POLITECNICO DI TORINO

Master Degree
in MECHATRONIC ENGINEERING

Master Degree Thesis

Model-based Design of an Automotive Control Code with a modified V-cycle and Modular Model approach

Supervisor
Prof. Stefano Carabelli

Candidate
Martina Coletta

A.A.2020/2021

Summary

The *methodology* is an important concept that holds in **System Engineering**. It is a transdisciplinary and integrative approach that enables the successful realization, use, and retirement of engineered systems. System Engineering provides facilitation, guidance and leadership to integrate the relevant disciplines and speciality groups into a cohesive effort, forming an appropriately *structured development process* that proceeds from concept to production, operation, evolution and eventual disposal. It integrates the **Model-based Design** approach. Concerning the System Engineering purposes, the methodology has been developed to enable the interaction among different disciplines by creating a standard procedure and providing tools, that are valid for the design of any system. The provided tools are the **Hybrid V-cycle** and the **Modular Technical Model**. The Hybrid V-cycle is a guideline for the development process and it is composed of different steps that involve the Modular Technical Model usage. The Hybrid V-cycle is developed as an extended version of V-cycle. It is used to distinguish the computing platforms that are involved at each step: *DWS (Development Workstation)*, *RCP (Rapid Control Prototyping)* and *VMU (Vehicle Management Unit)*. The Hybrid V-cycle as well as defining the platforms it describes in details the different models that have to be used at each step that are based on the Modular Technical Model. The Modular Technical Model provides a structured architecture composed by a number of parts (modules) that are intended to define the system components and to differentiate the skills involved in the development process. As a consequence, the Modular Technical Model specifies the *interfaces* between the system parts (and the relative discipline) to enable the interaction. The idea is to create the Modular Technical Model as a standard layout with a pre-defined structure that is not dependant on the application and that is composed by several empty blocks that are the modules.
Using the Hybrid V-cycle and the Modular Technical Model is of a great importance when dealing with the code generation. The code is automatically extracted from the Control Logic module (devoted to the control algorithm) and it has to be deployed on the target hardware. In order to differentiate the target hardware a Control Logic frame is included into the Control module to represent the platform interfaces. These are used to create the interaction between the control algorithm and the system. Then the frame is the only Modular Technical Model component that changes along with the Hybrid V-cycle steps.
In order to test the methodology accuracy and consistency a "*Filters in series*" application is developed first. Then the "*Vehicle Management Unit code for an hybrid car*" application is considered.
The *dSPACE MicroAutoBox II* is used as Rapid Control Prototyping system.

Acknowledgement

I would like to express my profound gratitude to the Professor Stefano Carabelli that supported me and that really encouraged me during the thesis work. I would like also to thank the PANDA G1 Electrical team for the support. Finally I would express my gratitude to my family and all the people that believed in me and encouraged me till the end.

Contents

List of Figures

Table of acronyms

1 Methodology **1**

2 V-cycle **2**
2.1 V-cycle main characteristics . . . 3
2.2 Hybrid V-cycle . . . 5
2.2.1 Hybrid V-cycle description . . . 6

3 Modular Technical Model **10**
3.1 MTM components . . . 11
3.1.1 Modules . . . 11
3.1.2 Additional components . . . 12
3.2 MTM implementation . . . 13
3.2.1 Simulink implementation details . . . 14
3.2.2 Blocks description . . . 15
3.2.3 Simulink Template Generation . . . 19
3.2.4 Simulink Template Usage . . . 19

4 Control module structure **22**
4.1 Control module characteristics . . . 23
4.2 Introduction to the code generation . . . 25
4.3 Multi-task application . . . 26
4.3.1 Multi-task implementation . . . 29
4.3.2 Simulink implementation . . . 30

5 Control code generation **33**
5.1 SIL (Software-in-the-loop) . . . 33
5.1.1 SIL simulation . . . 34
5.2 PIL (Processor-in-the-loop) . . . 37

6 HMI module structure **41**

7 Rapid Control Prototyping (RCP) 42
7.1 dSPACE TargetLink . . . 50
7.1.1 MIL simulation with TargetLink . . . 50
7.1.2 SIL simulation with TargetLink . . . 52
7.1.3 PIL simulation with TargetLink . . . 52

8 Tutorial Example 54
8.1 Deeper analysis through the Hybrid V-cycle . . . 54
8.1.1 System Overall Specifications . . . 55
8.1.2 System Design . . . 56
8.1.3 RCP Code Production . . . 59
8.1.4 RCP on Test Bench . . . 60
8.2 Filters in series results . . . 62
8.2.1 MIL results . . . 63
8.2.2 SIL simulation . . . 65
8.3 dSPACE results . . . 66
8.3.1 Simulated Dashboard components . . . 66
8.3.2 Real Dashboard components . . . 69
8.4 Conclusions . . . 70

A Automotive application 72
A.1 Concept Model . . . 72
A.2 Modular Technical Model . . . 74
A.3 Code Generation . . . 75
A.4 dSPACE MicroAutoBox II . . . 75

Bibliography 79

List of Figures

2.1 Theoretical V-cycle . . . 2
2.2 MATLAB V-cycle [3] . . . 4
2.3 Hybrid V-cycle . . . 5
2.4 System Design step . . . 6
2.5 RCP Code Production step . . . 7
2.6 RCP on Test Bench step . . . 8
3.1 General architecture of the Modular Technical Model . . . 10
3.2 Simulink implementation of the MTM as a template . . . 13
3.3 Control Block content . . . 16
3.4 HMI content . . . 17
3.5 Button . . . 17
3.6 Example of a Button connected to a constant value . . . 18
3.7 Content of a generic Monitor . . . 18
3.8 Template preview . . . 19
3.9 Example . . . 20
3.10 Simulink Starting Page . . . 20
3.11 Templates . . . 20
3.12 Example . . . 21
3.13 Template Information . . . 21
4.1 Control and Control_Logic . . . 22
4.2 Hybrid V-cycle . . . 23
4.3 Example of dSPACE ADC/DAC modelling [**14**] . . . 24
4.4 Example of Control_Logic frame when considering the dSPACE MicroAutoBox II as RCP platform and when performing the HIL procedure . . . 25
4.5 Generic multi-task application . . . 27
4.6 Example of dSPACE overrun situation [**18**] . . . 28
4.7 Multi-task application with different task sample rates . . . 29
4.8 Example of a model with Rate Transition blocks - Digital Filters in series . . . 30
4.9 Example of Control_Logic content that represents a multi-task application with three different synchronous tasks and an aperiodic process modelled by different Stateflow charts . . . 31
5.1 Test Harness Example . . . 35
5.2 Example of Runs pane . . . 36
5.3 Control badge . . . 36
5.4 PIL Verification Mode . . . 37

5.5 Test Harness error . . . 37
7.1 dSPACE MicroAutoBox II . . . 42
7.2 Example of dSPACE AD/DA converters . . . 43
7.3 Math and Data Types . . . 44
7.4 Simulation Target . . . 44
7.5 RTI load options . . . 44
7.6 RTI variable description . . . 45
7.7 Platforms/Device icon . . . 45
7.8 Register Platforms . . . 46
7.9 Procedure . . . 46
7.10 .sdf file selection . . . 46
7.11 Measurement Configuration icon . . . 47
7.12 Measurement Configuration menu . . . 47
7.13 Properties . . . 47
7.14 Model Root . . . 47
7.15 Block selection . . . 48
7.16 Layout example . . . 48
7.17 Stop RTP . . . 49
7.18 Measuring buttons . . . 49
7.19 dSPACE example of a generic model in MIL simulation mode . . . 51
7.20 TargetLink Plot Overview Window considering a generic dSPACE example . . . 51
7.21 dSPACE example of a generic model in SIL simulation mode . . . 52
7.22 TargetLink Main Dialog code informations . . . 53
7.23 dSPACE example of a generic model in PIL simulation mode . . . 53
8.1 V-cycle . . . 54
8.2 System Overall Specifications step . . . 55
8.3 Filters in series - concept model . . . 55
8.4 System Design step . . . 56
8.5 Control_Logic content with the tasks that are executed at different sample times. A1 ⇒ *Asynchronous* D1 ⇒ *Discrete, sample time 1* D2 ⇒ *Discrete, sample time 2* D3 ⇒ *Discrete, sample time 3* . . . 57
8.6 Filters in series application - Control_Logic content without asynchronous source . . . 58
8.7 RCP Code Production step . . . 59
8.8 RCP on Test Bench step . . . 60
8.9 DS1541 - dSPACE I/O board . . . 61
8.10 Concept model results . . . 63
8.11 DWS MIL results . . . 63
8.12 DWS MIL results . . . 64
8.13 RCP MIL results . . . 64
8.14 SIL results of the 500 Hz filter . . . 65
8.15 SIL results of the 100 Hz filter . . . 65
8.16 Test Bench . . . 66
8.17 Control content - simulated dashboard components . . . 66
8.18 System OFF . . . 67

8.19 System ON - Emergency OFF 68
8.20 System OFF - Emergency ON 68
8.21 Control content - real dashboard components 69
8.22 System OFF 69
8.23 System ON - Emergency OFF 70
8.24 System OFF - Emergency ON 70
A.1 Wired logic schematic of the vehicle 73
A.2 Inverter logic schematic 73
A.3 Environment module - road slope 74
A.4 Plant module - vehicle representation 74
A.5 Control module of the step 2.1 of the Hybrid V-cycle with the frame and the Control Logic module 74
A.6 Control Logic module - Supervisor and Task for the torque command computation 75
A.7 Frame representing the input signals to the Control Logic module . . 76
A.8 Frame representing the output signals from the Control Logic module 76
A.9 Forward test with an acceleration command that come from the external DC power supply. 77
A.10 Backward Test with the acceleration command that come from the external DC power supply 77
A.11 Emergency situation 78
A.12 AC charger plugged 78

Table of acronyms

RCP	Rapid Control Prototyping
VMU	Vehicle Management Unit
MIL	Model-in-the-loop
SIL	Software-in-the-loop
PIL	Processor-in-the-loop
HIL	Hardware-in-the-loop
DWS	Development Workstation
HMI	Human-Machine-Interface
MTM	Modular Technical Model
AD	Analogical-Digital
DA	Digital-Analogical
PWM	Pulse-Width-Modulation
ADC	Analogical-Digital converter
DAC	Digital-Analogical converter
FFT	Fast Fourier Transform
API	Application Programming Interface
PC	Personal Computer
RTI	Real-Time Interface
sdf	System Description File
RTP	Real-Time Platform

Chapter 1

Methodology

The *methodology* is an important concept that holds in **System Engineering**. It is a transdisciplinary and integrative approach that enables the successful realization, use, and retirement of engineered systems. It makes use of systems principles and concepts, and scientific, technological and management methods. System Engineering provides facilitation, guidance and leadership to integrate the relevant disciplines and speciality groups into a cohesive effort, forming an appropriately *structured development process* that proceeds from concept to production, operation, evolution and eventual disposal. The approach generates and evaluates alternative solution concepts and architectures and it focuses on modelling requirements and selected solution architecture for each phase of the endeavour and performing design synthesis and system verification and validation [**1**].
System Engineering integrates the **Model-based Design** approach. It is related to models that describe the system. Models provide clear structures to deal with the system complexity. Moreover, they are a simple, understandable and standardized language for all involved engineers and managers [**2**]. Models allow simulation and prediction of behaviour and properties of a generic system in early phases, resulting in the acceleration of the design process and the increase of the efficiency of the product development.
Concerning the System Engineering purposes, the methodology has been developed to enable the interaction among different disciplines by creating a standard procedure and providing tools, that are valid for the design of any system. The provided tools are the **V-cycle** and the **Modular Technical Model**.
The V-cycle aims at defining the process of the system development. It makes a differentiation of *phases* (time-based) and *stages* (content-based) from the *System Requirements* to the *System Decomposition* (based on the Modular Technical Model) and *System Integration* and their iteration.
The Modular Technical Model provides a structured architecture composed of a number of parts (modules) that are intended to define the system components. This allows to split the skills involved in the system development process. As a consequence, the Modular Technical Model specifies the *interfaces* between the system parts (and the relative discipline) to enable the interaction.
The methodology is needed in order to gain in *re-usability*, *safety*, *system* and *timing performances* as well as *costs* trade-off.

Chapter 2

V-cycle

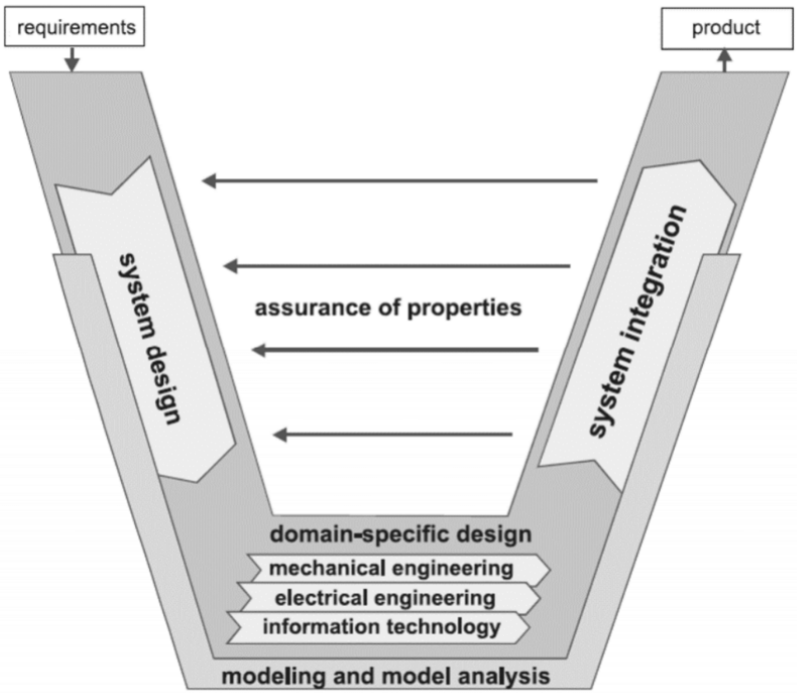


Figure 2.1: Theoretical V-cycle

The V-cycle is a guideline that divides the development process into three conceptual phases (Figure 2.1). The phase depicted on the left side of the V-model describes the transformation of requirements, which are considered as an input when designing a system. In the second phase the project is split into sub-projects. Each sub-project is managed by a domain-specific team that can better address the critical aspects related to that engineering area. The third section integrates the previously split sub-projects during the system integration, verification and validation. The result or output of the V-cycle is the product [**2**].

2.1 V-cycle main characteristics

1. *Integration of Model-based Development approach.*
   The model is a system representation and it is designed according to specific modelling languages. It is the starting point of the software development. The modelling phase purpose is to generate code for the final application. The code is generated from the model in an automatic fashion (**Automatic Code Generation**) by a transformation tool.
   The code generation introduces the **Functional Safety** concept, i.e., the software can include redundant functionalities in order to guarantee safety.
   Once the software is developed, it has to be deployed on different target hardware (or computing platforms) that are the RCP device and VMU.

2. *Life cycle representation.*
   The V-cycle illustrates the whole life cycle of the system development starting from the requirements up to the final product, describing accurately each step and how it has to be performed.

3. *Sequential or iterative approach.*
   The V-cycle steps have to be carried out in the right order, following the V-shape. During the verification and validation phases some discrepancies between the expected output and the obtained one might be detected. To solve these inconsistencies, the process has to be restarted from the phase in which the problems may have originated. It may also happen that the development process has to be started again from the beginning of the V-cycle.
   The iterations are grouped in the *'-in-the-loop' procedures*, that are:

   - **Model-in-the-loop**.
     The system is designed and simulated *non in real-time*. Real-time means that a system must respond to external events both correctly and within a finite, specified period of time called *deadline*. Being in a non real-time application means that fulfilling the deadline is not mandatory, but the correctness of the output must be guaranteed.
   - **Software-in-the-loop**.
     It is the process used to develop the software as a result of the automatic code generation and to check whether it behaves as expected. The idea is the same as the MIL approach, but the system is simulated non in real-time along the code resulting from the code generation process.
   - **Processor-in-the-loop**.
     This procedure aims at integrating the software into the target hardware. Then the software behaviour is checked, being executed non in real-time on the hardware.
   - **Hardware-in-the-loop**.
     In general the Hardware-in-the-loop procedure has a first step in which a dedicated hardware or emulation hardware is used to mimic the physical interfaces. In this way a virtual real-time implementation of physical components is created and then simulated to check the system behaviour.

Then, all the system components are tested in real-time using Test Bench that physically reproduce the system.

4. *Verification and validation.*

   - *Verification*: evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process.
   - *Validation*: assurance 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.

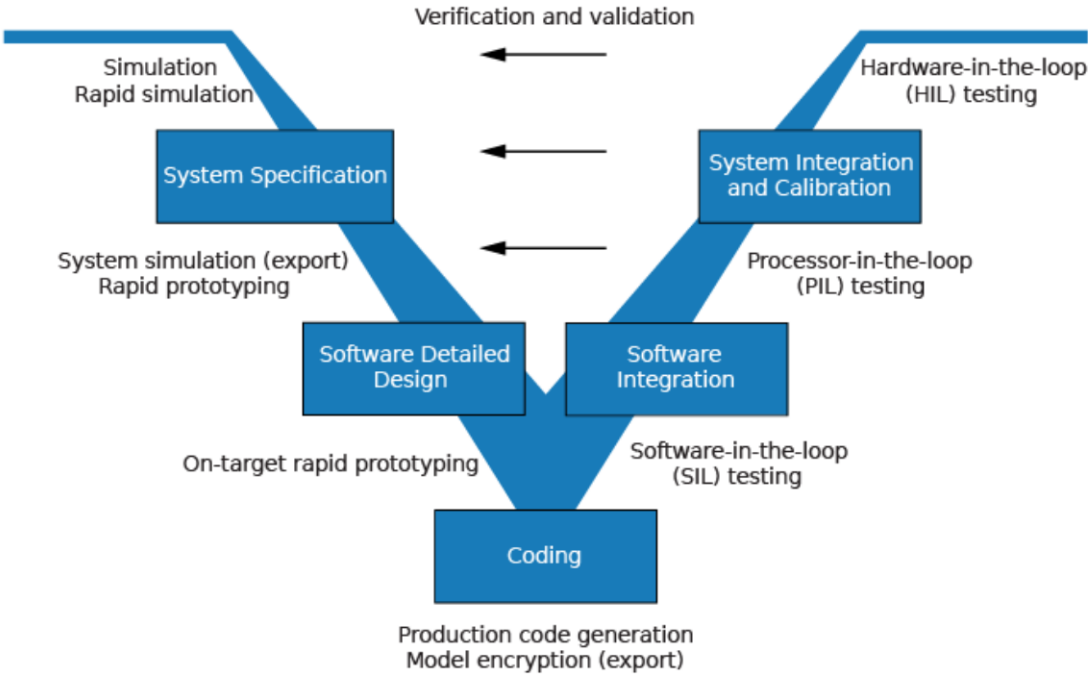


Figure 2.2: MATLAB V-cycle [3]

There are different interpretations of V-cycle that share the same features described in Section 2.1. Expanded versions of V-cycle can be found in literature, e.g., Figure 2.2 that have the purpose of highlighting further critical aspects. Another example concerns the V-model of the United States Department of Transportation (DoT) (U.S. Department of Transportation, 2009) that emphasizes the meaning of a careful preparation and definition of the verification process by arrows pointing from left to right [**2**].
Taking into account the above considerations, the idea is to create a *Hybrid V-cycle* as another expanded version of V-cycle to distinguish the computing platforms and the different models that have to be used at each step.

2.2 Hybrid V-cycle

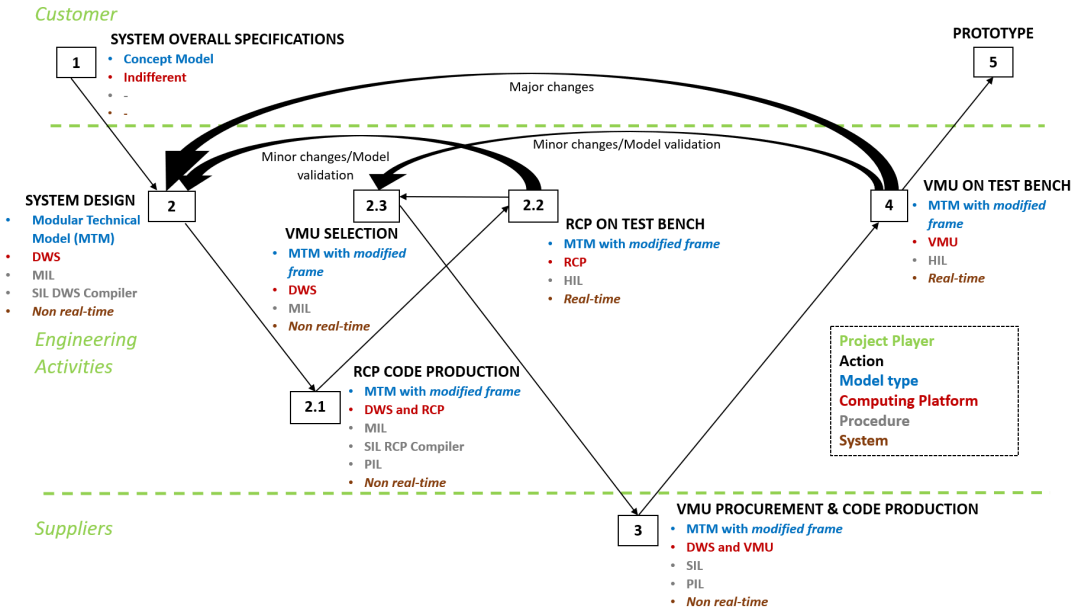


Figure 2.3: Hybrid V-cycle

It is composed by several steps that are carried out considering different computing platforms and different models:

- *DWS (Development Workstation).*

  It is intended for design and simulation purposes. The first step consists in modelling the system non in real-time, without any considerations on the target hardware. After that, the model is designed including all the target hardware characteristics and then it is simulated.

- *RCP (Rapid Control Prototyping)*

  This platform is used to repeat the MIL simulation. Then it is needed to perform the SIL and PIL simulations with the system non in real-time and then for testing (HIL), considering the system in real-time. As a final step the system behaviour is analysed to verify whether it fulfils the requirements.

- *VMU (Vehicle Management Unit)*

  The approach is the same as RCP, but using the VMU.

- *Concept Model*
  It is a model developed by the customer and it is used to collect the system specifications neglecting the implementation details.

- *Modular Technical Model*
  This model is used as a standard architecture when considering the implementation details and the hardware characteristics. The different Hybrid V-cycle steps use different versions of MTM. The *frame* is the only element that

changes among the versions. The frame is a MTM component responsible of the interfaces between the system and the *control algorithm*. The control algorithm (or *control law*) is the code that implements the control logic of the system.

2.2.1 Hybrid V-cycle description

1. *System Overall Specifications*
   In this phase the concept model is produced by the customer and it has to be implemented. The platform that is used to develop the concept model is not relevant. No particular procedure is applied at this step and the concepts of real-time and non real-time can be neglected.

2. *System Design*
   The System Design step involves *only* the DWS as computing platform. The system is considered *always non in real-time* and the Modular Technical Model is used as a standard architecture to perform the MIL and SIL procedures.

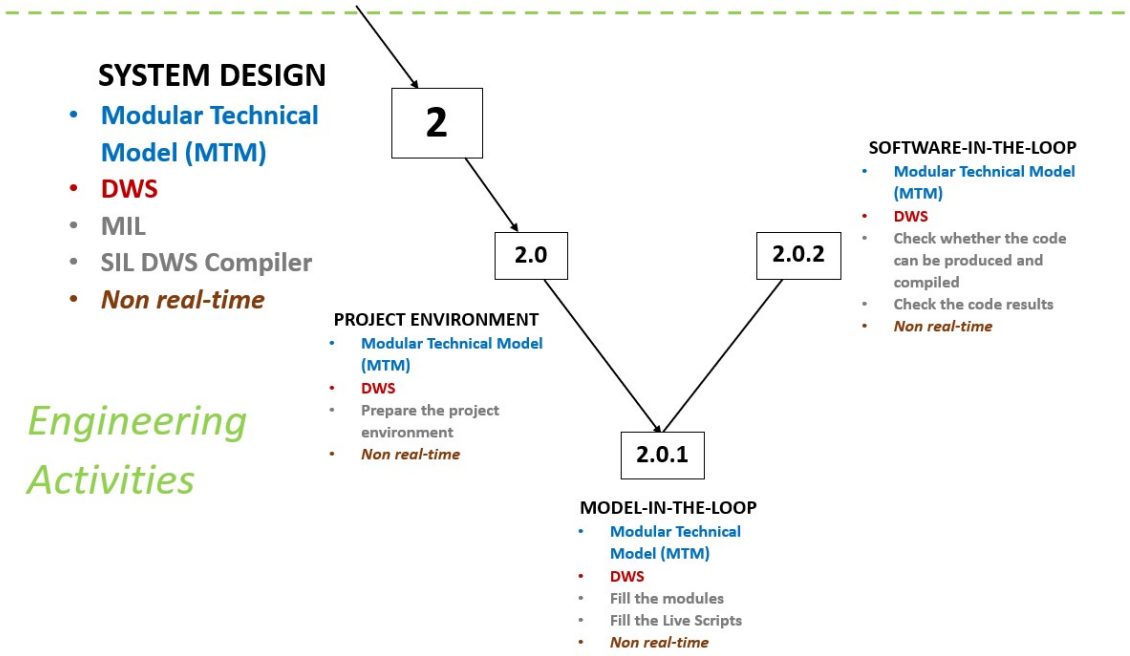


Figure 2.4: System Design step

2.0 *Project Environment*
It is a preliminary step intended to prepare the project environment. The idea is to use a predefined project as an architecture that contains folders associated to the different Hybrid V-cycle steps. This means that each step has to be carried out using the corresponding folder. As a consequence the skills involved in the system development can be split.

2.0.1 *Model-in-the-loop*
The procedure consists in developing a model that includes all the detailed characteristics of the system and then to simulate that model to

check the system behaviour. The results have to match those of the concept model. At this step generic interfaces have to be included into the frame. They are still not associated to the RCP platform. This is needed to detect the requirements that are used to choose the most appropriate RCP platform among all the available devices.

2.0.2 *Software-in-the-loop*

The procedure is composed of two steps which are respectively mandatory and recommended. First the code that implements the control logic of the system has to be automatically generated and compiled. During this procedure the control algorithm runs on the DWS and the DWS *System Target File* is selected. The System Target File is related to the transformation tool that is used to extract the code from the model. Then the SIL simulation can be run to check the results of the code generation as an additional procedure.

2.1. *RCP Code Production*

The RCP Code Production step involves both the DWS and the RCP platform. The system in considered *always non in real-time* and the Modular Technical Model is used as a standard architecture for the MIL, SIL and PIL procedures.

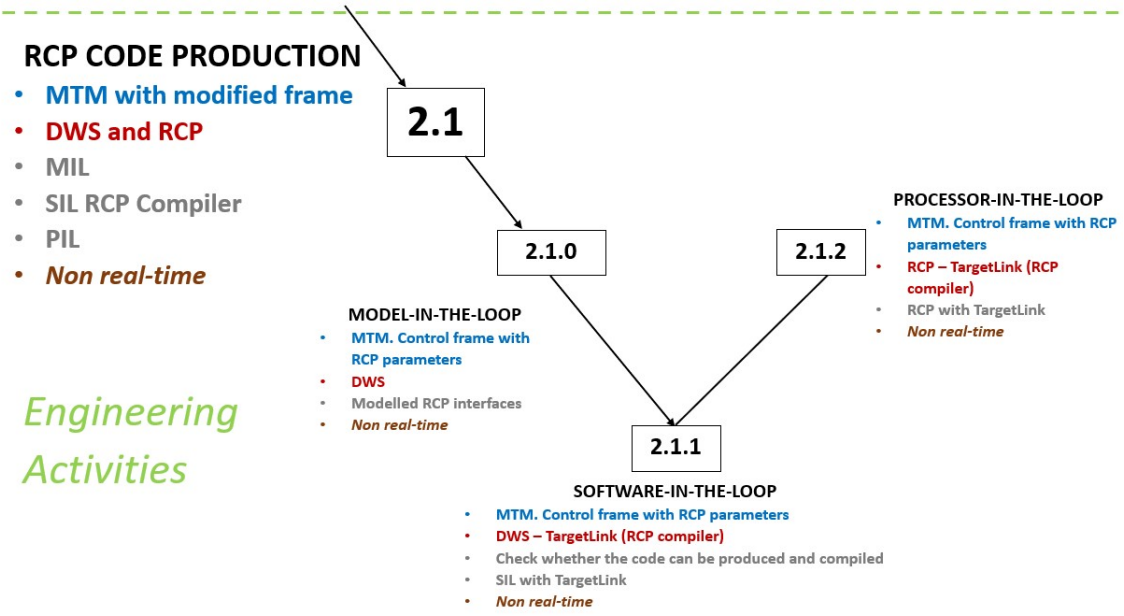


Figure 2.5: RCP Code Production step

2.1.0 *Model-in-the-loop*

The model designed at the step 2 is modified including all the characteristics of the RCP hardware. The MIL procedure is repeated because the RCP interfaces have to be included into the frame substituting those of the step 2. Then the model is simulated in order to check whether the MIL results are compatible with those of the MIL simulation performed at the step 2.

