4: Acquisition timing measurements about the triple of signals in the BCTU conversion list (test 3).

4.6 SPI Communication Testing

The software component SPI Communication was tested to verify the correct behaviour of the SPI communication between the MCU and the resolver-to-digital converter. The requirements described in section 3.5.1 about the communication timings, signal polarities, configuration frame sending, and data frame receiving were verified using the serial decoding feature of the oscilloscope to observe and decode the bus signals, and the Lauterbach to access to the MCU debug and trace ports. After some specific hardware tests, the software component was tested executing the following tests:

1. **Generic data sending**
   This test aims to verify the correct sending of an easy-to-measure data frame through SPI communication. During this test, the correct peripheral and timing configurations, signal polarities, and generic frame sending were verified.

2. **Generic data acquisition**
   This test is performed to request and read a frame sent through SPI communication. The purpose of this test is to verify the correct peripheral configuration and frame acquisition. In order to perform this test, a loopback approach was used: the MISO and MOSI signals of the LPSPI peripheral were connected together to acquire the data frame sent by the MCU itself.

3. **R2D converter configuration frame sending**
   This test sends a specific configuration frame through SPI communication to the resolver-to-digital converter. Its purpose is to verify the correct sending of the configuration frame that shall be received by the R2D converter during system initialization. The test is repeated two times - first, it is performed connecting the MCU to the oscilloscope, and once the sent configuration frame is checked, the complete test program is executed.

4. **R2D data acquisition through SPI interface**
   This test aims to periodically acquire data frames sent by the R2D converter through SPI communication. In order to be executed, the R2D converter must be connected to the MCU and configured through SPI before to be used. The R2D converter was configured according to the specifications described in 3.5.1 to simulate conditions as close as possible to the nominal operating ones.

These tests were considered sufficient to complete verify the functional behaviour of the software component and the corresponding SPI communication between the MCU and the resolver-to-digital converter.

4.6.1 Test procedure

Two different approaches were used to perform the tests about generic (test 1 and 2) and R2D specific (test 3 and 4) data sending and receiving. In the first case, the oscilloscope probes were set up to acquire the output pins of the LPSPI peripheral. The MISO and MOSI lines of the LPSPI peripheral were connected together to make a loopback connection. The signals observed by the oscilloscope were the MOSI, SCK, SSCS, and SCSB of the SPI interface of the R2D converter. The acquisition of the MISO line was not required because corresponds to the MOSI one due to the loopback configuration. During this test, the software Trace32 was used to set the generic frames to be sent by the MCU and observe the loopback acquisition. It is important to remember that this test was possible due to the presence of a hardware selector able to connect or disconnect the R2D device to the MCU. In the second case, the R2D converter was connected to the MCU and the acquired data read through Trace32. Of course, in order to acquire meaningful data from the R2D converter, the resolver must be connected to the target platform. This setup is represented in Fig. 4.20.

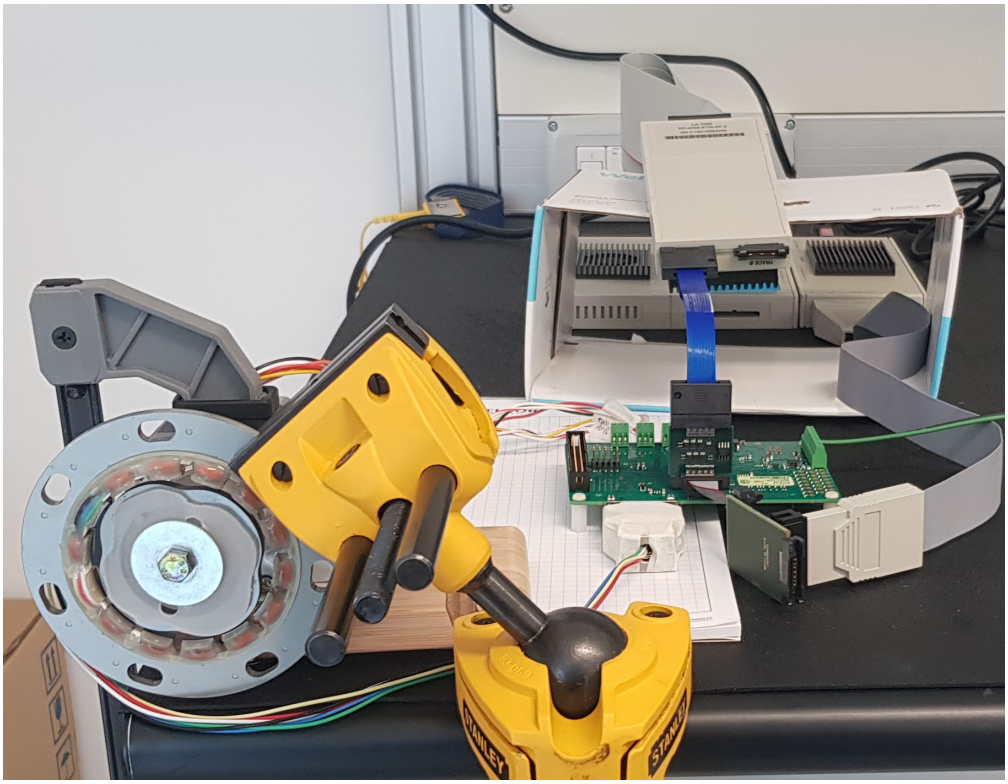

Figure 4.20: Setup used to test the software component SPI Management when the target platform is connected to the resolver and the R2D converter.

Two test programs were developed to perform the test listed above. More precisely, the first one was developed to implement the generic data sending and acquisition through the loopback connection (test 1 and 2), and the second one to verify the correct communication between the R2D converter and the MCU.

Since the first test program was designed mainly to verify the bus timings and signal characteristics of the communication, a virtual button was used to decide when to start the transmission. This mechanism was implemented in an infinite loop inside the main function. The source code is the following:

Listing 4.7: Infinite loop to implement the first test program for SPI Communication.

```
int main(void)
{

    /* Initialization sequence */
    ..............................

    /* Send or acquire SPI data */
    while(1)
    {
        if(SpiSendEnabled)

            switch(SpiTx)
            {
                case SpiTx_cfg:
                    R2dCfgError = SpiCom_R2dSetCfg(R2dCfg);
                    break;
                case SpiTx_data:
                    R2dDataError = SpiCom_R2dReqData();
                    break;
                default:
                    break;
            }

            SpiSendEnabled = 0;
        }
        else
        {
        }
}
```

As can be seen in the source code, SpiTx is a global variable used to control the test program execution flow. More precisely, When its value is SpiTx_cfg or SpiTx_data, the tests performed are respectively the the generic data sending (test 1) and receiving (test 2). The data frame is sent through SPI using the function SpiCom_R2dSetCfg(). The global variable R2dCfg is used as parameter to change the values in the data frame to be sent. Instead, the data receiving is performed through the function SpiCom_R2dReqData(). In normal operating condition, this function puts the MOSI line to 0 when a data acquisition transmission is started. Instead, in this test program, it is also responsible to send the frame to be acquired back. The data requested by the function are stored in the variable R2dSpiData declared as global in the file .c related to the software component SPI Communication. Both functions return variables of type status_T which are used to check if errors occur during the program execution.

The previously described test program was used to perform the tests 1 and 2 according to the following test procedure:

1. Connect the oscilloscope to the SPI bus signals of the target platform, and the Lauterbach Debug and Trace to the debug and trace ports;
2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32 software;
3. Start the test program;
4. Set a desired value in the variable R2dCfg to be sent through SPI and the value SpiTx_cfg in the variable SpiTx to start test 1;
5. Set the value 1 in the variable SpiSendEnabled to start the SPI transmission and acquire the frame sent through the oscilloscope;
6. Repeat step 5 with different value for the variable R2dCfg - these steps constitute the execution of test 1;
7. Connect the MISO and MOSI signals of the SPI bus to make a loopback connection;
8. Set a desired value in the variable R2dTxFrame (defined as global static variable in SpiCom.c) to be sent through SPI and the value SpiTx_data in the variable SpiTx to start test 2;
9. Set the value 1 in the variable SpiSendEnabled to start the SPI transmission, acquire the frame sent through the MOSI signal using the oscilloscope, and observe the MCU acquisition using Trace32;

10. Check the correspondence between the frame acquired by the MCU and the one by the oscilloscope;

11. Repeat from step 9 with different value for the variable R2dTxFrame - these steps constitute the execution of test 2;

12. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

The second test program was developed to reproduce conditions as close as possible to the nominal operating ones. The configuration data sending was performed immediately after the initialization of the LPSPI peripheral and the re-enabling of the global interrupts. Then, a Periodic Interrupt Timer (PIT) was configured to manage the periodic acquisition of the data coming from the R2D converter. The application software will require the motor angular position given by the resolver and R2D converter with a frequency of 16kHz. For this reason, the PIT peripheral was configured to generate an interrupt each 62.5μs. Therefore, in order to execute the tests 3 and 4, the resolver was connected to the target platform. The interrupt service routine (ISR) of the PIT implemented for the test program is reported in the following:

Listing 4.8: Interrupt service routine for the test program of SPI Communication.

```
void PIT_0_Handler(void)
{
    /* Peripheral interrupt flags check and clear */
    ..........................

    /* Request data from the R2D converter through SPI */
    R2dDataError = SpiCom_R2dReqData();
}
```

As done for the first test program, the global variable R2dDataError was added to monitor if errors occurred during the test program execution related to SPI transmissions. As can be seen in the source code, the function SpiCom_R2dSetCfg() is not present in the ISR because the configuration sending is done only one time at startup and not periodically. It is important to remember that the data requested by the function SpiCom_R2dReqData() are stored in the variable R2dSpiData declared as global in the file SpiCom.c.

In order to perform the tests 3 and 4, listed in the previous section, a test procedure was designed taking into account the connection between the target platform and the resolver. The test procedure is described by the following steps:

1. Connect the resolver to target platform, the oscilloscope to SPI bus signals, and the Lauterbach Debug and Trace to the debug and trace ports;

2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32 software;

3. Before to start the test program, set a breakpoint immediately before the call of the function SpiCom_R2dSetCfg();

4. Start the test program, which will stop after sending the R2D configuration frame, and check that the configuration frame and the SPI bus signals respect the requirements - this step corresponds to test 3;

5. If the previous step has been passed, continue the execution of test program;

6. Check the SPI bus signals using the oscilloscope to verify that they behave according to their requirements;

7. Move the resolver to change the sensed angular position and check the correspondence between its position and the data acquired by the MCU - these steps corresponds to test 4;

8. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

During the execution of the test program, particular test points, designed for the purpose in the PCB of the target platform, were used to acquired the interested signals constituted by the MOSI, SCK, SSCS, and SCSB of the R2D SPI interface.

4.6.2 Results

The test procedures described in the previous section about the software component SPI Communication were performed on the target platform to verify the requirements listed in section 3.5.1 and simulate conditions as close as possible to the operating ones. The acquisition of a generic frame sent through SPI (test 1) is reported in Fig. 4.21. This frame acquisition was used to measure the timings of the SPI communication when the configuration frame is sent to the R2D. The measurements are reported in Tab. 4.5. The set and hold timings of the related chip select signal are much greater than the minimum required ones, instead the ones related to the frame sending and clock signals were set as small as possible to increase the communication speed.

Then, test 2 was executed to verify the acquisition of the SPI frames. The test results are reported in Fig. 4.22. In the figure, it is reported the oscilloscope

Figure 4.21: Screenshot of the oscilloscope reporting a generic frame sent by the MCU.

acquisition of the frame sent through the MOSI line and the value acquired back through the MISO one stored in the variable R2dSpiData. For this particular case, the value acquired back by the MCU was observed through the corresponding SPI data buffers. As can be seen, there is a correspondence between the buffer R2dBufferRx and the frame decoded by the oscilloscope. Also in this case, the oscilloscope was used to measure the timings of the SPI communication when the data frame is requested by the MCU. The measurements are reported in Tab. 4.5. As seen for the configuration frame sending, the set and hold timings of the chip select signal related to the data acquisition are much greater than the minimum required ones, instead the ones related to the frame sending and clock signals were set as small as possible to increase the communication speed. Therefore, the timing requirements and signal characteristics were considered satisfied both for configuration frame sending and data requesting.

