POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Elettrica

"Development of Simulator of Logic Control Blocks for Electromagnetic Actuators"

Supervisor: Prof. Enrico RAGAINI
Co-Supervisor: Marco RIVA

Master of Science Thesis of:
Ahmar ALI
Matr. 897137

Academic Year 2018_2019

Acknowledgments

I would like to pay my gratitude to the helping hand during the entire process of my thesis work, Professor Enrico Ragaini. I am also grateful for the opportunity given to me ABB S.p.A and my company supervisor Mr. Marco Riva to not only work in the company but also use the state-of-the-art ABB products present in their labs which brought my thesis work to life.

Moreover, I have to thank my collaborators Lorenzo Verniani and Carlo Taborelli for working together with me in the accomplishment of Software-In-the-Loop simulation system, during the testing phase.

Lastly, I would like to acknowledge my family and my fiancé, Zoya for their much-needed support and boost at every step of my life and through this thesis work.

Abstract

We define the need to design and development of simulator of logic control blocks for medium voltage apparatus. The function of the simulator is to reproduce high level logic of apparatus control unit (ACU) platform. Analysis and optimizations were carried out on Outdoor Vacuum Recloser (OVR) configuration in order to augment its performance.

Furthermore, there has been an update in the Simulink user defined block library. The update covers logic blocks for motor controls, position encoders and communication peripherals, also the likelihood of starting the simulation from a saved status has been incorporated into the simulator. Additionally, based on the user input characteristics, a graphical user interface has been structured to launch the simulation.

A concept system based on this architecture was built and tested by means of a software-in-the-loop test for VD4-AF1 configuration. Several tests cases were done on the system to learn about the working of the system. All of the tests performed were successful.

Sommario

Descriviamo la necessità di progettare e sviluppare un simulatore di blocchi di controllo logico per apparecchi di media tensione. Il simulatore viene utilizzato nella simulazione della logica di alto livello della piattaforma dell'unità di controllo dell'apparato (ACU). Le sue prestazioni sono state ottimizzate sulla base di diverse valutazioni condotte sulla configurazione OVR (Outdoor Vacuum Recloser).

Inoltre, la libreria di blocchi definita dall'utente di Simulink è stata aggiornata per i blocchi logici per il controllo del motore, encoder di posizione e periferiche di comunicazione, la possibilità di avviare la simulazione da uno stato salvato è stata integrata nel simulatore, un'interfaccia utente è stata progettata per iniziare la simulazione; in base alle caratteristiche di input dell'utente.

Un sistema di concetto basato su questa architettura è stato creato e testato utilizzando un software nella configurazione ad anello per VD4-AF1. Le prestazioni del sistema sono state quindi testate da più casi di test sul sistema. Tutti i test sono stati completati correttamente.

Contents

List of Figures

Introduction

In today's era everything we can experience through our senses has been in a continuous loop of growth, we live in an industrial world with smarter products at our disposal and since there is evolution in every sector of this global village, we call earth, the electrical sector is also no exception. If we perceive the shift that takes place even in the simplest of things around us, for example the energy meter has an advanced version with better and simpler options than the older versions or systems such as smart home that are IoT based. With the dawn of smart grids, smarter electrical products are in dire need. So, the new generation of products must feature the elementary tasks of a circuit breaker as well as having more flexibility and innovative functionalities (e.g. real-time controls, diagnostics, communication). The aforementioned functionalities make the product more performing and reliable. Therefore, the real-time simulation of a physical apparatus is a valuable instrument to comprehend the behavior and response of a system in an arbitrary point in time. One of its key system development tools, Software-In-The-Loop simulation, allows us to review and realize the execution of physical components minus the expense and complication of actually developing the real hardware. It also gives us the basic idea of the behavior we should expect from the hardware so in that way it can also help us in diagnosing the hardware.

This thesis work has been completed at the Technology Center of ABB S.p.A. in Dalmine, Italy. The objective of the thesis work is to realize a simulator, which can provide us with the ability to verify and validate the functionalities and logical applications of the apparatus. The simulator system is based on MATLAB Simulink as the primary design and development tool. Simulink is used for creating higher level of apparatus logic, this logic is generated as a user defined configuration file, the apparatus control unit (ACU) is running the algorithms for the interface between Simulink and firmware we have used Texas Instruments' (TI) Code Composer studio (CCS). The user defined S-Functions library tool is used to link the above two elements in a flexible and effective way: MATLAB Simulink is used in the model design phase, then the model can simulate the firmware behavior of the apparatus.

The thesis is divided in six chapters:

First chapter discusses the platform used for simulating the behaviors of apparatus i.e. actuator control unit (ACU), the details about the basic hardware od ACU central processing unit (CPU), power electronics and data communication channels have been explained, also the possibility of master slave configuration working as single module or multi module systems have been discussed.

Second chapter reviews the definition of Configuration Programmable Logic and the usability of the logic with ACU firmware modules. Also, the detail of configuration developments tools such as Simulink and functions of the application i.e. Level-2 MATLAB S-function and MEX function, and other tools like Code Composer Studio, Programmable Logic and Configuration File.

Third chapter discusses optimization of simulator performance, studies made for an efficient and better performing simulator with improved storage. Additionally, Outdoor Vacuum Recloser (OVR) configuration is explained thoroughly with analog and binary inputs for OVR, analysis for improvement of OVR is studied by learning total simulation time and its results obtained. Improvement of performance is suggested by selecting a faster simulation mode or by not saving all the outputs or by selecting important blocks for output saving.

Fourth chapter explains the expansion of simulator library for final SIL test such as Motor Control Blocks, Motor load identification, Motor calibration, Motor driver diagnostic. Communication I/O blocks such as CAN input, CAN output, GPIO Output. Position sensors; Absolute Encoder and Servo blocks.

Fifth chapter reviews the use of MATLAB app designer with components like common components, containers and figure tools, instrumentation and toolbox components. Additionally, the study of graphical user interface of the simulator and its details i.e. model select, simulation mode and simulation state.

Sixth chapter discusses Software-In-Loop Simulation of VD4-AF, testing configuration of VD4-AF and detailed test cases like open close test which was test case 1 and close open test which was test case 2, capacitor voltage test which was test case 3 and test case 4, 5 and 6 which were reties failed, CAN supervisor, encoder mismatch and undervoltage respectively

Chapter 1

Apparatus Control Unit (ACU)

1.1 Introduction

The ACU (Apparatus Control Unit) system is an electronic device used to control electromechanical actuators used in Medium Voltage applications, e.g. Circuit breakers, switchgears, sectionalizers, capacitor switches. The main functionalities of ACU are the following:

- Driving magnetic actuators and motors that perform the basic switching functionalities of an Apparatus.
- Implementing customized application logics.
- Allowing system parameters configuration, monitoring, data acquisition and communication.

The ACU system is made of a main CPU board, a power drive board and other optional boards (e.g. Communication board, supplementary I/O board). The purpose of this document is to describe the software architecture, the communication interfaces, data and file structures, interaction with the debug terminal for the ACU system.

1.2 Detail of Hardware

Detail of all hardware utilized to achieve simulator development of logic control blocks for Electromagnetic Actuators for the Apparatus Control Unit (ACU) and then to test the achieved system are detailed in subsequent segments of this chapter.

The ACU hardware is depicted in the figure 1

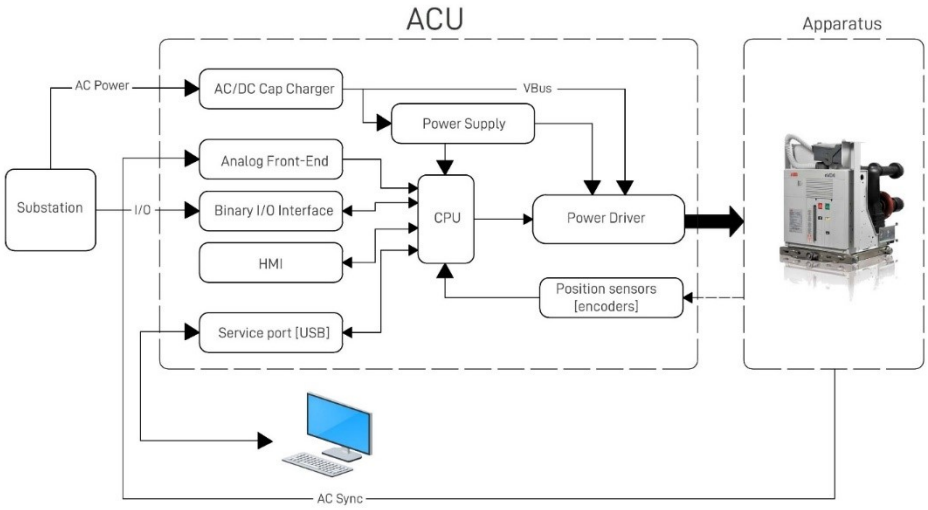

Figure 1 Basic Hardware Configuration of Apparatus Control Unit

A fundamental explanation of ACU unit structure is illustrated in Figure 2, An ACU unit is made of two boards: the CPU board and the Power drive board.

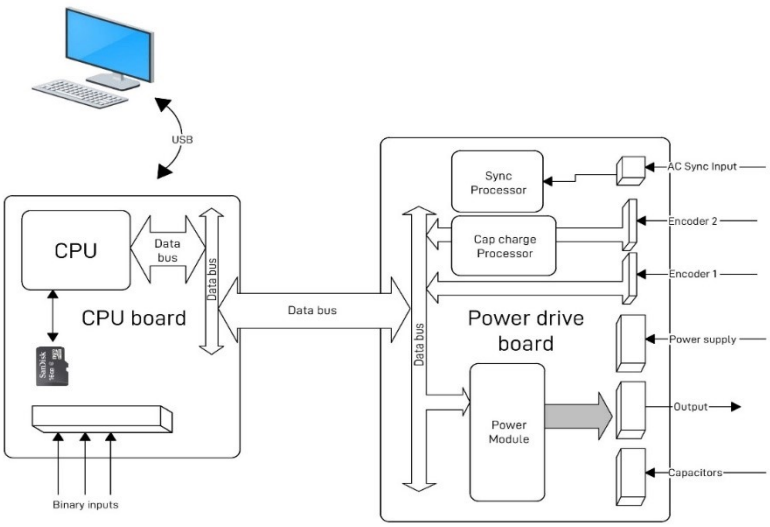

Figure 2 ACU Unit Structure

1.2.1 CPU Board

The following hardware devices related with software are mounted on the CPU board:

- Main processor options:
  Texas Instruments F28M35 dual core chip featuring an ARM Cortex M3 processor and a floating point C28 DSP (Texas Instruments C2000 family). The ARM processor runs a set of supervision, communication and data storage functionalities. The DSP runs real-time controls, programmable logic and internal diagnostics, reading analog signals through its A/D converters and the position of mechanical devices (motors) using incremental and absolute encoders.
  Texas Instruments TIVA TM4C129NCPDT: ARM Cortex M4 single core.
- Mass Storage options:
  Two slots for SD-cards used as non-volatile memory storage handled by M3 core.
  SPI Flash memory
- Binary inputs hardware interface for wide range voltage measurement used by main
  CPU as “power digital inputs” to issue commands to the Apparatus.