2.1.1 *Software-in-the-loop*

The control algorithm runs on the DWS. The first step consists in check whether the code can be produced and compiled considering the RCP System Target File. Then the SIL results can be checked by means of a specific RCP tool.
The DWS is not connected to the RCP platform.

2.1.2 *Processor-in-the-loop*

The control algorithm runs on the RCP platform physically connected to the DWS. The RCP System Target File is selected and the procedure is carried out using the specific RCP tool.

2.2. *RCP on Test Bench*

The RCP on Test Bench involves both the DWS and the RCP platform that are physically connected. The system is considered *in real-time* and it is represented by the Test Bench (real interfaces). The part of the Modular Technical Model that represents the frame and the control algorithm is *only* considered.

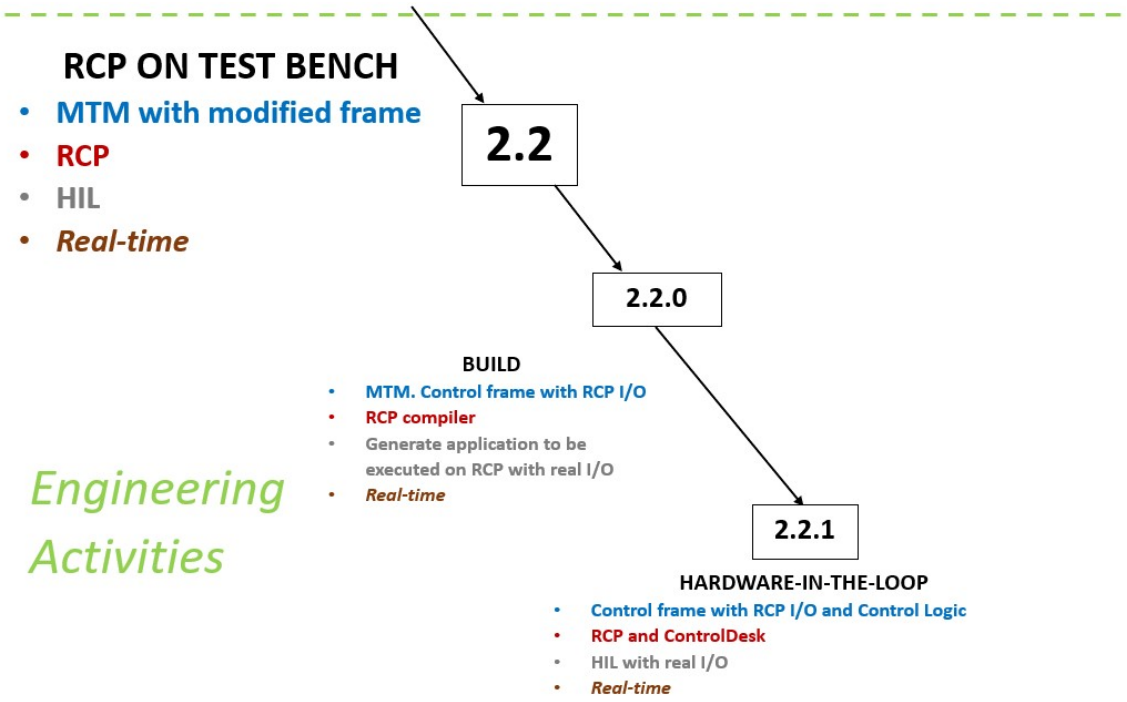


Figure 2.6: RCP on Test Bench step

2.2.0 *Build*

The control algorithm runs on the RCP platform. The RCP System Target File is selected and the code implementing the control algorithm is automatically generated and compiled.

2.2.1 *Hardware-in-the-loop*

The code is deployed on the RCP device and the system is tested by

means of the Test Bench. A specific RCP software is used to handle the platform and its signals.

2.3. *VMU Selection*
This is conceptually equal to the step 2. The MIL procedure is repeated considering the VMU characteristics. This allows to identify the more appropriate VMU among all the available devices for the specific application.

3. *VMU Procurement & Code Production*
The model developed at the step 2.3 is used to perform the SIL and PIL procedures as the step 2.1 but considering the VMU instead of the RCP platform.

4. *VMU on Test Bench*
The VMU is physically connected to the DWS. The control algorithm is deployed on the VMU in order to test the system that is represented by the Test Bench in real-time (HIL).

5. *Prototype*
Vehicle tuning with Log & Tune real-time interfaces.

Chapter 3

Modular Technical Model

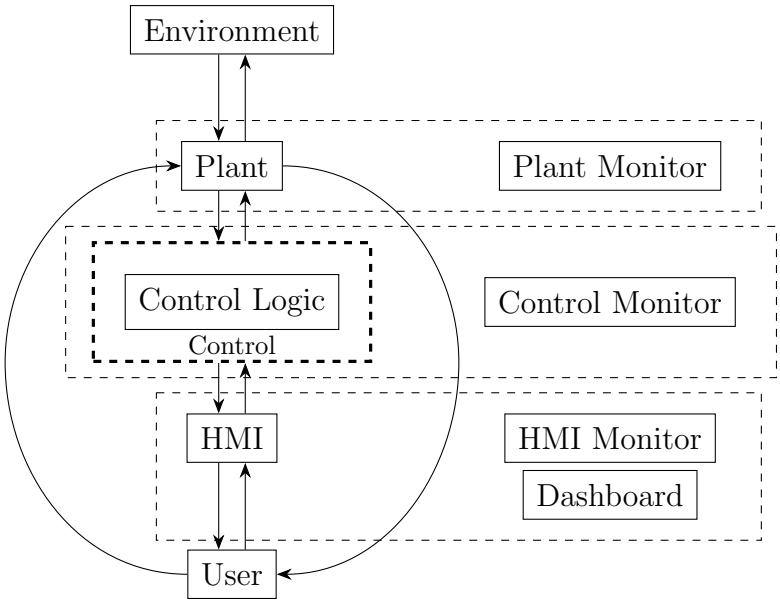


Figure 3.1: General architecture of the Modular Technical Model

The Modular Technical Model is as a standard layout (Figure 3.1) with a predefined structure that is not dependant on the application and that is composed by several empty blocks that are the modules (subsystems). They just need to be filled with the desired system characteristics. The result is a *standard* and *reusable* architecture that needs to be implemented.
The model is **technical**. This means that it contains all the detailed characteristics of the system, e.g., sampling times, tasks, quantization intervals that the concept model does not take into account. In fact the Modular Technical Model is used along with the Hybrid V-cycle steps needed to translate the requirements provided by the customer into an effective representation of the system.
The model is **modular**. It is composed of different modules associated to the parts that derive from the system decomposition. They own specific characteristics that are analysed by different speciality groups. The result is the separation of all the disciplines involved in the design of the system components, as the System

Engineering approach suggests. The Modular Technical Model is represented as a *main model*. It is intended to collect all the modules including the monitors and the dashboard components. The purpose is to provide a model that describes the whole system and that allows to easily access the content of the subsystems. It is mainly used to create the interaction among the modules through the *interfaces*. In particular, the main model comprises the Control that includes the *frame* used to contain all the target hardware interfaces, e.g., converters. They are needed to make the Control Logic compatible with the system. The other system parts interact through simple connections (lines) that can be associated to physical cables in reality. They are interfaces as well. Moreover, when simulating the main model all the system performances can be evaluated. They are the result of the modules interaction. As a consequence the effectiveness of the modules development can be checked. This can be achieved by means of the monitors that are inside the Plant, Control, and HMI modules.

Modularity also means that the different modules are potentially independent of the main model. This results in storing the subsystems content in models that are separated from the main one. Nevertheless, the different parts are still visible in the main model because the separation is needed to differentiate the skills involved in the system design. Then it allows a split modules development that is performed in the apposite models. The result is that the modules are used for design purposes while the main model is needed to create the interaction between the subsystems. Figure 3.1 shows the monitors that are intended to check the modules behaviour and then they are included into the different subsystems. Another important aspect regards the *dashboard*. It represents specific HMI components, e.g., buttons used to translate the user actions into signals for the Control subsystem.

3.1 MTM components

The Modular Technical Model is composed of different modules that are: Environment, Plant, Control, Control Logic and User. It also comprises additional components that are the monitors and the dashboard.

3.1.1 Modules

Environment

The Environment is the set of noises, disturbances and loads. The module comprises continuous (analogue) signals.

Plant

The Plant is used to characterize the system that has to be controlled and it comprises continuous signals. It also contains the Plant monitor.

Control & Control Logic

The Control Logic comprises discrete (digital) signals. It is intended to define a control law that has to be applied on the Plant, in order to obtain a certain system behaviour that has to be checked, to verify whether it is equal to the desired behaviour as much as possible. The module has a *frame* that is represented by the dashed line (Figure 3.1). It decouples an I/O device from the control algorithm. In particular, the frame is a layer that enables the interaction between the Control Logic and the target hardware by including device specifications without affecting the control law. In this way the MTM is adapted to the target hardware. These considerations are of great importance when dealing with the code generation process that extracts from the Control Logic module the application code.
The Control module is the set of the Control Logic, the frame and the Control Monitor. It comprises analogue and digital signals depending on the application.

HMI

The HMI comprises either continuous signals or discrete signals. It is the *Human Machine Interface*, therefore it is intended to translate the user action into a signal for the Control module. It contains also the dashboard components and the HMI monitor.

User

The User is a collection of multiple possible user actions.

3.1.2 Additional components

Monitors

The monitors are used to check the system behaviour and in particular that of the different modules. In fact they are included in the Plant, Control, HMI subsystems to respectively detect their functioning. The monitors are connected to specific components inside the subsystems to create a virtual link between the monitor and the module.

Dashboard

The dashboard is intended to represent the HMI components responsible of the interaction between the user and the system. As an example, the push button is a dashboard element that if pressed it causes a system reaction.

3.2 MTM implementation

Simulink has been chosen as modelling language for the MTM implementation, creating a **template** that comprises all the MTM characteristics. The template is a particular Simulink object that can be used to create control project with model-based design and automatic code generation for RCP and VMU platforms with and from the same model.

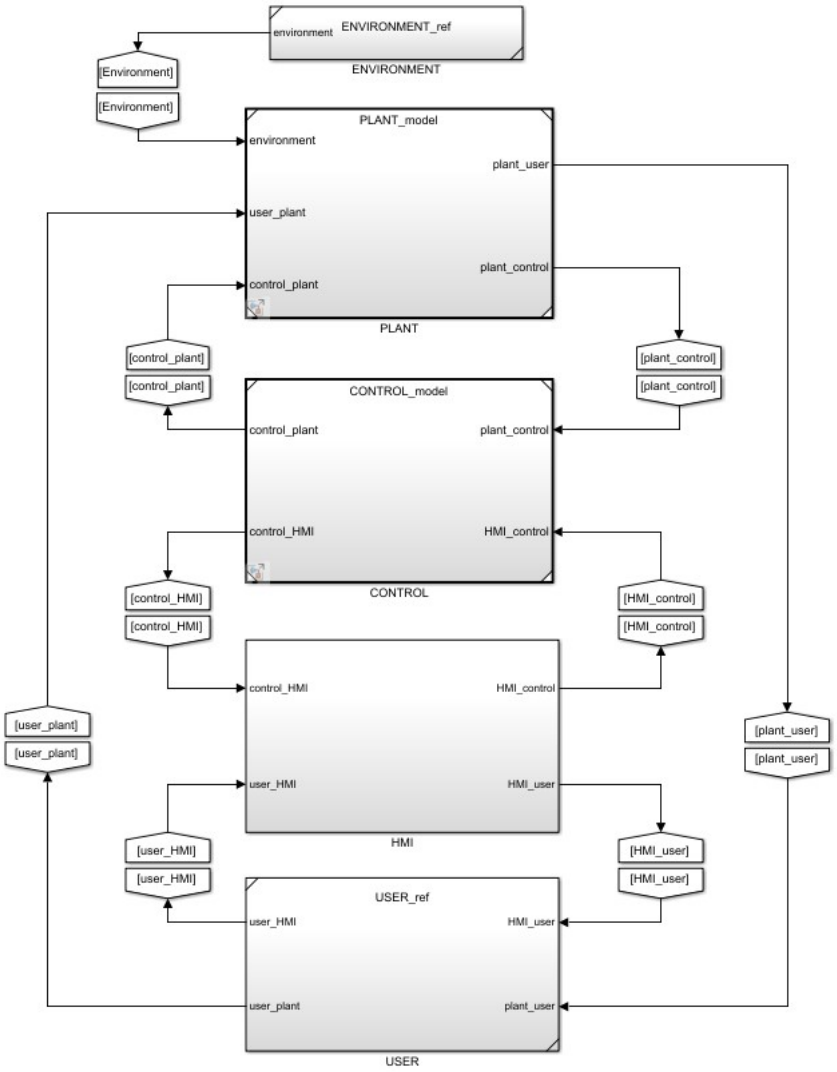


Figure 3.2: Simulink implementation of the MTM as a template

The structure in Figure 3.2 is the MTM implementation. It consists in 5 subsystems that represent the parts that compose a general system.

3.2.1 Simulink implementation details

- Template created in Matlab release 2020a.
- Two procedures have to be concurrently performed:
  - *Parametrization*
    It regards the definition of all the data or parameters that are involved in the MTM. The data are associated to the different Simulink blocks and they have to be stored in either MATLAB *Script* (.m file) or MATLAB *Live Script* (.mlx file).
  - *Modelling*
    It is the implementation process that uses the Simulink blocks to represent the system and to describe its functionalities. The implementation process result is a Simulink model or .slx file.
- The main model is composed of different subsystems that represent the modules. They are associated to the system parts.
- The contents of the subsystems are stored in separate models to differentiate the skills involved in the system development process. This allows also to use different solvers to evaluate the contents of the modules if needed.
  In order to store the content of the modules in separate models, the subsystems have to be converted into either *referenced subsystem* or *referenced model*. The content of the former cannot be simulated thus it cannot be associated to a different solver with respect to the one of the main model. The content of the latter is stored in a model that is independent of the main one and then it can be simulated with the desired solver.
  Note: the HMI is neither designed as referenced subsystem or as a referenced model because the dashboard elements are not supported in these kind of components.
- *Environment, Control, User* are blocks that have been rotated for organization purpose. This seems to be in contrast to the *MAB guidelines* [**12**]. However, the rules are respected since in each subsystem the stream of information is always guaranteed, from the left (inputs) to the right (output).
- Two different subsystems are linked by only one connection that contains all the possible signals that are being exchanged between two modules. Then by using a *Signal Routing* block all the signals can be packed together and the general architecture left untouched.
- *Sequence Viewer* can be added to check the system behaviour and the different chart transitions with their occurrences and their timing behaviour, in case of Stateflow usage.
- *Environment, User* designed as *referenced subsystem*. This means that a *reference* to a reusable group of blocks with a dynamic interface, which can be visual or functional is created [**13**]. In this way the content of the User is

stored in a separate model (.slx). This allows the disciplines to be split as System Engineering suggests.

- *Control, Control_Logic, Plant* designed as *referenced model*. The purpose is to create separate models (.slx) that store the Control and the Plant contents and with a well-defined interface. It is functional and independent of the main model. In order to convert a subsystem in a referenced model it has to be *atomic*, so that it functionally groups the blocks and executes them together. Functionally grouping the blocks makes it easier to convert the subsystem to a referenced model [**13**]. This is useful when dealing with a subsystem that is expected to grow.
  Note: only subsystem blocks can be converted in referenced subsystem or referenced model.

- *Solver* selection. When modelling the Control_Logic subsystem the solver has to be selected as *fixed-step* because the module comprises digital signals. A fixed-step solves the model at regular time interval (fixed-step-size) that is chosen at priori making a trade-off with the simulation accuracy.
  When considering the Plant, the solver can be selected as *variable-step* because the module comprises analogue signals. A variable-step solves the model choosing the step-size at each solver iteration. This results in the step-size adaptation, being high at lower frequency to avoid evaluating the system during the transient and then to avoid taking unnecessary steps and being low at higher frequency to see the dynamics of the system that change much faster.
  There are different kinds of solver that can be grouped in two main categories: *continuous solver*, that evaluates the model by performing an approximation in an infinite number of points and *discrete solver*, that uses a quantized phenomena with a finite approximation.
  When evaluating the overall model that includes all the system components, the fixed-step discrete solver is suggested.

3.2.2 Blocks description

1. *Environment*
   The Environment is implemented as a subsystem containing a block that generates a noise signal.

2. *Plant*
   The Plant has three inputs. The *control-plant* input is related to a signal that has been converted to the Plant domain before entering the subsystem. This is performed in the Control block. The other inputs are connections with other modules.
   It has also two outputs. The *plant-control* output is connected to a signal that enters the Control block and that will be converted in the Control domain. The other output is a connection with the HMI module.
   The Plant module contains also reference signals for the Plant Monitor.

3. *Control*
   The Control module comprises the Control Logic frame and the Control Logic itself. It is implemented as a set of two nested subsystems to differentiate the Control_Logic module and the frame components.

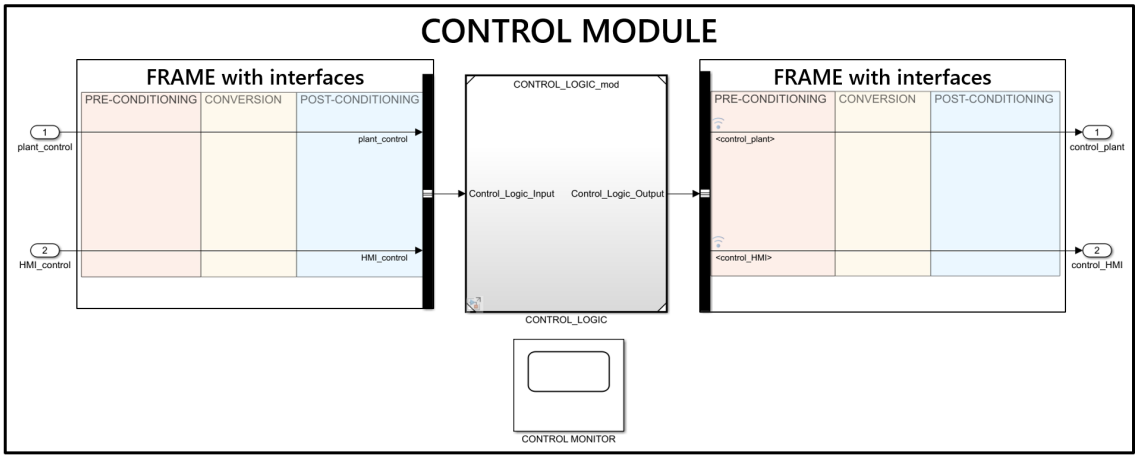


Figure 3.3: Control Block content

Note: The Control_Logic is devoted **only** to the **control algorithm** and it is **independent of the target hardware**.
The frame represents the **interfaces** and it **strictly depends on the target hardware**. It is the **only** part of the MTM dependent of the target hardware.

3.1 *Frame*
Part of the Control module that represents the interfaces needed to create the interaction between the Control_Logic and the other subsystems.

Interface component	Description
ADC	Analogue input
DAC	Analogue output
Digital input I/O	Digital input & digital output • FPGA - based digital I/O • BIT I/O • PWM generation/measurements • Incremental Encoder

Table 3.1: Interface components

Table 3.1 shows different interfaces that have to be included into the frame depending on the application and on the target hardware.

4. *HMI*
   The HMI subsystem contains the Dashboard and HMI Monitor including other elements used to create the virtual connection to those components, e.g., constant blocks.

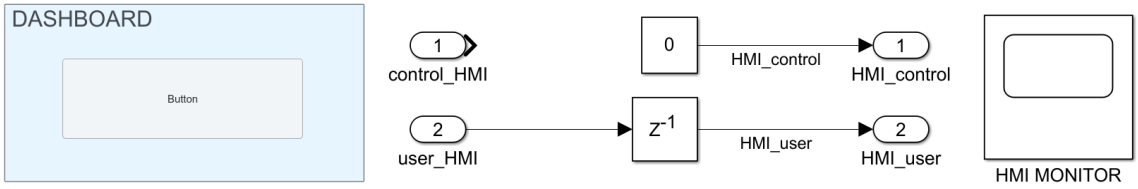


Figure 3.4: HMI content

Figure 3.4 shows an example of HMI content. The module is almost empty because the template is considered. The constant block is used to create the virtual connection to the Button that is a Dashboard element. The Dashboard is a specific Simulink library that contains control and indicator blocks to interact with simulations. Since the template is being analysed only a Button is considered as Dashboard component. In a specific application there are multiple Dashboard components, e.g., led.

4.1 *Button*
The Button is implemented as a block taken from *Simulink Library Browser* ⇒ *Dashboard* ⇒ *Push Button*.

Procedure:

1. Select the Button and click on the colon in order to connect the block with the desired signal.

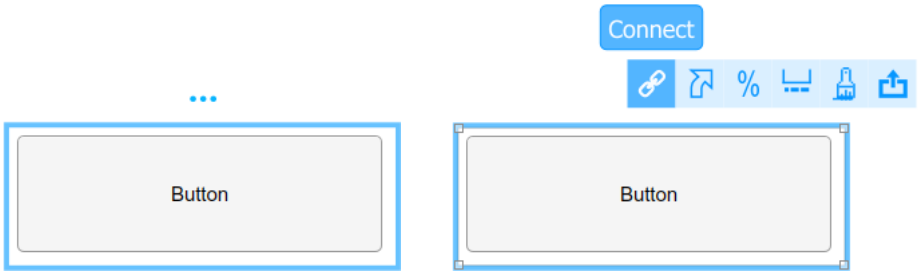


Figure 3.5: Button

2. Double click on the Button to open the window representing the content of this block.

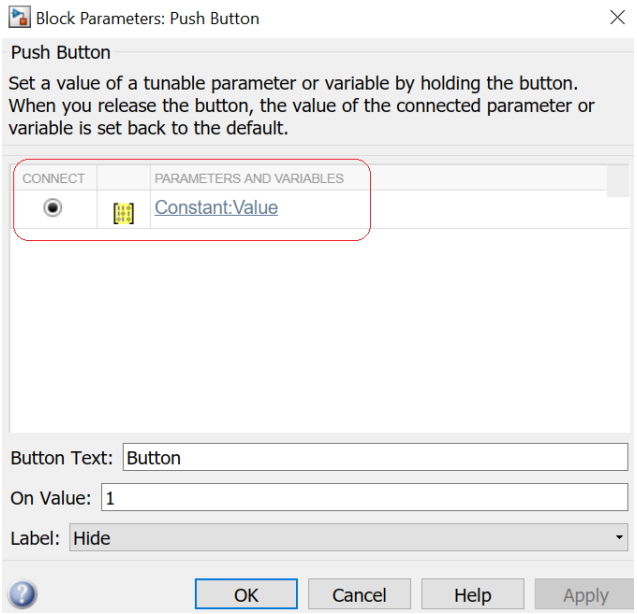


Figure 3.6: Example of a Button connected to a constant value

5. *User*
   The User is a subsystem that contains the user inputs.
   Consider as an example the vehicle as the Plant block. The User module contains the driver actions described by blocks as the *Longitudinal Driver*.

6. *Monitors*
   The monitors are included in the Plant, Control and HMI modules.
   They are taken from *Simulink Library Browser* ⇒ *Sinks* ⇒ *Floating Scope*.

   Procedure:

   1. Double click on the Floating Scope in order to open the window representing the content of the block.

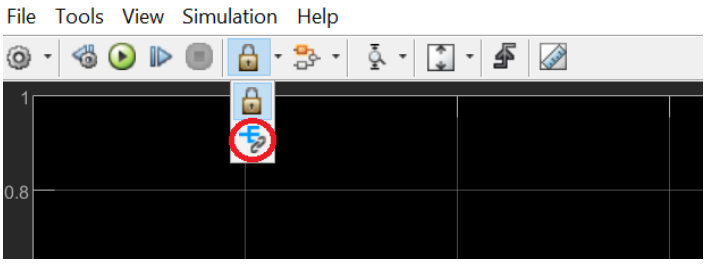


Figure 3.7: Content of a generic Monitor

   2. Click on the *Signal Selector* (in red) in order to connect the scope to the signals that have to be shown in this block.
      - (Optional) Click on *View* ⇒ *Layout* in order to show multiple signals separately with no overlap.

3.2.3 Simulink Template Generation

1. Open Simulink and create a model.
2. On the Simulation tab, select *save* ⇒ *Template*.
3. A window will be opened, called *'Export template-name to Mode Template'*.
   - 3.1 Edit the *template title*, select or create a *group*, and enter a *description* of the template.
   - 3.2 The *file location* is already present:

     'C:Users-username-Documents-MATLAB-untitled.sltx', rename the title of the template.
   - 3.3 (Optional) Specify a thumbnail image for the template by clicking Change and selecting an image file.
   - 3.4 Click *Export* to save the template.
4. In *Simulink Start Page* ⇒ *My Templates* the preview of the exiting templates will be shown. It includes all the created templates with the above procedure.

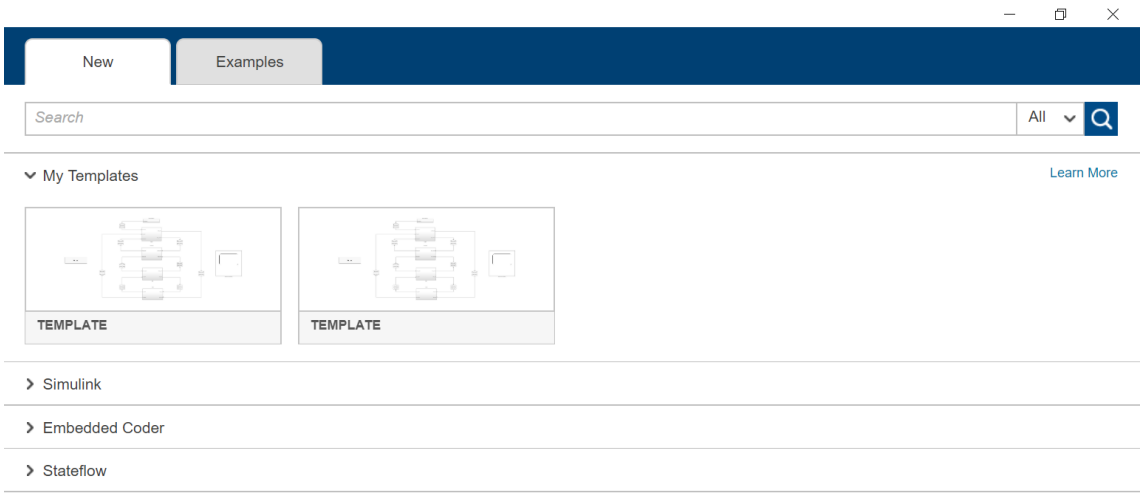


Figure 3.8: Template preview

3.2.4 Simulink Template Usage

In order to use a predefined *template*, there is a procedure that has to be followed:

1. Download the desired *template* and save it.
   - (a) In *Documents* ⇒ *MATLAB* there will be the *template*, that is a file *.sltx*.