Test 3 was performed using the second test program and its results are reported in Fig. 4.23. More precisely, the figure shows an oscilloscope acquisition of the configuration frame sent to the R2D converter at startup. The sent frame is constituted by 16 bits even if only 12 are required for the configuration. In this

Figure 4.22: Correspondence between the frame sent through the MOSI line and the one acquired back by the MCU itself.

SPI Communication Timing Requirements				
Symbol	**Parameter**	**Min.**	**Max.**	**Measured**
t_{hSCK}	SPI clock high time	100ns		100ns
t_{lSCK}	SPI clock low time	100ns		100ns
$t_{setSCSB}$	Data Chip select setup time	100ns		1μs
$t_{holSCSB}$	Data Chip select hold time	100ns		1μs
$t_{disSCSB}$	Data Chip deselect time	200ns		1μs
$t_{setSSCS}$	Setting Chip select setup time	100ns		1μs
$t_{holSSCS}$	Setting Chip select hold time	100ns		1μs
$t_{disSSCS}$	Data Chip deselect time	200ns		1μs
$t_{setSSDT}$	Setting input setup time	100ns		100ns
$t_{holSSDT}$	Setting input hold time	100ns		100ns

Table 4.5: Timing requirement and measurements of SPI communication between the MCU and the resolver-to-digital converter.

case, only the last 12 bits sent are considered by the R2D converter to configure the device. Moreover, the figure demonstrates that the set configuration corresponds to the required one highlighted in Fig. 3.16.

Figure 4.23: SPI frame sent by the MCU to configure the R2D converter at startup.

Test 4 simulates the acquisition of the motor angular position sensed by the resolver and converted by the R2D device. The resolver was connected to an external stepper motor. Its rotational speed was maintained constant and the data acquired with a frequency of 16kHz. The variable R2dSpiData was plotted using Trace32 using the feature Trace.Draw. Its time evolution is reported in Fig. 4.24. The angular speed of the step motor connected to the resolver was set to be equal to about 850rpm. As can be seen in the figure, the profile of R2dSpiData has a triangular shape spanning from 0 to 4096 (2^{12}) with a period of about 70ms corresponding to about 850rpm. Therefore, the data acquisition from the R2D converter was successfully verified. The same test was repeated without moving the resolver, acquiring a constant angular position, and comparing the real position with the acquired one, but the results are not reported in this section.

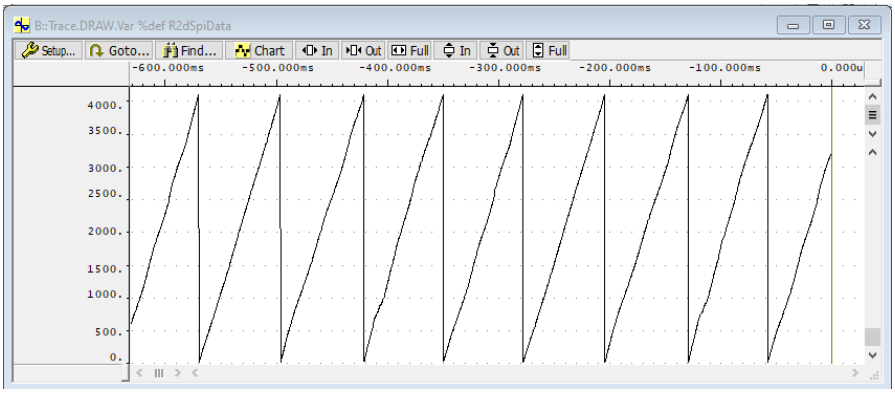

Figure 4.24: Time evolution of the variable R2dSpiData acquired through the trace feature using the Lauterbach debugging tools.

4.7 Digital Signal Management Testing

The software component Digital Signal Management was tested to verify that the digital output and input signals connected to the target platform work as expected. The requirements about their acquisition and driving, described in section 3.6.1, were verified using the signal generator to generate high or low logical signals for the digital inputs, the oscilloscope to observe the digital outputs, and Trace32 to access to the MCU debug and trace ports. For the parallel port of the resolver-to-digital converter, a dedicated test was developed independent from the one developed to test the general features of Digital Signal Management. The tests required to verify the implemented features are listed in the following:

1. **Digital input signal acquisition**
   Test 1 aims to verify the digital signal acquisition. More precisely, the digital input pins were stimulated with high (5V) and low (0V) logic signals. The corresponding values were acquired and stored in dedicated variables using the API functions described in section 3.6.3. During this test, the correct polarity configurations were verified.

2. **Digital output signal driving**
   Test 2 aims to test the digital signal driving. More precisely, the digital output pins were driven to high (5V) and low (0V) logic signals. The corresponding values were driven using specific virtual buttons and the API functions described in section 3.6.3. As done for test 1, during this test, the correct polarity configurations were verified.

3. **R2D data acquisition through digital parallel interface**
   This test aims to test the data acquisition through the digital parallel port

of the resolver to digital converter. This test must be performed after the software component SPI communication has been successfully tested because a preliminary configuration of the R2D converter, done through SPI, is required. Then, the data acquired through the digital parallel interface are compared with the one acquired through SPI.

The listed tests were executed driving and acquiring the digital signals by means of a polling mechanism. This approach was used to simulate the conditions in which the application software interacts with the digital signals of the target platform. Indeed, this implementation was chosen because it is as close as possible to the one used in operating conditions.

4.7.1 Test procedure

In order to perform the tests described in the previous section, one test program was developed. First, the test program was used to perform the acquisition of the digital inputs and the driving of the digital outputs corresponding respectively to test 1 and 2. Then, the data acquisition through the digital parallel connected to the R2D converter (test 3) was tested. In order to perform this test, the software component SPI Communication must be previously successfully tested.

The test program was developed to acquire and drive digital signals periodically with the API functions of the software component. The period of this process was set to be 62.5μs, and as usual, implemented through a periodic interrupt timer (PIT). As already said, this development choice was carried on to simulate the same conditions that occur when the system works in the field. The acquired signals were stored in dedicated global variables observable with Trace32. Global variables were used as virtual button to set the desired values in the digital output signals. The source code of the interrupt service routine (ISR) of the PIT peripheral is the following:

Listing 4.9: Interrupt service routine for the test program of PWM Management.

```
void PIT_0_Handler(void){

    /* Peripheral interrupt flags check and clear */
    ..........................

    /* Set the digital output signals */
    DigSgnMgm_SetSignalName1(Out_SignalNameValue1);
    DigSgnMgm_SetSignalName2(Out_SignalNameValue2);
    DigSgnMgm_SetSignalName3(Out_SignalNameValue3);
```

```
    ..........................................

    /* Get the digital input signals */
    In_SignalNameValue1  = DigSgnMgm_GetSignalName1();
    In_SignalNameValue2  = DigSgnMgm_GetSignalName2();
    In_SignalNameValue3  = DigSgnMgm_GetSignalName3();
    ..............................................

    /* Acquire data from the R2D converter */
    if(R2dDataEnabled){
        R2dDataErrorDig = DigSgnMgm_R2dReadData();
        R2dDataErrorSpi = SpiCom_R2dReqData();
    }
}
```

This implementation allows to acquire the input signals in the dedicated global variables In_SignalNameValueX and to set the output signals changing the values of the virtual buttons Out_SignalNameValueX. As can be seen in the source code, the acquisition of the data provided by the R2D can be enabled acting on the virtual button R2dDataEnabled. The data are acquired through the digital parallel and SPI interfaces using respectively the functions DigSgnMgm_R2dReadData() and SpiCom_R2dReqData(). Both functions return variables of type status_T which were used to monitor if errors occurred in the data acquisition. The acquired data are stored in the global variables R2dSpiData and R2dDigData. In order to correct interface with the R2D converted, its configuration procedure is performed after the initialization sequence. The test procedure, used to perform all the tests described in the previous section, is described by the following steps:

1. Connect the signal generator output to a digital input signal, the oscilloscope to a digital output signal, the resolver to the target platform, and the Lauterbach Debug and Trace to the debug and trace ports;

2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32 software;

3. Configure the signal generator to generate a constant signal with amplitude of 0V or 5V to simulate low or high digital values, and set the value of virtual button associated to the digital output pin connected to the oscilloscope;

4. Start the test program;

5. Observe the acquired value of the digital input signal connected to the signal

generator and the measured voltage level of the digital output signal connected to the oscilloscope to verify their correct configurations;

6. Repeat from step 3 to 5 for all the digital input and output signals to be tested;

7. Enable the R2D data acquisition using the virtual button R2dDataEnabled to start test 3;

8. Compare the data acquired through SPI (previously tested) with the one acquired through the digital parallel port using different configuration for the resolver to verify their coherence - steps 7 and 8 constitutes test 3;

9. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

4.7.2 Results

The test procedure described in the previous section about Digital Signal Management testing was performed on the target platform. The results of only one digital input and one digital output were reported in this section, but all the digital signals were tested with the same approach.

The input and output signals considered in this section are both with negative polarities - this means that a 5V electric voltage signal corresponds to a logic 0 and 0V to a logic 1. The results of the test for the output digital signal are reported in Fig. 4.25 and 4.26. As can be seen in the figures, the oscilloscope shows a 0V constant voltage level when the virtual button used to set the output value of the digital signal, corresponding to the signal SafeStateReq_Do in the Control Setup - switch windows, is equal to 1, and a 5V voltage level in the opposite case. These results demonstrate that the API functions used to set the digital output signals and the polarity configuration are working according to the requirements described in section 3.3.1.

The results of the test for the input digital signal are reported in Fig. 4.27 and 4.28. As can be seen in the figures, the Control Setup - parameter window in Trace32 shows that the variable HvDcOvv_Di is 0 when the input signal has a voltage level of 5V and 1 when the input signal has a voltage level of 0V. In this partical case, the input signal HvDcOvv_Di was connected to the output signal TriggerMapping_Do in a loopback fashion to use the virtual button of TriggerMapping_Do to drive the input of the acquired signal.

Also in the case, the results demonstrate that the API functions used to read the digital input signal and the polarity configuration are working according to the requirements described in section 3.3.1. This approach was used for all the digital input and output signals, but the results were not reported for a brief discussion.

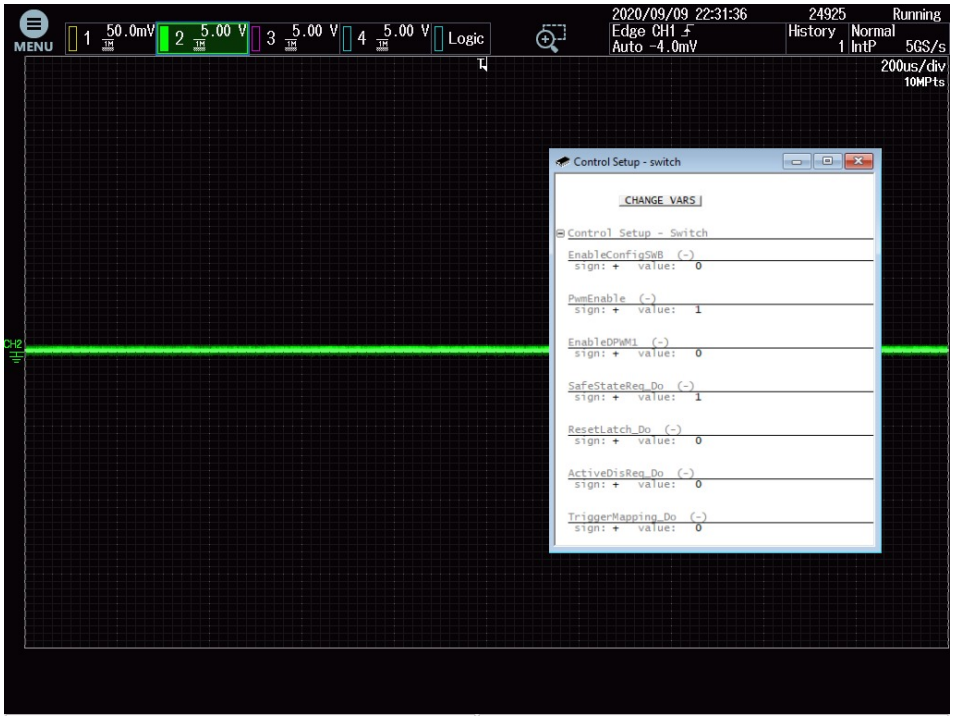

Figure 4.25: Screenshot of the acquisition of a digital output signal with negative polarity when the corresponding virtual button is set to 1.