1.2.2 Power Drive Board

The following hardware devices related with software are mounted on the Power drive board:

- Synch Processor: Texas Instruments TMS320F28027 DSP runs the AC Synchronization real-time algorithm needed by synchronized switches. It reads the AC voltage supplied to a dedicated channel in order to detect sine wave zero crossing events used in synchronous devices.
- Capacitor charger processor: Texas Instruments MSP430 MCU runs a real-time application to monitor and control charging of external capacitors used as energy storage for apparatus electromagnetic or electromechanical actuators.
- Battery Charger processor: Texas Instruments MSP430 MCU runs a real-time application to monitor and control charging of a battery for auxiliary power supply backup.
- Power Module: a power device driven by the PWM outputs of the main CPU DSP code delivers power to the actuators that physically perform the Apparatus operations (driving motors or magnetic actuators).

1.2.3 Single-Module and Multi Module Systems

An ACU-based system may be built using either one ACU module only or several ACU modules connected each other through a CAN bus. In this case one of the modules is the Master of the system and the others are Slaves. The application configurations downloaded on each unit configures it as master or slave, executing different logics as required by the application. A typical multi-module system uses three modules, one for each pole of a three-phase circuit breaker.

1.2.4 Data Communication Channel

Data are exchanged between hardware/software modules through several physical channels. A block diagram of data communication channels is shown in Figure 3.

- Shared RAM between C28 and M3 cores. This channel is built in the main CPU silicon.
- UART between main CPU and peripheral processors (Synch and Cap charge).
- CAN bus between main CPU of different modules. CAN bus is owned by M3 core.
- ECAP (Capture-Compare peripheral) between binary input interface and C28 CPU.
- QEP (Quadrature Encoder peripheral) between incremental optical encoder andC28 CPU.
- SPI between absolute magnetic encoder and C28 CPU

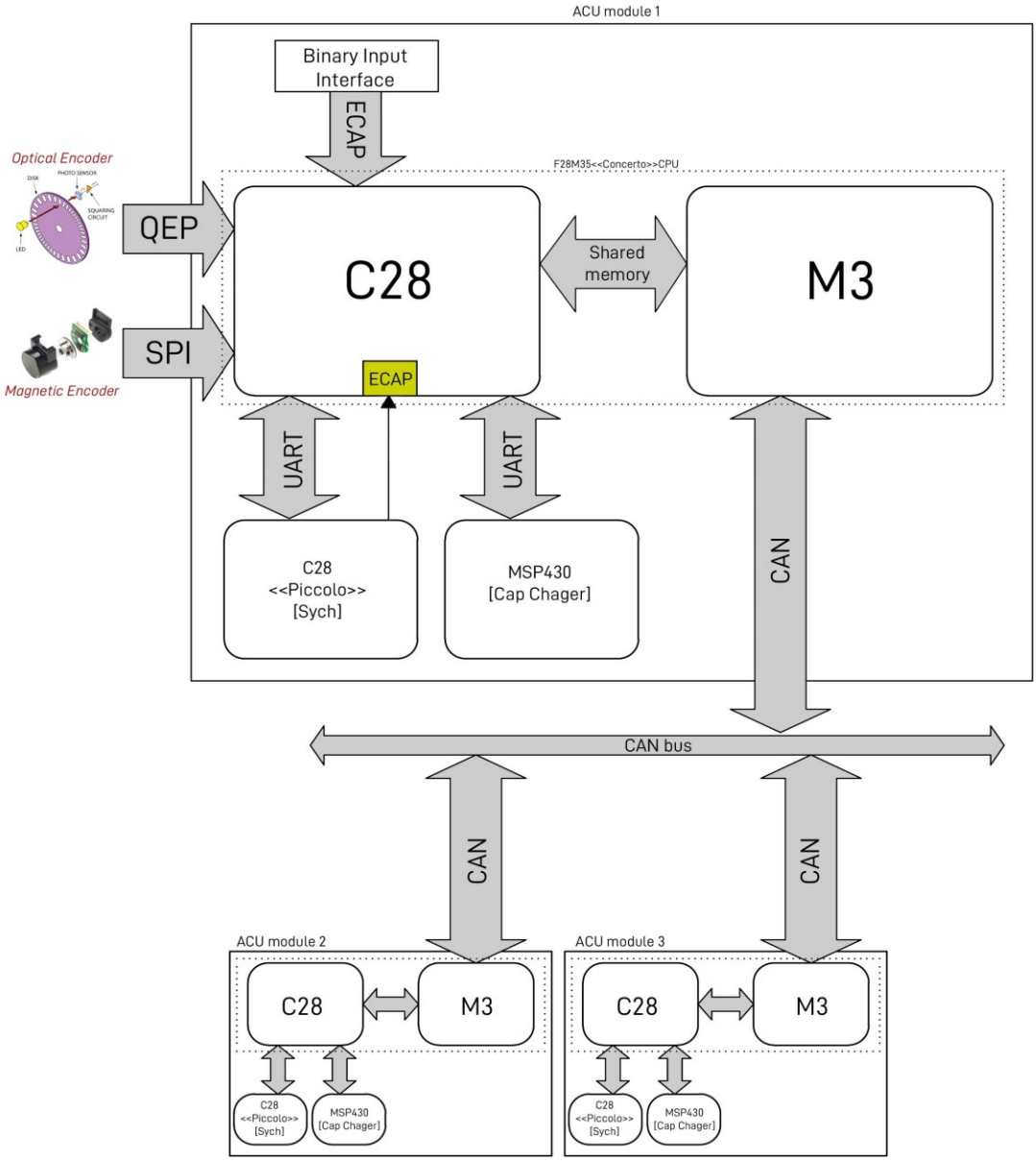

Figure 3 Block diagram of data communication channels of ACU

Chapter 2

Configuration Programmable Logic

The ACU firmware modules do not execute any application specific logic, i.e. an ACU based electronic device does not have any built-in high-level logic. The application behavior is defined through the Configuration file, i.e. description of the application logic created using a Simulink block diagram.

2.1 Detail of configuration development tools

2.1.1 Simulink

Simulink is a MATLAB-based graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. Its primary interface as shown in figure 4 is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in automatic control and digital signal processing for multidomain simulation and model-based design.

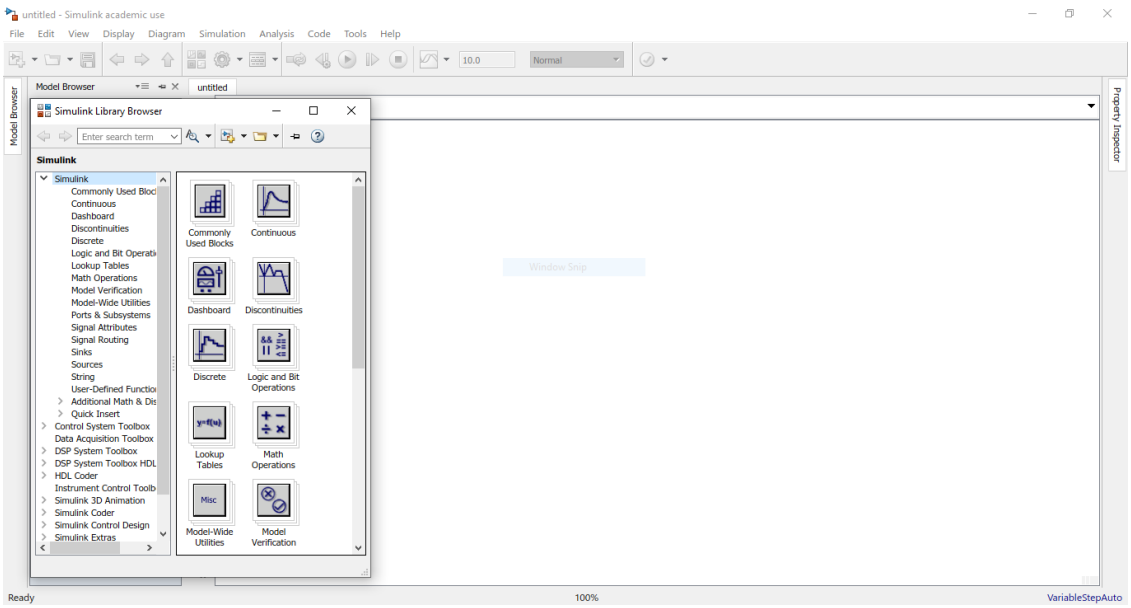

Figure 4 Simulink graphical programming environment for modeling.

Configuration file for ACU is created by User defined Level 2 S-Functions, The Level-2 MATLAB® S-function API allows you to use the MATLAB language to create custom blocks with multiple input and output ports and capable of handling any type of signal produced by a Simulink® model, including matrix and frame signals of any data type. The Level-2 MATLAB S-function API corresponds closely to the API for creating C MEX S-functions.

2.1.1.1 Level-2 MATLAB S-function

A Level-2 MATLAB S-function is MATLAB function that defines the properties and behavior of an instance of a Level-2 MATLAB S-Function block as shown in figure 5 that references the MATLAB function in a Simulink model. The MATLAB function itself comprises a set of callback methods that the Simulink engine invokes when updating or simulating the model. The callback methods perform the actual work of initializing and computing the outputs of the block defined by the S-function.