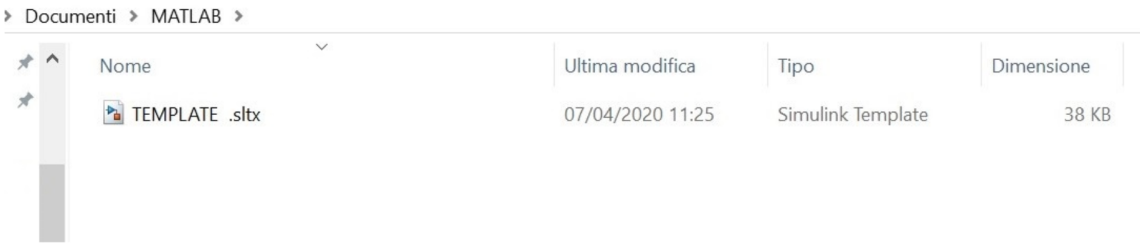


Figure 3.9: Example

2. Open Simulink
   (a) The following window is shown.

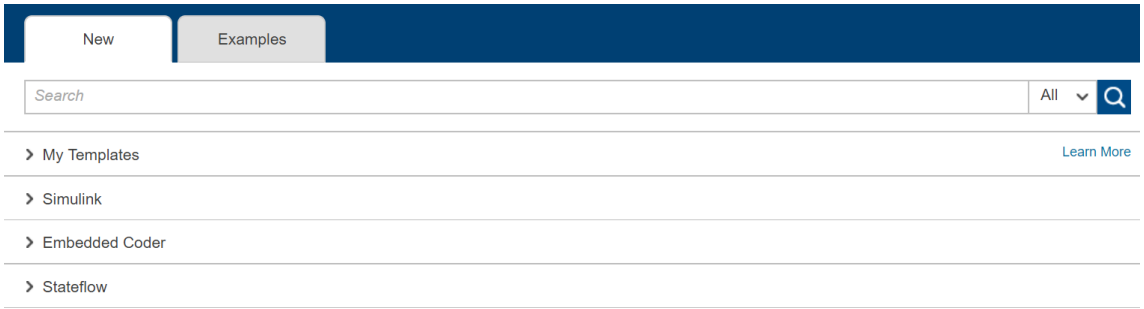


Figure 3.10: Simulink Starting Page

3. Click on *My Templates*.
   (a) This will show the existing templates.

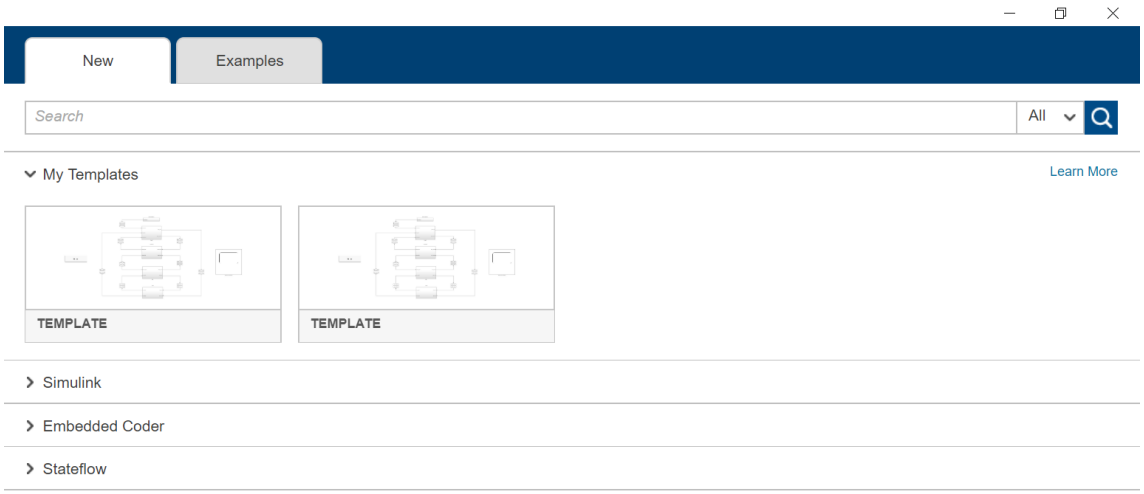


Figure 3.11: Templates

4. Click on the *template* that is intended to use.
   (a) Click on the icon named *Create Model*.

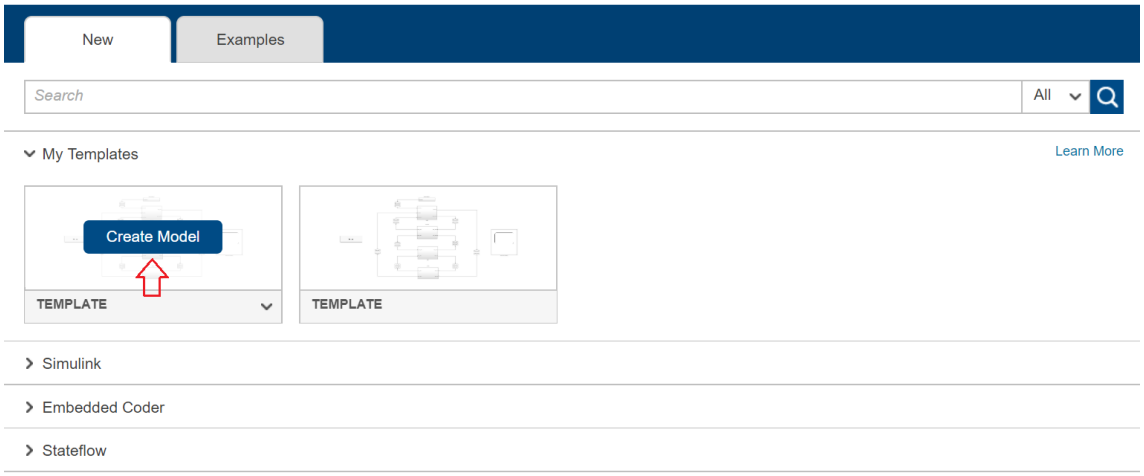


Figure 3.12: Example

(b) By clicking the down arrow, a window containing the information about the template is opened.

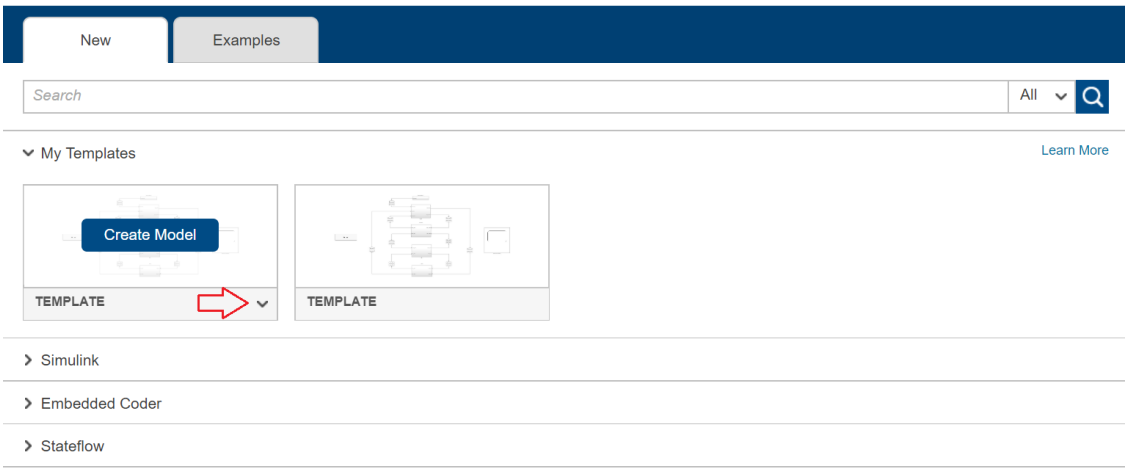


Figure 3.13: Template Information

(c) The standard project structure is opened and the model can be designed.

5. The same procedure can be done by clicking directly on the *template* file *.sltx* in the *Documents* ⇒ *MATLAB* directory.

Chapter 4

Control module structure

The Control module comprises the Control Logic and its frame intended to collect the target hardware characteristics. This is needed to separate the computing platforms with their I/O devices from the Control Logic module. The Control Logic module is used for the control law implementation and to generate the code that has to be deployed on the target hardware.
As mentioned in Chapter 3, the idea is to differentiate the Control module from the Control Logic one. This allows a clear identification of the interfaces that make the other system parts and the control algorithm compatible. The distinction between Control and Control Logic module is needed to identify the target hardware that is responsible of the Control Logic frame. In order to differentiate the modules two different subsystems are created: the Control_Logic and the Control. The former inside the latter.

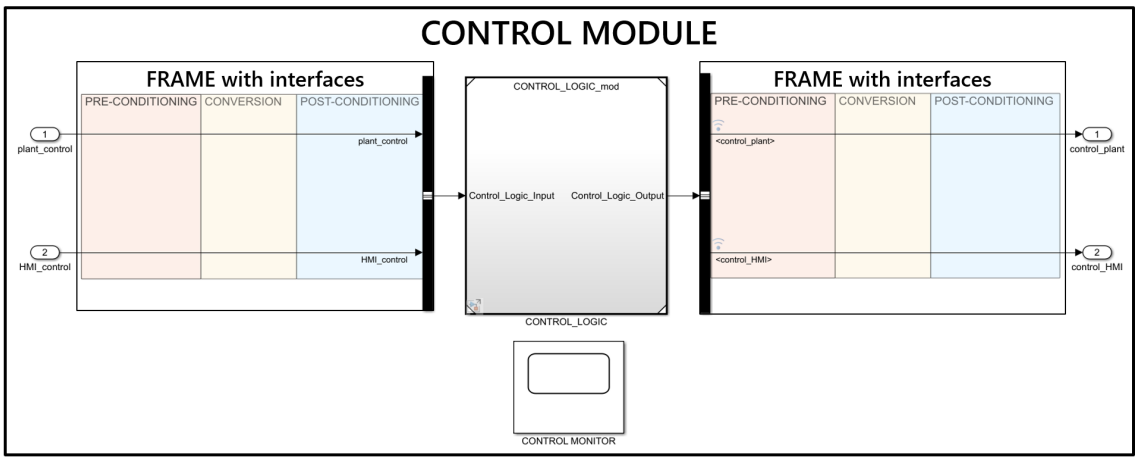


Figure 4.1: Control and Control_Logic

Figure 4.1 shows the two different subsystems where the frame is represented by the coloured areas that are empty because no target hardware is considered. When the hardware is selected its characteristics have to be included in the Control module. In particular, in the apposite areas (Figure 4.1) representing the Control_Logic frame, the target hardware interfaces have to be modelled with the appropriate

Simulink library components when dealing with the steps 2, 2.1, 2.3 and 3 of the Hybrid V-cycle. When performing the steps that involve the Test Bench the interfaces that have to be included in the Control_Logic frame are related to the specific hardware libraries, e.g., dSPACE RTI1401 - MicroAutoBoxII DS1513. As a consequence, all the blocks used at the aforementioned steps have to be substituted with the ones belonging to the specific libraries. In addition, the Modular Technical Model inputs and outputs have to be removed because the hardware library components will constitute the input and the output signals of the computing platforms. Removing the MTM signals is allowed because the Control is a referenced model that is independent of the main model. Moreover, when deploying the application code on the target hardware only the Control_Logic subsystem is involved. The MTM does not add any other contributes. Therefore it is not considered and only the model that comprises the Control content is used to manage the computing platforms. In particular, the Control_Logic for the code generation and the frame (Control excluding the Control_Logic) to represent the hardware interfaces.

4.1 Control module characteristics

- The control law is *not dependent* on the development platform.
- The Control content (excluding that of the Control_Logic) is the frame and it is the only part that is dependant on the target hardware.
- The Control_Logic subsystem has to be evaluated by means of the *fixed-step solver*.
- The Hybrid V-cycle has to be used as a guideline for the Control module development.

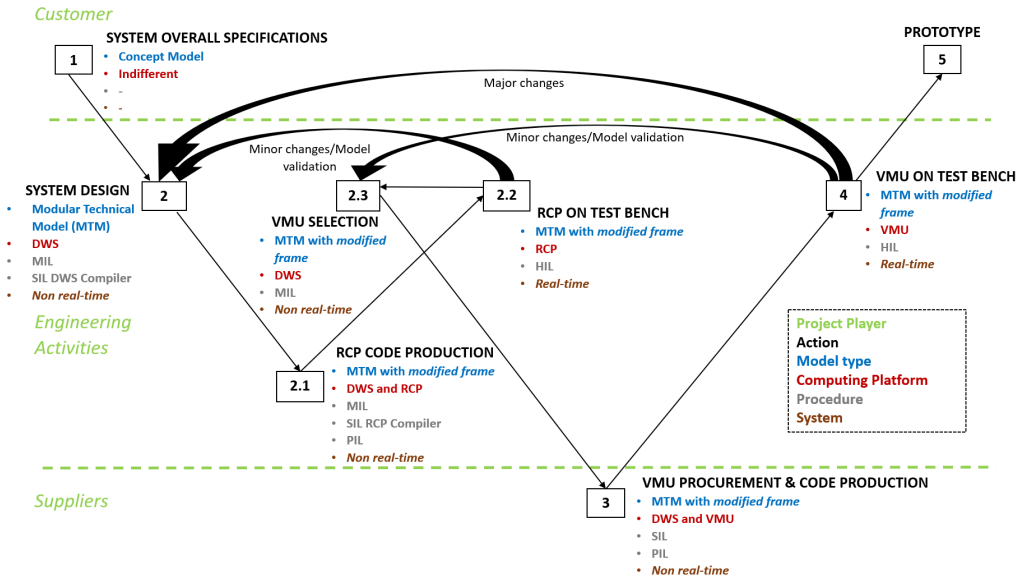


Figure 4.2: Hybrid V-cycle

2. *System Design*
   The hardware interfaces have to be included in the Control module. Since the DWS is the only computing platform involved at this step Simulink blocks are used to model generic interfaces (Table 3.1) that are still not associated to the RCP platform. This is needed to detect the requirements that are used to choose the most appropriate RCP platform, e.g., dSPACE MicroAutoBox II among all the available devices. The SIL procedure is carried out considering the DWS compiler.
   The next steps are used to differentiate the RCP platform and the VMU characteristics starting from the interfaces model of the step 2. The frame will comprise almost the same components of the Control module developed at step 2 but with different parameters.

2.1. *RCP Code Production*
   The frame comprises the RCP platform interfaces. The Control model developed at the step 2 has to be modified considering the device characteristics. The corresponding technical manual is used as a reference. In the specific case of the dSPACE MicroAutoBox II as RCP platform either the configuration manual [**8**] or the brochure [**14**] can be used.

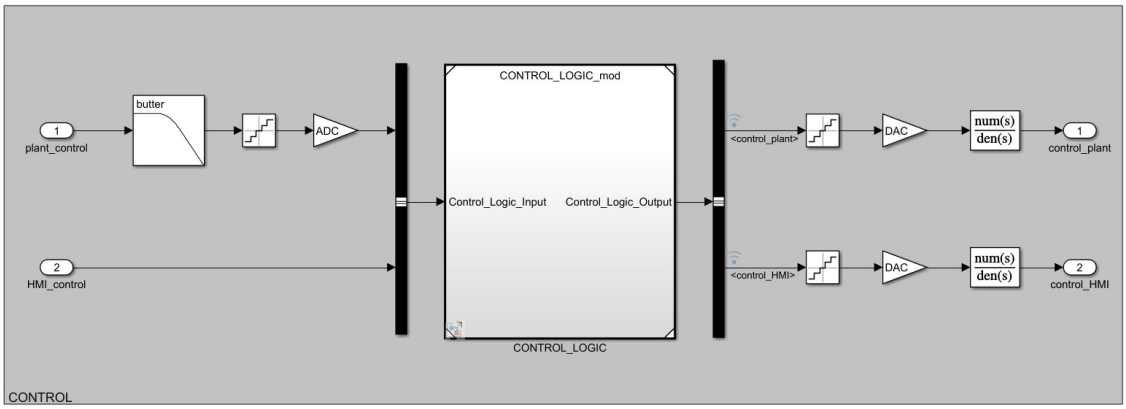


Figure 4.3: Example of dSPACE ADC/DAC modelling [**14**]

2.2 *RCP on Test Bench*
   At this step only the model that comprises the Control content is used regardless of the MTM. All the frame components of the step 2.1 have to be removed including the MTM inputs and outputs. All these elements have to be replaced by specific blocks of the RCP platform library. These blocks are needed to create the interaction between the RCP hardware and the Test Bench because they represents the physical interfaces. As a result the Control model will contain the Control_Logic and the specific blocks of the RCP platform library.

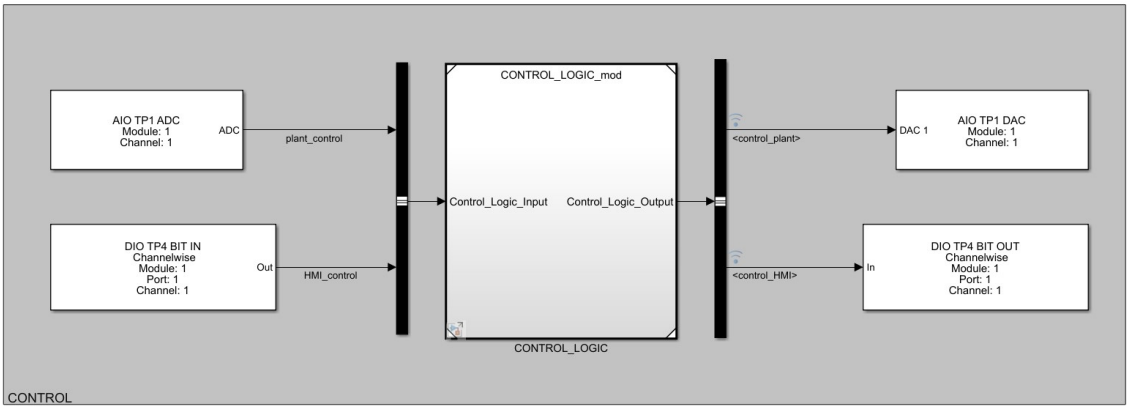


Figure 4.4: Example of Control_Logic frame when considering the dSPACE MicroAutoBox II as RCP platform and when performing the HIL procedure

2.3 *VMU Selection*
This is conceptually equal to the step 2 but considering the VMU. The interfaces have to be modelled and included into the frame. In the specific case of Ideas & Motion VMU the *Compass ECU - Product Brief* [**15**] can be used as a reference.

3 *VMU Procurement & Code Production*
The frame is the same as the one of the 2.3 step.

4 *VMU on Test Bench*
The content of the frame has to be changed removing the components included at the step 2.3. The specific VMU blocks have to be used to represents the physical interfaces (as in the 2.2 step). As a result the Control will contain the Control_Logic and the specific VMU components.

- The Control_Logic subsystem is used to automatically extract the code (Figure 3.1).

4.2 Introduction to the code generation

The Automatic Code Generation is the outcome of the following procedures:

- *Model-in-the-loop*
  The system is modelled and simulated on the DWS *non in real-time.* This type of simulation lets verify the control algorithm design and log the simulation results as a reference for the next verification step (SIL).

- *Software-in-the-loop*
  The code implementing the control algorithm is automatically extracted and compiled. Then it can be simulated to check the results. This type of simulation allows the fixed-point effects analysis like quantization error or saturation and overflows.

- *Processor-in-the-loop*
  The generated code is compiled with *target compiler* and simulated on target hardware (RCP, VMU) *non in real-time*. This type of simulation lets perform final verification on the *target processor*, e.g., any target related issues. In addition, informations on the code size, stack consumption and execution time of the generated code can be determined.
  When performing the PIL procedure the DWS and the target hardware are physically connected. The Control_Logic code is deployed on the RCP platform and VMU. During the simulation the evolution of the model is not constant. The different iterations (samples) are executed whenever the DWS communicates with the target hardware.

- *Hardware-in-the-loop*
  The system behaviour is checked by means of dedicated Test Bench, being executed *in real-time*. During the HIL testing the target hardware is synchronized with the system. The frame is represented by the real interfaces (Figure 4.4) that are used to create a connection between the Control_Logic responsible of the control algorithm and the external instruments. Those instruments are the Test Bench and they are used to represent the physical system which is no longer simulated on the DWS.

4.3 Multi-task application

The Control_Logic module can be designed in different ways. The idea is to model the Control_Logic as a multi-task application. In this case, some considerations have to be taken into account. A multi-task application is a set of separate tasks or processes with priorities that can be assigned arbitrarily in relation to the importance of the single task. The tasks are *periodic* or *synchronous* and *aperiodic* or *asynchronous*. There is also a background task that is not a task in the common sense, meaning that it has no priority.
The Control_Logic is composed of a generic number of tasks that are responsible of the system functionalities and that are regularly executed if an *interrupt* does not stop them. The interrupt is a request for the execution of a particular task and it can come from different sources, e.g., I/O device. Interrupts can be *internal* or *external events* and periodic or aperiodic. They can also be differentiated into *software interrupt* triggered by the timer device on the processor and *hardware interrupt* triggered by external events. Each model has at least a task driven by an interrupt block [**16**].
Each task is associated to a *state*, e.g., running, ready, idle. It defines the process working condition during its execution in fact as a task runs it changes the state. This leads to define the *scheduler* of the processor. The scheduler implements the task state transitions. It performs two operations that are: a *scheduling algorithm* that is used to decide the state of the tasks and a *context switch* needed to implement the actions that the scheduling algorithm has to perform to change the process that is running.
An example of scheduler is the *priority-based preemptive* that supports the *rate-*

monotonic scheduling (RMS) strategy. By means of the priority of a task, the scheduler decides whether it should be started immediately or if it has to wait until a higher-priority task is completed. Hence, higher-priority tasks interrupt lower-priority tasks. This is the scheduler of the dSPACE MicroAutoBox II platform.

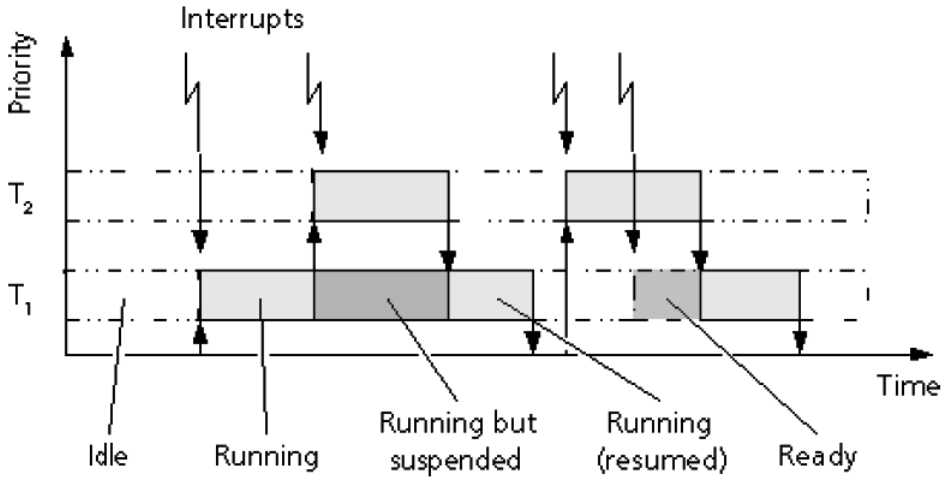


Figure 4.5: Generic multi-task application

Figure 4.5 shows a dSPACE generic multi-task scenario with two periodic tasks (T1 and T2), a background task and asynchronous interrupts.
Depending on the priorities and current states of the tasks, the scheduler executes them according to the following rules:

- A high-priority task that is triggered always suspends a low-priority task that is currently running.
- If no high-priority task is triggered, the suspended low-priority task resumes execution.
- Tasks of the same priority do not suspend each other if they are triggered, but follow a first come, first served policy.
- As long as all other tasks are idle, the background task is executed [**17**].

When designing a multi-task application an *overrun* situation can occur. This means that a task is requested to start but has not finished its previous execution yet [**18**].

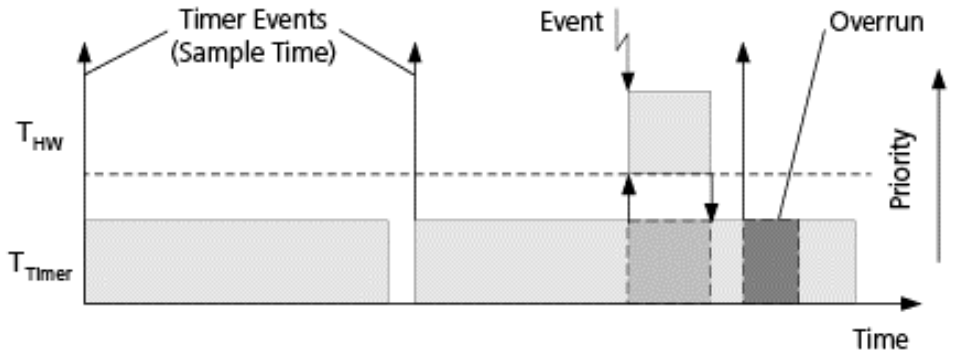


Figure 4.6: Example of dSPACE overrun situation [**18**]

Considering the Figure 4.6 between the first and the second timer interrupt there is no overrun, but if the hardware-interrupt-block-driven task T_{HW} needs to be calculated, there is not enough time for the timer task T_{Timer} to finish until it is requested to start again. To avoid overrun situations time values have to be correctly setted. For each task the sample time has to be greater than the sum of the *task-switching time* that is the delay between the occurrence of an interrupt on the hardware and the execution of the corresponding task's first statement and the *turnaround time* that is the time that passes between the triggering and the end of the task execution. (It can include the time required by higher-priority tasks that interrupt it), [**18**].
In order to make the application more compliant with the Control_Logic, the tasks have to be accurately differentiated:

- *Synchronous tasks*
  The periodic tasks are used to characterize the system. The amount of tasks depends on the application.
  - *Supervisor*
    It is a synchronous task that starts or stops the application. In particular, it manages the execution of the other synchronous tasks responsible of the system functionalities.
- *Asynchronous tasks*
  An aperiodic task interrupts the execution of all the tasks, mainly during emergency situations. This is important for *safety reason*: an application must be stopped if a misbehaviour is detected.

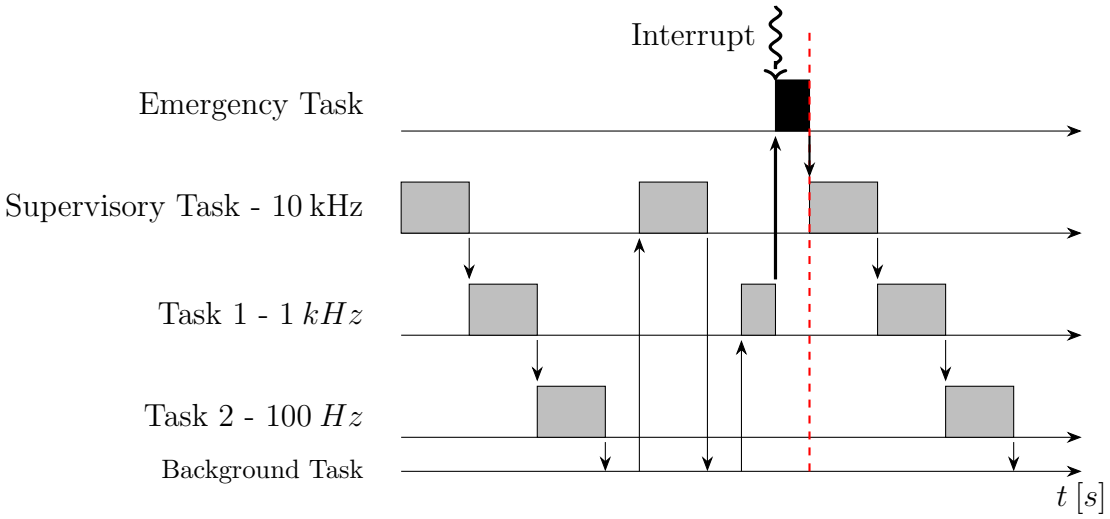


Figure 4.7: Multi-task application with different task sample rates

Figure 4.7 shows the multi-task application that has to be implemented. It consists in three synchronous tasks that are the Supervisor, the Task 1 and the Task 2 and an asynchronous task that is the Emergency Task. The priorities are: $Emergency\,Task > Supervisor > Task\,1 > Task\,2$.
The Emergency Task is used to stop the application in case of potential dangerous situations. It is sporadically run during the execution of the periodic tasks. It is invoked when the interrupt signal is generated. The interrupt informs the processor that an emergency task must be executed as soon as possible. At the end of the Emergency Task the task queue is cleared and restored depending on the priority of each task.
The application that has to be implemented has the Emergency Task that runs in response to an asynchronous interrupt generated by external components, e.g., toggle switch.