After the requirements about digital pins were verified, test 3 was started to verify the digital parallel interface with the R2D converter reproducing the R2D data acquisition test described in section 4.6. As described by the test procedure, this was done enabling the data acquisition through the virtual button R2dDataEnabled. Therefore, the acquisition of the angular position, sensed by the resolver and converted by the R2D converter device, was done moving the resolver. It was connected to an external stepper motor maintaining its rotational speed constant and acquiring data with a frequency of 16kHz. The angular speed of the stepper motor connected to the resolver was set to be equal to about 850rpm. The variables R2dSpiData and R2dDigData were plotted using Trace32 with the feature Trace.Draw. Their time evolution are reported in Fig.4.29. As can be seen in the figure, the profile of R2dSpiData and R2dDigData have a triangular shape spanning from 0 to 4096 (2^{12}) with a period of about 70ms corresponding to about 850rpm. A further coherence check was performed in static condition, and in figure 4.30, the result is reported - the same value coming from the resolver-to-digital converter was acquired both through the digital and parallel interfaces. Therefore, the data acquisition from the R2D converter and the data coherence between the SPI and digital parallel acquisitions were successfully verified.

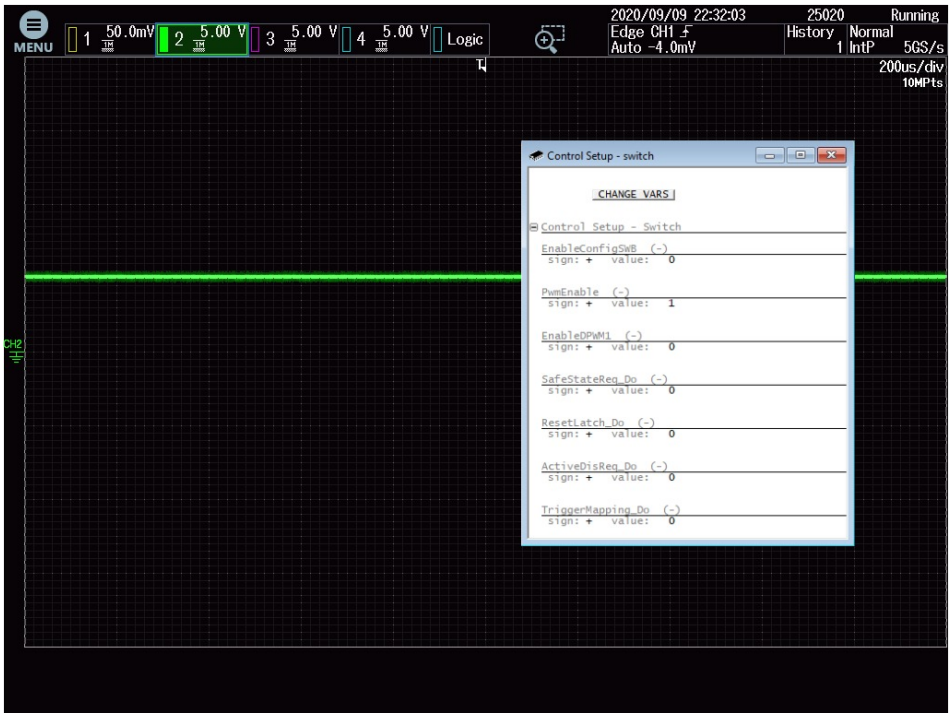

Figure 4.26: Screenshot of the acquisition of the digital output signal with negative polarity when the corresponding virtual button is set to 0.

4.8 CAN Communication Testing

The software component CAN Communication was tested to verify the correct configuration of the in-chip CAN peripheral used to allow the MCU to work as a node in the CAN network. The requirements of the CAN communication, described in section 3.7.1, were verified utilizing the Peak PCAN-USB and oscilloscope. The Peak PCAN-USB was fundamental to simulate the behaviour of the test bench with which the MCU will interact during the overall proof of concept tests. Moreover, this device allowed to easily analyze the messages transmitted through the bus. For the purpose, the following tests were executed:

1. **Periodic CAN messages sending**
   This test aims to send periodically messages over the CAN network. The physical connections were done in such a way to connect both the MCU node and the PCAN-USB node to the CAN bus. As already said, the PCAN-USB allowed to monitor the messages sent through the bus. This test allowed to verify the correct configuration of the bit timing segmentation for the FlexCAN peripherals, the sending of the TX messages, and the packaging of the variables into the data segment of the sent messages.

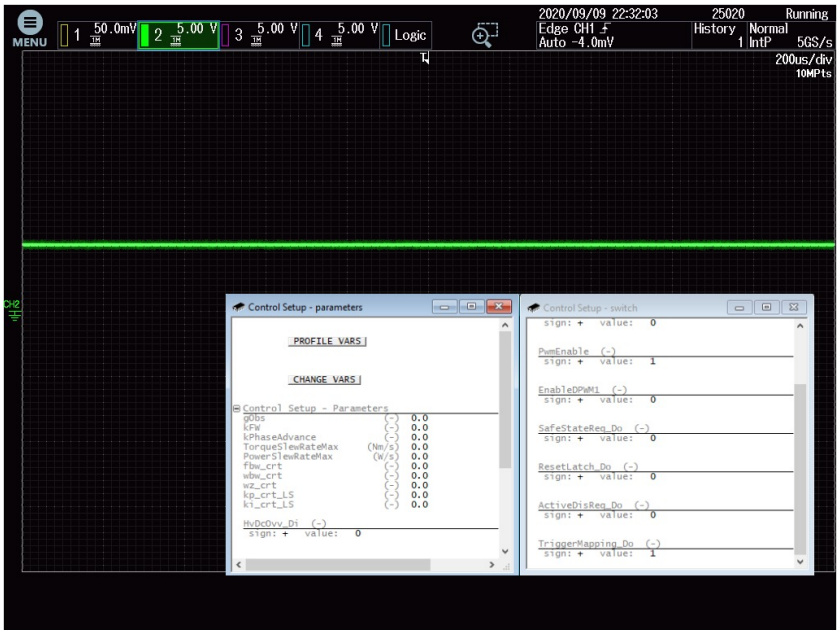

Figure 4.27: Screenshot of the acquisition of the digital input signal when the corresponding voltage level is set to 1.

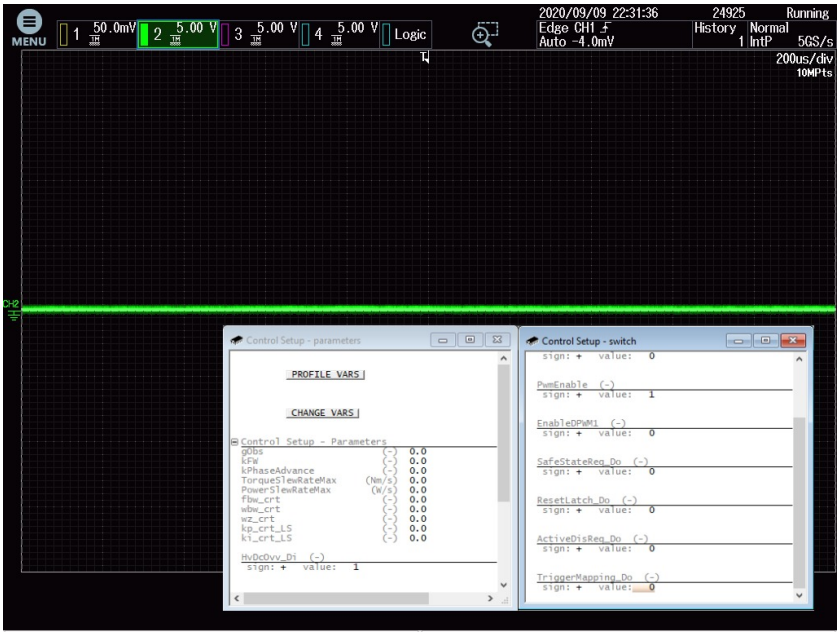

Figure 4.28: Screenshot of the acquisition of the digital input signal when the corresponding voltage level is set to 0.

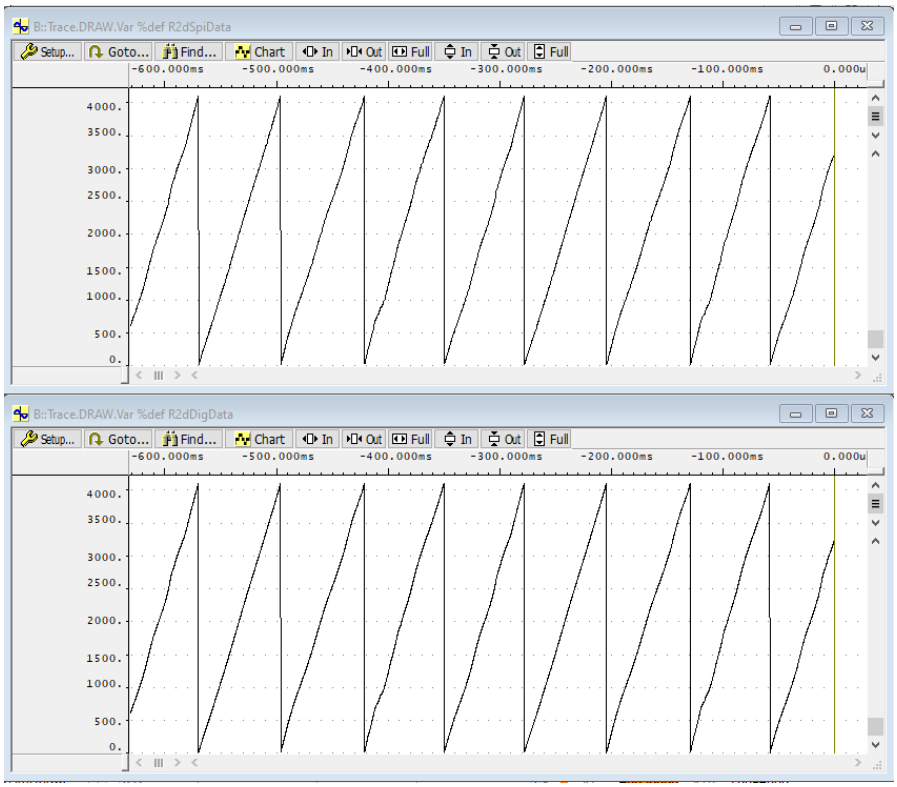

Figure 4.29: Time evolution of the variable R2dSpiData and R2dDigData acquired through the trace feature using the Lauterbach debugging tools.

2. **CAN messages sending disabling and enabling**
   The enabling and disabling features of the periodic sending of CAN messages was tested through a dedicated virtual button. The C functions of the API, that implement these features, were called by the test program with the same rate of the periodic task which will manage the CAN Communication in the application software.

3. **Periodic CAN messages receiving**
   This test aims to verify the correct reception of CAN messages. Once the correct peripheral configuration was verified through test 1, the PCAN-USB device was configured to send the CAN messages corresponding to the ones that the test bench will send during nominal operating conditions. As done in test 1, the receiving of the RX messages and the unpackaging of the message data segments into predefined variables were verified.