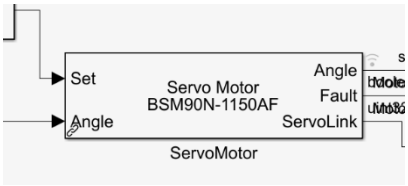

Figure 5 Example of Level 2 S-function block

The mask and dialog parameters can be set to user value as shown in figure 6

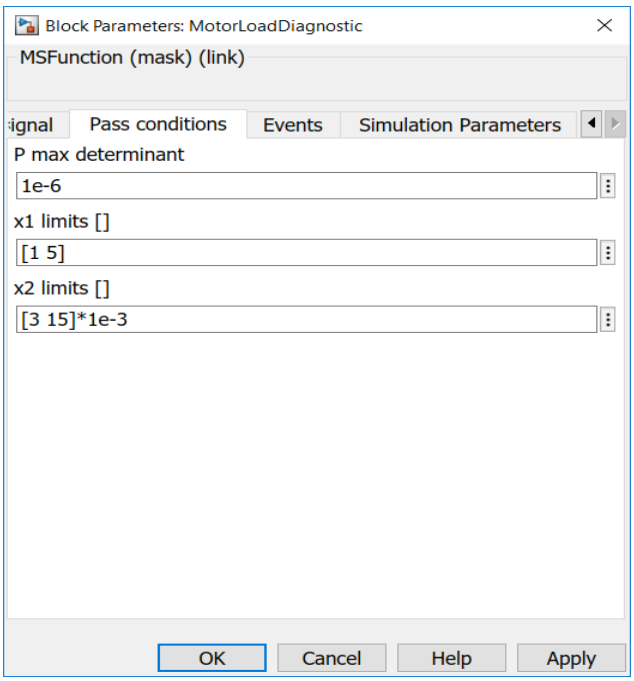

Figure 6 Sample mask of Level 2 s function block

To facilitate these tasks, the engine passes a run-time object to the callback methods as an argument. The run-time object effectively serves as a MATLAB proxy for the S-Function block, allowing the callback methods to set and access the block properties during simulation or model updating.

The body of the setup method in a Level-2 MATLAB S-function initializes the instance of the corresponding Level-2 MATLAB S-Function block. The setup method performs the following tasks:

- Initializing the number of input and output ports of the block.
- Setting attributes such as dimensions, data types, complexity, and sample times for these ports.
- Specifying the block sample time
- Setting the number of S-function dialog parameters.
- Registering S-function callback methods by passing the handles of local functions in the MATLAB S-function to the RegBlockMethod method of the S-Function block's run-time object.

The following steps illustrate how to write a simple Level-2 MATLAB S-function. When applicable, all lines of code use the variable name block for the S-function run-time object.

1. Modify the setup method to initialize the S-function's attributes.
2. Set the run-time object's `NumInputPorts` and `NumOutputPorts` properties to any number in order to initialize input ports and output ports.
3. Set the run-time object's `NumDialogPrms` property to the required number in order to initialize S-function dialog parameter.
4. Call the run-time object's `RegBlockMethod` method to register the following four callback methods used in this S-function.
    - `PostPropagationSetup`
    - `InitializeConditions`
    - `Outputs`
    - `Terminate`

Example in Figure 7 shows the body of setup function.

```
FILE  NAVIGATE  EDIT  BREAKPOINTS  RUN
13  function setup(block)
14  global simulation
15
16  %simulation is global variable so can be changed from workspace
17
18
19      % setup blocks I/O
20      % blockSetup(block, <NUM_INPUTS>, <TYPES_INPUTS>, <NUM_OUTPUTS>, <TYPES_OUTPUTS>);
21      blockSetup(block,       ...
22                 2, {         ...
23                 'boolean'    ...
24                 'single'},   ...
25                 1, {         ...
26                 'single'});
27
28      % Register the parameters.
29      block.NumDialogPrms = 2;
30  %% if simulation is defined and its value is 1 then implement Output function, else Start function
31
32      sim_not_defined =  isempty(simulation);  % checking if the simulation variable is defined?
33
34      if (sim_not_defined == 0 && simulation == 1)
35          block.RegBlockMethod('Outputs',                 @Outputs);
36          block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
37          block.RegBlockMethod('InitializeConditions',    @InitConditions);
38          block.RegBlockMethod('Terminate',               @Terminate);
39      else
40          block.RegBlockMethod('Start', @Start);
41      end
42
43  end
```

Figure 7 Example of Setup function.

A Level-2 MATLAB S-function stores discrete state information in a DWork vector. occur in a model, your S-function must use DWork vectors instead of global or static memory to store instance-specific values of S-function variables. Otherwise, your S-function runs the risk of one instance overwriting data needed by another instance, causing a simulation to fail or produce incorrect results. The ability to keep track of multiple instances of an S-function is called *reentrancy.*

You can create an S-function that is reentrant by using DWork vectors that the engine manages for each instance of the S-function.

DWork vectors have several advantages:

- Provide instance-specific storage for block variables
- Support floating-point, integer, pointer, and general data types
- Eliminate static and global variables
- Interact directly with the Simulink engine to perform memory allocation, initialization, and deallocation

2.1.1.2 C MEX File Application

We can call your own C programs from the MATLAB command line as if they were built-in functions. These programs are called MEX functions and the function name is the MEX file name, mex filenames compiles and links one or more C source files written with the MATLAB Data API into a binary MEX file in the current folder. To call a MEX function, we can use the name of the MEX file, without the file extension. The MEX file contains only one function or subroutine. The calling syntax depends on the input and output arguments defined by the MEX function. The MEX file must be on your MATLAB path. Example of a sample Mex function call from MATLAB is given in the figure 8.

For compiling mex files we have to use MATLAB Supported and Compatible Compilers. We can use the command:
`mex -setup` for displaying the options for our version.

```
function Outputs(block)
global TM4;
if (isempty(TM4))
    [Output_value,Counter_Update] = Counter_Mex( block.InputPort(1).Data,block.InputPort(2
else
    [Output_value,Counter_Update] = TM4_Counter_Mex( block.InputPort(1).Data,block.InputPo
end

block.OutputPort(1).Data      =     logical(Output_value);
block.Dwork(1).Data           =     Counter_Update;
%%
%Storing the Outputs in the Dwork for variable structure in workspace
dw_Output_for_workspace(block);

end
```

Figure 8 Example of Mex function call from .m file.

Refer to appendix 1 for mex code template.

2.1.2 Code Composer Studio

Code Composer Studio (CCStudio or CCS) is an integrated development environment (IDE) to develop applications for Texas Instruments (TI) embedded processors. Development environment for CCS is shown in the figure 9. CCS is used for firmware code for algorithms running on the processors, these programs are called from

MATLAB Mex functions respectively and the output of blocks is updated after implementing the behavior as it in firmware.

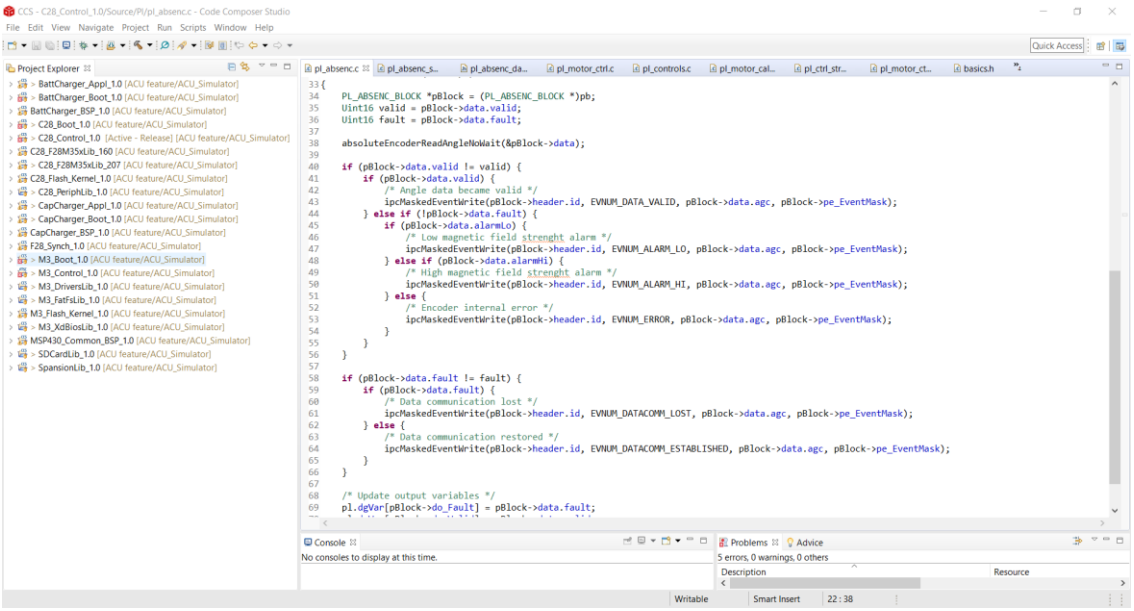

Figure 9 Code Composer IDE.

2.1.3 Programmable Logic

The ACU firmware modules do not execute any application specific logic, i.e. an ACU based electronic device does not have any built-in high-level logic. The application behavior is defined through the Configuration file, i.e. description of the application logic created using a Simulink block diagram, the interaction with the hardware is shown in the figure 10.

The logic application, i.e. the functional behavior of an ACU-based apparatus, is designed by creating a Simulink block diagram using a suitable ACU Simulink blocks library, a set of special blocks intended to create a PL logic. The application is the result of blocks behaviors and connections between blocks. Connections are shown by wires with arrows in the diagram, routed from output ports of a block to input ports of another block (loops are allowed using a special block called *Loop Breaker*).

Three types of wires are available:

- *Digital*: the wire carries 0/1 (FALSE/TRUE) values.
- *Analog*: the wires carries floating-point values.

- *Link*: the wire carries a reference to a block that allows the source block to access data structures belonging to the destination block.