4.3.1 Multi-task implementation

- *periodic* or *synchronous* ⇒ by default the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority tasks is used for processing lower priority tasks [**5**].
  The periodic tasks has to be included into the design process to define the system functionalities. The most important synchronous process is the Supervisor. It is devoted to the tasks management and it is used to handle the external events and commands, e.g., it receives the signals from buttons. In particular, it responds to two different types of external phenomena. The external signal can come from a push button that is used as an ON/OFF strategy to enable the system and the corresponding periodic tasks. Therefore this scenario is related to the user dynamics represented by the HMI module.

As the push button is activated by user actions, the Supervisor receives the corresponding signal through the HMI. It reacts to the user commands by enabling (ON) or disabling (OFF) the synchronous tasks.
The external signals can also be generated by sporadic events related to potential dangerous situations. As the event is produced an asynchronous interrupt reaches a task specially created for emergency reasons. This emergency task generates a signal that is received by all the tasks including the Supervisor and that is used to stop their execution.

- *aperiodic* or *asynchronous* ⇒ there may not necessarily be a relationship between sample rates and task priorities. The tasks with the highest priority need to have a sample rate that is not necessary the fastest one. The sense of what priority numbers mean can be switched by selecting or deselecting the Solver option *Higher priority value indicates higher task priority* [**5**].
The aperiodic task that is related to emergency situations has to be included into the system development for safety reasons. This is sporadically executed in response to asynchronous interrupts triggered by external events. The events are related to specific switches used in case of dangerous situations. In order to ensure safety, the emergency task has to be associated to the highest priority so that when the event is generated the execution of all the other tasks is stopped and it can be resumed only when the emergency is terminated.

4.3.2 Simulink implementation

The procedures needed to correctly implement a Simulink multi-task application have to be differentiated.

Modelling phase

When modelling a multi-task application there are two main approaches that can be used to handle the tasks. The *explicit* approach includes all the processes in the same model and at the same level (Figure 4.8). This means that the user can directly interact to all the tasks.

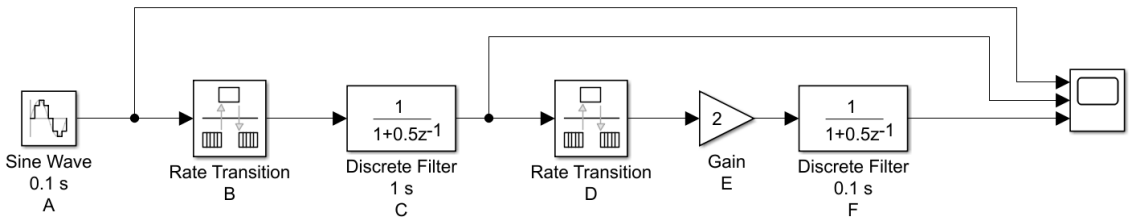


Figure 4.8: Example of a model with Rate Transition blocks - Digital Filters in series

The processes may have different sample rates and then the *Rate Transition Blocks* have to be used with the *Ensure data integrity during data transfer* and *Ensure deterministic data transfer (maximum delay)* options on. This allows the interaction among all the tasks both synchronous processes and asynchronous ones.

The *implicit* approach uses Stateflow to define the processes. Each task is represented by a Stateflow chart. This allows to clearly separate the periodic processes *without using the Rate Transition blocks*. When dealing with asynchronous tasks the Rate Transition blocks have to be used to create the interaction between the aperiodic processes and the periodic ones.

The chart settings have to be selected:

- Open the chart.
- Click on *Modelling* ⇒ *Chart Properties*.
- Click on Update method ⇒ *Discrete* ⇒ *Sample Time* ⇒ write the desired value.

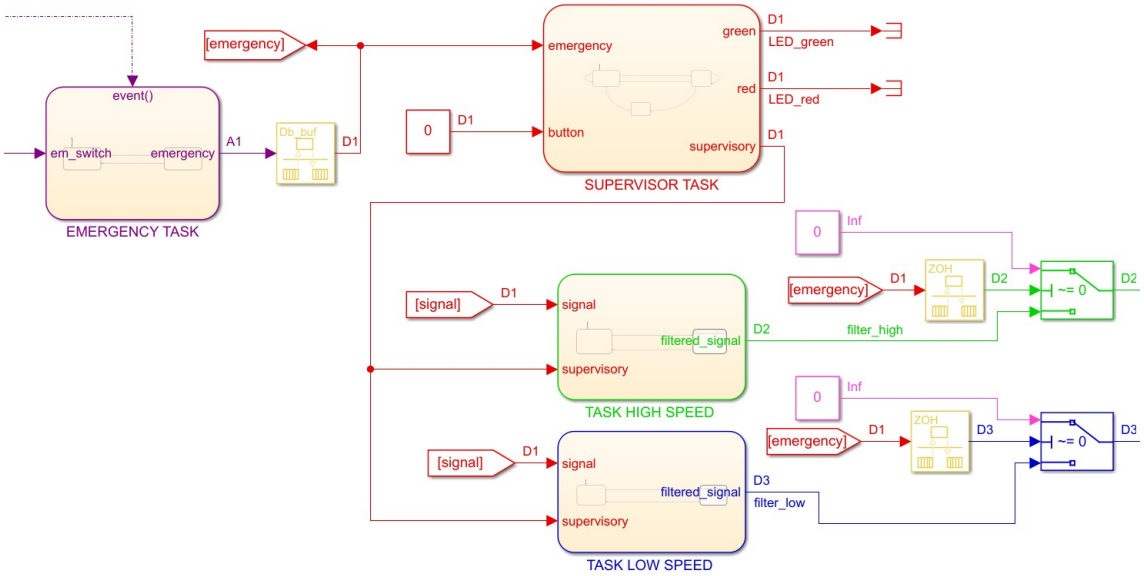


Figure 4.9: Example of Control_Logic content that represents a multi-task application with three different synchronous tasks and an aperiodic process modelled by different Stateflow charts

Configuration phase

- Click on *Configuration Parameters* ⇒ *Solver*.
  - *Solver Selection* ⇒ *Type* ⇒ select *Fixed-step* ⇒ restrictions:
    - The sample rate of a block must be an integer multiple of the base (that is, the fastest) sample period.
    - When *Periodic sample time constraint* is *Unconstrained*, the base sample period is determined by the *Fixed-step size*.
    - When *Periodic sample time constraint* is *Specified*, the base rate fixed-step size is the first element of the sample time matrix that you specify in the companion option *Sample time properties*.

    - Continuous blocks execute by using an integration algorithm that runs at the base sample rate. The base sample period is the greatest common denominator of all rates in the model only when *Periodic sample time constraint* is set to *Unconstrained* and *Fixed-step size* is *Auto*.
    - The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part and is an integer multiple of the base sample rate [**5**].

- *Solver Selection* ⇒ *Solver* ⇒ a possible solution is *auto*. If the latter is selected, the model has to contain at least two different sample times, to create a multitasking environment.
- *Solver details* ⇒ *Fixed-step size* ⇒ a possible solution is *auto*. If this quantity has to be associated with a particular value, it has to be different with respect to the discrete sample time in a model containing continuous and a discrete sample time. Otherwise the model runs in single-tasking mode.
- *Solver details* ⇒ *Tasking and sample time options* ⇒ select *Treat each discrete rate as a separate task* [**5**].

Chapter 5

Control code generation

The code of the Control_Logic module has to be automatically generated and compiled. Then the results of the code generation can be checked to verify whether the code behaves as expected.
Procedures:

- *SIL (Software-in-the-loop).*
- *PIL (Processor-in-the-loop).*

When dealing with target hardware that have a different System Target File with respect to that of the DWS, a specific tool has to be used to perform the SIL and PIL simulations. The RCP platform has a tool devoted to the SIL and PIL simulations.

5.1 SIL (Software-in-the-loop)

The Software-in-the-loop aims at generating and compiling the code implementing the control algorithm of the Control_Logic module. It is related to the steps 2, 2.1 and 3 of the Hybrid V-cycle. In order to perform the SIL process, the first step consists in check whether the code can be produced. *Not all the Simulink blocks can be used for code generation because they may not be supported for that functionality.* Then the code can be compiled. In order to accomplish the task, the Simulink configuration parameters have to be correctly setted:

- *Hybrid V-cycle step 2*
  In *Code Generation* ⇒ *System Target File* ⇒ select the DWS compiler, e.g., *grt.tlc* (Generic Real-Time Target).
- *Hybrid V-cycle step 2.1*
  In *Code Generation* ⇒ *System Target File* ⇒ select the RCP compiler, e.g., *rti1401* (dSPACE MicroAutoBox II).
- *Hybrid V-cycle step 3*
  In *Code Generation* ⇒ *System Target File* ⇒ select the VMU compiler.

The Control_Logic code has to be generated and compiled. Simulink allows to generate and compile the code by clicking the apposite "*Build*" icon.

5.1.1 SIL simulation

The SIL simulation is used to verify the behaviour of the production source code. The output of software-in-the-loop code have to match that of the Control_Logic subsystem. This procedure can be executed at the steps 2, 2.1 and 3 of the Hybrid V-cycle.
The following procedure cannot be executed considering the RCP platform. A specific RCP tool has to be used to perform the SIL simulation.

Procedure

1. Open the MTM.
2. Consider the subsystem (Control_Logic block) that is intended to be used for the SIL simulation and set the correct parameters.
   - In *Block Parameters* ⇒ *Main* ⇒ select *Treat as atomic unit.*
3. In *Configuration Parameters* ⇒ *Code Generation* ⇒ *System target file* ⇒ select *ert.tlc.*
4. *Create a SIL Verification Harness*
   (a) Enable signal logging for the model. At the command prompt, enter:

```
set_param(bdroot,'SignalLogging','on','SignalLoggingName',...
'SIL_signals','SignalLoggingSaveFormat','Dataset')
```

   (b) Right-click the input signals into Controller ports, and select *Properties.* In the Signal Properties dialogue box, for the Signal name, enter a desired name (e.g. 'in1', 'in2', etc). Select *Log signal data* and click OK.
   (c) Right-click the signals out of Controller ports, and select *Properties.* In the Signal Properties dialogue box, for the Signal name, enter a desired name (e.g. 'out1', 'out2',etc). Select *Log signal data* and click OK.
   (d) simulate the model.
   (e) Get the logged signals from the simulation output into the workspace. At the command prompt, enter:

```
out_data = out.get('name of the model');
control_in1 = out_data.get('name of the input signal of the
    controller');
control_out1 = out_data.get('name of the output signal of the
     controller');
```

These statements are written assuming only one input and one output.

(f) Create the SIL test harness.
Right-click the Controller subsystem and select *Test Harness* ⇒ *Create Test Harness (Controller)*.

(g) set the Harness properties

Name: e.g. *SIL_harness*
Sources and Sinks: *Inport and Outport*
Select ⇒ *Open harness after creation*
Advanced Proprieties ⇒ Verification Mode: *Software-in-the-loop (SIL)*

(h) Click OK. The resulting test harness has a SIL block.

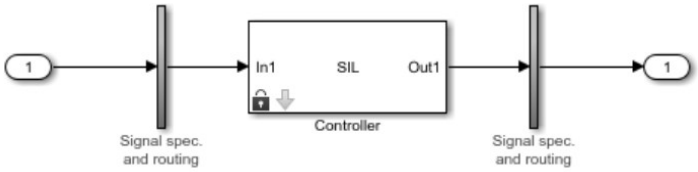


Figure 5.1: Test Harness Example

5. *Configure and Simulate a SIL Verification Harness*

(a) Configure the test harness to import the logged controller input values. From the top level of the test harness, in the model *Configuration Parameters* dialogue box, in the *Data Import-Export* pane, select *Input*. Enter *control_in*1.*Values* (considering one input signal) as the input and click OK.

(b) Enable signal logging for the test harness. At the command prompt, enter:

```
set_param('harness name','SignalLogging','on','
    SignalLoggingName',...
'harness_signals','SignalLoggingSaveFormat','Dataset')
```

(c) Right-click the output signals of the SIL block and select *Properties*. In the Signal Properties dialogue box, for the Signal name, enter a desired name e.g. *SIL_block_out*. Select *Log signal data* and click OK.

(d) simulate the harness.

6. *Compare the SIL Block and Model Controller Outputs*

(a) In the test harness model, in the *Review Results* section, click *Data Inspector* to open the Simulation Data Inspector.

(b) In the *Simulation Data Inspector*, click *Import*. In the Import dialogue box,

Set Import from to: *Base workspace*.

Set Import to to: *New Run.*
Under Data to import select *Signal Name to import data from all sources.*

(c) Click Import.

(d) Select the output signals of the SIL model and of the Controller in the *Runs* pane of the *Data Inspector* window.

Figure 5.2: Example of Runs pane

If the 2 signals overlap, there is an equivalence between the SIL code and the Controller code. This is the expected result.

(e) Close the test harness window. This results into a return to the main model.
The badge on the Controller block indicates that the SIL harness is associated with the subsystem [**6**].

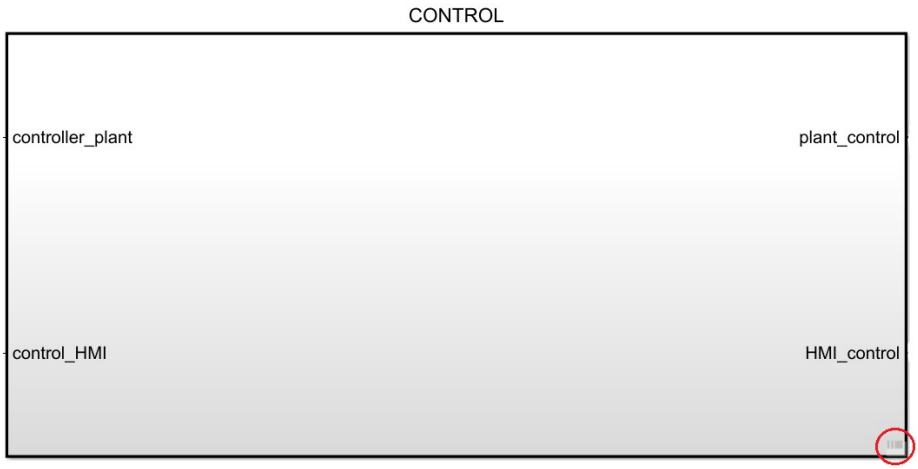


Figure 5.3: Control badge

5.2 PIL (Processor-in-the-loop)

The PIL simulation is used to verify the compiled object code that is intended to be deployed in production. The PIL object code is run on real target hardware (dSPACE, VMU). It is performed at the steps 2.1 and 3 of the V-cyle in order to check the technical model that considers the RCP (step 2.1) and VMU (step 3) characteristics.
The following procedure cannot be applied on the dSPACE RCP platform, a specific tool belonging to the RCP software has to be used.

Procedure

The procedure is the same as the SIL simulation, except for a Harness property:

Advanced Properties ⇒ *Verification Mode: Processor-in-the-loop (PIL)*

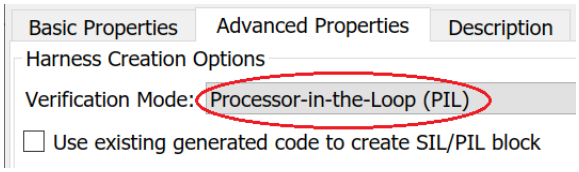


Figure 5.4: PIL Verification Mode

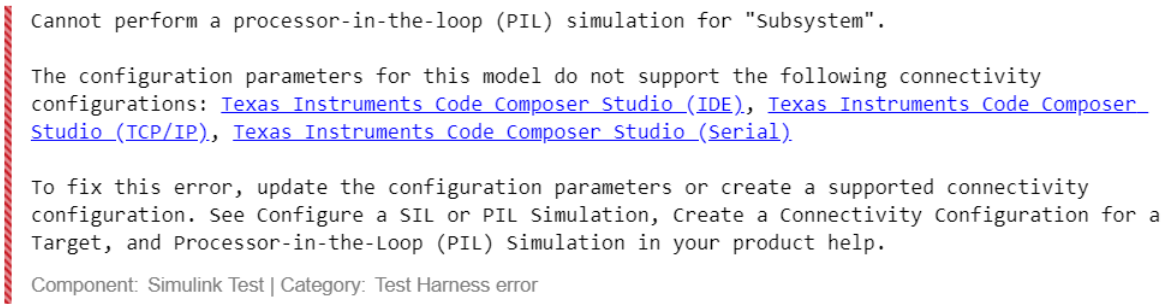


Figure 5.5: Test Harness error

A very common *error* can occur during the simulation (Figure 5.5) and to solve this problem an intermediate step has to be performed.

Configuration of Processor-In-The-Loop (PIL) for a Custom Target

This procedure is used to create a target connectivity configuration by using target connectivity APIs. With a target connectivity configuration PIL simulations can be run on custom target hardware.
The idea is to start from a model configured for SIL simulation and then create a target connectivity configuration, so that the model can be simulated in PIL mode.
The procedure is performed considering the DWS characteristics to keep complexity low.

1. *Preliminaries*

   - Add a folder to the search path. Create the folder path.

```
sl_customization_path = fullfile(matlabroot,...
  'toolbox',...
    'rtw',...
    'rtwdemos',...
    'pil_demo');
```

   - If this folder is already on the search path, remove it.

```
if strfind(path,sl_customization_path)
   rmpath(sl_customization_path)
end
```

   - Reset customizations.

```
sl_refresh_customizations
```

2. *Test Generated Code with SIL Simulation*

   - Simulate a model configured for SIL. Verify the generated code compiled for the host computer by comparing the SIL simulation behaviour with the normal simulation behaviour.

3. *Target Connectivity Configuration*

   - Start work on a target connectivity configuration for PIL.
   - Make a local copy of the target connectivity configuration classes.

```
src_dir = ...
fullfile(matlabroot,'toolbox','coder','simulinkcoder','+coder
    ','+mypil');
if exist(fullfile('.','+mypil'),'dir')
rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+
    mypil');
copyfile(fullfile(src_dir,'ConnectivityConFigurem'), '+mypil'
    );
```

   - Make the copied files writeable.

```
fileattrib(fullfile('+mypil', '*'),'+w');
```

   - Update the package name to reflect the new location of the files.

```
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConFigurem',...
    'coder.mypil',...
    'mypil');
```

- Verify that you now have a folder +mypil in the current folder, which has the files Launcher.m, TargetApplicationFramework.m, and ConnectivityConFigurem.

```
dir './+mypil'
```

4. *Review Code to Launch the PIL Executable*

   - The class that configures a tool for launching the PIL executable is mypil.Launcher. Open this class in the editor.

   ```
   edit(which('mypil.Launcher'))
   ```

   Review the content of this file. The method setArgString supplies additional command-line parameters to the executable. These parameters can include a TCP/IP port number. For an embedded processor implementation, you can choose to hard-code these settings.

5. *Configure the Overall Target Connectivity Configuration*

   - View the class mypil.ConnectivityConFigure

   ```
   edit(which('mypil.ConnectivityConfig'))
   ```

   Review the content of this file. You should be able to identify:

   – The creation of an instance of rtw.connectivity.RtIOStreamHost Communicator that configures the host side of the TCP/IP communications channel.
   – A call to the setArgString method of Launcher that configures the target side of the TCP/IP communications channel.
   – A call to setTimer that configures a timer for execution time measurement.

   To define your own target-specific timer for execution time profiling, you must use the Code Replacement Library to specify a replacement for the function code_profile_read_timer. Use a command-line API or the crtool user interface.

6. *Review the Target-Side Communications Drivers*

   - View the file rtiostream_tcpip.c.

   ```
   rtiostreamtcpip_dir=fullfile(matlabroot,'toolbox','coder','
       rtiostream','src',...
       'rtiostreamtcpip');
   edit(fullfile(rtiostreamtcpip_dir,'rtiostream_tcpip.c'))
   ```

Scroll down to the end of this file. See that this file contains a TCP/IP implementation of the functions rtIOStreamOpen, rtIOStreamSend, and rtIOStreamRecv. These functions are required for the target hardware to communicate with the host computer. An implementation for each of these functions has to be provided. This is specific to the target hardware and communication channel.

7. *Add Target-Side Communications Drivers to the Connectivity Configuration*

   - The class that configures additional files to include in the build is mypil.Target ApplicationFramework. Open this class in the editor.

```
edit(which('mypil.TargetApplicationFramework'))
```

8. *Use sl_customization to Register the Target Connectivity Configuration*
   To use the new target connectivity configuration, an sl_customization file has to be provided. The sl_customization file registers the new target connectivity configuration and specifies the required conditions for its use. The conditions specified in this file can include the name of the system target file and the hardware implementation settings.

   - You can view the sl_customization file.

```
edit(fullfile(sl_customization_path,'sl_customization.m'))
```

   - Add the sl_customization folder to the search path and refresh the customizations.

```
addpath(sl_customization_path);
sl_refresh_customizations;
```

9. *Test Generated Code with PIL Simulation*

   - Run the PIL simulation as the SIL one.

Chapter 6

HMI module structure

The *Human-Machine-Interface (HMI)* is a subsystem that represents the interfaces between the user and the Control module. It is related to all those instruments that can be used by humans to interact with the Control module, e.g., *Dashboard* and *Sinks* components.
As mentioned in Section 2.2, different target hardware are needed to develop a generic application. It is important to remember that the model of the system is *always* inherited from Simulink. When considering the dSPACE RCP platform it is *always* imported to the specific software related to the dSPACE device. This means that the Simulink model has to be adapted to the hardware (Chapter 4).
Concerning the Dashboard instruments (e.g., Push Button) there are two approaches, i.e., the components can be either *simulated* or *physical*. In the simulated case on the DWS, no particular attention have to be paid except for the choice of the components that are gained to be used (e.g., Simulink Push Button block is used to model a real button). Concerning the software that is used to manage the target hardware, the situation is exactly the same as the Simulink approach on the DWS. The RCP software has an *Instrument Selector* window in which several tools are available to be used for the specific purpose, e.g., Push Button and Knob. From those blocks, which simulate the behaviour of the instruments, the signal will be sent to the corresponding target hardware.
In the second approach, the simulated instrument of the Simulink model is substituted with blocks belonging to the specific target hardware library. Then, other considerations have to be taken into account. Noise, disturbances and tolerances affect the real instruments. When modelling an analogue button with the *Push Button* block in Simulink, no particular care is taken on the voltage (within reasonable limits) during the MIL simulation. Considering that the real instrument is connected to the target hardware, there is a tolerance that has to be considered (it depends on physical components). As a consequence, it has to be modelled in Simulink by means of additional blocks (e.g., *Relays* between the specific platform blocks and the Control module). After that all the previous considerations have been addressed, the target hardware can be used in conjunction with the external real Dashboard instruments.

Chapter 7

Rapid Control Prototyping (RCP)

This Chapter shows how to handle a RCP platform. There are different types of device, in this case the **dSPACE *MicroAutoBox II*** is considered.

Figure 7.1: dSPACE MicroAutoBox II

1. Configuration

This step is necessary to connect the dSPACE platform to the DWS.

- Power the dSPACE, considering the acceptable voltage range and check the colour of the corresponding led. It must be red.
- Power off the platform and connect it to the DWS via Ethernet, performing the following steps:
  - Disconnect the DWS from the network.
  - Open the *Control Panel*.
  - In the *Control Panel* select ⇒ *Network and Internet* ⇒ *Network Connections and Sharing Centre* ⇒ *Change adapter settings*.
  - Double click on *Ethernet* ⇒ *Proprieties* ⇒ double click on *TCP/IPv4*.
  - Click on *Use the Following IP address*.

- In the *IP Address* edit field ⇒ enter a value in the range 192.168.140.2 ... 192.168.140.254.
- In the *Subnet Mask* edit field ⇒ enter the value 255.255.255.0, [**8**].
- Power the dSPACE platform and check the connection to the DWS, looking at the *host PC* led. It must be green and lamping.

2. Verifications

This step is based on very simple models that are intended to verify that the results coming from the hardware testing match the characteristics reported on the technical manual of the dSPACE. Models that can be used for verification:

- *Sending signal from a generator to the dSPACE.*
- *Receiving signal on the oscilloscope from the dSPACE.*
- *Sending signal from a generator to the dSPACE and receiving that signal on the oscilloscope.*

3. dSPACE usage

A procedure has to be followed to use the *MicroAutoBox II* both at the verification step and then with the desired application.

1. *Replace the Control Block converters.*
   (a) Open the *dSPACE RTI14041 Library.*
      - This can be accessed via Simulink Library or via MATLAB, writing on the Command Window *rti*1401 (the latter method is suggested).
   (b) Use the dSPACE library components to model the interfaces.
      - The blocks are associated to different *DS numbers* (e.g. DS1513), that depend on the platform connector.

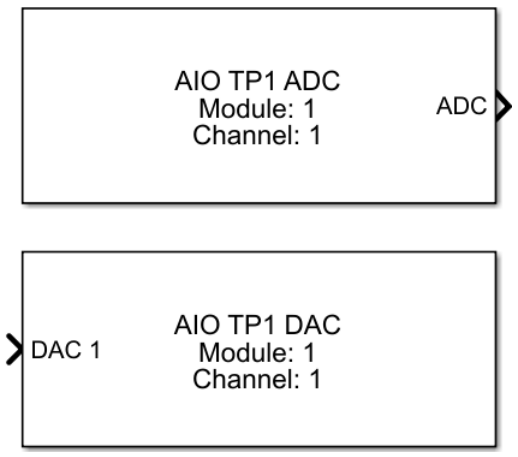


Figure 7.2: Example of dSPACE AD/DA converters

(c) Connect the interface blocks to the Control_Logic module, that contains the algorithm that is intended to be deployed on the dSPACE platform.

2. *Change the settings in the Configuration Parameters*

   (a) In *Solver* ⇒ *Solver selection* ⇒ choose ⇒ *Fixed-step discrete*

   (b) In *Math and Data Types* ⇒ deselect the following statement (recommended):

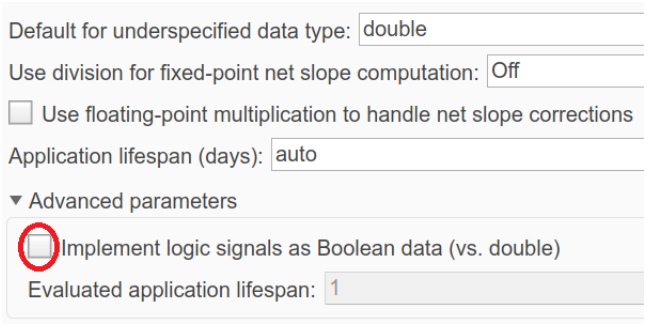


Figure 7.3: Math and Data Types

   (c) *Simulation Target* ⇒ deselect the following statements:

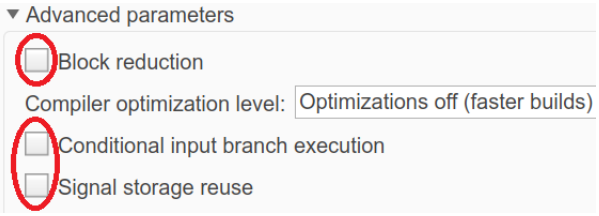


Figure 7.4: Simulation Target

   (d) In *Code Generation* ⇒ *System target file* ⇒ *Browse* ⇒ choose *rti1401.tlc*

   (e) In *Code Generation* ⇒ *RTI simulation options* ⇒ *Initial simulation state* ⇒ select *RUN*

   (f) In *Code Generation* ⇒ *RTI load options* ⇒ deselect the following statement (not mandatory) to avoid loading the application on the dSPACE platform before using the ControlDesk:

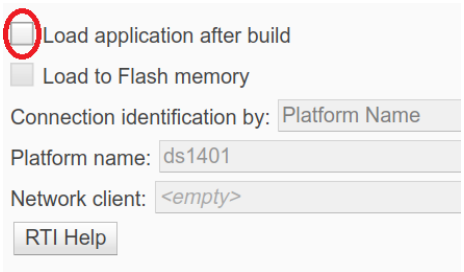


Figure 7.5: RTI load options

(g) In *Code Generation* ⇒ *RTI variable description* ⇒ follow the instructions (recommended):

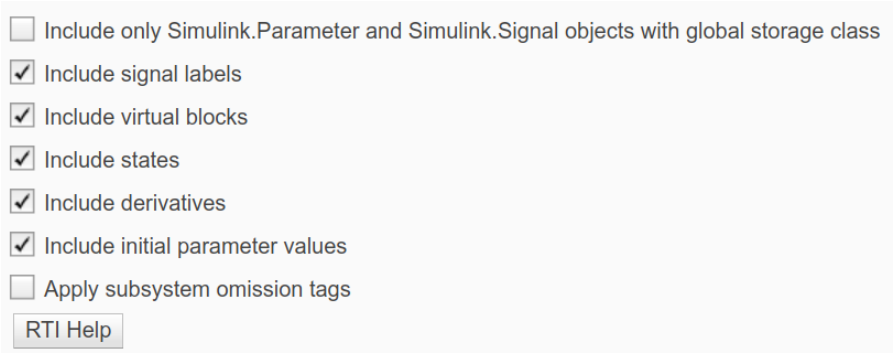
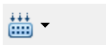


Figure 7.6: RTI variable description

3. *Generate the system description file (.sdf).*

   - Build the model by pressing *ctrl+B* or by clicking the appropriate button.