4. **Test bench environment simulation**
   Once all the implemented features were tested and the requirements were

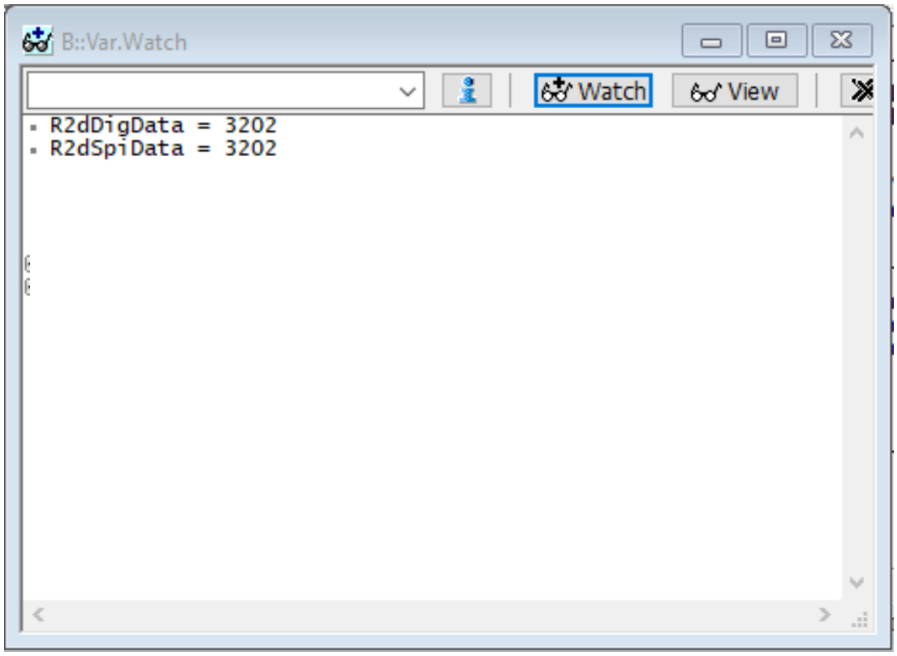

Figure 4.30: Values of the variables R2dSpiData and R2dDigData acquired respectively through the serial and digital ports of the resolver-to-digital converter.

verified, the operating conditions of the CAN network expected for the overall powertrain tests were simulated. In such conditions, both the test bench node, simulated by the PCAN-USB, and the MCU node send and receive data messages. Moreover, the PCAN-USB allowed to measure the bus load in these conditions.

The listed tests were considered sufficient to test all the requirements to be implemented about CAN communication and verify the correct MCU behaviour in conditions similar to the operating ones. The oscilloscope was only used to analyze the quality of the electric signals transmitted through the bus, but was not necessary to test the features implemented in the software component CAN Communication.

4.8.1 Test procedure

As done for other software components, only one test program was developed to implement the tests described in the previous section. Dedicated global variables, accessible through Trace32, were used to store the data of the received messages and the data to be sent. In such a way, the user can modify the variables to be sent over CAN network by the MCU in an automatic way using custom C functions or

manually by means of Trace32. A virtual button was used to enable or disable the periodic sending of CAN messages.

In order to reproduce conditions as close as possible to the nominal operating ones, a Periodic Interrupt Timer (PIT) was configured to manage the enabling and disabling features related to the periodic sending of CAN messages. The application software will interact with the software component CAN Communication through a periodic task executed with a frequency of 1kHz. For this reason, the PIT peripheral was configured to generate an interrupt each 1ms. The reception disabling function was not implemented because the feature was not listed in the requirements about CAN Communication. Therefore, in order to execute the test 1, the PCAN-USB was configured to not send any messages. These considerations allowed to make the test program less intrusive and less complex. The interrupt service routine of the PIT implemented for the test program is reported in the following:

Listing 4.10: Interrupt service routine for the test program of CAN Communication.

```
void PIT_0_Handler(void)
{
    /* Peripheral interrupt flags check and clear */
    ..........................

    /* Enable/Disable periodical sending of CAN messages */
    if (CanSendEnabled) {
        CanCom_SendEnable();
    } else {
        CanCom_SendDisable();
    }
}
```

As already seen in the other test programs, the function PIT_0_Handler() is the ISR associated to the interrupt periodically generated by the hardware instance 0 of the PIT peripherals. The variable CanSendEnabled is a global variable which acts as virtual button to enable or disable the periodic sending of TX messages over the CAN network. The global data structures CanTxDataMsg01, CanTxDataMsg02, CanRxDataMsg03, and CanRxDataMsg04, described in section 3.7.3, were used to store the data frames of the received messages and to set the data to be sent by the MCU node.

In order to perform all the tests listed in the previous section with this simple test program, a precise test procedure shall be followed. The test procedure is described by the following steps:

1. Connect the PCAN-USB and target platform to the CAN bus and the Lauterbach Debug and Trace to the debug and trace ports;
2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32 software;
3. Open the PCAN-View software to monitor the CAN bus and interact with the PCAN-USB;
4. Before to start the test program, use the variable CanSendEnabled to enable the periodic sending of CAN messages by the MCU node;
5. Start the test program;
6. Use the PCAN-USB to acquire the messages sent by the MCU and verify that no errors occurred on the CAN bus - the message acquisition allows to monitor the data sent by the MCU node and the mean time distance between their sending (called cycle time in PCAN-View software);
7. Repeat point 6 setting different values for the data structures CanTxDataMsg01 and CanTxDataMsg02 - steps 6 and 7 allows to completely execute test 1;
8. Change the value of the variable CanSendEnable at run-time to disable periodic sending of CAN messages by MCU node - this step contributes to execute test 2;
9. Use the PCAN-USB to send the messages to the MCU node and verify that the messages are correctly received - the PCAN-View software shall be used to send messages according to the corresponding requirements;
10. Repeat step 9 setting different values in the data sent by the PCAN-USB and verify the correspondence with the data structures CanRxDataMsg03 and CanRxDataMsg04 - steps 9 and 10 allows to completely execute test 3;
11. Change the value of the variable CanSendEnable at run-time to re-enable periodic sending of CAN messages from MCU side - this step contributes to completely executes test 2;
12. Repeat simultaneously points 6 and 9 changing the data sent both from MCU and PCAN-USB sides to simulate an environment as close as possible to the one reproducing the conditions of the overall powertrain tests and monitor the bus load using PCAN-View - this step allows to perform test 4;
13. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

In order to be more confident about the electrical signals transmitted by the MCU over the CAN network, a further optional step can be performed: it consists in observing the electrical signals transmitted over the CAN bus by means of an oscilloscope. This step is not fundamental in case all the tests are passed, but can be useful as debugging approach or redundant check.

4.8.2 Results

The test procedure described in the previous section about CAN Communication was performed on the target platform. The results of test 1 are reported in Fig. 4.31. More precisely, in the figure, it is shown the correspondence between the data sent by the MCU node and the ones received by the PCAN-USB node. The software PCAN-View shows that the bus status is "OK". The detected bit-rate is 1Mbps and it is correct configured according to the requirements described in section 3.7.1. In the Receive/Transmit window, the messages with IDs 04 and 01 corresponds respectively to the TX messages 01 and 02 defined in the CAN database. Their measured cycle times (mean time distance between the occurrence of two consecutive messages with same ID) corresponds to 50ms for TX message 01 and 10ms for TX message 02. There values are exactly equal to the nominal ones which are respectively 50ms and 10ms. Therefore, test 1 was considered passed.

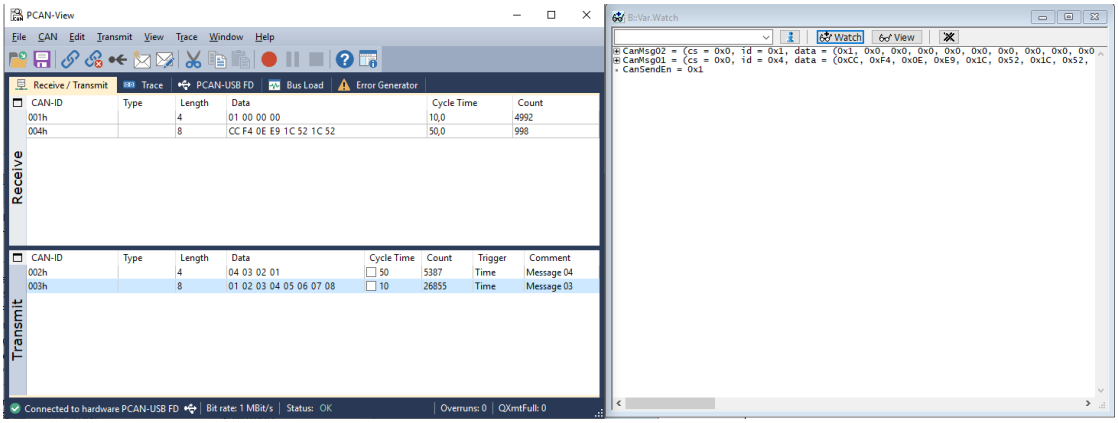

Figure 4.31: Screenshot representing the correspondence between the messages sent by the MCU node and the ones acquired by the PCAN-USB node.

Step 6 of the test procedure, which consists in disabling the periodic sending of TX messages, was verified observing the value of "Count" in the Receive/Transmit window of PCAN-View software. Indeed, the value of "Count" stops to increase once the value of the virtual button CanSendEnable was changed.

Before to start step 7 of the step procedure, corresponding to the execution of test 3, the Receive/Transmit window was restored to highlight only the messages to

be received from MCU node. The two RX messages were configured in PCAN-View software to be sent by means of the PCAN-USB device. Their IDs, transmission time rates, and data frame lengths were set according to the defined CAN database. After being configured, the message sending from the PCAN-USB point of view was started and the correspondence between the data frames sent and the ones received by the MCU checked. As suggested by the test procedure, the test was repeated with different data frames changed at run-time. The results are reported in Fig. 4.32, and as can be seen by the figure, the receiving of CAN messages worked as expected. Therefore, the test was considered passed.

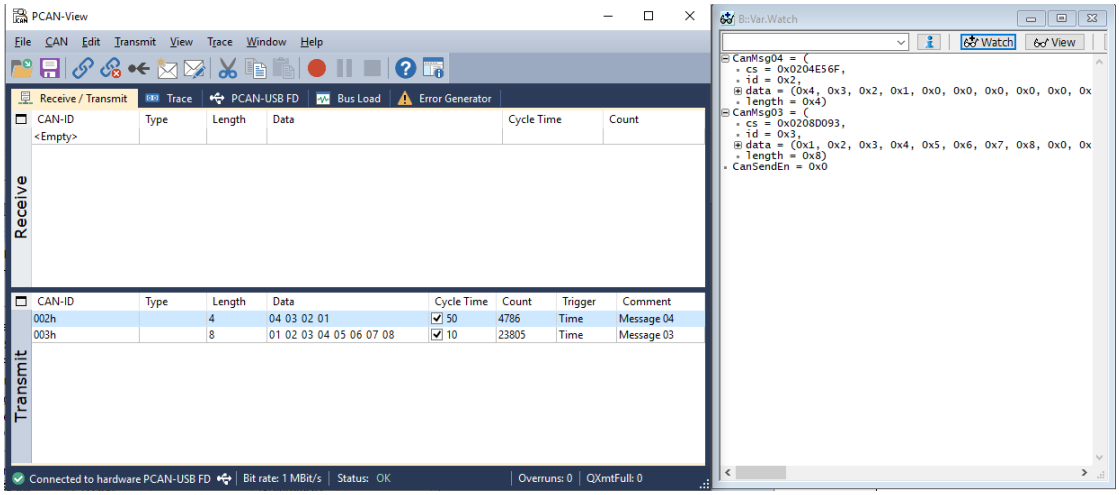

Figure 4.32: Screenshot representing the correspondence between the messages sent by the PCAN-USB node and the ones acquired by the MCU node.

Then, as done previously, the Receive/Transmit windows was restored maintaining the same settings for the RX messages 03 and 04. Test 4 was started re-enabling the periodic sending of the CAN messages both by MCU and PCAN-USB nodes. Also in this scenario, the correspondence between the sent data frames and the received ones was maintained. Indeed, the test was considered passed and the results are reported in Fig. 4.33.

As already explained, these conditions are as close as possible to the nominal operating ones and this scenario simulates the behaviour of the system when it will be connected to the external test bench through CAN network. The bus load was measured by means of the PCAN-View software to estimate the traffic over the network in nominal operating conditions. The results are reported in Fig. 4.34. The measured mean bus load is equal to 2.4%, and as expected, it is very low. This means that the CAN database designed for the network can be implemented in the system without any issues.