Each block of the set is associated to a matching firmware module made by a data structure and (at least) two functions (the so-called "init-function" and "run-function") that must be available in the main ACU firmware in order to execute the block's intended behavior. The "init-function" is called at system start-up; it initializes the block's internal data structures. The "run-function" is called every execution cycle of the logic and implements the logic specific behavior. Each block has also an associated S-function (like usual Simulink blocks) that is executed when the simulation is run to create the configuration file downloaded to the ACU device. At application start-up, the ACU firmware PL engine scans the configuration file building an internal list of the blocks and executes once each block's init-function. After the initialization, the PL engine runs called

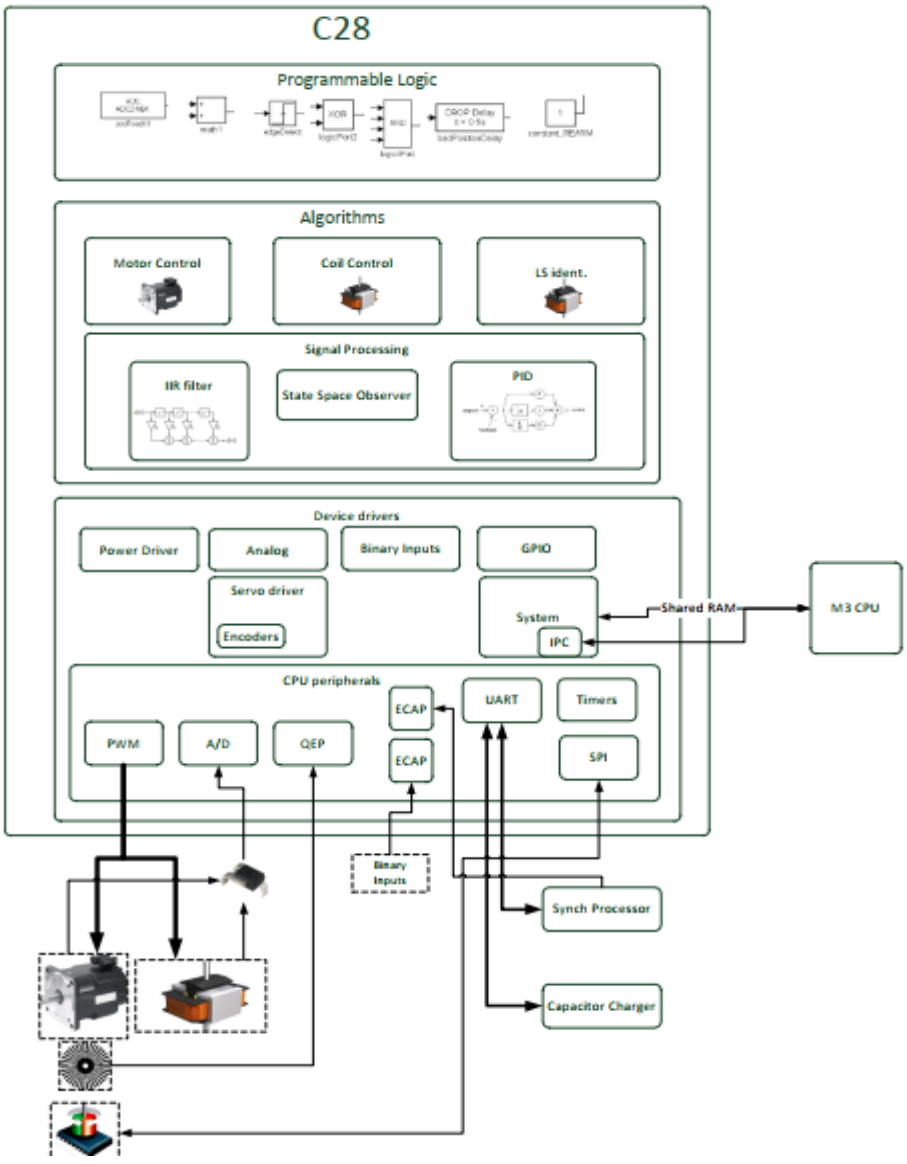

Figure 10 PL interaction with firmware algorithms.

periodically (1ms tick time) calling each run-function in the order defined by the block diagram connections, using the content of the configuration file as working data.

2.1.4 Configuration File

Once the block diagram is done in the Simulink graphic design environment, the first step of ACU configuration creation process is starting the Simulink block diagram simulation (Figure 11, button **Run** or menu **Simulation/Run**).

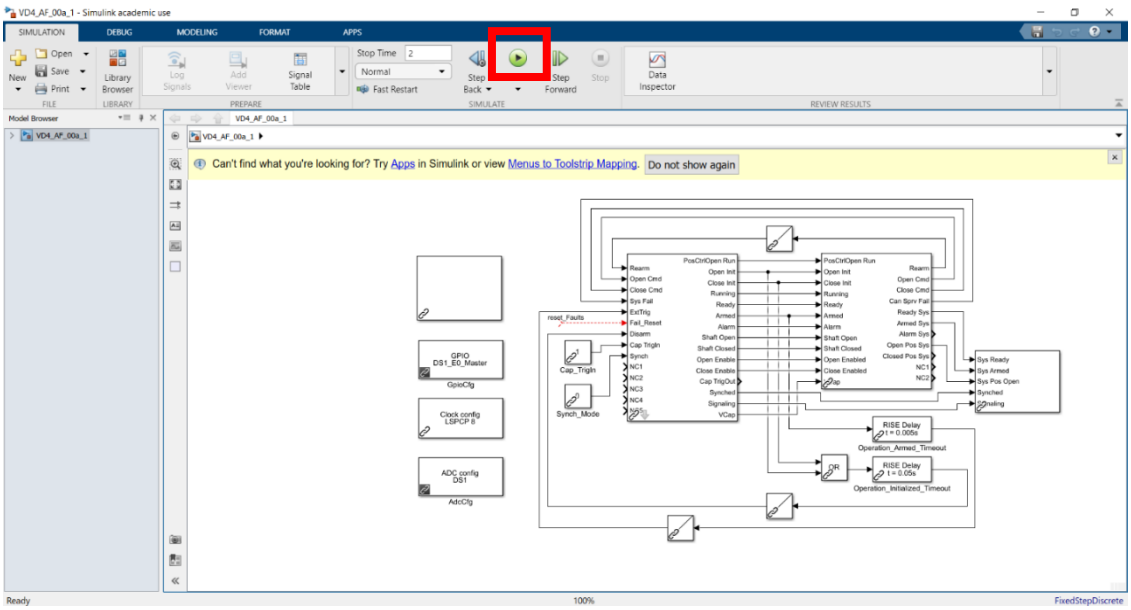

Figure 11 Run button for Generation of Configuration file.

The Simulink engine scans the diagram, building suitable data structures, which represent the connections between blocks and the execution order of the code related with each block. Furthermore, the S-functions associated to each block are executed (as for a standard Simulink simulation), along with a set of utility m-functions and S-functions supplied with the ACU library. The S-functions execution produces a C-language source code derived from the diagram as intermediate result of the process. Such C-code is a sequence of statements that declare, for each block of the configuration, an instance of the corresponding **data structure** used in the firmware to execute the block's logic. Note that **the C-source code is not the code that executes the block application logic but only a mean to automatically create the working data structures used by the firmware.**

The second step is done by compiling and executing the generated C-code. The execution writes a binary file, which contains all instances of the data structures declared by the code as a sequence of memory images of each block's data structure filled with the

corresponding parameters. Such binary file is part of the configuration file that will be downloaded to the ACU to implement a specific ACU logic. The binary file is copied in the ACU main CPU's memory, i.e. it is the image of the working memory used by the ACU PL engine while running the block firmware code.

Besides the binary file, the Simulink diagram execution adds to the configuration file some text information describing configuration parameters and monitoring signals, in terms of name, data type and memory placement. This additional information is in text/XML format and is used by the ACU Remote Tool to display/edit the configuration and to perform the real-time monitoring of PL logic running by the ACU device.

Chapter 3

Simulator Performance Optimization

Analysis of the current simulation platform performance, evaluation of more efficient solutions for signal management and storage have been discussed in this chapter. First the details of the product simulated are discussed.

3.1 Outdoor Vacuum Recloser (OVR) configuration

The outdoor vacuum recloser is a medium voltage vacuum recloser for outdoor applications. The vacuum interrupters of this circuit breaker are operated by a single coil magnetic actuator; the low end recloser includes protection and control functionalities that are supported by voltage and current sensors and a protection relay.

The control unit for the vacuum recloser drives the magnetic actuator and provides uninterruptible power supply. It performs the opening and closing commands generated by the protection relay and converts the auxiliary power to supply the whole system. Furthermore, the control unit for the OVR includes diagnostic and protection functions that allows preventive and selective maintenance of the system.

The ACU configuration layout of OVR is shown in the figure 12.

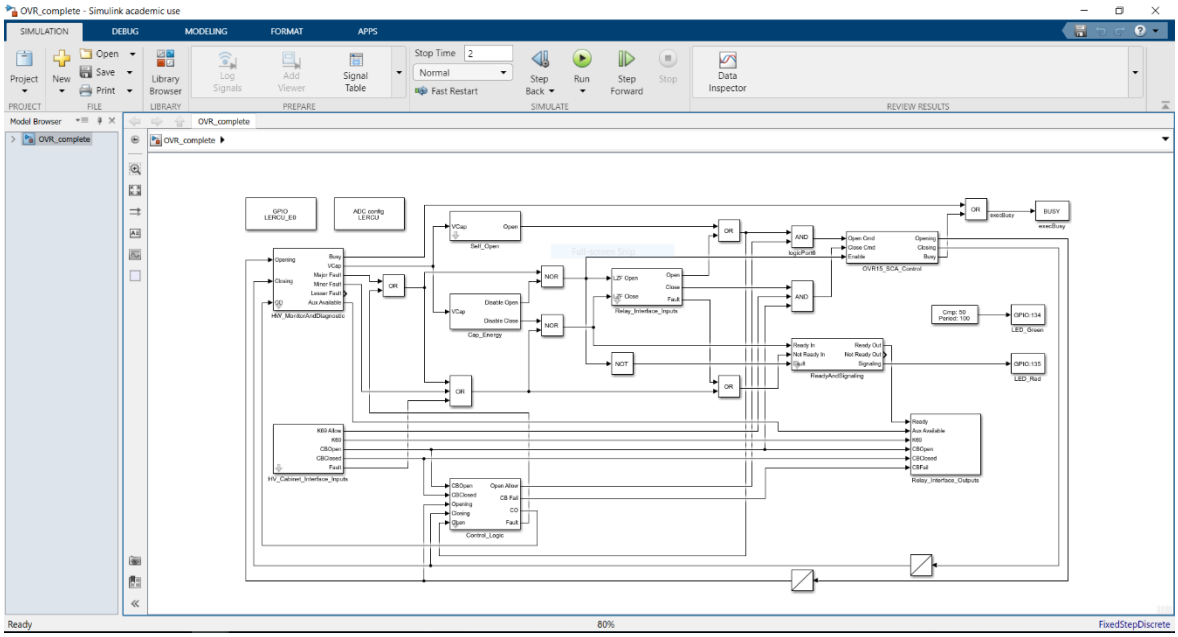

Figure 12 OVR Simulink Model

3.2 Analog and binary inputs for OVR

The analog and binary inputs for OVR are generated by a separate input file, all the peripherals of OVR are simulated to write array values to MATLAB workspace. Input configuration is depicted in the below figure 13 and 14.