     – If 'Load application after build' option is selected the dSPACE has to be switched on before building the model.

4. *Open the ControlDesk software and use it to manage the signals and the dSPACE platform.*
   The ControlDesk is the dSPACE experiment software for seamless ECU development. It performs all the necessary tasks and gives you a single working environment, from the start of experimentation right to the end [**19**]. In particular, it can be used as the '*real-time version*' of the Control Monitor for model validation.

   (a) Register the platform.

      i. Click on *Platform/Devices* icon, at the bottom of the main window. A blank window will be opened.

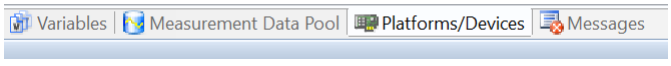


Figure 7.7: Platforms/Device icon

      ii. Right click on the blank window ⇒ click on *Register Platforms.*

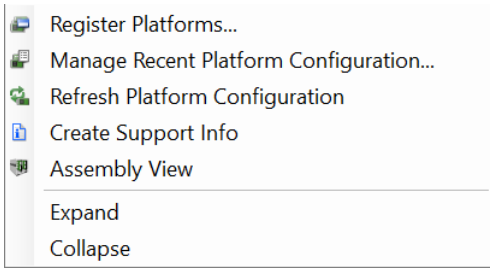


Figure 7.8: Register Platforms

iii. Start the registration procedure and choose the *MicroAutoBox II*.

iv. If 'Load application after build' option is selected the dSPACE is automatically registered in the ControlDesk.

(b) Click on *New* ⇒ *Project+Experiment* ⇒ perform the red steps:

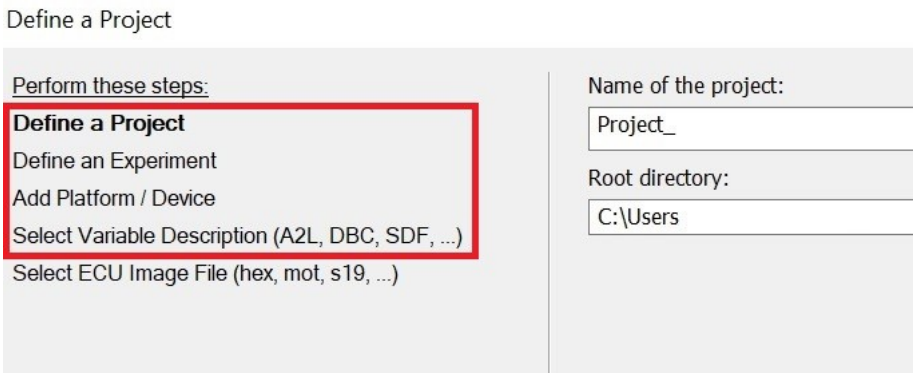


Figure 7.9: Procedure

i. A single project can contain different experiments. If a project already exists only the experiments will be created as new components.

ii. At the *Select Variable Description* step, import the desired *.sdf* file.

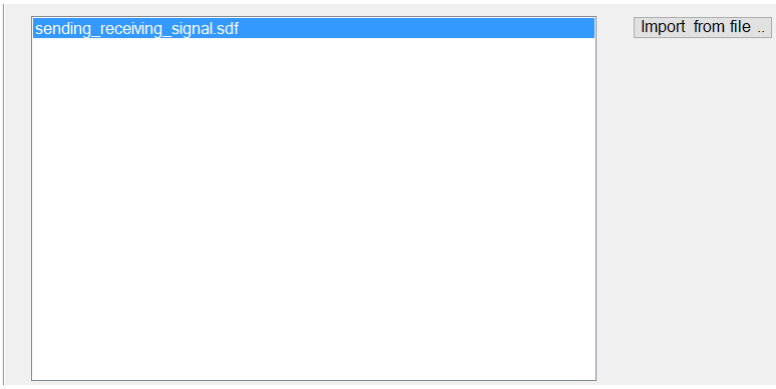


Figure 7.10: .sdf file selection

(c) Click on *Measurement Configuration* ⇒ *Acquisition* ⇒ *Platform* ⇒ *Host-Service* to check whether the *Sampling period* (on the *Properties* window that appears on the right) match that of the Simulink model.

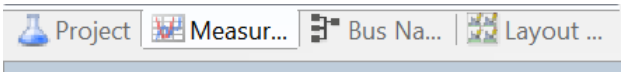


Figure 7.11: Measurement Configuration icon

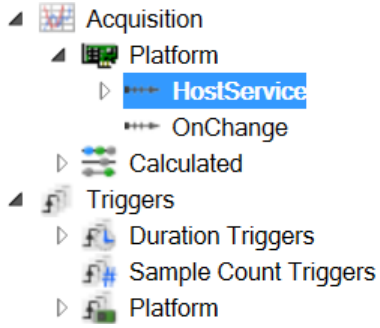


Figure 7.12: Measurement Configuration menu

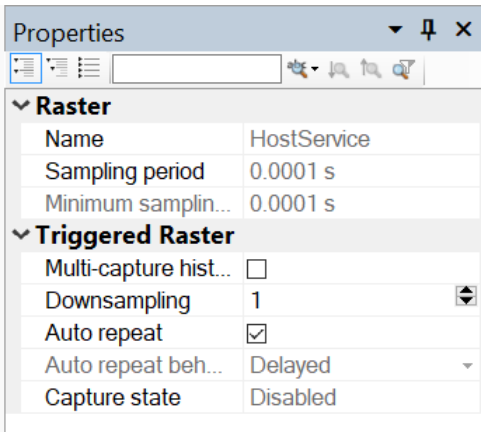


Figure 7.13: Properties

(d) Click on the *Variables* icon (Figure 7.7) ⇒ *Model Root* to see the Simulink blocks.

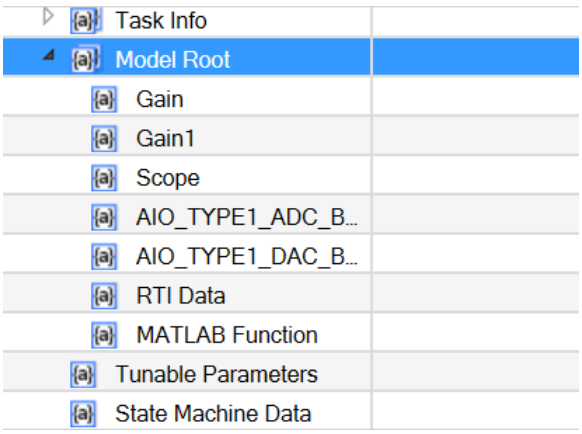


Figure 7.14: Model Root

i. Click on the block that is intended to use to make the *MicroAutoBox II* and the DWS interact, e.g., a *scope*.
ii. Drag the corresponding line on the window on the right to the empty grey space that is the ControlDesk layout.

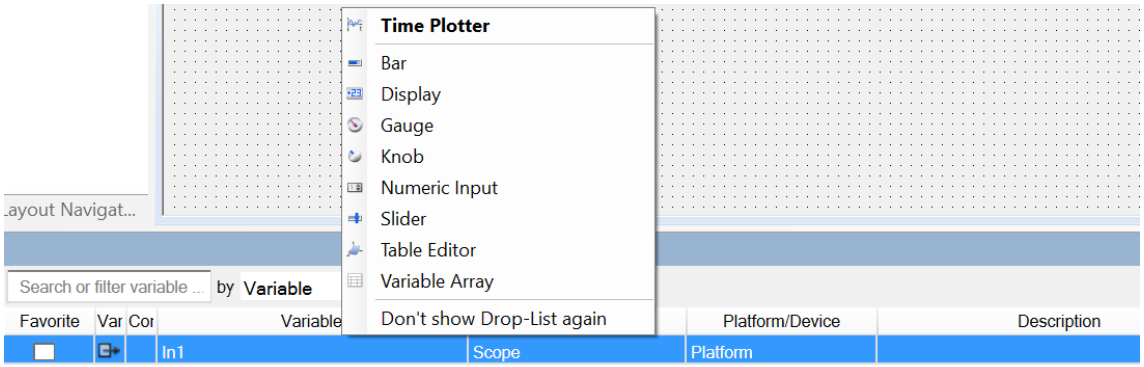


Figure 7.15: Block selection

iii. This opens a short menu, containing the *ControlDesk instruments* that can be associated to the Simulink blocks. In that case, the *Time Plotter* is suitable, because the objective is to see the time behaviour of the signal exiting the dSPACE platform.
iv. A further example is related to the button simulation. A button can be simulated via DWS and the signal can be sent to the *MicroAutoBox II*. The Push Button signal is modelled as a constant block in Simulink and associated to the ControlDesk Push Button (or Knob), located in the *Instrument Selector* window (on the right). Running the simulation the button is activated and, if pushed, it allows to check the dSPACE platform behaviour.

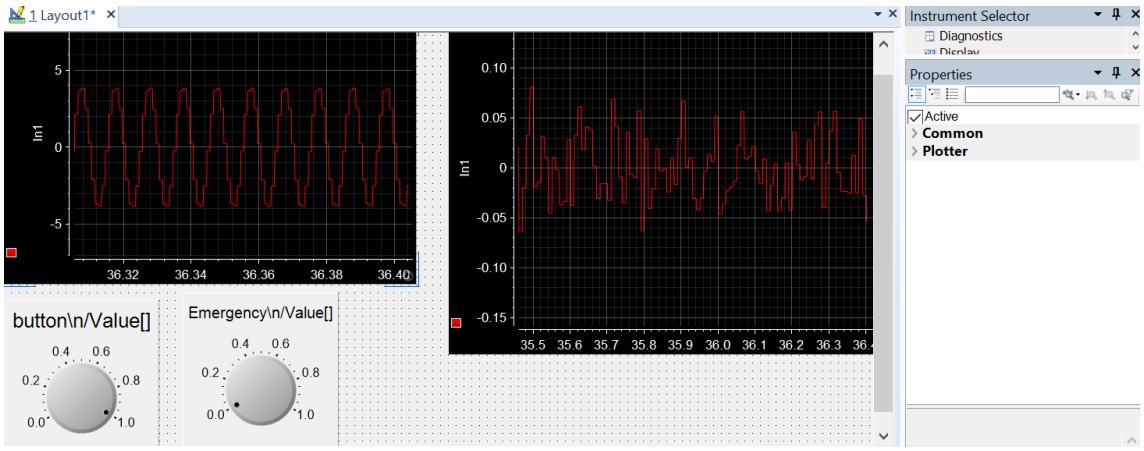


Figure 7.16: Layout example

Figure 7.16 shows an example of layout. In this case, the used buttons are simulated and they are associated to knobs, to see the value that makes the application start.

v. The ControlDesk intruments have different properties. On the layout, click on the desired instrument to open its properties (on the right, under the Instruments Selector as in Figure 7.16) and to manage them.

vi. Concerning the Time Plotter, the X and Y axes can be calibrated and subdivided in the desired range. Click slightly under the X axis and slightly left to the Y axis, when a particular shape of the mouse cursor appears.

Then on the *Properties* window (on the right), the axes settings will be opened.

(e) Click on the *Devices/Platform* icon ⇒ right click on *ds1401* ⇒ click on *Stop RTP* (the led near the connector will become red).
Before starting the measurements, it is recommended to stop the communication between the platform and the I/O board. (It depends on whether the 'Load application after build' option is selected).

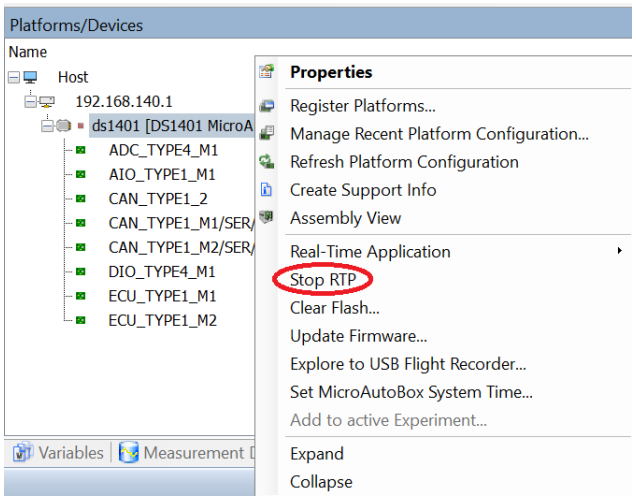


Figure 7.17: Stop RTP

(f) Click on *Start Measuring*

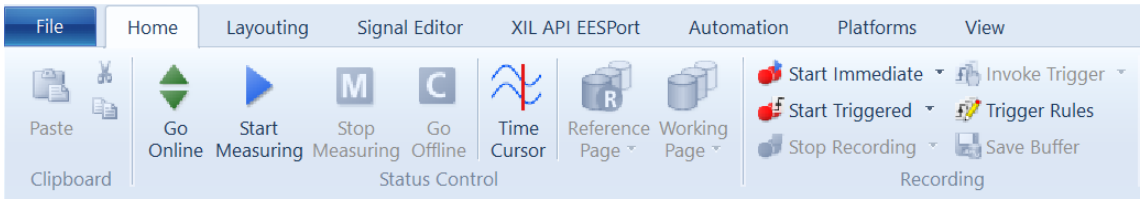


Figure 7.18: Measuring buttons

- The application will run on the dSPACE platform. Use the instruments in the layout to handle the hardware.

(g) Click on *Stop Measuring* to stop the application.

- Click again on *Start Measuring* to resume the application.
- Click on *Go Offline* and then on *Stop RTP*, to stop the *MicroAutoBox II*.

(h) If no measuring instruments are included in the ControlDesk layout, the 'Start Measuring' button will be not available. Clicking on 'Go Online' and then on 'Go Offline' will be sufficient to respectively make the application start and stop.

7.1 dSPACE TargetLink

TargetLink is a production code generator that generates highly efficient C code straight from MathWorks Simulink/Stateflow and allows early verification through built-in simulation and testing. It supports efficient, modular development [**9**]. In particular, TargetLink allows the code generation for MATLAB code contained in Simulink blocks, a faster testing and validation of production code in real environment and new features for improved distributed development [**10**].
For test and verification purposes early in the development process, TargetLink provides a powerful 3-steps built-in simulation support:

- *Model-in-the-loop (MIL) simulation.*
- *Software-in-the-loop (SIL) simulation.*
- *Processor-in-the-loop (PIL) simulation.*

Switching between the different simulation modes requires just one mouse click and thanks to the integrated data logging and plotting concept there is no need of a separate test model, manually insertion on doing a test harness model or writing the own plotting script. Furthermore, SIL and PIL simulation can be enhanced by code coverage analysis to identify code branches that were never executed or tested [**11**].
In order to use the dSPACE TargetLink, the corresponding library in the Simulink Library Browser has to be opened. The library contains all the needed blocks to perform the different mentioned simulations.

7.1.1 MIL simulation with TargetLink

In order to use TargetLink to perform a MIL simuation, the corresponding Target Link subsystem should be in model-in-the-loop simulation mode.

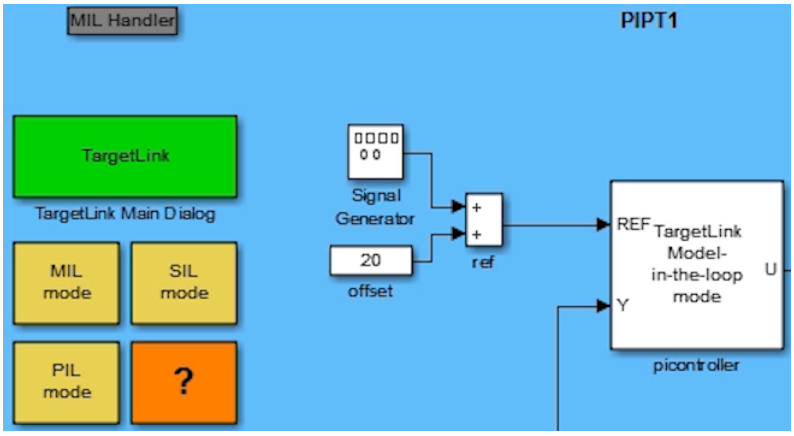


Figure 7.19: dSPACE example of a generic model in MIL simulation mode

For a quick overview of the simulation results TargetLink blocks have an integrated data logging functionality. This means that the model should not to be modified and the block dialogs can be used to specify if and how the output signal is logged during the simulation. As a result, during the next simulation run the TargetLink Plot Overview Window is displayed, showing the simulation results of all the logged signals.

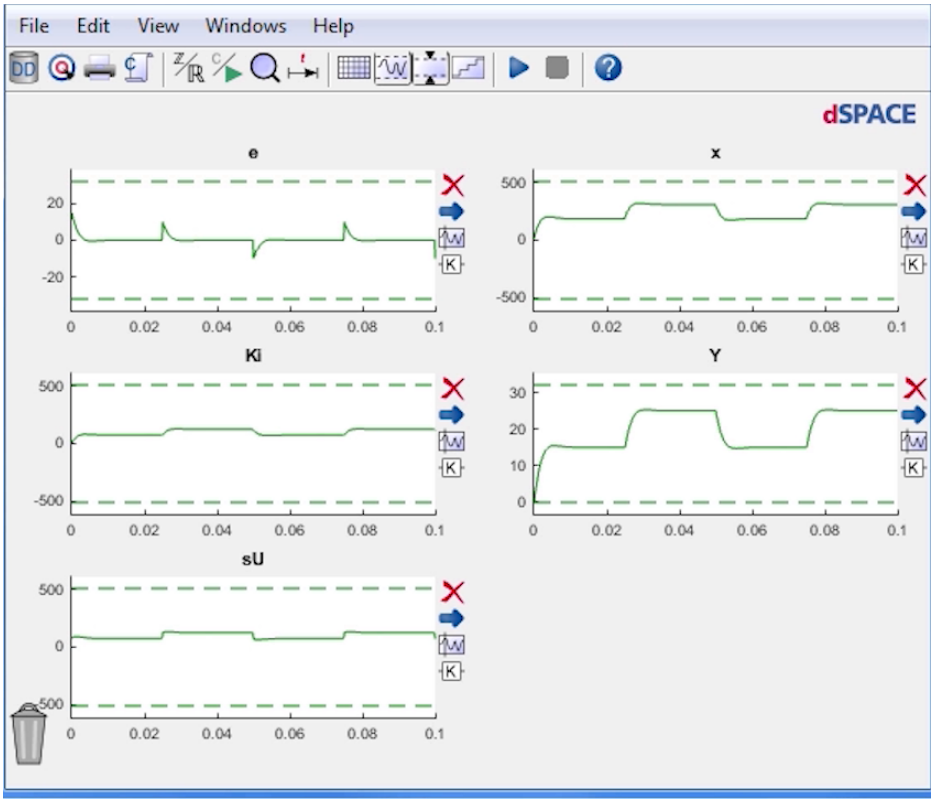


Figure 7.20: TargetLink Plot Overview Window considering a generic dSPACE example

A MIL simulation can serve, for instance, as a reference for overflow detection and autoscaling of fixed-point models or as a reference of subsequent production code simulations, e.g., SIL simulation [**11**].

7.1.2 SIL simulation with TargetLink

In SIL simulation the generated code of the Control Logic module is compiled and simulated again on the DWS. With this simulation, the fixed-point effects can be analysed, e.g., quantization errors or saturation and overflows. To perform the SIL simulation, first a suitable application has to be built. This is easily done in TargetLink by double-clicking the SIL mode button.

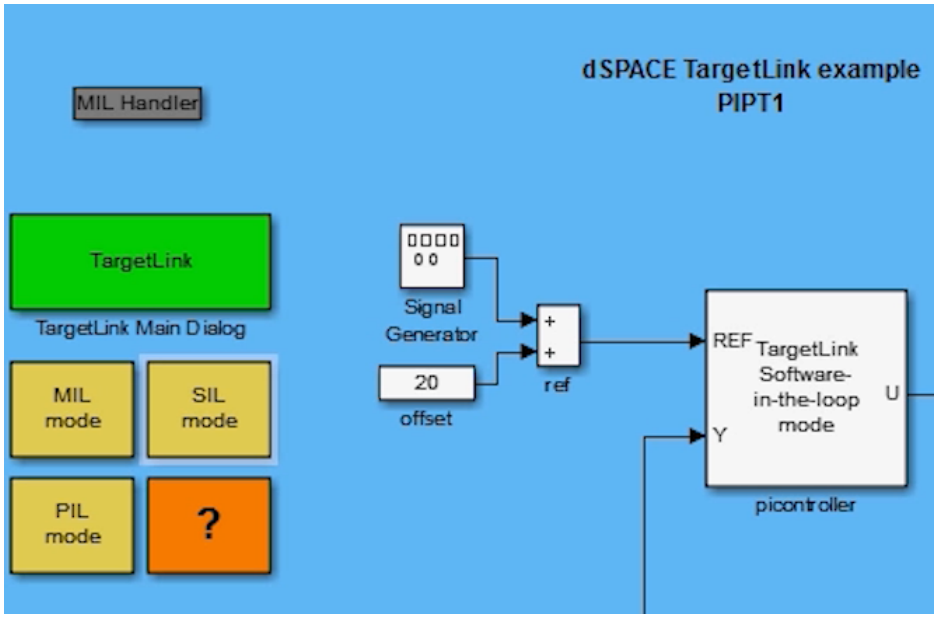


Figure 7.21: dSPACE example of a generic model in SIL simulation mode

The following steps are then executed automatically: the production code for the TargetLink subsystem is generated and the files are compiled and linked to a simulation application. The TargetLink subsystem is then in software-in-the-loop simulation mode and when the model is simulated again, the generated code of the Control Logic module is simulated instead of the Simulink blocks.
In the Target Plot Overview Window, the SIL simulation results are plotted on top of each MIL simulation result. Several helpful features are available, e.g., signals can be quickly compared by dragging them from one subplot to another. For detailed signal analysis, the additional plot windows can be opened showing, e.g., details on deviations of the SIL simulation from reference simulation, the numerical values of two compared signal amplitudes over time and much more.

7.1.3 PIL simulation with TargetLink

In PIL simulation, the generated code of the Control Logic module is compiled with the target compiler and simulated on the target hardware. Performing a PIL simulation is very easy with TargetLink. First, the target hardware has to be connected to the DWS and in the TargetLink Main Dialog, the appropriate combination of target compiler and target hardware have to be selected.

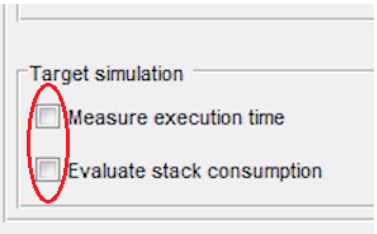


Figure 7.22: TargetLink Main Dialog code informations

For execution time and/or stack consumption of the generated code informations, the corresponding checkbox has to be selected (Figure 7.22).
Double click the PIL mode button in the model to generate the production code and compile it for the PIL simulation. TargetLink automatically manages the download and communication process between the DWS and the target hardware, so no further user interaction is required. The TargetLink subsystem is now in processor-in-the-loop simulation mode.

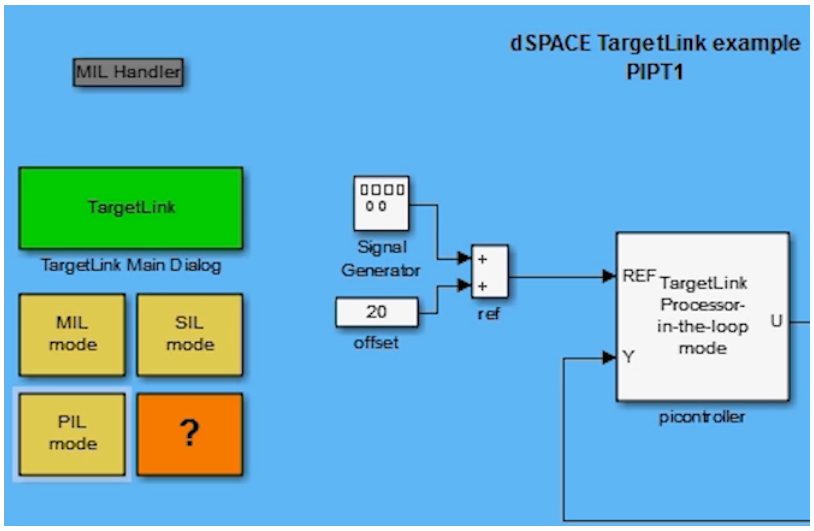


Figure 7.23: dSPACE example of a generic model in PIL simulation mode

This means the generated code of the Control Logic module is simulated on the target hardware. In the TargetLink Plot Overview Window, the PIL simulation results are plotted on top of the previous simulation results. Additional subplot show the execution time and stack consumption if the corresponding checkboxes have been previously selected. TargetLink also provides a code summary, listing the size of each generated C file as well as the RAM and ROM consumption of the generating code [**11**].

Chapter 8

Tutorial Example

The tutorial example is used both to guide the user to the development process of a generic system and to test the methodology consistency. It has the following general characteristics:

- *Filters in series* as multi-task application.
- *dSPACE MicroAutoBox II* is the RCP platform.
- *Template* used to guarantee a defined architecture.
- *V-cycle* used as a guideline for the development process.

8.1 Deeper analysis through the Hybrid V-cycle

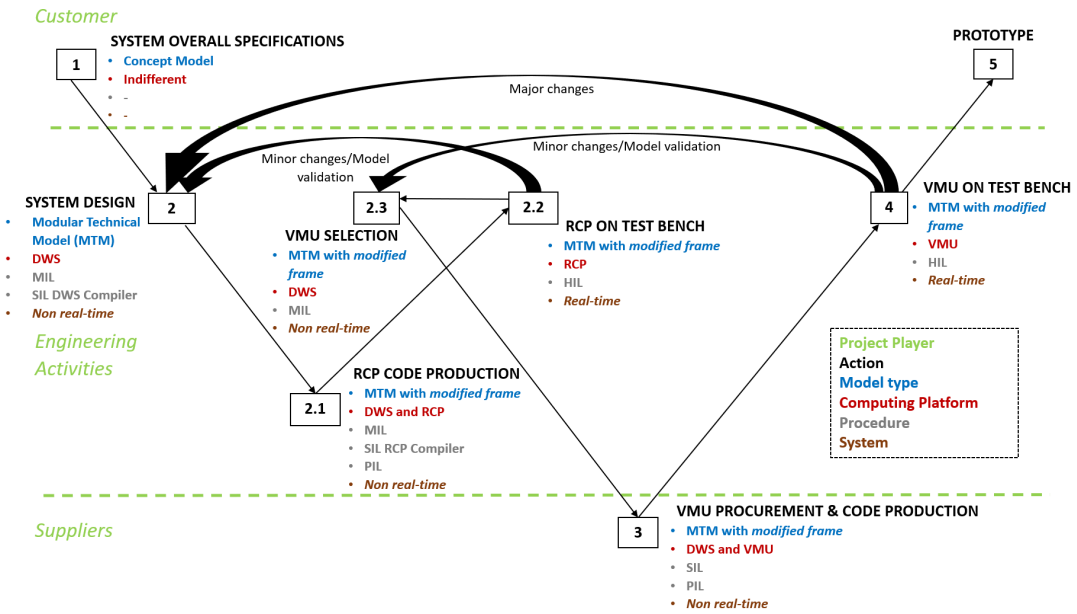


Figure 8.1: V-cycle

8.1.1 System Overall Specifications

Figure 8.2: System Overall Specifications step

In this phase the concept model is produced by the customer with the desired system specifications. The computing platform used by the customer is not relevant because it will be considered along with the next Hybrid V-cycle steps. The concept model has to be implemented by means of the engineering activities.