After all the tests were successfully executed, the electrical signals transmitted

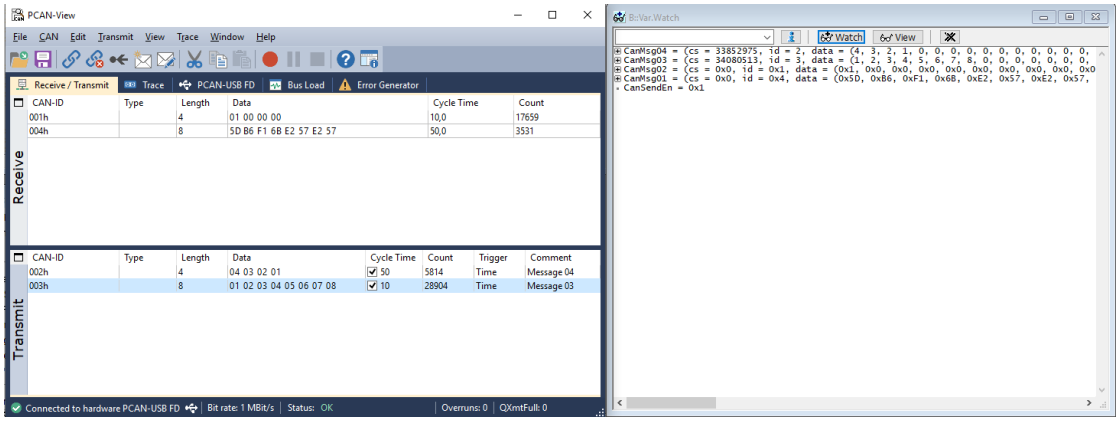

Figure 4.33: Screenshot representing the correspondence between the messages exchanged between the PCAN-USB node and the MCU node.

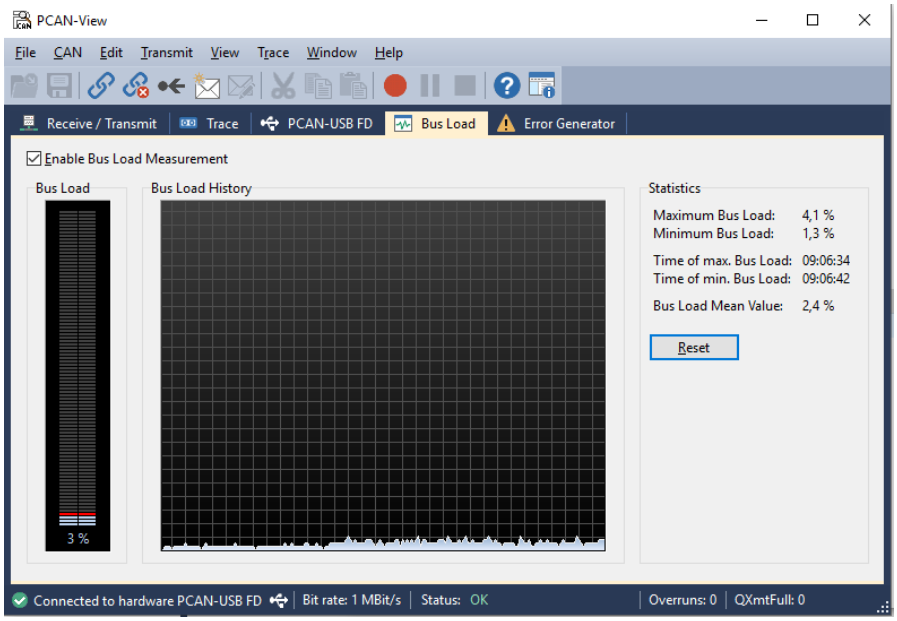

Figure 4.34: Measurement of the bus load in conditions as close as possible to the nominal operating ones.

through the CAN bus were analyzed using the oscilloscope to perform a redundant check about the sent messages and verify their integrity. A screenshot of the acquired signals is reported in Fig. 4.35.

Figure 4.35: Screenshot of the acquisition of the TX message 02 sent by the MCU node.

Chapter 5

Software Integration Process

System integration is defined in engineering as the process of bringing together the component sub-systems into one system (an aggregation of subsystems cooperating so that the system is able to deliver the overarching functionality) and ensuring that the subsystems function together as a system, and in information technology as the process of linking together different computing systems and software applications physically or functionally, to act as a coordinated whole[8].

In the context of this thesis, the whole software is composed by the different software components, which constitute the firmware, and the application software obtained through a code generation process in Matlab/Simulink environment. The first step was to integrate together the software components constituting the low-level software. Then, the code generated from the application software was run on the target platform to check the presence of differences with respect to the simulated environment. Lastly, the integration of these two software layers was performed. In detail, the overall integration process was constituted by the steps listed in the following:

1. **Software components integration**
   The software components composing the firmware were tested one at a time as described in chapter 4. This approach was motivated by the absence of communication between the different peripheral drivers with the only exception of Pin Management, which functions are called by the firmware itself to configure the related MCU pins. For this reasons, before to perform the integration between the firmware and application software, it was necessary to integrate all the software components together to verify that all their features continue to work according to their requirements in such conditions. This integration process is described in section 5.1 and the previously developed

test programs were reused to implement the required tests.

2. **Processor-in-the-Loop simulation of application software**
   The application software was developed through a model-based approach. Therefore, its behaviour was verified and validated in simulation. Once its code was generated through Embedded Coder in Matlab/Simulink environment, a further verification step was required to check that no errors occurred in the code generation process. Indeed, the application software went through software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. In this thesis, only the PIL simulation was analyzed. It was fundamental not only to validate the behaviour of the application software, but also to demonstrate the feasibility of its implementation into the specific target platform measuring its worst case execution time (WCET).

3. **Firmware and application software integration**
   Once both the application and firmware were verified separately as whole, a final integration process was performed. In detail, a new software component was developed to allow the application software to access to the firmware global variables through specific get and set functions. As it will be described in section 5.3, these functions are also responsible to map the firmware variables into variables with the correct unit measurements required by the application and vice-versa. Then, a dedicated environment was developed to run the tasks representing the motor control algorithm of the application on the low-level software layer.

The integration process was carried on using the same laboratory instrumentation described in section 4.2. Furthermore, the trace features provided by Lauterbach Trace32 were fundamental to compute a statistical figure about the worst case execution time of the task related to the application software.

5.1 Software Components Integration

The software components, constituting the firmware of the target platform, were developed following a modular approach. Indeed, as described in chapter 3, the firmware architecture was structured grouping together similar features which use the same peripherals. As result of this approach, each software component was developed and tested maintaining its independence with respect to the others. Therefore, at this point of the development, an integration process was required before to prepare the software environment that will host the application software.

Since all the software components were already developed and their functions verified, the integration process was strictly related to repeat the already executed tests merging the different test programs in a single executable file. In such a way, all the tests can be repeated to verify if the different peripherals and features interfere each other when they work simultaneously.

5.1.1 Initialization and Startup Sequence

To integrate all the software components, all the initialization functions shall be called before to start test programs or the application software. The same initialization sequence of section 4.1.1 was used to prepare the MCU to run a program that can use all the functions provided by the firmware. The resulting code, used to implement the described initialization sequence, is the following:

Listing 5.1: Initialization and startup sequence of all the software components.

```
int main(void)
{
    status_T l_err = status_success;

    /* Disable global interrupts */
    INT_SYS_DisableIRQGlobal();

    /* Clock initialization */
    l_err |= ClkMgm_Init();

    /* SWC initialization */
    DigSgnMgm_Init();
    l_err |= PwmMgm_Init();
    l_err |= AdcMgm_Init();
    l_err |= SpiCom_Init();
    l_err |= CanCom_Init();
```

```
    /* Initialization check */
    while(l_err != status_success){}

    /* Enable global interrupts */
    INT_SYS_EnableIRQGlobal();

    /* Startup sequence R2D */
    l_err |= SpiCom_R2dSetCfg(R2dCfg);

    /* Startup sequences check */
    while(l_err != status_success){}

    ..............................
}
```

As can be seen above, the global interrupts were disabled during the initialization procedure to avoid interruptions. As already discussed in previous chapters, the MCU clock signals and the clock gating of the used peripherals is configured as first operation through the function ClkMgm_Init(). Then, the software components and the corresponding peripherals are initialized with their corresponding functions and a check is performed to verify the correct initialization. It is important to underline that the software component PWM Management shall be initialized before ADC Management to start the eMIOS timebase required to trigger the analog-to-digital conversions. The only startup sequence corresponds to the initialization and configuration of the resolver-to-digital converter. It is performed after the enabling of global interrupts through the its dedicated function - since it works using interrupts, the function responsible to configure the R2D converter is called with interrupts enabled.

This initialization sequence was tested without noticing any particular issues. Therefore, it was used as starting point to develop a software environment including all the software components.

5.1.2 Software Components Integration Test

The main purpose of the integration process at firmware-level was to merge together all the software components which were developed and tested separately. Indeed, the behaviour of all their functions and features was checked to verify that they are working correctly when considered as a whole. This process leads to a further development of the environment that will be used to run the application software. More precisely, a new test program was developed combining the test routines

described in chapter 4 which were used to perform a verification process about the single software components.

The test program was developed using a periodic task to perform the required actions according to the application software requirements. As done previously, a Periodic Interrupt Timer (PIT) was configured to generate periodic interrupt with the same rate (16kHz) of the one required by the application software. In the corresponding interrupt service routine (ISR), the programs previously developed as independent routines were merged. Their virtual buttons and control variables were maintained to affect the test program execution flow and interact with the target platform performing the required steps of the test procedure described in this section. The source code used for the integration process was reported in the following:

Listing 5.2: Interrupt service routine for the integration test program.

```
void PIT_0_Handler(void)
{
    /* Peripheral interrupt flags check and clear */
    ..........................

    /* Enable/Disable PWM signal outputs */
    if (PwmEnabled)
    {
        PwmMgm_OutEnable();
        PwmMgm_SetDuty(Duty_a, Duty_b, Duty_c);
    }
    else
    {
        PwmMgm_OutDisable();
    }
    /* Set the digital output signals and get the digital
       input signals */
    if(DigSgnEnabled)
    {
        DigSgnMgm_SetSignalName1(Out_SignalNameValue1);
        ..........................................

        In_SignalNameValue1  = DigSgnMgm_GetSignalName1();
        ..........................................
```

```
        /* Acquire data from the R2D converter through the
           digital parallel port */
        R2dDataErrorDig = DigSgnMgm_R2dReadData();
    }
    /* Acquire data from the R2D converter through SPI */
    if(R2dDataEnabled)
    {
        R2dDataErrorSpi = SpiCom_R2dReqData();
    }
    /* Enable/Disable periodical sending of CAN messages */
    if (CanSendEnabled)
    {
        CanCom_SendEnable();
    } else
    {
        CanCom_SendDisable();
    }
}
```

As described in chapter 4, the function PIT_0_Handler() is the interrupt handler responsible to run the ISR associated to the peripheral PIT0. Therefore, all the tests designed and executed on the single software components can be repeated using this service routine where the previously analyzed test programs are merged. The only exception are some tests about the SPI Communication (generic data sending and receiving) which can't be executed. Despite this, the peripheral LPSPI2 associated to the communication with resolver-to-digital converter can be tested anyway through the functions SpiCom_R2dSetCfg() and SpiCom_R2dDataReq() which respectively set the required configuration in the device and read the data through SPI. All the global variables added in this program and used to interact with the firmware are used following the same approach of the previous tests. More precisely, the virtual buttons PwmEnabled, R2dDataEnabled, and CanSendEnabled are used in the same way discussed in the sections describing the corresponding software component testing. Furthermore, a new virtual button called DigSgnEnabled was added to enable or disable the interaction with the digital input and output signals. It is important to notice that the acquisition of the data through the digital parallel port of the R2D convert is enabled with the new virtual button DigSgnEnabled differently from the test described in section 4.6 where R2dDataEnabled is used.

In order to test the correct integration of the software components, a test procedure was developed. Therefore, the following steps shall be executed to successfully test the integration:

1. Connect Lauterbach Debug and Trace to the target platform;

2. Download the test program from the version control system, build the project and download the executable file into the MCU flash memory using Trace32;
3. Disable all the virtual buttons and perform the test procedure about ADC Management described in section 4.5.1;
4. Disable all the virtual buttons except for PwmEnabled to perform the test procedure about PWM Management described in section 4.4.1;
5. Disable all the virtual buttons except for DigSgnEnabled to perform the test procedure about Digital Management described in section 4.7.1;
6. Disable all the virtual buttons except for R2dDataEnabled to perform the test procedure about SPI Communication described in section 4.6.1 considering only test 4;
7. Disable all the virtual buttons except for DigSgnEnabled and R2dDataEnabled to check the coherence between the acquired data from the resolver-to-digital converter;
8. Disable all the virtual buttons except for CanEnabled and perform the test procedure about CAN Communication described in section 4.8.1;
9. Enable all the virtual buttons and repeat the previous tests to verify if all the features are working according to their requirements when all the software components are stressed;
10. Compare the acquired results to the expected ones and determine if the tests were passed or failed.