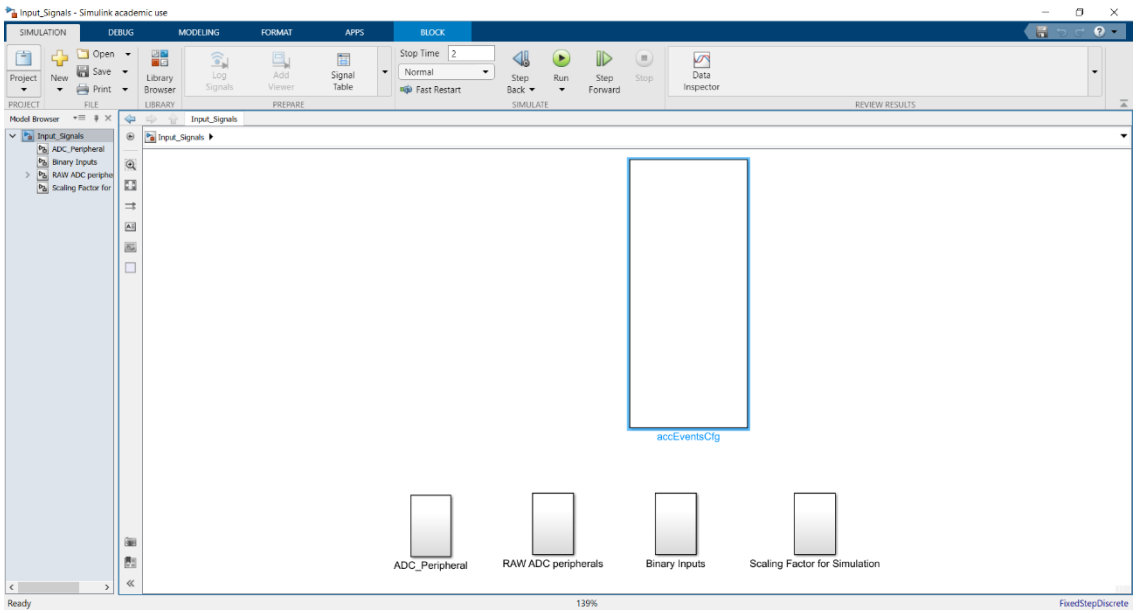

Figure 13 Input File for Generation of inputs.

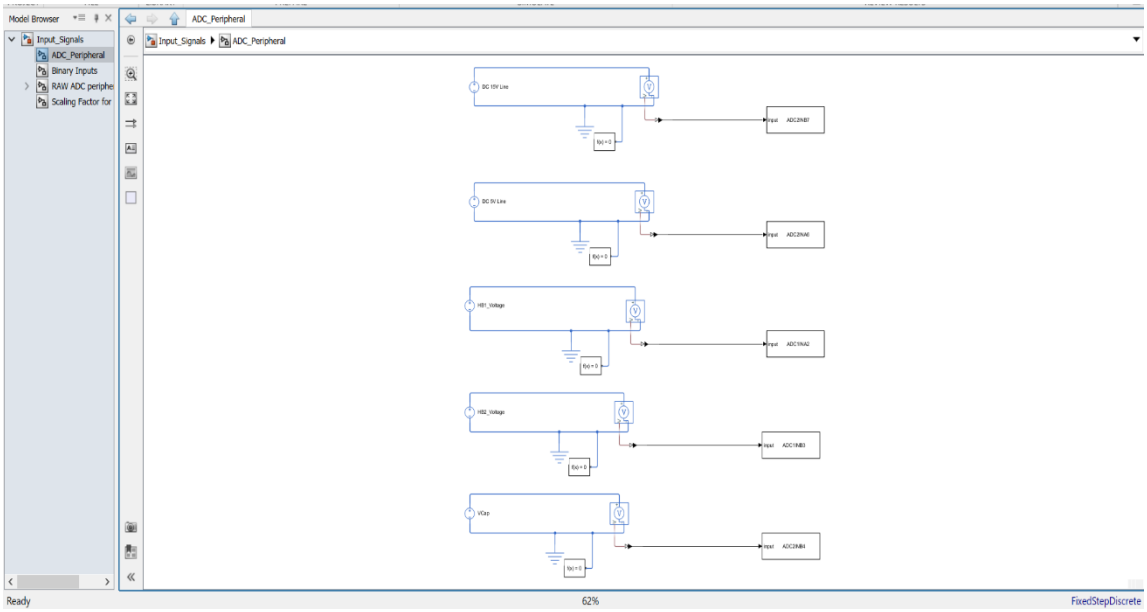

Figure 14 Analog inputs for Simulation

3.3 Performance analysis and improvement of the OVR

Analysis of the current simulation platform performance, evaluation of more efficient solutions for signal management and storage have been discussed in this section.

3.3.1 Analysis of the system total simulation time.

Total time of OVR is calculated by using MATLAB standard commands i.e. tic toc. Characteristic of system running Simulink.

- Processor: Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz 2.70 GHz
- Installed memory (RAM): 16.0 GB (15.6 GB usable)
- System type: 64-bit Operating System

Time analysis of OVR full configuration using **Simulink Profiler Report** is given in figure 15. Total time is around **240 seconds** for **2 seconds** simulation.

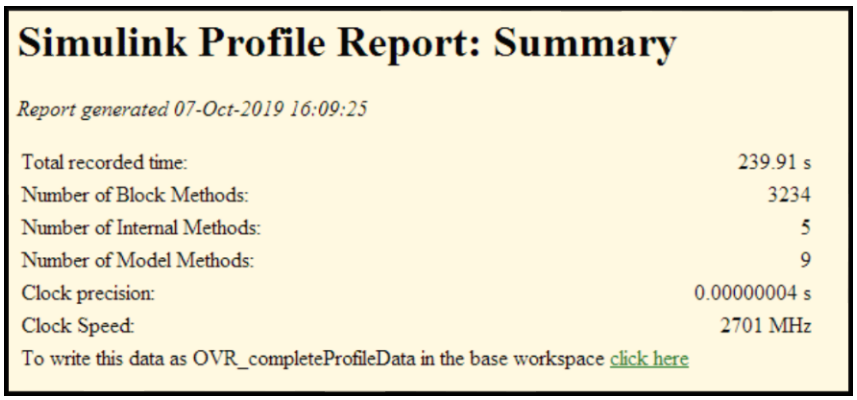

Figure 15 Simulink Profiler Report Summary OVR

Further analysis shows **'OVR_complete.Outputs.Major'** function is taking **up to 86%** of the total simulation time i.e. **206 seconds** as shown the figure 16 This is the function which executes at every iteration, taking in all the inputs and calculating the outputs to be transferred to the wires.

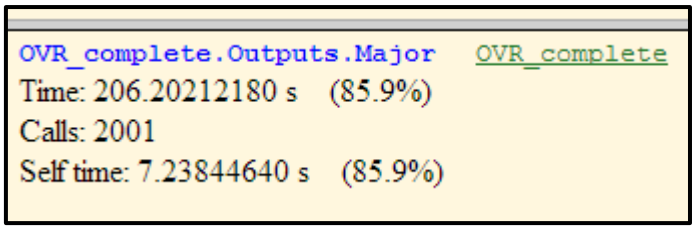

Figure 16 OVR total Output function time.

More investigation into the output() functions of two most complex and time-consuming blocks in the configuration.

- Coil Load Diagnostic
- SCA control

Coil Load Diagnostic

Total time for output() for Coil Load Diagnostic without tic toc is **2.449s** as shown in the figure 17.

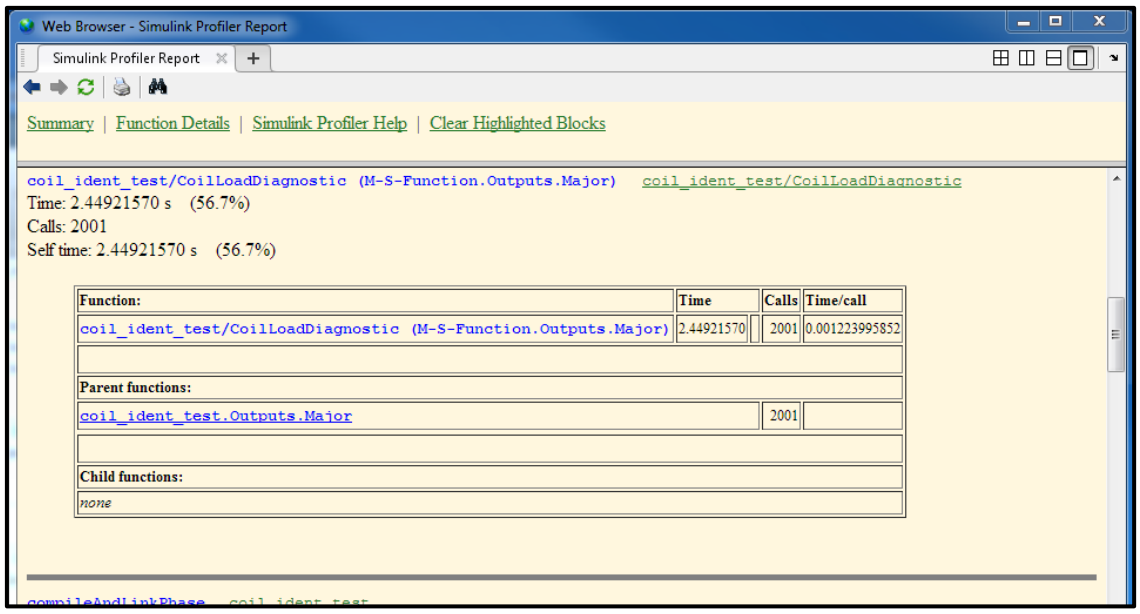

Figure 17 Coil Load Diagnostic output function time.

Total time for output for Coil Load Diagnostic with tic toc is **2.418 s** as shown in the figure 18.