Example of concept model:

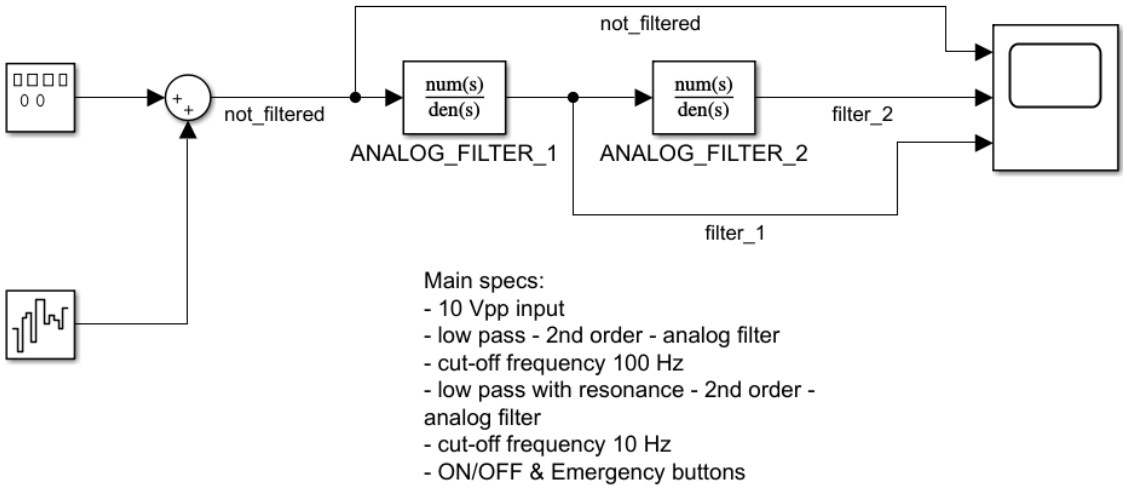


Figure 8.3: Filters in series - concept model

8.1.2 System Design

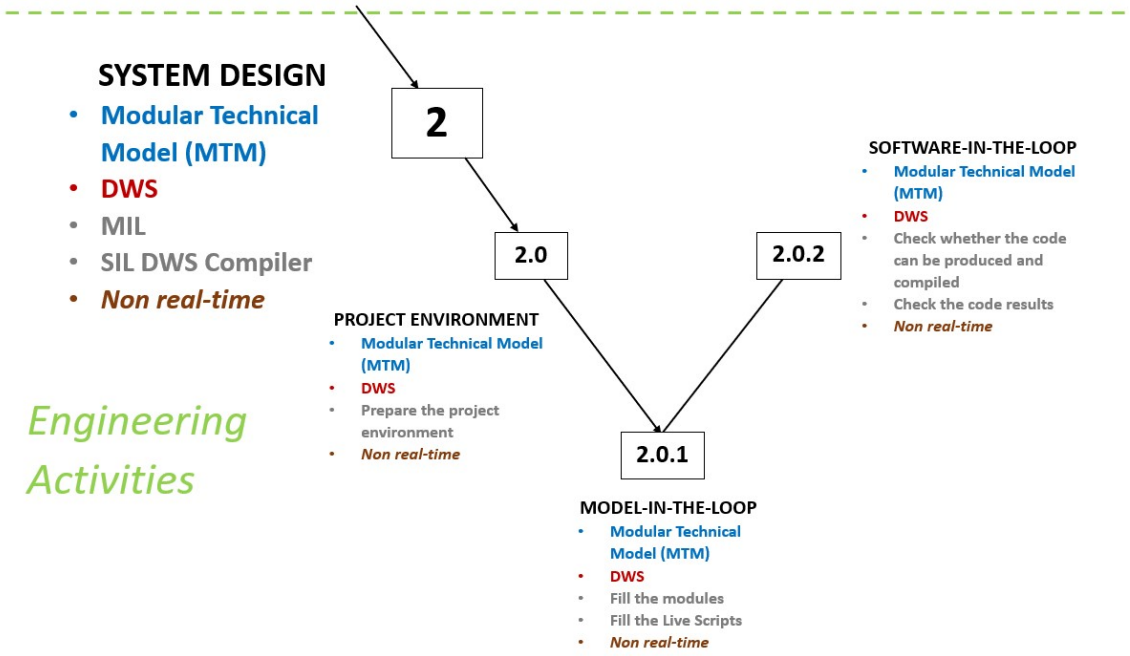


Figure 8.4: System Design step

2.0 *Project Environment*

(a) Click on the project that is intended to be used (*.mlproj* file in "Modular Technical Model – template" folder).

(b) Extract the project to the desired location.
All the folders will be created.

(c) Under Details change the name of the project.

2.0.1 *Model-in-the-loop*

(a) Use the Modular Technical Model to develop the system with the detailed characteristics.
The idea is to use the MTM as a predefined architecture (template) and fill the different subsystems (modules) with the needed blocks that aims at representing the system specifications.

- Located into "*2.0 System design MTM*"⇒ "*2.0.1 MIL*" folder.

(b) *Fill the MTM modules and the Live Scripts with the data.*
The multi-task application model can be designed according to different approaches, that have been already discussed. In this case, a combination of Simulink and Stateflow has been used so that the Stateflow charts are associated to the different tasks. As mentioned in Section 4.3, there are multiple tasks that are involved in the application. The Supervisor is synchronous and it is used to manage the overall behaviour of the system.

In this example, it is needed to start and stop the application. The other periodic tasks are the 500 Hz and 100 Hz filters.
When the application has to run, the Supervisor makes the first filter (500 Hz task) and the second filter (100 Hz task) start. In particular, a push button has to be pressed in order to activate the Supervisor which enables the other tasks.
The application must be stopped if a misbehaviour is detected. Then all the tasks receive an emergency signal when the toggle switch is activated. This component stops the execution of the Supervisor and it acts on the outputs of the other tasks if an emergency situation occurs. The toggle switch activation is an external event that triggers the asynchronous interrupt. The interrupt enables the Emergency Task that produces the emergency signal for the Supervisor and for the other tasks. In particular, it makes the output of the filter tasks equal to zero meaning that it disconnects the outputs.

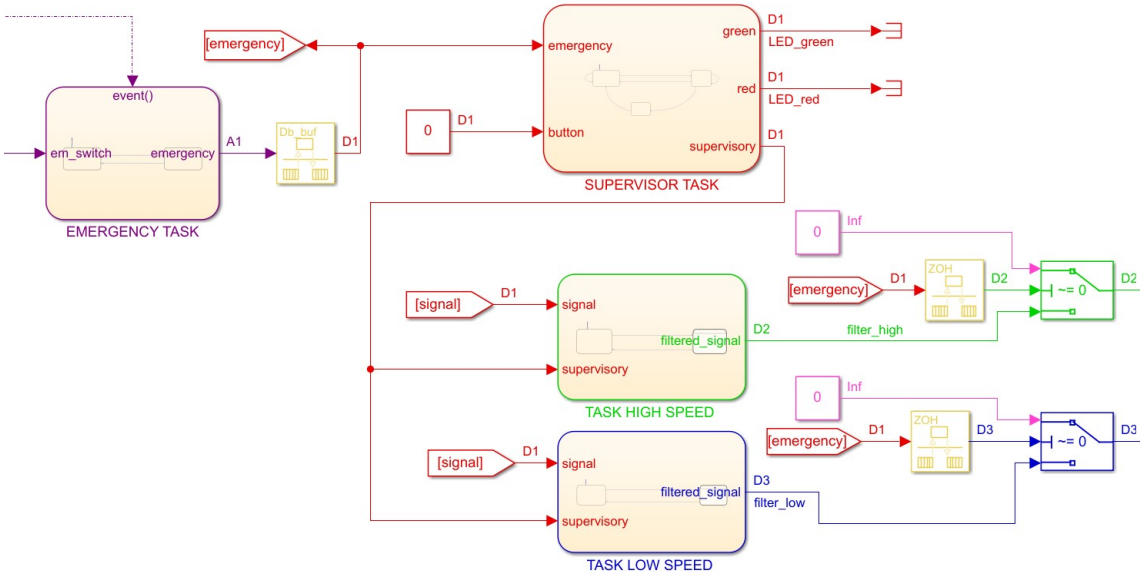


Figure 8.5: Control_Logic content with the tasks that are executed at different sample times. A1 ⇒ *Asynchronous* D1 ⇒ *Discrete, sample time 1* D2 ⇒ *Discrete, sample time 2* D3 ⇒ *Discrete, sample time 3*

Figure 8.5 shows the Control_Logic content with the asynchronous source that is represented by the *asynchronous interrupt*. As a consequence the Emergency Task is aperiodic. The other tasks are synchronous. Then the Rate Transition block is used to create the interaction between the aperiodic task and the periodic processes. Since the simulation of a referenced model that comprises interrupt blocks cannot be executed, the MTM cannot be simulated with the asynchronous source unless some modifications on the MTM itself are performed. These are not allowed because the MTM structure of Section 3.2 has to be respected. In order to overcome this inconvenient the content shown in Figure 8.5 can be simulated considering only the Control model that is generated by the

conversion of the module into referenced model. This is reasonable because the code that has to be deployed on the target hardware is the one related to the Control_Logic. Since the Control module is a referenced model it can be simulated and built independently of the main model that is the MTM.

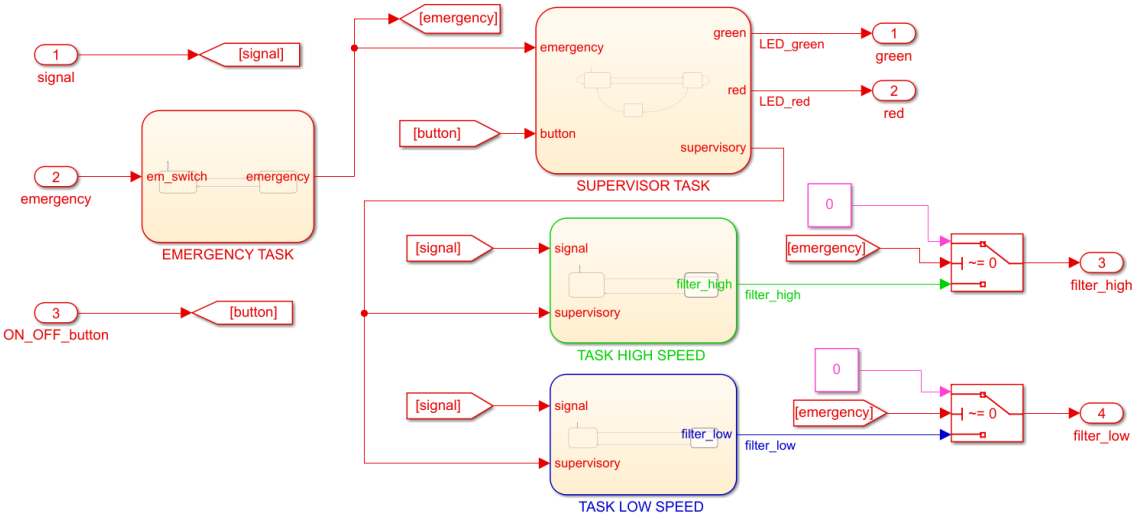


Figure 8.6: Filters in series application - Control_Logic content without asynchronous source

In order to simulate the Modular Technical Model that comprises all the system parts the asynchronous source has to be removed (Figure 8.6).

- Modules (except Control and Control_Logic) located into "*2.0 System design MTM*" ⇒ "*0 MODULES*" folder.
- Control module for MIL simulation located into "*2.0.1 MIL*" ⇒ "*CONTROL_INTERFACE*" folder with contain the frame of the control logic.
- Control_Logic module located into "*0 Control Logic*" folder. It is in the root folder because the control algorithm does not change among the computing platforms.

(c) Fill the frame with generic hardware interfaces (Table 3.1) to detect the most appropriate RCP platform among all the available devices.

(d) Simulate the model (non in real-time) and compare the results to those of the concept model.

2.0.2 *Software-in-the-loop*

The code related to the Control_Logic module has to be automatically generated and compiled.

(a) Set the "*2.0 System design MTM*" ⇒ "*2.0.2 SIL*" ⇒ "*0 CODE*" folder. The code will be saved in this location.

(b) Click on *Build* to generate and compile the code.

- Click on *View Code* to review the code.

- Click on *Open Report* for further details about the code generation process.

(c) (Recommended) Create the Test Harness to check the SIL results and whether they match those of the MIL model (Section 5.1.1).

8.1.3 RCP Code Production

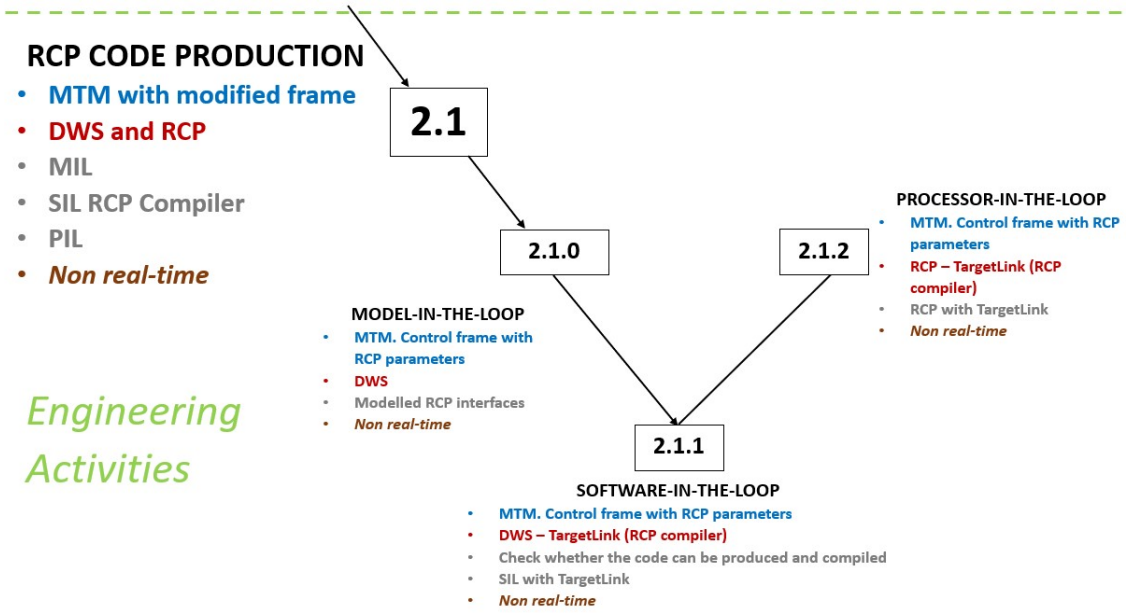


Figure 8.7: RCP Code Production step

2.1.0 *Model-in-the-loop*

This step consists in replacing the frame components of the Modular Technical Model (step 2.0.1) with the RCP interfaces.

(a) Use the Modular Technical Model located into the "*2.1 CODE PRODUCTION*" folder.

(b) Change the frame with the RCP characteristics.

- Fill the Live Script in the "*0 DSPACE_interfaces*" folder with the RCP parameters.

(c) Simulate the model and compare the results to those of the System Design.

2.1.1 *Software-in-the-loop*

A first step consists in check whether the Control_Logic code can be produced and compiled. Then the SIL simulation can be performed to check the results of the code generation.

(a) Click on *Build* to generate and compile the code.

- Click on *View Code* to review the code.

- Click on *Open Report* for further details about the code generation process.

(b) Perform the SIL simulation to check the results of the code generation.

- The **TargetLink** has to be used.
- This step is *not performed* because the TargetLink is not included in the license.

2.1.2 *Processor-in-the-loop*

This step is needed to deploy and simulate the extracted code on the target hardware. This means that the code of the Control_Logic is deployed on the RCP platform that is physically connected to the DWS. If the '*Load application after build*' option is selected, the code will be deployed on the dSPACE MicroAutoBox II. However, to execute the PIL simulation and check the results the TargetLink has to be used.
The simulation is not performed because the TargetLink is not included in the license.

8.1.4 RCP on Test Bench

The step that involves the emulation hardware is skipped. The HIL approach considers the system in real-time. The frame of the Control Logic module has to be modified according to the dSPACE MicroAutoBox II characteristics. Then the system can be tested in real-time by means of the Test Bench representing the real interfaces.

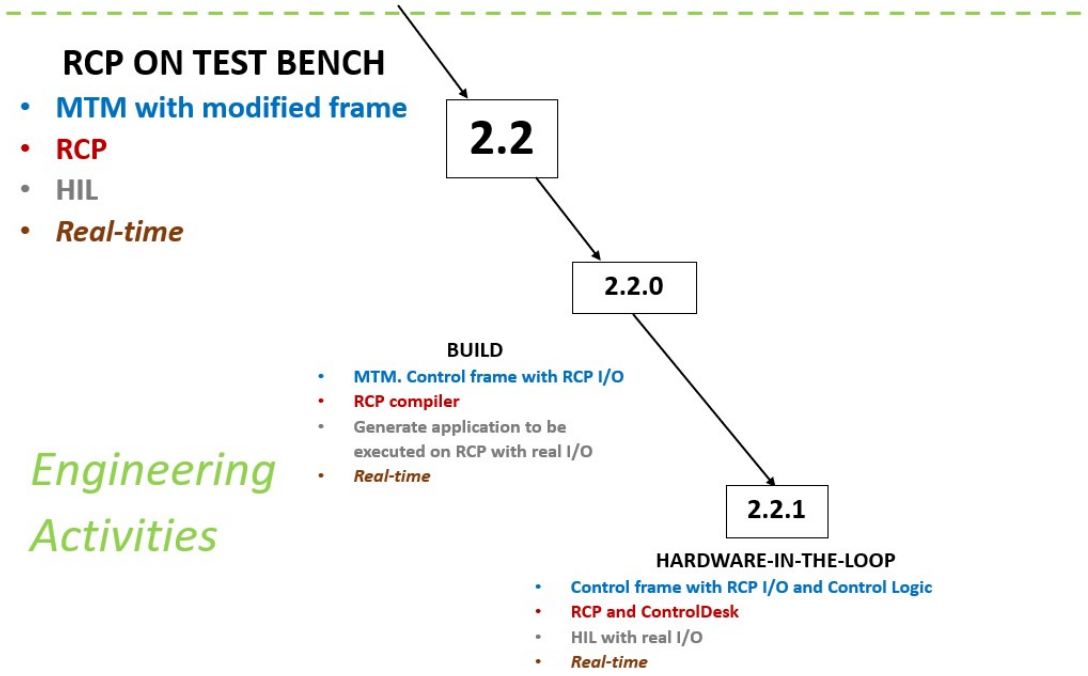


Figure 8.8: RCP on Test Bench step

2.2.0 *Build*

This procedure is used to generate and compiled the code to be executed on the dSPACE platform. The code can be also directly deployed on the hardware if requested.

(a) Change the frame of the Control_Logic ("*2.2 RCP on Test Bench HIL*" folder).
The Control subsystem has to be considered (not only the Control_Logic one) because it will contain the real dSPACE interfaces used to create the interaction between the control algorithm and the system. The Control module has been designed as a referenced model. This means that the content is stored in a different model that can be simulated separately and independently of the main model. As a consequence all the needed operations can be made on the Control model leaving the MTM untouched. The input and the output signals of the frame are now dSPACE signals. This is the reason why the dSPACE library has to be used. It contains interface modules that have to be inserted into the Control model in place of the MTM interfaces, inputs and outputs. The result is an independent model used specifically to deploy the Control_Logic code on the dSPACE platform.

(b) Change the configuration parameters (Chapter 7).

(c) Prepare the Test Bench.

i. Connect the dSPACE to the *I/O board* (DS1541) to the external instruments. The I/O board represents an interface device that makes the connection between the MicroAutoBox II and the external instruments simpler. The board has several pins corresponding to the input and output signals. The external instruments have to be linked to the appropriate pins to establish the physical connection to the dSPACE.

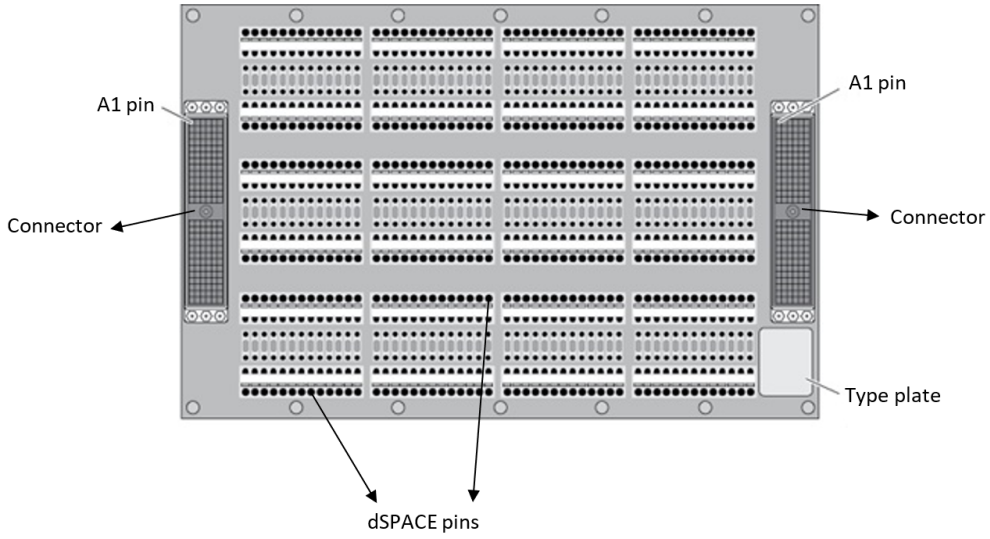


Figure 8.9: DS1541 - dSPACE I/O board

ii. Switch on all the instruments.

(d) Click on *Build* to generate and compile the code ("*2.2 RCP on Test Bench HIL*"⇒ "*0 CODE*" folder).
This produces the system description file for the ControlDesk that is used to manage the dSPACE signals.

- If '*Load the application after build*' option is selected the code will be also deployed on the dSPACE MicroAutoBox II.

2.2.1 *Hardware-in-the-loop*

(a) Open the ControlDesk.

(b) Import the .sdf file into the ControlDesk.
From now on the ControlDesk is used to manage the signals (Chapter 7) and to produce the results.

8.2 Filters in series results

The results are related to the different Hybrid V-cycle steps. The MIL simulation performed at the step 2 considers the DWS. The results have to match those of the concept model.
The MIL procedure of the step 2.1 considers the RCP platform characteristics. The outcomes have to match those of the MIL simulation of the step 2.
The results of the SIL simulation are related to the step 2 performed on the DWS. The SIL simulation that implies the dSPACE compiler is not carried out. The same consideration applies on the PIL simulation with the dSPACE compiler.
The HIL results (step 2.2) consider two different scenarios:

- *Simulated dashboard components*
- *Real dashboard components*

When considering the MIL and the SIL simulations of the step 2 performed with the DWS, the Control Monitor results are shown. This is sufficient because the Plant is only represented by the signal generator and the HMI contains the signals of the buttons. Moreover, the Plant results are already shown at concept model level.
The ControlDesk is used to show the HIL results that are related to the dSPACE MicroAutoBox II.

8.2.1 MIL results

In order to check the consistency of the MIL results of the step 2 of the Hybrid V-cycle they have to be compared with the ones of the concept model.

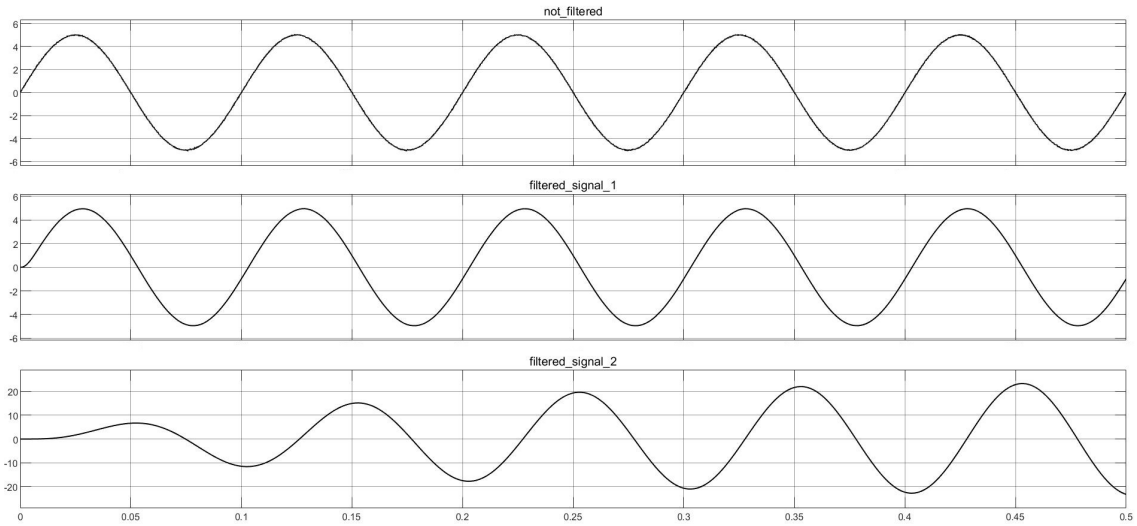


Figure 8.10: Concept model results

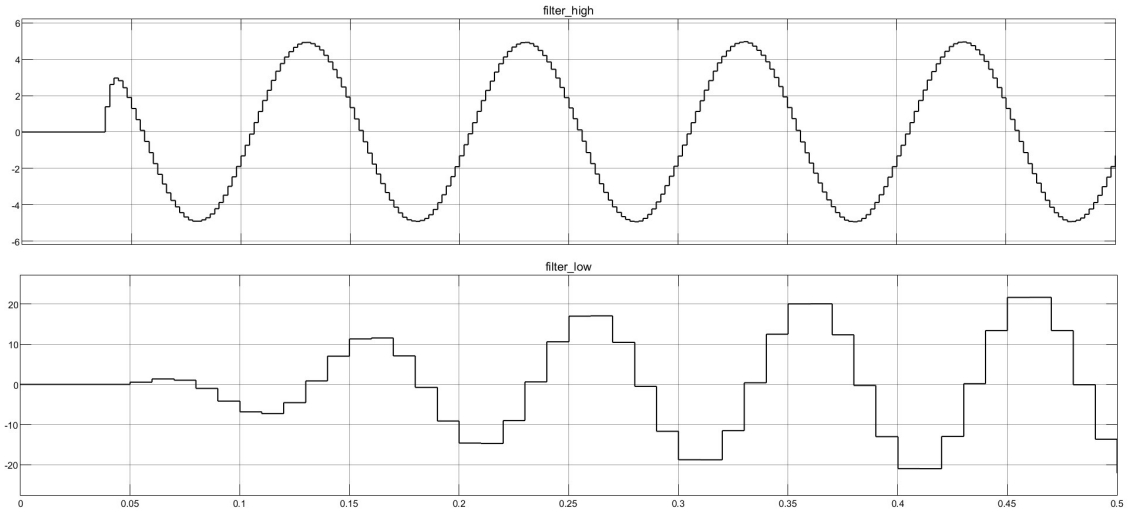


Figure 8.11: DWS MIL results

The concept model has continuous filters while the MIL results comprise digital signals. Moreover, in the MIL scenario a delay seems to be present. The signal is not filtered exactly since the simulation starts. This is reasonable because it represents the button functioning. Only when the button is activated the system is enabled and the filters can start filtering the signal. This occurs at about 0.04 s in this case. Nevertheless the results are acceptable and they are exactly as expected.

Note: the '*not_filtered*' signal is the sinusoidal wave of the generator. The '*filtered_signal_1*' is associated to the 500 Hz filter and the '*filtered_signal_2*' represents the 100 Hz filter.

The MIL simulation of the step 2.1 of the Hybrid V-cycle considers the RCP characteristics. The results have to be compared to those of the MIL simulation performed at the step 2 of the Hybrid V-cycle.