As can be noticed by the test procedure, new tests were not developed to perform the verification of the firmware integration process. This procedure was performed on the target platform aiming to detect any bugs related to the simultaneous execution of the different software components. During the test, no problems were found obtaining results close to the one described in chapter 4.

5.2 Processor-in-the-Loop Simulation

Processor-in-the-Loop (PIL) is a test technique that allows designers to evaluate if the embedded software runs properly when compiled and running on the chosen MCU, from the computation results point of view. PIL tests are designed to expose problems with execution in the embedded environment.

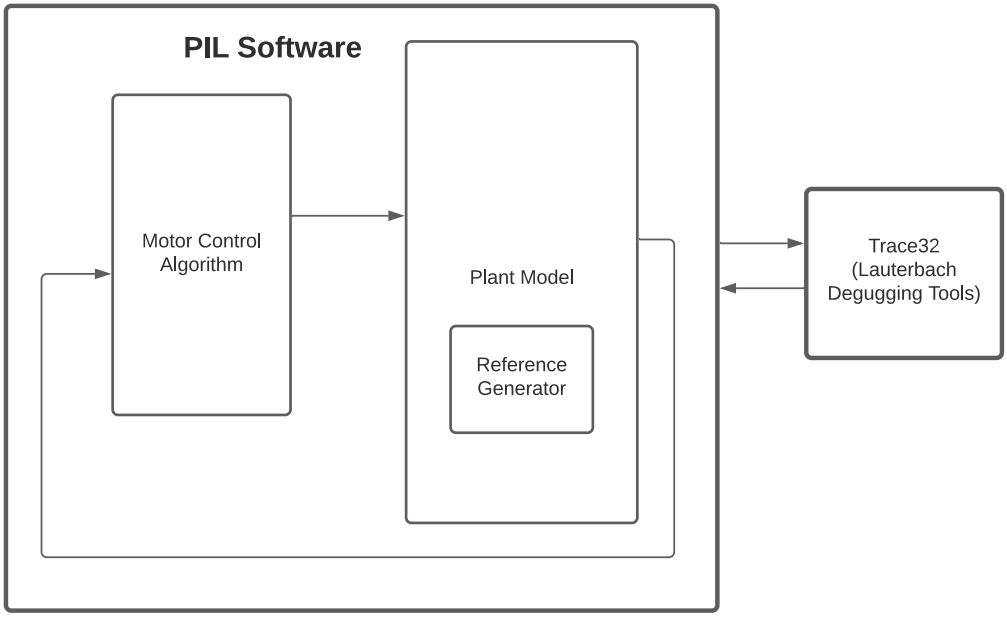

Figure 5.1: Block scheme of the processor-in-the-loop simulation of the task representing the motor control algorithm of the application software implemented on the target platform.

In the PIL simulation performed in this thesis, the plant model, developed in Matlab/Simulink environment, was integrated and run in the target platform. More precisely, the source code about the task related to the motor control algorithm and the plant model were generated through automatic code generation procedures. Then, they were integrated and executed in the target platform hosting the MCU. In this way, the plant model was simulated directly on the embedded hardware without using any external computer. Indeed, the time evolution of its state and output variables was computed on the base of the control action of the motor control algorithm which outputs correspond to the plant inputs. The control actions of the motor control algorithm were computed starting from the plant outputs and reference signals generated by a suitable generator which simulates the test bench. From practical point of view, this generator was obtained through the automatic code generation process performed on the plant model. Of course, since running the plant model of the e-powertrain creates a delay in the overall

software execution, this type of simulation is not performed in real-time. Moreover, the considered task of the application software interacts with the simulated plant without using the physical in-chip peripherals and/or in-board devices of the target platform. Therefore, this simulation can be only used to verify the correct functional behaviour of the source code generated through automatic code generation tools when it is run on the target platform. A block scheme about the implementation of the described processor-in-the-loop simulation is reported in Fig. 5.1.

For our specific application, unusually, PIL was fundamental to verify the feasibility of the implementation of the motor control algorithm on the chosen target platform. More precisely, since the task related to the motor control algorithm shall be executed in a periodic fashion (62.5μs) with hard real-time constraints, its worst case execution time (WCET) must be sufficient small to guarantee no deadline misses. It is important to remember that a deadline miss occurs when the considered task instance is not terminated before the activation of the next instance, indeed when its execution time is bigger than its period.

5.2.1 Processor-in-the-Loop Implementation

A software environment was developed to implement the describe PIL simulation. As explained in the previous section, both the plant model and the application software were obtained through automatic code generation and executed on the target platform. Therefore, the PIL simulation was developed through the following source code:

Listing 5.3: Structure of the implementation of the PIL simulation.

```
int  main(void)
{
    status_T l_err = status_success;

    /* Disable global interrupts */
    INT_SYS_DisableIRQGlobal();

    /* Clock initialization */
    l_err |= ClkMgm_Init();

    /* Initialization check */
    while(l_err != status_success){}

    /* Models initialization */
    Plant_Initialize();
```

```
    Control_Initialize();

    /* Run PIL simulation */
    for (int i = 0; i <= 160000 * SIM_LENGTH; i++)
    {
        /* Call the step function of the plant */
        Plant_step();

        /* Set control inputs */
        Control_U.sig1 = Plant_Y.sig1;
        Control_U.sig2 = Plant_Y.sig2;
        ..........................

        /* Call the step function of motor control
           algorithm with the correct rate */
        if (i % 10 == 1)
        {
            Control_step();
        }

        /* Set plant inputs (control actuation)*/
        Plant_U.sig1 = Control_Y.sig1;
        Plant_U.sig2 = Control_Y.sig2;
        ..........................
    }

    while(1){}

    return 0;
}
```

As can be seen in the code, only the software component Clock Management is initialized to perform the PIL simulation. Indeed, since this test runs the motor control algorithm of the application software interacting with the simulated plant, no peripherals are needed to execute the simulation. For the same reasons, global interrupts were not enabled to avoid interruptions during the program execution. The functions Plant_Initialize() and Control_Initialize(), obtained from the corresponding source files of the generated code, are called before to start the test. Then, the plant model and the task of the application software are run. More precisely, the plant is simulated through the function Plant_step() and the application software through the function Control_step(). Before to call

these functions, the corresponding input data structures (Plant_U and Control_U) are written with the required values on the base of the block diagram of Fig. 5.1 that describes this implementation. More precisely, since the implemented model simulates a closed-loop system, the input data structure of the controller is written with the output data structure of the plant and vice-versa.

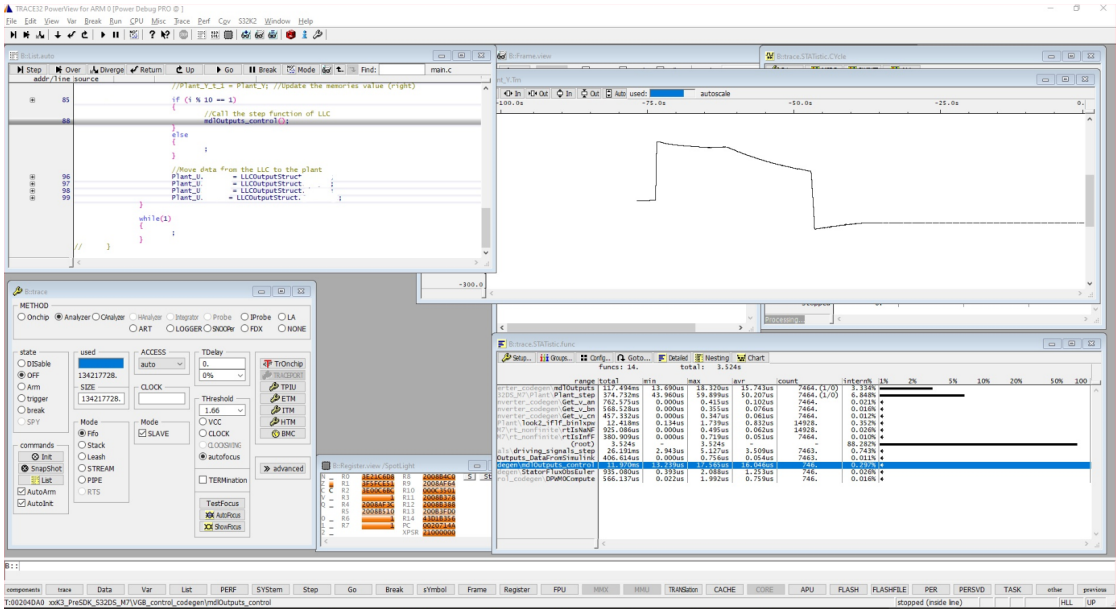

Figure 5.2: Lauterbach Trace32 environment used to perform the PIL simulation. In the figure, the output data structure of the application software, corresponding to data structure Control_Y, is called LlcOutputStruct.

It is important to notice that the two step functions simulating the plant and the controller are run at different rates. Indeed, the plant model is run 10 times faster than the task of the application software. Therefore, since the execution rate of the application software was chosen equal to the demonstrative frequency of 16kHz, the plant model was discretized to be simulated at 160kHz. The choice to run the plant model at a faster rate is motivated by the physical characteristics of the system which requires smaller time discretization to be correctly simulated.

5.2.2 Processor-in-the-loop Results

The results obtained by this test were compared with the ones related to the previously performed software-in-the-loop (SIL) simulation (SIL simulation was not reported because outside the purposes of this thesis) to identify any differences in the functional behavior of the controller.

From a control point of view, the purpose of the application software is to control the powertrain to produce the right torque according to its reference signal. Therefore, the time evolution of the torque produced by the model of the e-powertrain, representing the controlled output variable of the system, was used to verify the functional behaviour of the generated code of the application software running on the target platform. The results of the PIL and SIL simulations are reported respectively in Fig. 5.3 and 5.4. As can be seen in the figures, there

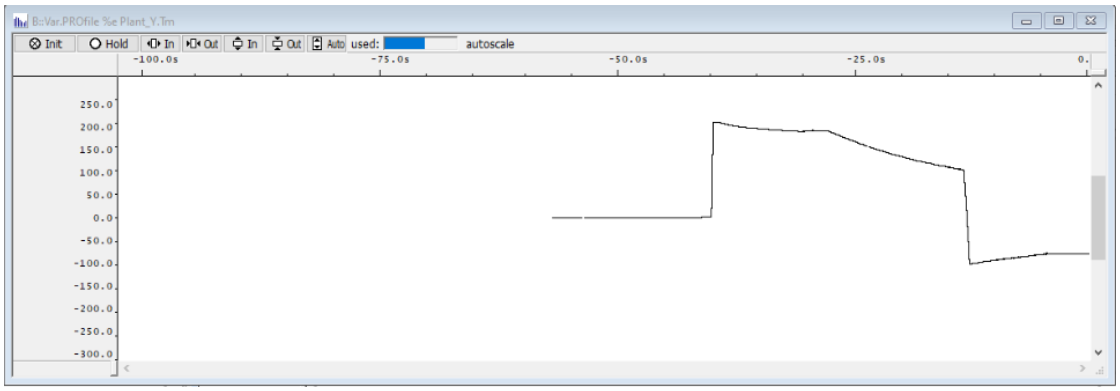

Figure 5.3: Time evolution of the controlled variable of the plant obtained by PIL simulation.

is a correspondence between the data obtained by the SIL and PIL simulations. Indeed, these results validate the behaviour of the application software when it is run on the target platform.

In order to verify that the MCU provides enough computation power to run the application software according to its real-time requirements, the WCET was computed using the feature Trace.STATistic.func provided by the Lauterbach debugging tools. A statistical analysis of the execution time of the step function of the application software is reported in Fig. 5.5 where the function mdlOutputs_control(), highlighted in blue, corresponds to the step function of the application software previously called Control_step() - its average and maximum execution times are respectively 16.046μs and 17.565μs. Indeed, from this empirical analysis, it can be demonstrated that the implementation of the control task, obtained by the code generation process performed on the application software, is feasible. It is important to underline that the computed maximum execution time doesn't correspond exactly to the real WCET, but it is a reasonable estimation of its value. Furthermore, the computed maximum execution time of the function mdlOutputs_control() is much lower than the nominal period of the control task allowing to be comfortable about its feasibility due to the margin of tolerance about this estimation.