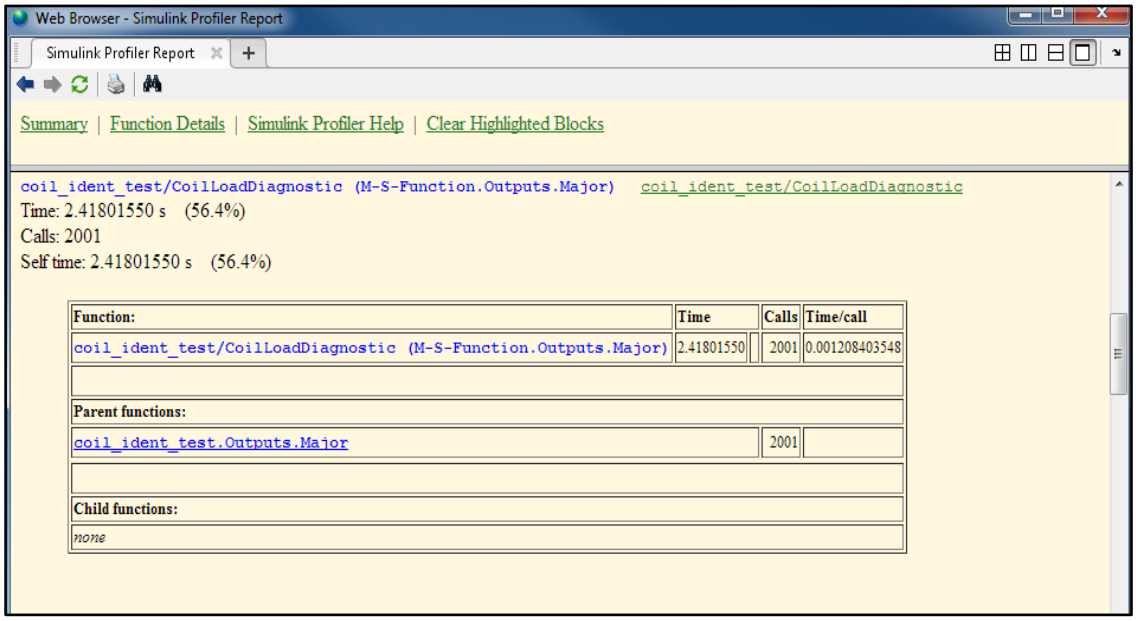

Figure 18 Coil Load Diagnostic output function time with tic toc.

Break down of Output() function is calling **Mex() function** and defining Dwork for variable structure in workspace **dw_DoPost_for_workspace(block)**, time distribution is shown in the figure 19.

Mex() function time to total time = **67.3%**

dw_DoPost_for_workspace(block). of total time = **21.79%**

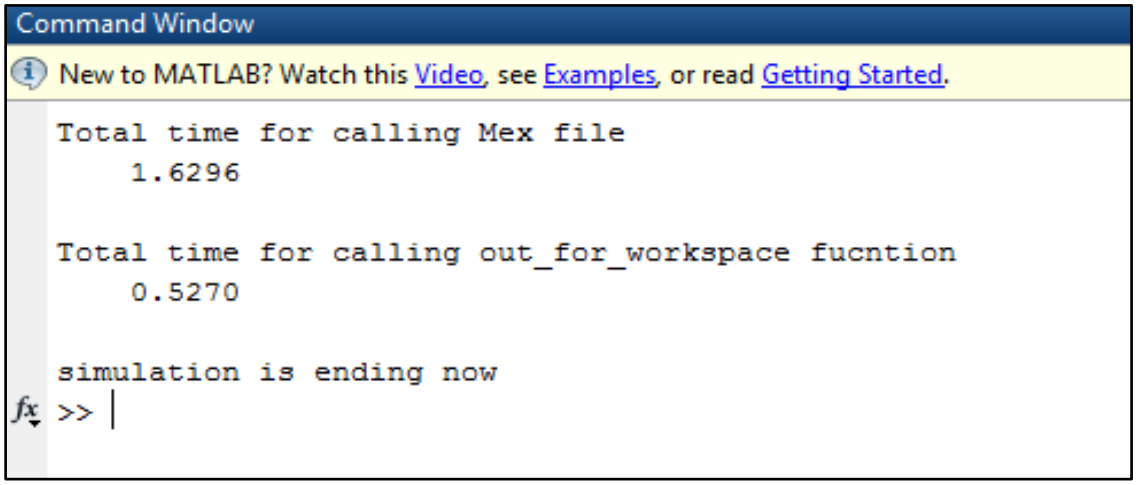

Figure 19 Breakdown of Output function.

Single Coil Actuator

Total time for output() for **SCA control** without tic toc is **3.0264 s**, as shown in the figure 20

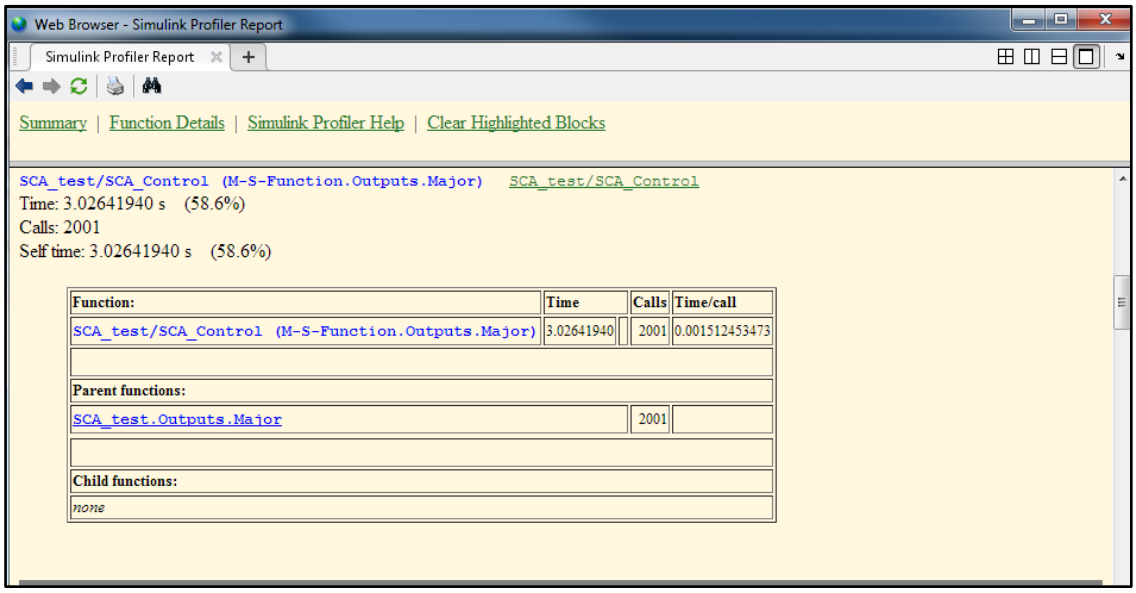

Figure 20 SCA control output function time.

Total time for output() for **SCA control** without tic toc is **2.38 s**, as shown in the figure 21.

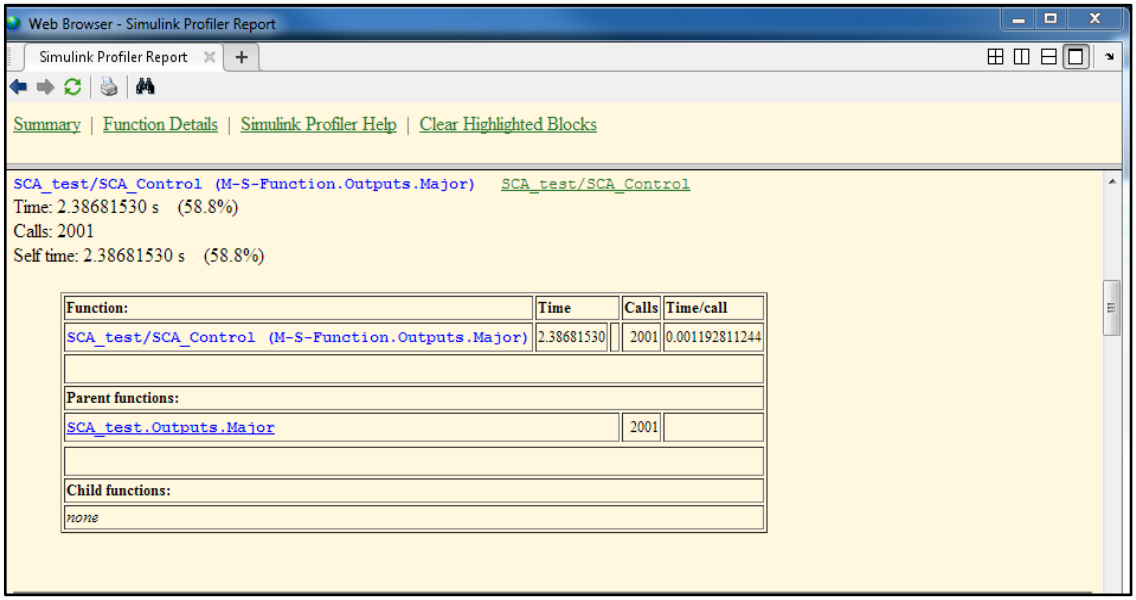

Figure 21 SCA control output function time with tic toc.

Break down of Output() function is calling **Mex() function** and Defining Dwork for variable structure in workspace **dw_DoPost_for_workspace(block** time distribution is shown in the figure 22.

Mex() function time to total time = **57.59%**

dw_DoPost_for_workspace(block). of total time = **29%**

Figure 22 Breakdown of output function.

3.3.2 Results obtained from the analysis

From this analysis this clear that **dw_DoPost_for_workspace(block)** is taking around **30%** of time in the execution of output function, if we choose blocks, we want to save output in the workspace we can save significant time in overall simulation execution.

3.3.3 Performance Improvement

To improve the performance speed, options are to select a faster simulation mode, not saving all the outputs to workspace and Select only important blocks before and simulation and publish output for only those blocks

3.3.3.1 Selection of Simulation Mode

- In **Normal mode**, Simulink interprets your model during each simulation run. If you change your model frequently, this is generally the preferred mode to use because it requires no separate compilation step.

- In **Accelerator mode**, Simulink compiles a model into a binary shared library or DLL where possible, eliminating the block-to-block overhead of an interpreted simulation in Normal Mode. Accelerator mode supports the debugger and profiler, but not runtime diagnostics.