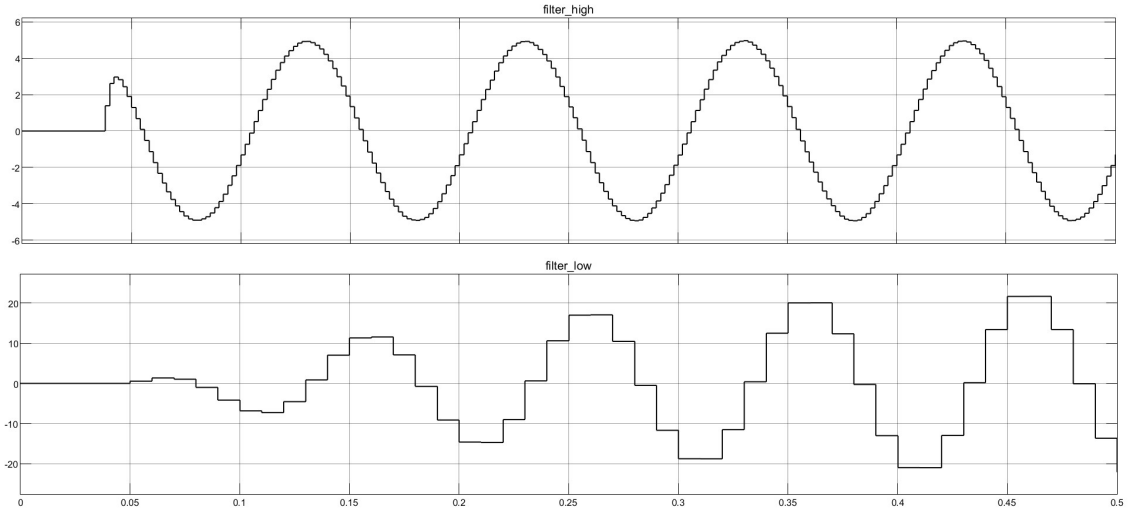


Figure 8.12: DWS MIL results

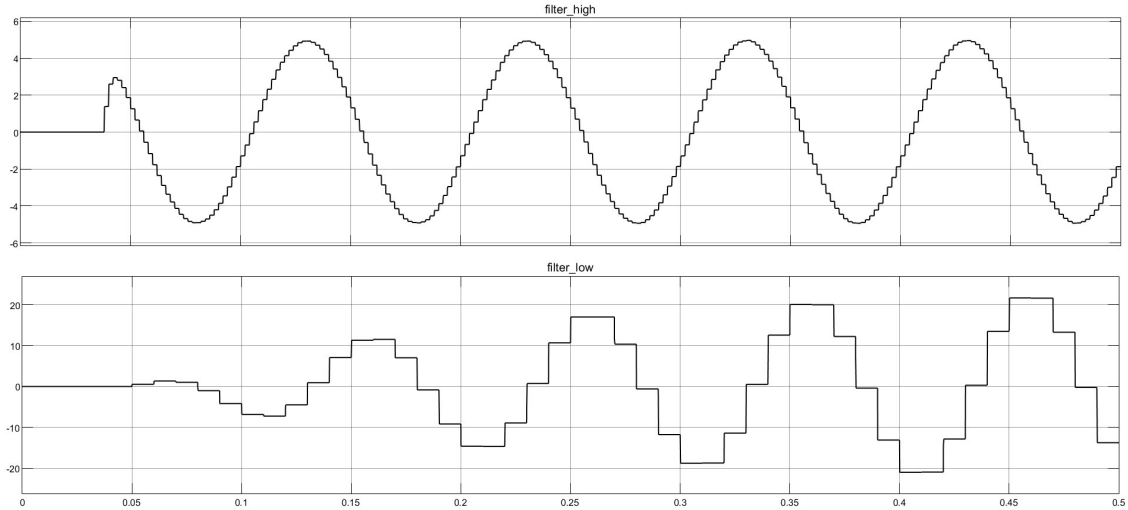


Figure 8.13: RCP MIL results

As shown the MIL results of the steps 2 and 2.1 of the Hybrid V-cycle are identical. Then they are acceptable.

8.2.2 SIL simulation

The SIL results (that are the ones of the extracted Control Logic code) have to be compared with those of the MIL procedure.

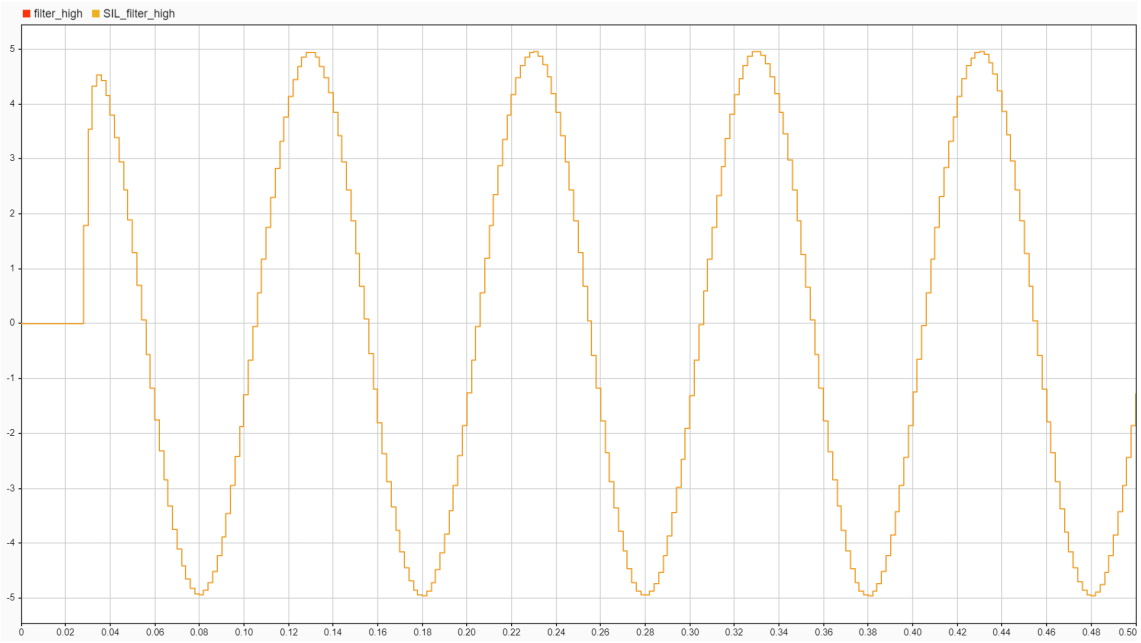


Figure 8.14: SIL results of the 500 Hz filter

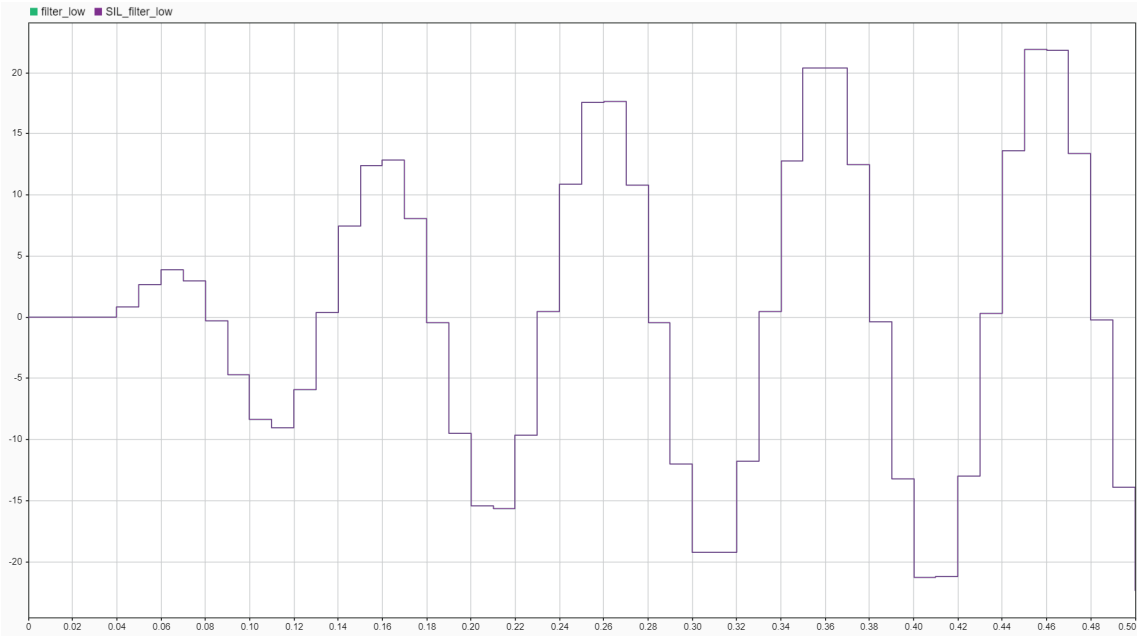


Figure 8.15: SIL results of the 100 Hz filter

As shown the SIL and the MIL the results are almost identical then they are acceptable.

8.3 dSPACE results

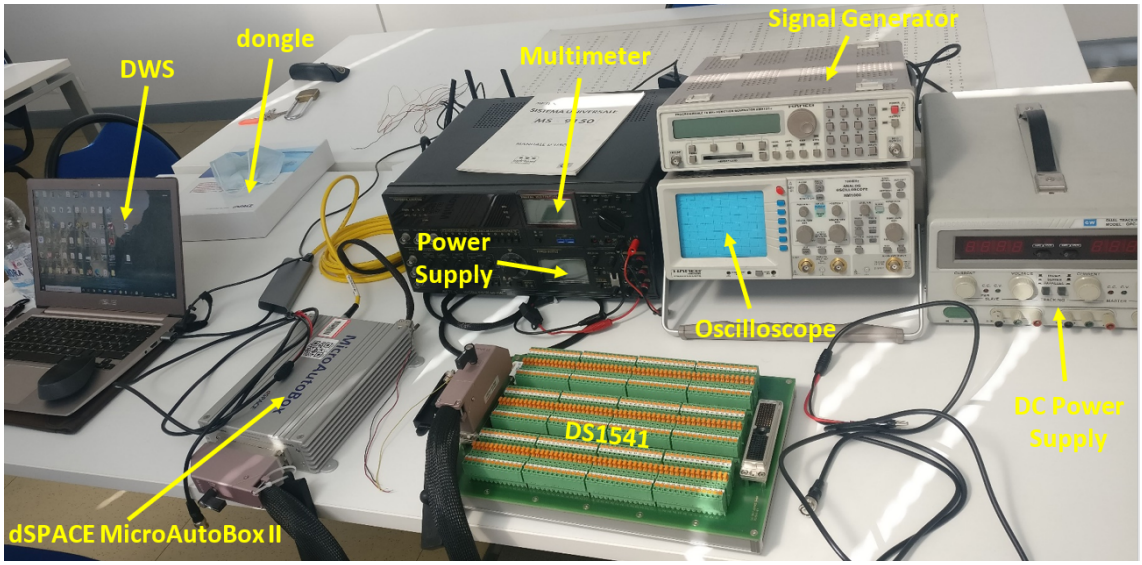


Figure 8.16: Test Bench

The dSPACE MicroAutoBox II is used to perform the HIL procedure and it is a component of the Test Bench (Figure 8.16).

8.3.1 Simulated Dashboard components

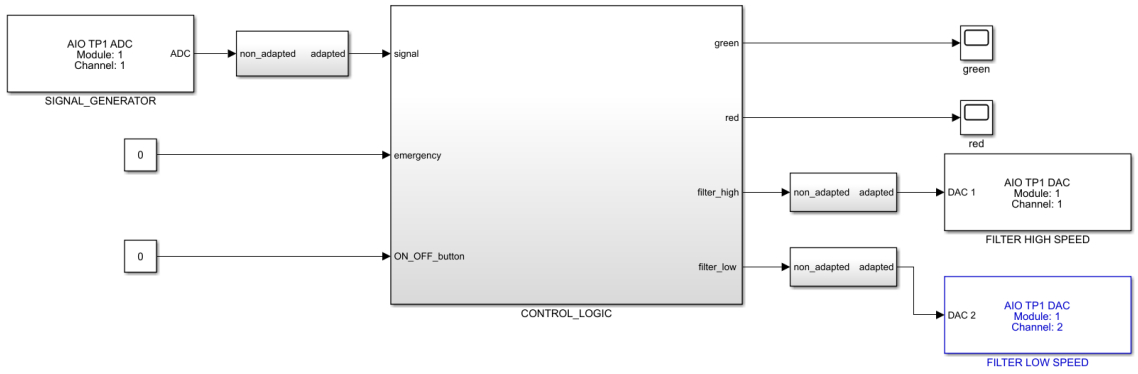


Figure 8.17: Control content - simulated dashboard components

Figure 8.17 shows the Simulink model used to generate the code for the dSPACE platform. In particular, it is the content of the Control subsystem. The frame has been changed according to the dSPACE library modules to represent the real interfaces. The button, the switch and the led are still modelled with Simulink library components because they have to be simulated. This model can be used to investigate the dSPACE library modules and the ControlDesk instruments that have to be used to simulate the dashboard components in particular the ControlDesk *Instrument Selector*. Moreover, this model can be used to manage complex systems

that may have several dashboard components and to directly interact with those elements to analyse the behaviour of the system in different scenarios.
Building the model results in the system description file generation. It will be imported into the ControlDesk to manage the dSPACE signals. In order to simulate the dashboard components the Simulink blocks representing those elements have to be associated to the ControlDesk instruments.

ControlDesk results

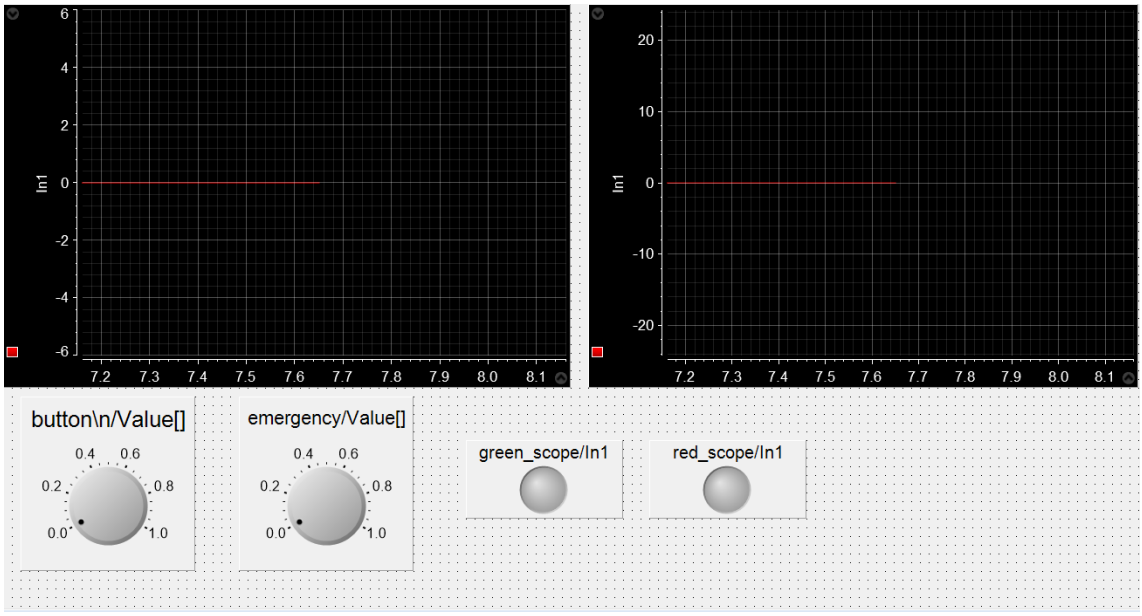


Figure 8.18: System OFF

Figure 8.18 shows that the system is not enabled because the push button has not been activated. This means that the filters do not produce any filtered signal.

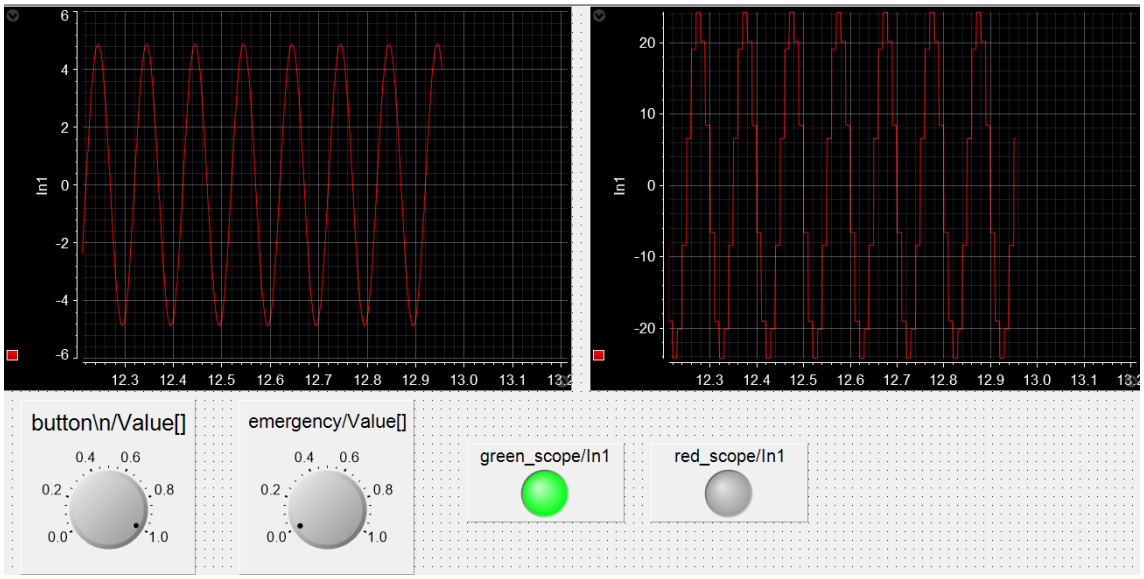


Figure 8.19: System ON - Emergency OFF

In Figure 8.19 the push button is activated while the emergency switch is off. The result is that the system is enabled and it correctly filters the input signal. As a consequence the led becomes green.

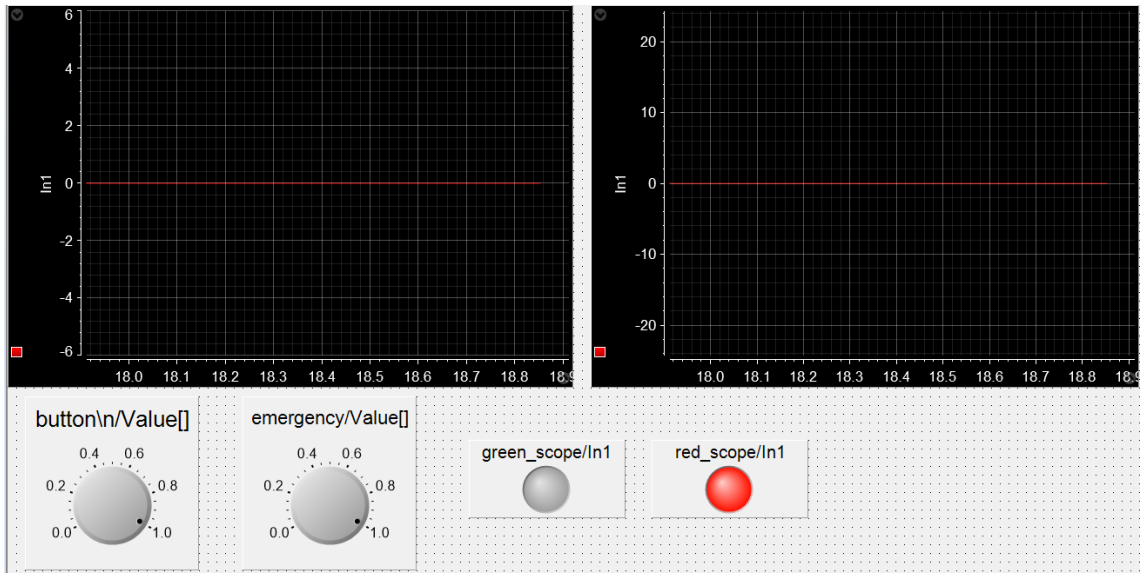


Figure 8.20: System OFF - Emergency ON

Figure 8.20 shows the emergency situation. The corresponding switch is activated. As a consequence the system is stopped although the push button is still on. Moreover, the led becomes red.

8.3.2 Real Dashboard components

At this step the Test Bench (Figure 8.16) represents the whole system. The dashboard elements are real and then they are represented by physical components: led and a generator used for the push button and the switch.

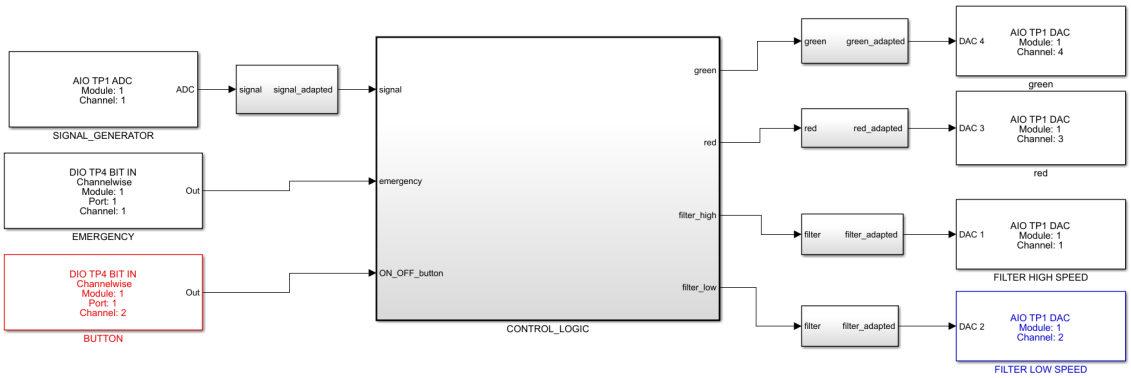


Figure 8.21: Control content - real dashboard components

In Figure 8.21 the Control content comprises synchronous tasks and it does not contain any asynchronous source. This means that the Emergency Task is synchronous. The system does not behave exactly as mentioned in Section 4.3. Nevertheless this model can be used as a first representation of the system and it can be used to investigate the dSPACE library modules for the real dashboard components. Next experiment will include also the asynchronous source.
The frame considers the real interfaces represented by the dSPACE library modules. In this case also the button, the switch and the led are real.

ControlDesk results

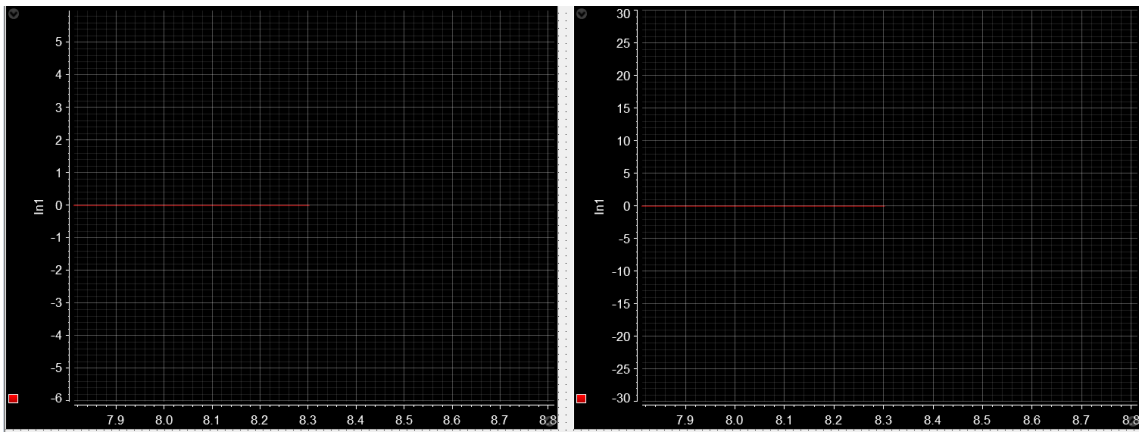


Figure 8.22: System OFF

Figure 8.22 shows that neither the push button or the emergency switch are activated. This means that system is not filtering any signal.

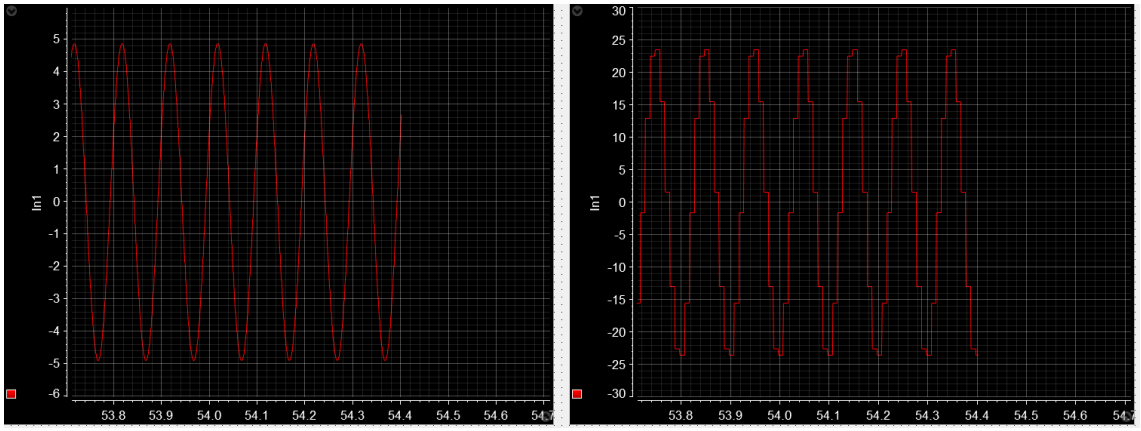


Figure 8.23: System ON - Emergency OFF

In Figure 8.23 the system is enabled by the push button and it correctly filters the input signal.

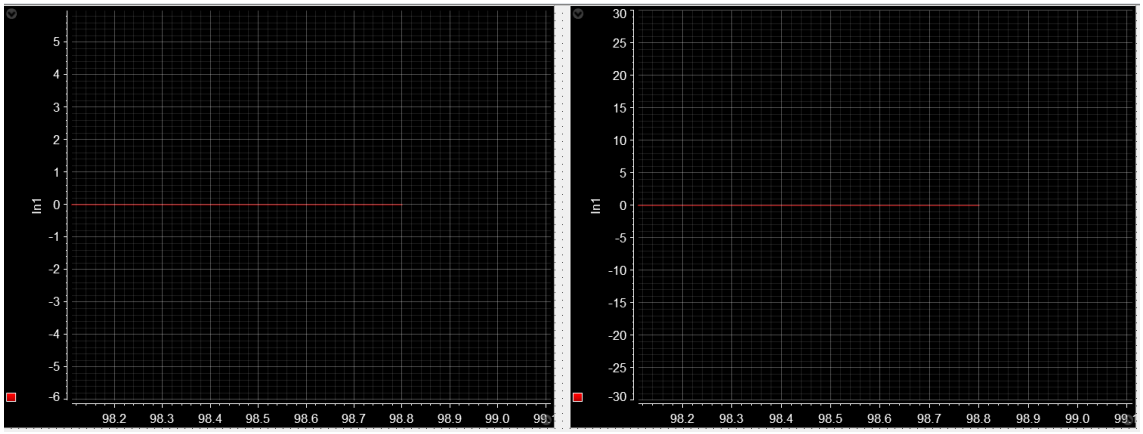


Figure 8.24: System OFF - Emergency ON

Figure 8.24 shows the emergency situation in which the system is stopped by means of the emergency switch activation.

8.4 Conclusions

The "*Filters in series*" application is an example used to test the methodology accuracy. As it shown the methodology allowed to properly define the system components with their characteristics and the interfaces needed to create the interaction among all the system parts including the Control Logic frame. The frame represents the computing platforms and then if it is correctly developed and configured it will allow the user to manage the platforms for testing the application in real-time. Then the main purpose of the methodology is the automatic code generation that results in the code implementing the control algorithm. It has to be deployed on the target hardware.
The code of the "*Filters in series*" application is correctly generated, compiled and

deployed on the dSPACE. As a consequence the dSPACE MicroAutoBox II results are exactly as expected. This means that the methodology provides effective procedures and tools for the system development process when dealing with relatively simple applications. In order to ensure the robustness of the methodology, it is applied on a more complex scenario that involves automotive issues (*Appendix A*). The result is that the methodology is really effective, accurate and robust because it allowed to achieve quite acceptable outcomes (from the automotive prospective) in a relatively short while. From the System Overall Specifications to the RCP on Test Bench steps of the Hybrid V-cycle.
Next objectives are related to applications that will be developed to investigate the dSPACE MicroAutoBox II details, e.g., HW interrupt. This will allow to produce even more sophisticated systems. Then the Hybrid V-cycle will be used as a guideline till the final step to introduce the VMU and to complete the system development process.