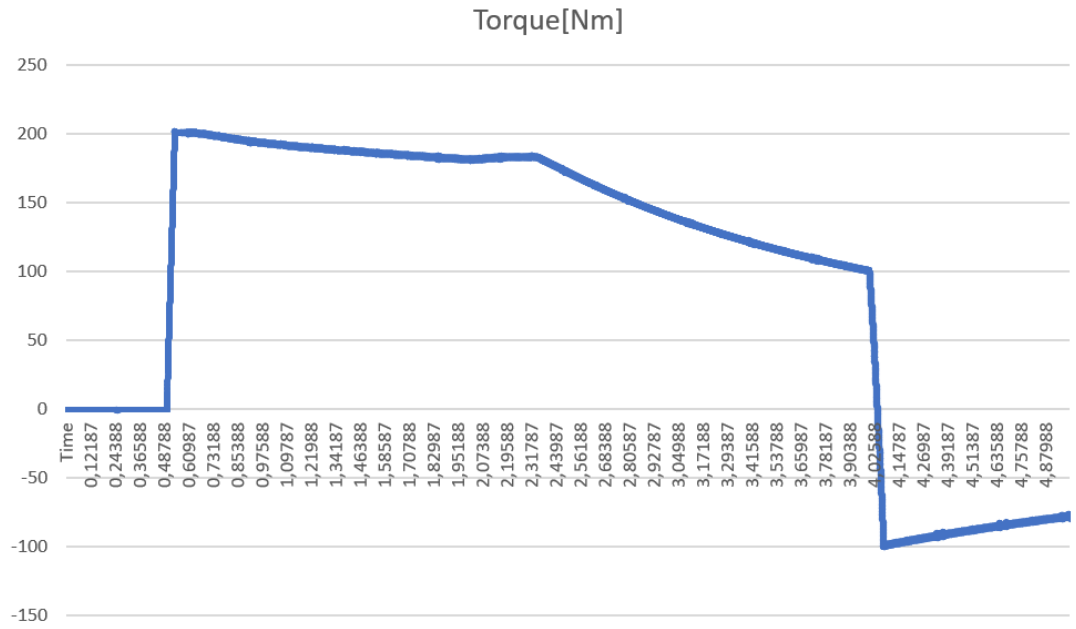

Figure 5.4: Time evolution of the controlled variable of the plant obtained by SIL simulation.

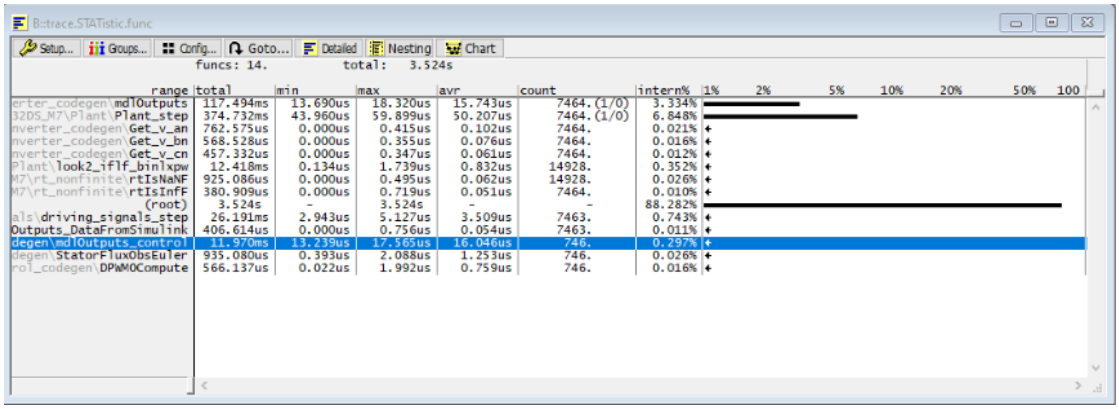

Figure 5.5: Statistical figure about the execution time of the control motor task of the application software.

5.3 Firmware and Application Integration

The final step of the software integration process is represented by the developing of a dedicated environment to run the tasks, which constitute the application, on the low-level software layer. The executable file obtained by this process is the

software that will be downloaded in the target platform to perform the tests on the overall system in its dedicated testing environment. In Fig. 5.6, it is reported the architecture of the overall software including the device drivers provided by the silicon-vendor, the firmware constituted by the developed software components, and the application software obtained through an automatic code generation process. The software integration process was carried forward on the base of this software architecture.

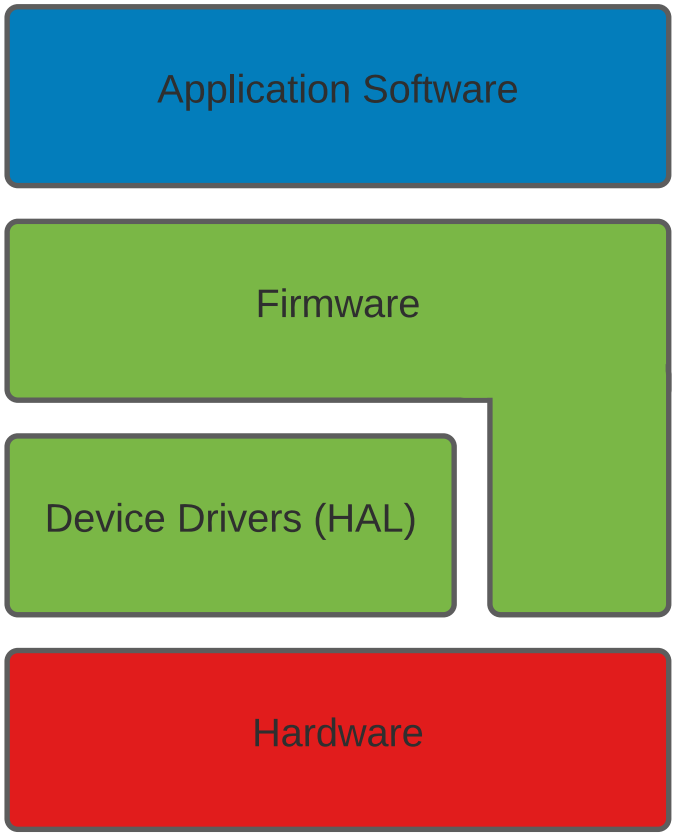

Figure 5.6: Software architecture including the device drivers provided by the silicon-vendor, the firmware constituted by the developed software components, and the application software obtained through automatic code generation process.

5.3.1 Software Environment Initialization

The software environment, developed to initialize all the software components providing the features required by the application software and perform the startup procedure, is described in section 5.1. This environment was fundamental to prepare the target platform to run the control task. Furthermore, the steps required to

initialize the application software, obtained by a code generation process, were tested during the execution of the processor-in-the-loop simulation described in section 5.2. Therefore, the two source codes were integrated to obtain a complete initialization of the software components, application software, and target platform devices. The resulting code, used for the purpose, is described in the following:

Listing 5.4: Initialization and startup sequence of all the software components and application software.

```
int main(void)
{
    status_T l_err = status_success;

    /* Disable global interrupts */
    INT_SYS_DisableIRQGlobal();

    /* Clock initialization */
    l_err |= ClkMgm_Init();

    /* SWC initialization */
    DigSgnMgm_Init();
    l_err |= PwmMgm_Init();
    l_err |= AdcMgm_Init();
    l_err |= SpiCom_Init();
    l_err |= CanCom_Init();

    /* Initialization check */
    while(l_err != status_success){}

    /* Enable global interrupts */
    INT_SYS_EnableIRQGlobal();

    /* Startup sequence R2D */
    l_err |= SpiCom_R2dSetCfg(R2dCfg);

    /* Startup sequences check */
    while(l_err != status_success){}

    /* Models initialization */
    Control_Initialize();
    Safety_Initialize();
```

```
    /* Start periodic timer */
    Scheduler_Start();

    /* Infinite loop */
    while(1){}
}
```

As can be seen in the source code, the initialization function Control_Initialize() of the application software was added to integrated code described in section 5.1. The function Safety_Initialize is responsible to initialize the safety and fault management task obtained by the application software. The function Scheduler_Start() is responsible to initialize the periodic timer used to start the execution of the step functions of the application software. This function is defined in the software component Scheduler which was specifically developed with the purpose to manage the periodic execution of the application and its communication with the firmware. The software component Scheduler is described in the following section.

5.3.2 Software Component Scheduler

As already explained, the software component Scheduler was specifically developed with the purpose to manage the periodic execution of the application software and its communication with the firmware software components. Therefore, the main requirements about its behaviour are the following:

1. Run the motor control algorithm with the chosen rate of 16kHz in a synchronous manner with respect to the PWM switching period and analog signal acquisition, and the safety and fault management task with its nominal rate of 1kHz - both tasks were obtained through the automatic code generation performed on the application software;

2. Implement the required signal scaling and mapping of the variable which stores the acquired quantities coming from the plant through the target platform;

3. Perform the actuation of the control actions required by the control algorithm and safety and fault management logic implemented through the developed firmware features.

Starting from the first requirement, a periodic interrupt, synchronous with respect to beginning of the switching period of the application software was configured to run the motor control algorithm. More precisely, the interrupt service routine associated to the peripheral eMIOS0, used to implement the PWM generation, was utilized to run the considered task. Indeed, the channel 0 of the eMIOS0,

responsible to generate the timebase for the PWM generation, was configured to generate an interrupt in the occurrence of the so-called reload event corresponding to the beginning of the switching period. In Fig. 5.7, a graphical representation of the instants where the application software shall be start (task activation) and terminated (task deadline) and the analog acquisition are triggered is reported. Therefore, the control algorithm of the application software will be run on the base of the data acquired at the center of the T_{ON} time interval of the previous switching period having available 62.5μs to complete its execution and actuate its control action before the activation of the next task instance.

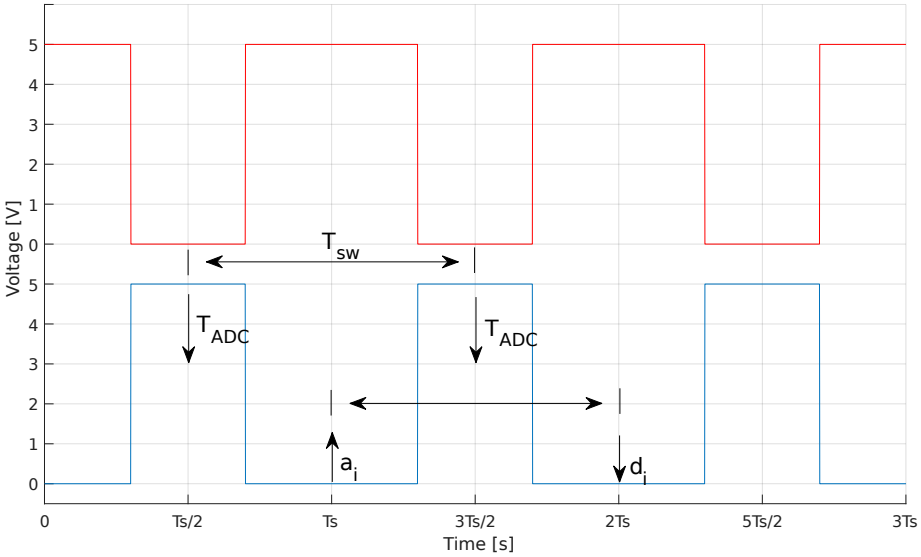

Figure 5.7: Graphical representation of the time instants related to the software application execution and firmware acquisitions.