Normal Mode Simulation time = 240 **seconds**

Accelerator mode Simulation time = 230 **seconds**

According to MATLAB, Accelerator mode may not improve performance much in the following situations:

- Your model contains compiled code
- Your simulation runs include initialization or termination phases
- You log large data sets.

3.3.3.2 Not saving all the outputs

For this purpose, we can do another analysis on a smaller configuration **(HW_Diagnostic configuration)** to see how much time we can save if suppress **dw_DoPost_for_workspace(block)** function the simulation.

HW_Diagnostic configuration with writing to workspace active, is shown in the figure 23

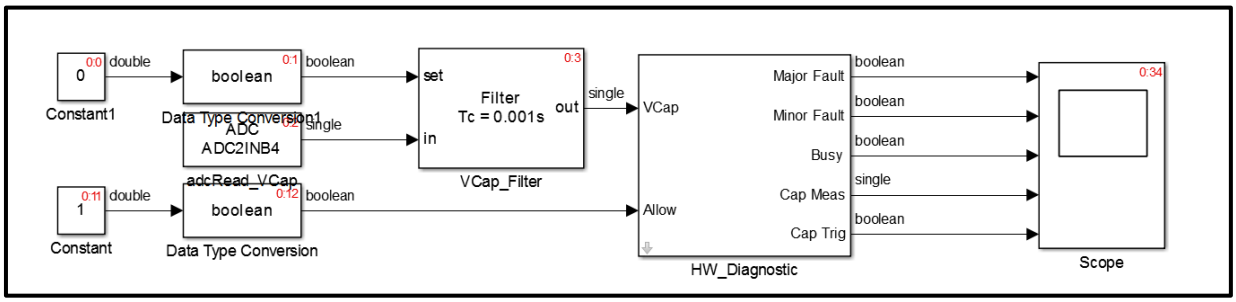

Figure 23 Hardware Monitor Model

Three consecutive tests are shown in the figure 24

Test No.	Time (s)	
1	**25.997033**	Mean Time
2	**26.741193**	26.528*s*
3	**26.848650**	

Figure 24 Test results for HW monitor.

HW configuration with writing to workspace inactive, results re shown in the figure 25

Test No.	Time (s)	
1	**16.737243**	Mean Time
2	**17.556115**	16.967s
3	**16.609024**	

Figure 25 Test results for HW monitor with no workspace output.

Total time saved without writing to workspace is:

Percentage difference 36.04 %

Difference 9.5615 seconds

3.3.3.2 Selecting important blocks for output saving

Select_output script (for detailed code see appendix 1) is written to select blocks which needs to be published before simulation starts. This script helps the user to select the blocks before the simulation by clicking on the blocks he wants to save the output. At the beginning, the number of blocks should be entered, the block to be saved are selected by clicking on them in Simulink model diagram. The selected block names and handles are saved and published in the workspace. When the number of selected blocks is equal to the input, the simulation starts.

For analyses purpose **HW_Diagnostic configuration** as shown in the figure 26 and 27 is used with modified output() functions for each block.

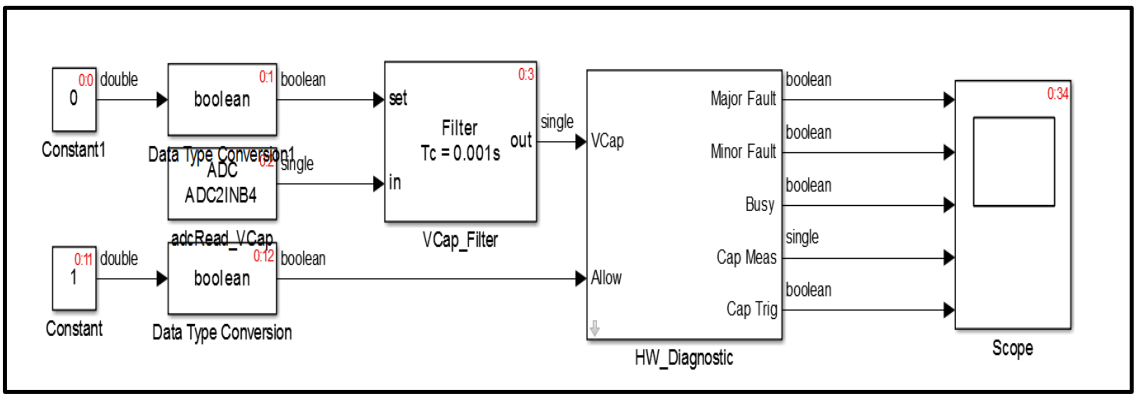

Figure 26 HW Diagnostic configuration.

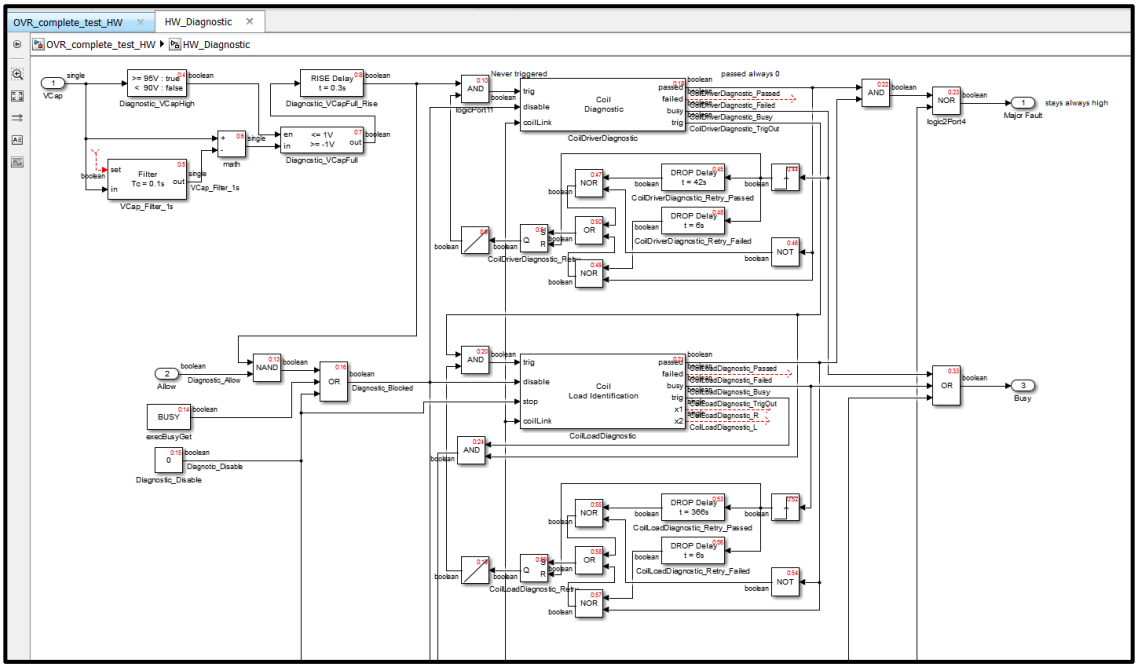

Figure 27 HW Diagnostic configuration.

For time analysis three complex blocks are chosen for simulation which are:

Selected block: OVR_complete_test_HW.HW_Diagnostic.CoilDriverDiagnostic

Selected block: OVR_complete_test_HW.HW_Diagnostic.CoilLoadDiagnostic

Selected block: OVR_complete_test_HW.HW_Diagnostic.CapDiagnostic

We know from our previous results that mean time for running this simulation is **26.5 seconds**Error! Reference source not found., whereas after selecting only important blocks w e reduce it to **18.47** seconds figure 28.

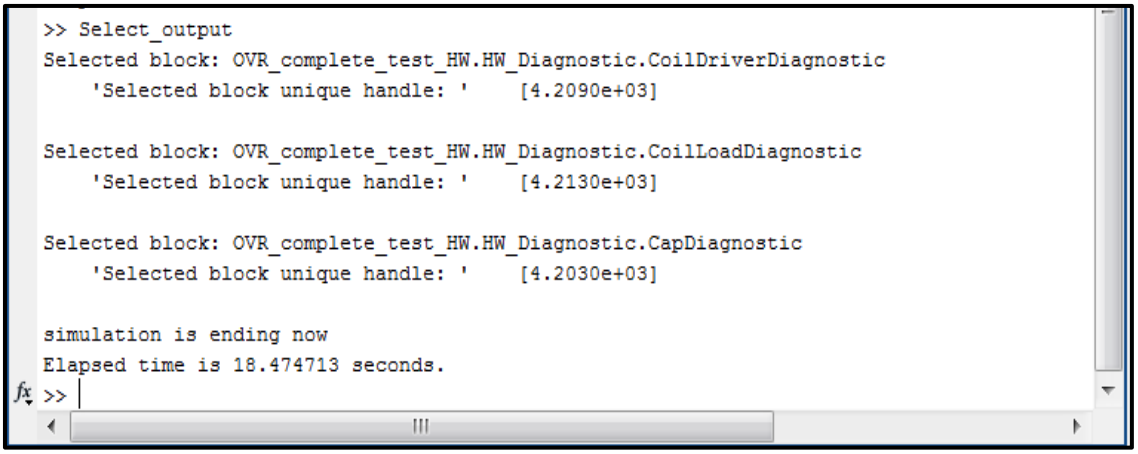

Figure 28 Simulation time after selecting specific blocks.

Chapter 4

Simulator library expansion for final SIL test

For the final test on ABB product VD4-AF, it is an exclusive product based on vacuum technology and an innovative actuation system to provide up to 150,000 close-open operations. Its technological breakthrough is based on servomotors, controlled by an efficient and smart electronics. The circuit breaker comprises:

Three poles in epoxy resin containing the vacuum interrupters , three brushless servomotors, one per phase with double encoder, three electronic controllers, one per phase, which communicate hierarchically with each other and where the first unit controls the entire system, an electronic supply unit, a capacitor for storing the energy required to operate the circuit breaker in the absence of auxiliary supply, three sensors and three mechanical position indicators, an operating lever seat for opening the circuit breaker in the manual mode

For simulating this apparatus, motor control, encoders, CAN bus and GPIO blocks had to be developed.

4.1 Motor Control Blocks

Motor Control blocks implement closed loop controllers for motor controls and identification, and diagnostic algorithms.