Appendix A

Automotive application

The application aims at producing the *Vehicle Management Unit code for an hybrid car* [**20**]. In order to accomplish this task the methodology is used as a guideline from the collection of the system specifications to the testing phase that involves the Test Bench and in particular the dSPACE MicroAutoBox II. The Hybrid V-cycle and the Modular Technical Model are used for the development process. This allows to obtain an effective and accurate application. Using the methodology during a more complex application development allows to ensure the robustness of the methodology itself.

A.1 Concept Model

The concept model and the system specifications are provided by the customer. The wired logic schematic of the vehicle and the inverter logic schematic.

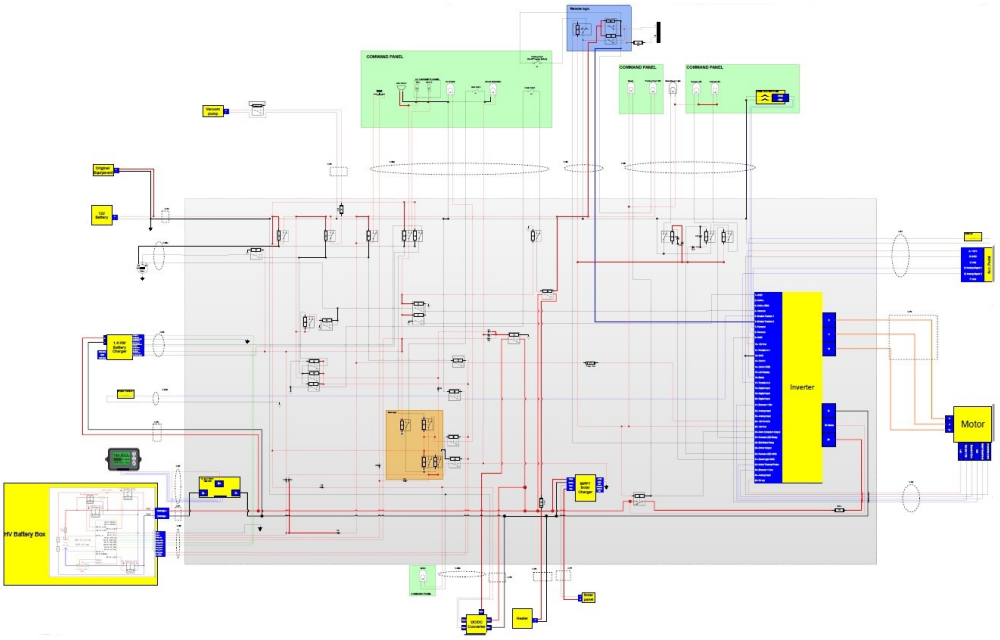


Figure A.1: Wired logic schematic of the vehicle

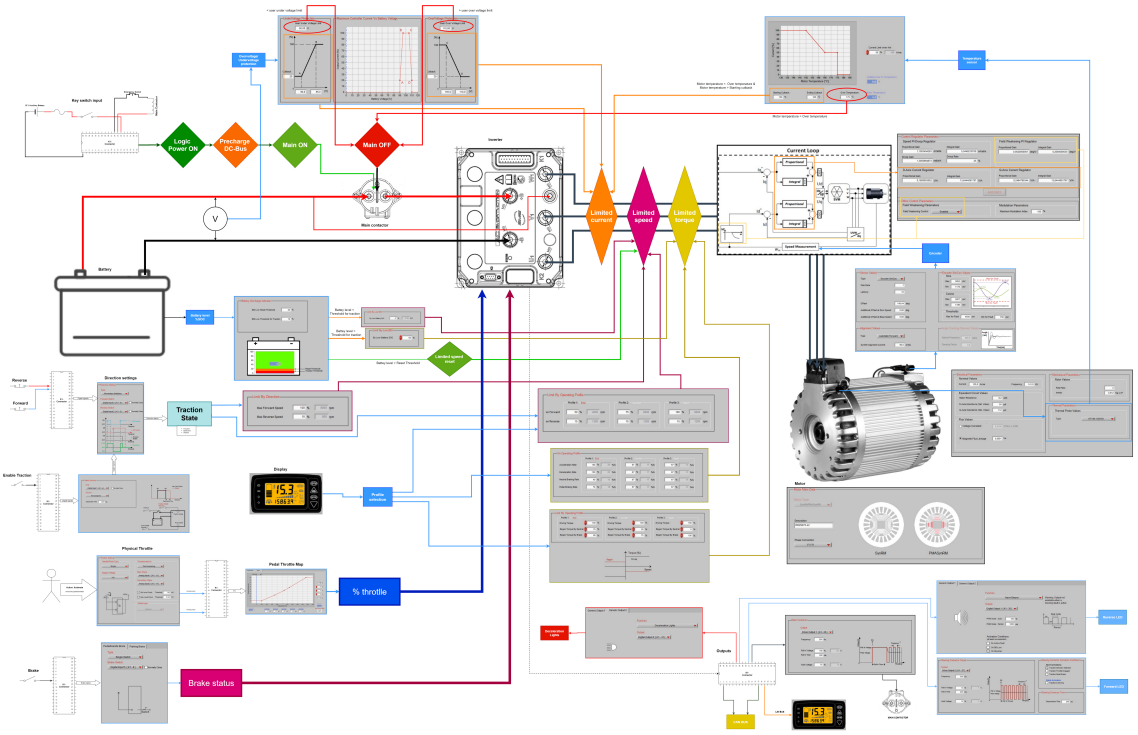


Figure A.2: Inverter logic schematic

A.2 Modular Technical Model

The modules are filled with the system characteristics as described in the Hybrid V-cycle. Some modules are shown.

Figure A.3: Environment module - road slope

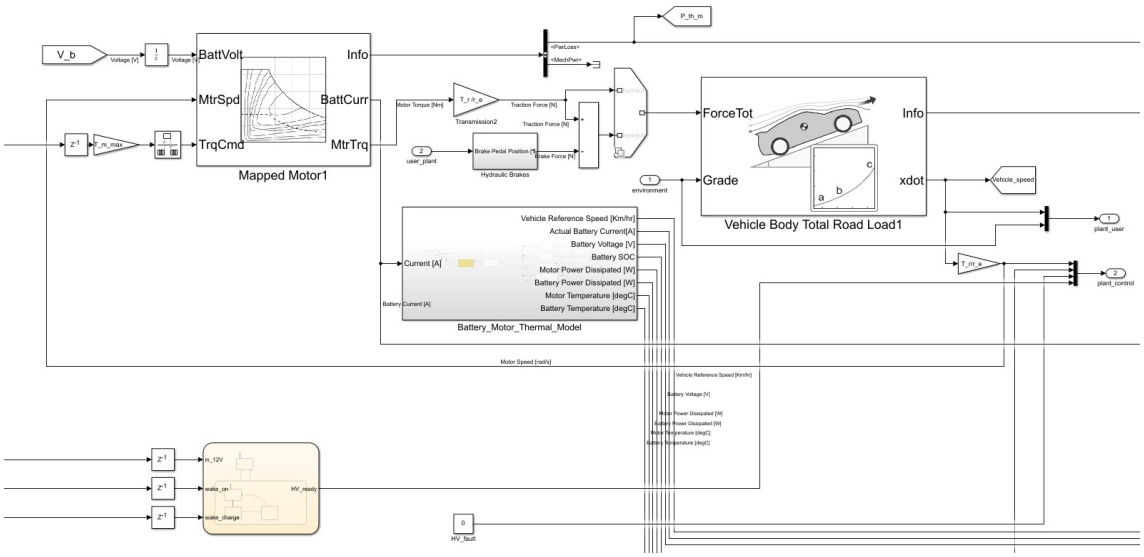


Figure A.4: Plant module - vehicle representation

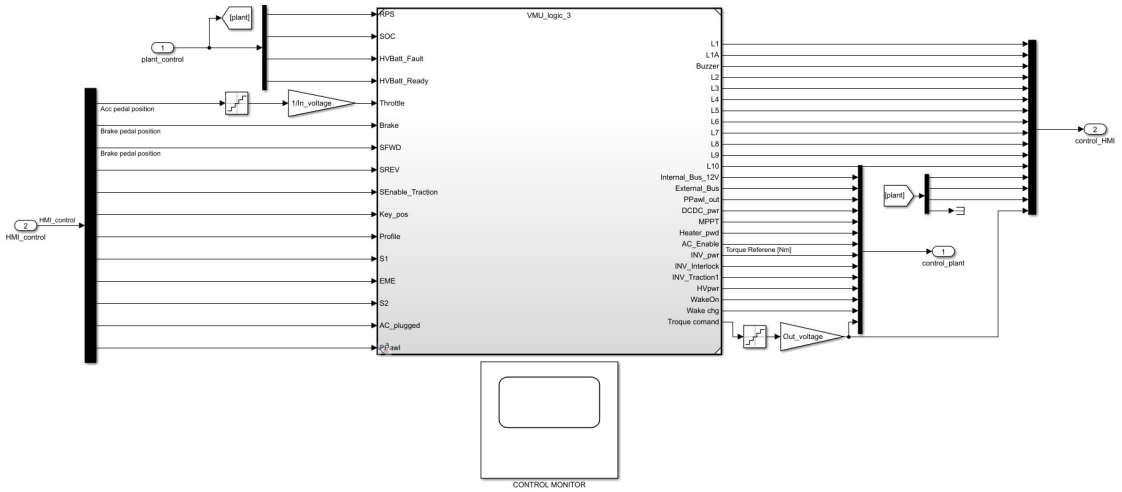


Figure A.5: Control module of the step 2.1 of the Hybrid V-cycle with the frame and the Control Logic module

A.3 Code Generation

The code is automatically generated and compiled when clicking the *Build* icon.

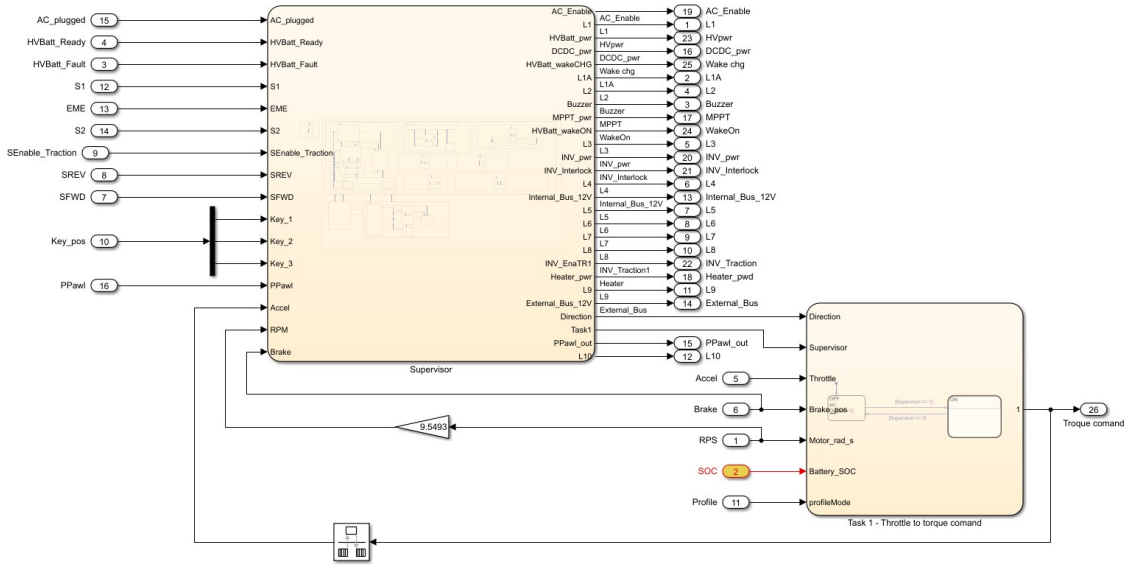


Figure A.6: Control Logic module - Supervisor and Task for the torque command computation

Figure A.6 shows the content of Control Logic module responsible of the code implementing the control algorithm. It will be deployed on the dSPACE MicroAutoBox II.

A.4 dSPACE MicroAutoBox II

The Control module is used to deploy the Control Logic code on the dSPACE MicroAutoBox II. The frame is composed of the real interfaces represented by the dSPACE rti1401 library. They have to be considered when dealing with the code generation in order to enable the interaction between the platform and the other Test Bench instruments.

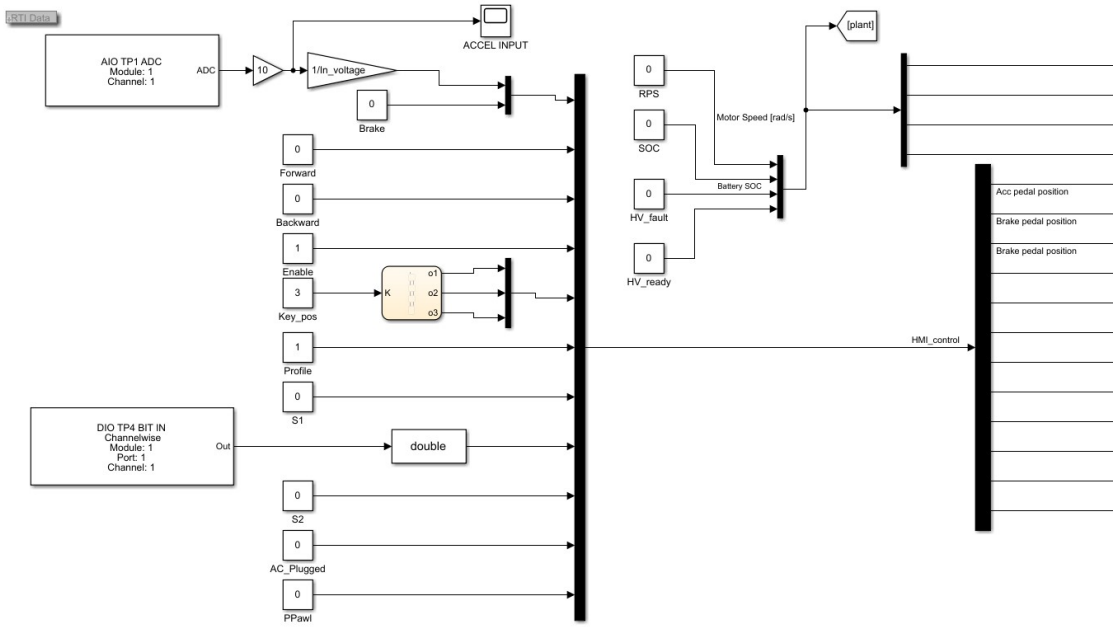


Figure A.7: Frame representing the input signals to the Control Logic module

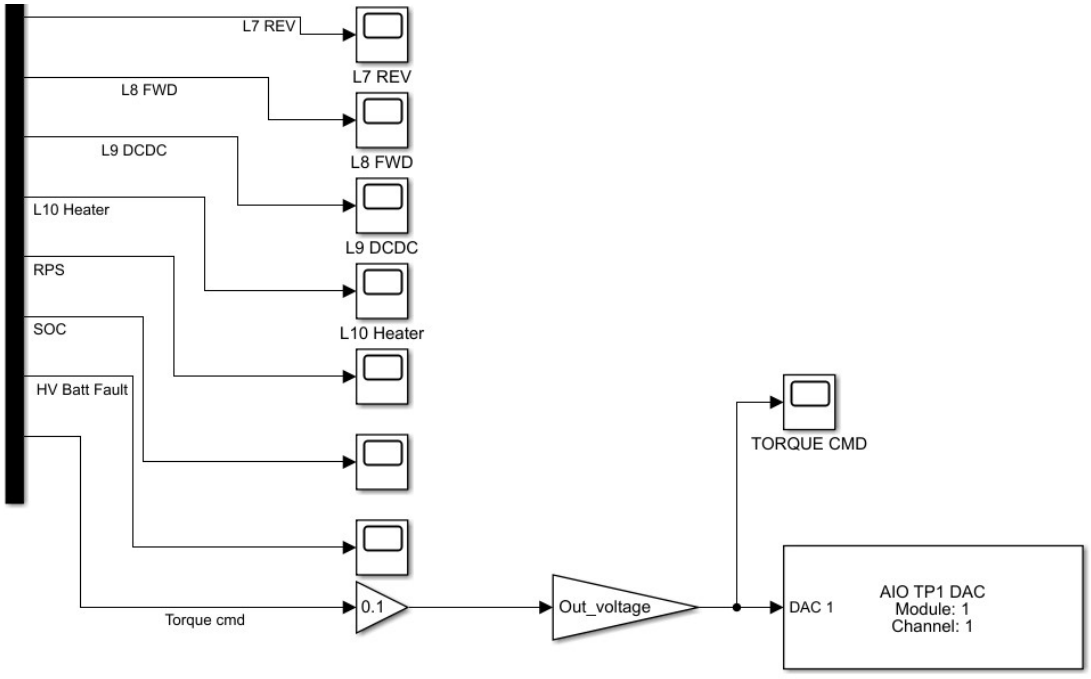


Figure A.8: Frame representing the output signals from the Control Logic module

Figure A.7 and Figure A.8 show a combination of real interfaces and simulated dashboard components due to the great amount of dashboard elements that are needed to create the interaction between the user and system.
Building the model results in the generation of the system description file that have to be imported into the ControlDesk. The ControlDesk is used to perform several tests to check the application functioning. The ControlDesk layouts will be shown. They represent the most relevant scenarios.

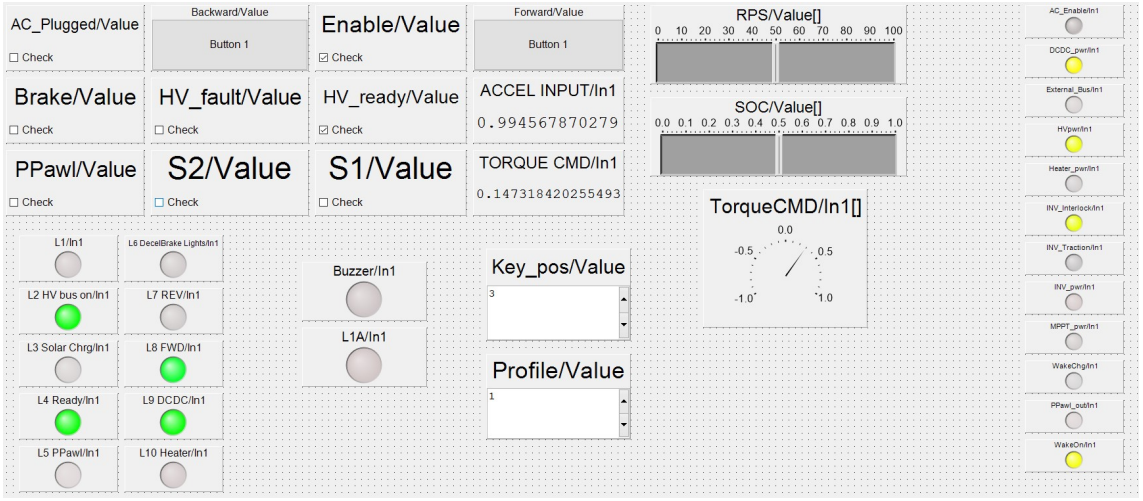


Figure A.9: Forward test with an acceleration command that come from the external DC power supply.

In Figure A.9 the *Enable* switch is on meaning that the traction is enabled and the corresponding led becomes on. The *HV_ready* switch is on meaning that the battery pre-charge has been completed. The corresponding led is activated. As the *Forward* button is pushed the *Torque CMD* (normalized torque command) becomes positive.

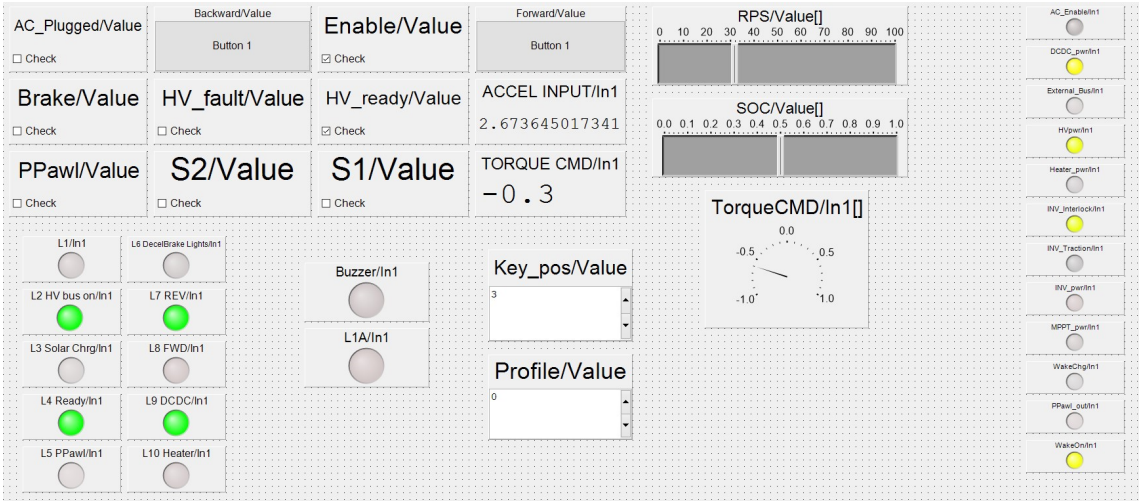


Figure A.10: Backward Test with the acceleration command that come from the external DC power supply

In Figure A.10 the *Backward* button is on, consequently the corresponding led is on. This results in a negative *Torque CMD*.

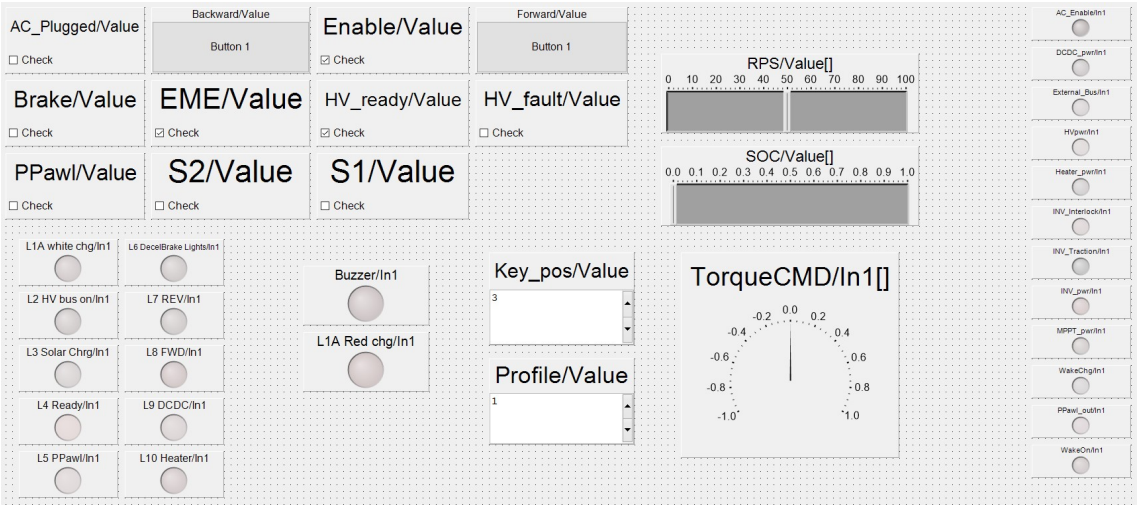


Figure A.11: Emergency situation

Figure A.11 shows the emergency situation. When the emergency switch is activated the application is stopped. No signal can be detected.

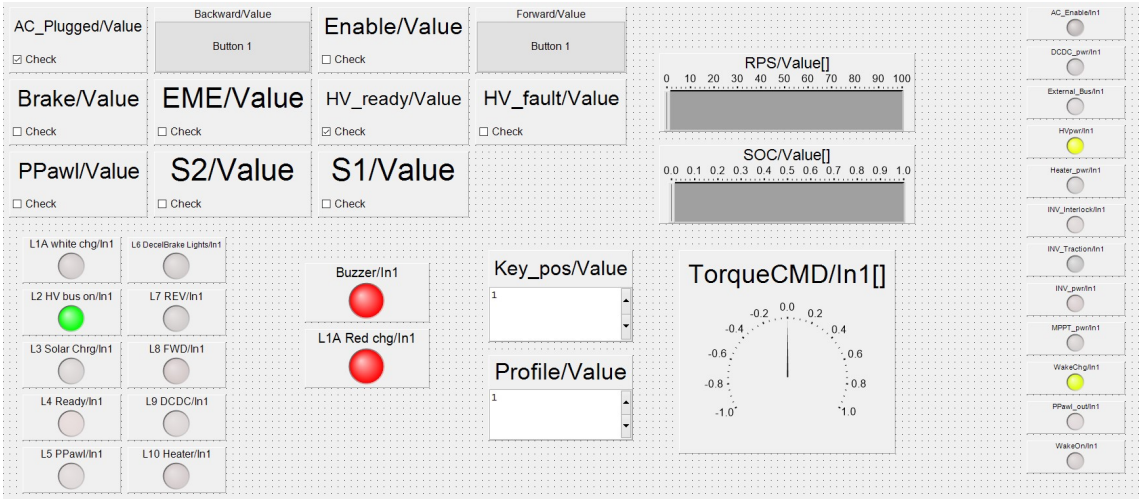


Figure A.12: AC charger plugged

In Figure A.12 the *AC_Plugged* switch is on. This means that the vehicle is charging and then it cannot be driven. As a consequence the *Buzzer* and the *L1A Red chg* led become red as a warning to the user.

Bibliography

[1] International Council on Systems Engineering (INCOSE). *System Engineering.* https://www.incose.org/about-systems-engineering/system-and-se-definition/systems-engineering-definition

[2] I. Graessler, J. Hentze and T. Bruckmann. *V-MODELS FOR INTERDISCIPLINARY SYSTEMS ENGINEERING.* Julian Hentze, Research Assistant, Paderborn University, Heinz Nixdorf Institut,Germany, 2018.

[3] The Mathworks Inc. *Validation and Verification for System Development.* https://www.mathworks.com/help/ecoder/gs/v-model-for-system-development.html?s_tid=srchtitle

[4] The Mathworks Inc. *Create a Template from a Model.* https://www.mathworks.com/help/simulink/ug/create-a-template-from-a-model.html

[5] The Mathworks Inc. *Modeling for Multitasking Execution.* https://www.mathworks.com/help/rtw/ug/modeling-for-multitasking-execution.html?s_tid=srchtitle

[6] The Mathworks Inc. *SIL Verification for a Subsystem.* https://www.mathworks.com/help/sltest/ug/silpil-verification-for-a-subsystem.html

[7] The Mathworks Inc. *Configure Processor-In-The-Loop (PIL) for a Custom Target.* https://www.mathworks.com/help/ecoder/ug/configure-processor-in-the-loop-pil-for-a-custom-target.html

[8] dSPACE. *MicroAutoBox II Hardware Installation Configuration.* Germany, 2018.

[9] dSPACE. *TargetLink.* https://www.dspace.com/en/pub/home/products/sw/pcgs/targetlink.cfm

[10] dSPACE. *dSPACE TargetLink 4.4 Provides New Functionalities: Production Code Generator Supports MATLAB Code in Simulink Models.* https://www.dspace.com/en/pub/home/news/dspace_pressroom/press/201903001.cfm

[11] dSPACE. *Simulating and Testing TargetLink Code.* https://www.dspace.com/en/pub/home/medien/videos/productvideos/video-tl-simulation.cfm

[12] The Mathworks Inc. *MAB Modeling Guidelines*. https://www.mathworks.com/help/simulink/mab-modeling-guidelines.html

[13] The Mathworks Inc. *MAB Modeling Guidelines*. https://www.mathworks.com/help/simulink/ug/types-of-model-components.html

[14] dSPACE. *dSPACE_MicroAutoBox-II-Brochure_2020-08_01_200811_E*. Germany, 2018.

[15] Ideas & Motion S.r.l. *Compass ECU - Product Brief - Rev 02*. Italy, 2019.

[16] dSPACE. *Handling Task - Introducing Task Handling*. Germany, 2018.

[17] dSPACE. *Handling Task - Task States and Execution Order*. Germany, 2018.

[18] dSPACE. *Handling Task - Overrun Situation and Turnaround Time*. Germany, 2018.

[19] dSPACE. *ControlDesk*. https://www.dspace.com/en/pub/home/products/sw/experimentandvisualization/controldesk.cfm

[20] Serrano Thesis. *Project for the implementation and validation of the Vehicle Management Unit (VMU) code of a hybrid car following the V-Cycle development strategies described in ISO 26262*. Italy, 2018.