A second periodic interrupt was configured to run the safety and fault management task of the application software. It was implemented using a periodic interrupt timer (PIT) configured to generate an interrupt every 1ms. The scheduling approach used to run the two task is based on the rate-monotonic algorithm. Rate-monotonic scheduling (RMS) is a priority assignment algorithm used in real-time operating systems with a static-priority scheduling class, where the static priorities are assigned according to the cycle duration of the task, so a shorter cycle duration results in a higher task priority [9]. Therefore, the priority of the safety and fault management task was set lower than the one corresponding to the motor control task, which means that the PIT interrupt priority was set lower than the eMIOS0 one through the NVIC core peripheral. Using this approach, real-time constraints were of the considered tasks were respected. An example of timeline

for the described algorithm is represented in Fig. 5.8

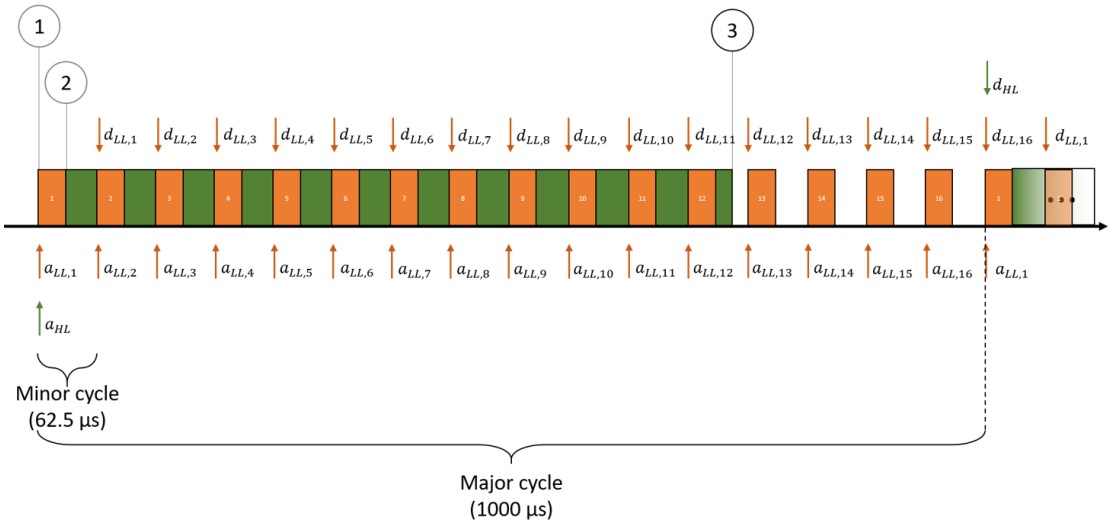

Figure 5.8: Timeline of the scheduling algorithm implemented through the software component Scheduler.

As can be seen in the figure, every time the motor control algorithm task needs to be executed and the safety and fault management is running, a preemption occurs. It is important to remember that preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such changes of the executed task are known as context switches[10].

In order to implement the correct data exchange between the considered tasks, two other data structures were created. The first one, called Control2SafetyData, was used to store the data to be transferred from the from the control task to the safety and fault management task and the second one, called Safety2ControlData, to store the data to be transferred in the opposite way. Moreover, to avoid concurrent access to the data structures the following rules were implemented:

- The control algorithm task transfers the data from its output data structure to the data structure Control2SafetyData at the end of its instances;
- The safety and fault management task transfers the data from its output data structure to the data structure Safety2ControlData at the end of its instances;
- The input data structure of the control algorithm is updated with the data coming from the firmware at the beginning of each instance and with the data

coming from the safety and fault management task at the beginning of each 16^{th} instances (corresponding to time instances a_{LL1} in figure 5.8);

- The input data structure of the safety and fault management task is updated with the data coming from the firmware and with the data coming from the control task at the beginning of each instance (corresponding to time instances a_{HLi} in figure 5.8) - during this operation the interrupt are disable to avoid multiple concurrent access to the data structure Control2SafetyData possible due to preemption.

The corresponding interrupt service routines (ISR), used to run the considered tasks, were used to perform the same steps described in section 5.2 related to the execution the motor control algorithm. In a formal way, the following steps shall be performed:

1. **Upload the task input structure**
   Before to run a periodic task obtained from the application software, its input data structure shall be uploaded with the data acquired by the target platform through the features implemented by the firmware.

2. **Run the task**
   Once the input data structure of the considered task is upload with the last acquired data, its step function is run. It is important to remember that the step function is the code representing the operations performed by the application software obtained through the automatic code generation process.

3. **Actuate the control action**
   Terminated the execution of the step function of the application software, its uploaded output data structure is used to actuate the corresponding control action through the functions implemented in the firmware - at this time, the instance of the task is terminated.

As result of the previous considerations, the interrupt service routine implemented in the eMIOS_0_Requests_0_3_Handler is reported in the following as example:

Listing 5.5: Handler responsible to run the application task.

```
void eMIOS_0_Requests_0_3_Handler(void)
{
    /* Peripheral flag check and clear */
    ....................................

    /* Upload the task input structure */
```

```
    Control_U.sig1 = Scheduler_GetSig1();
    Control_U.sig2 = Scheduler_GetSig2();
    ....................................

    /* Run the task */
    Control_step();

    /* Actuate the control action */
    Scheduler_setSig1(Control_Y.sig1);
    Scheduler_setSig2(Control_Y.sig2);
    ..............................
}
```

As can be seen in the code, some dedicated functions were developed in the software component Scheduler to set and get the fields of the input and output data structures of the considered task. This approach was needed because the global data structures, defined in the software components and used as interface with the application software, contain raw values to be scaled and mapped into the correct measurements units. Therefore, in order to correctly interface this two software layers, their interactions were analyzed to avoid bugs due to wrong data conversions or variable accesses. An example of implementation of these functions is the following:

Listing 5.6: Get and set function implemented in the software component Scheduler.

```
/* Generic get function */
    static inline real32_T Scheduler_GetPhCurrA(void)
    {
        return (real32_T)((DataCurrRaw -
           V_OFFSET)/V_GAIN*CURR_CONV_FACTOR);
    }

    /* Generic set function */
    static inline void
       Scheduler_SetCanTxDataMsg01_1(real32_T value)
    {
        CanTxDataMsg01.Variable01 = (uint16_T)((value +
           CAN_OFFSET)*CAN01_CONV_FACTOR);
    }
```

Only examples about these functions were reported due to non-disclosure policies. Anyway, the get and set functions were always declared as static inline to avoid the introducing of function call overheads in the interrupt service routine related to the execution of the application software. The values of offset and gain related to each variable scaling and mapping were chosen according to their corresponding hardware conditioning in the acquisition or due to predetermined conversions.

The actuation of the control actions computed by the application software was performed using the previously described set functions or by means of the specific functions developed for the purpose. An example of this approach is the setting of the duty cycles for the generate PWM signals which is reported in the following as example:

Listing 5.7: Handler responsible to run the application task.

```
void eMIOS_0_Requests_0_3_Handler(void)
{
    /* Peripheral flag check and clear */
    ..................................

    /* Upload the task input structure */
    ..................................

    /* Run the task */
    Control_step();

    /* Actuate the control action */
    ..............................
    PwmMgm_SetDuty(Control_Y.dutyA, Control_Y.dutyB,
        Control_Y.dutyC);
}
```

As can be seen in the source code, the fields of the output data structures of the application tasks are used to execute specific actions on the system through the features implemented in firmware to interact with the system.

At this point, a complete integration between the application software and the software components constituting the firmware was performed. Furthermore, a software environment responsible to run the tasks, obtained by means of automatic code generation processes performed on the application software, was successfully developed according to the system requirements.

Chapter 6

Conclusions

The world around the electric propulsion of automotive vehicles is a multidisciplinary field including mechanics, electronics, software development and control theory. This thesis was inserted in this context aiming to develop and test the low-level software for the target platform integrated in a proof of concept of an electric powertrain. In this chapter, a brief summary about the activities carried out is provided describing the corresponding results and possible further developments are discussed.

First, a general description of the topics discussed during the firmware development was provided to the reader in the corresponding chapter. As explained, the firmware was developed as a Hardware Abstraction Layer (HAL) to be placed between the Electronic Control Unit (ECU) which hosts the microcontroller (MCU) and the application software developed through a Model-Based approach. More precisely, the development process was carried on diving the firmware in separated modules called software components to prepare a easily modifiable architecture for further development. Indeed, the choice to structure the firmware in software components turned out the best way to guarantee the possibility to release upgrades and further versions without affecting the overall software. This means that changes in the implementation of the features required by the application software do not affect the predefined interfaces, and at the same time, changes in the application software can be made without the necessity to modify the firmware.

As well as done for the firmware development, test procedures were developed to be independent from the specific implementations of the related features – they were designed to verify requirements rather than implementations. During the firmware testing and verification process, all the features required by the application software and implemented in the firmware were successfully tested.

Once the firmware was tested, the integration process of the different software components was performed obtaining, also in this case, the expected results. The feasibility of the implementation of the application software into the target platform

was demonstrate through a processor-in-the-loop simulation. At that point, the software component Firmware Application Integration was developed to guarantee a correct communication between the two software layers. A further step required for a complete verification and validation of the integration process consists in performing a Hardware-in-the-Loop simulation or testing the overall software in the field. It is important to remember that, since we are dealing with a proof of concept, the field is represented by a test bench or a controlled environment developed to test the overall powertrain.

Even if all the requirements were satisfied by this first firmware version, improvement are required to increase the code quality. For example, code optimization and standardization could be required to increase the performances and safety of the system. In fact, during the development, a significant overhead in the device drivers provided by the silicon-vendor was found. Substituting the functions of the device drivers with custom bare-metal ones could lead to a significant increase of the code efficiency. In general, since we are now developing only a proof-of-concept implementation of the system, good practices were followed but MISRA rules were not implemented in a formal way. Since the system application is a safety critical one, standards like ISO26262 shall be also implemented to guarantee functional safety.

In summary, the development of the low-level software for the target platform of the considered electric powertrain leads to expected and satisfactory results. In order to improve the software developed for the prototype to be ready for a final product, further developments could be required to formally implement MISRA rules, functional safety according to standards like ISO26262 and to increase the computation efficiency. Despite this, a clear software architecture and a first version of the required features were successfully developed and tested providing a starting point for further developments.

Bibliography

[1] C.C. Chan. «The State of the Art of Electric and Hydrid Vehicles». In: 90 (Feb. 2002), pp. 247–275 (cit. on p. 3).

[2] Wikipedia contributors. *Powertrain — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-April-2020]. URL: https://en.wikipedia.org/w/index.php?title=Powertrain&oldid=931304124 (cit. on p. 4).

[3] Mohan Undeland Robbins. *Power Electronics: Converters, Applications and Design*. Ed. by John Wiley and Sons Inc. Third Edition (cit. on p. 32).

[4] Wikipedia contributors. *Firmware - Wikipedia, The Free Encyclopedia*. [Online; accessed 02-September-2020]. URL: https://en.wikipedia.org/wiki/Firmware (cit. on p. 36).

[5] Wikipedia contributors. *C Programming Language - Wikipedia, The Free Encyclopedia*. [Online; accessed 02-September-2020]. URL: https://en.wikipedia.org/wiki/C_(programming_language) (cit. on p. 36).

[6] Wikipedia contributors. *Software verification and validation - Wikipedia, The Free Encyclopedia*. [Online; accessed 13-August-2020]. URL: https://en.wikipedia.org/wiki/Software_verification_and_validation (cit. on p. 84).

[7] Lauterbach Development Tools. *Product Overview*. [Online; accessed 13-September-2020]. URL: https://www.lauterbach.com/product-overview_flyer_web.pdf (cit. on pp. 92, 93).

[8] Wikipedia contributors. *System integration - Wikipedia, The Free Encyclopedia*. [Online; accessed 06-September-2020]. URL: https://en.wikipedia.org/wiki/System_integration#cite_note-Heat-1 (cit. on p. 143).

[9] Wikipedia contributors. *Rate-monotonic scheduling - Wikipedia, The Free Encyclopedia*. [Online; accessed 16-September-2020]. URL: https://en.wikipedia.org/wiki/Rate-monotonic_scheduling#:~:text=In%5C%20computer%5C%20science%5C%2C%5C%20rate%5C%2Dmonotonic,in%5C%20a%5C%20higher%5C%20job%5C%20priority. (cit. on p. 159).

[10] Wikipedia contributors. *Preemption (computing) - Wikipedia, The Free Encyclopedia*. [Online; accessed 16-September-2020]. URL: https://en.wikipedia.org/wiki/Preemption_(computing) (cit. on p. 160).