4.1.1 Motor load identification.

The Motor Load Identification block controls the Motor Load Identification function, which tests the actuator winding dynamics. The motor winding is excited to induce a small current to flow into the winding, observing coil voltage and current to estimate its transfer function parameters using the Least Squares algorithm. Actuator winding electrical resistance (R) and inductance (L) are estimated and compared to their minimum and maximum values allowed, specified by configuration parameters. Successful

execution of this diagnostic is required to perform subsequent diagnostic functions and Open/Close operations, otherwise a fault is raised.

The figure 29 shows the motor load diagnostic block with the test inputs from signal builder.

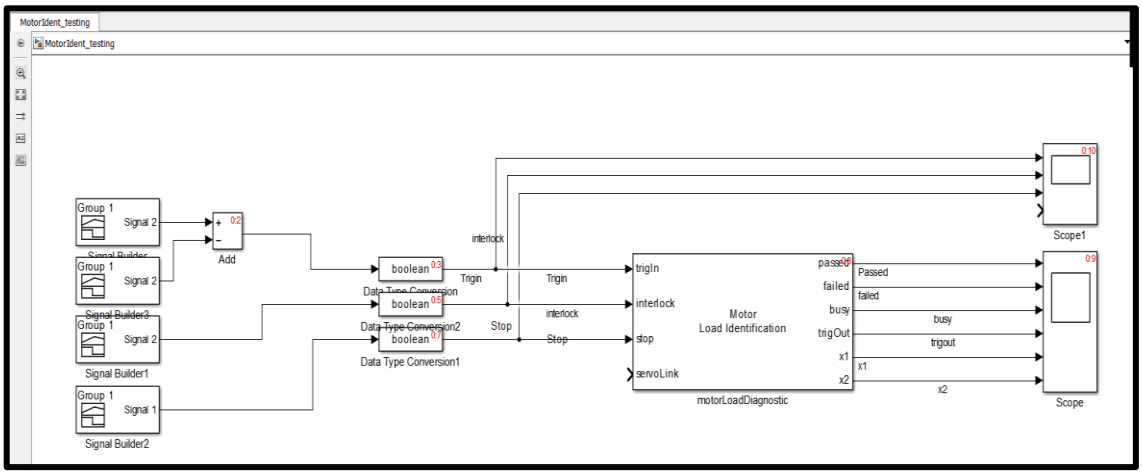

Figure 29 Motor Load Diagnostic block.

Figure 30 shows the results obtained from the independent simulation the block.

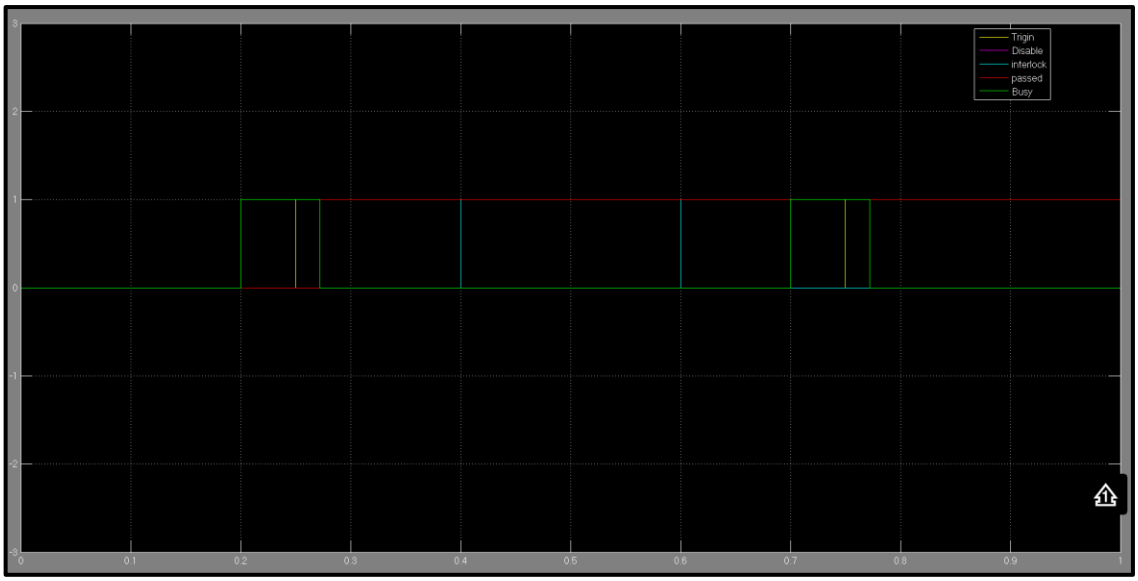

Figure 30 Motor Load Diagnostic Simulation results.

Figure 31 for inputs and outputs

	Type	Description
Inputs		
trigIn	D	Diagnostics trigger input (1 = start).
disable	D	Disable input (1 = disabled).
servoLink	L	Link to the *Servo* block (see 9.5.13) that describes the servomotor driven by the diagnostics algorithm.
Outputs		
passed	D	Diagnostic passed output. TRUE if the diagnostic has been run and successfully passed.
failed	D	Diagnostic failed output. TRUE if the diagnostics has been run and not passed.
busy	D	Busy flag. TRUE while the system is busy to run the diagnostics procedures.
trigOut	D	Trigger output, cleared at the beginning of diagnostics procedures execution and set to 1 at the end of execution, if the diagnostics passed.
x1	A	Motor R estimated value (Ohm)
x2	A	Motor L estimated value (Henry)

Figure 31 Inputs and Outputs.

4.1.2 Motor calibration

The Motor Calibration block implements the following procedures used to characterize a motor connected to the control system:

Find Index: the control system drives the motor in order to detect the index track of the absolute encoder. The motor is driven to perform N turns, where N is the number of motor's pole pairs. Driving is done using an indexless control algorithm, which controls the Id, Iq currents of the FOC control without sensing the angle position.

Calibrated Angle: the control system drives the motor in order to detect the Calibrated Angle value, i.e. the offset between the electrical angle associated with FOC algorithm transforms and the mechanical angle read by the encoder.

The figure 33 shows the motor calibration block with the test inputs from signal builder

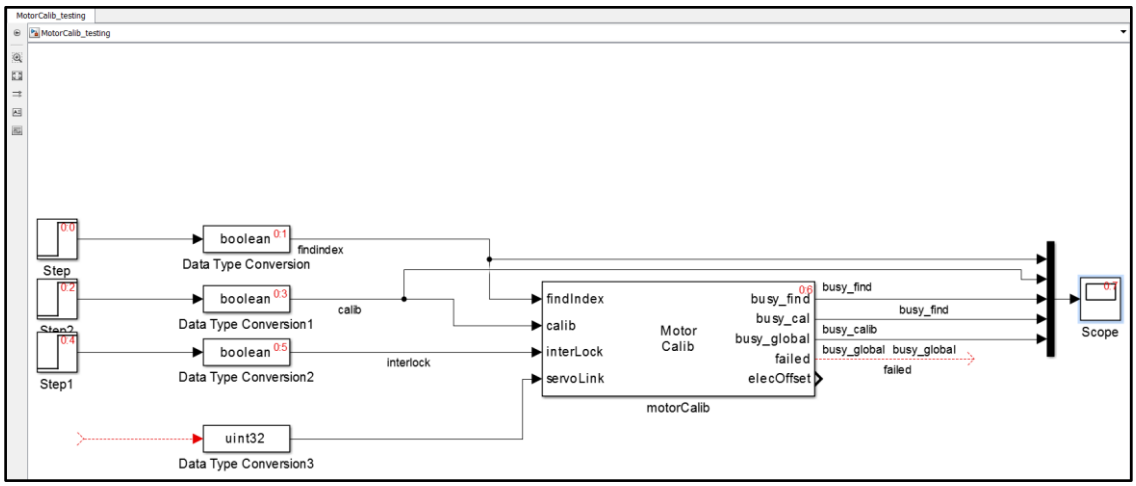

Figure 32 Motor Calibration Block.

Figure 33 shows the results obtained from the independent simulation the block.

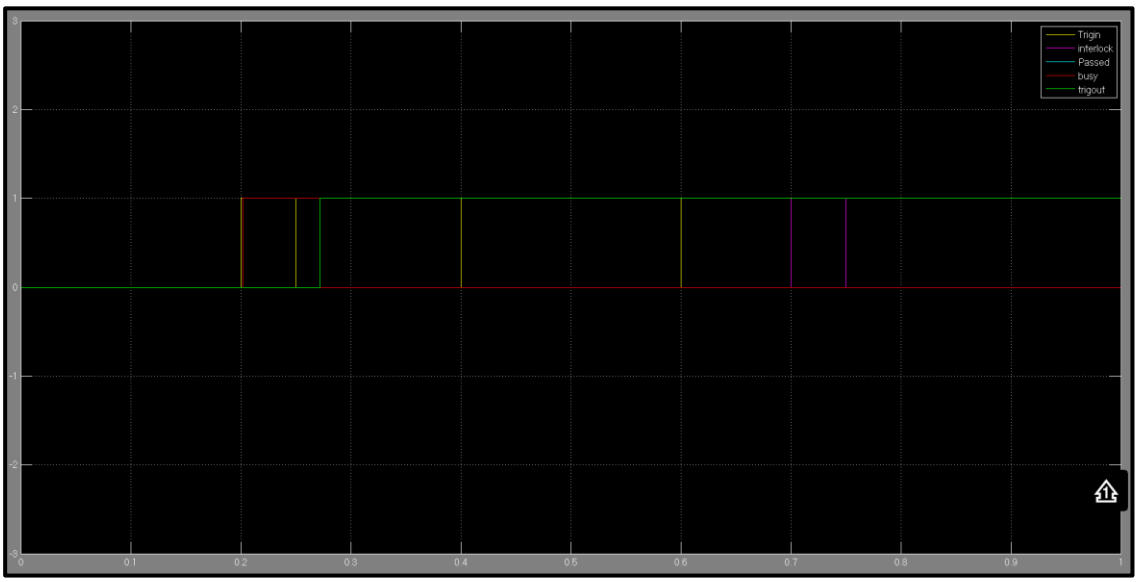

Figure 33 Motor Calibration block Simulation results.

Figure 34 shows f inputs and outputs for motor calibration block.

	Type	Description
Inputs		
findIndex	D	Find Index procedure start signal (1 = start).
Calib	D	Calibrated Angle procedure start signal (1 = start).
Interlock	D	Interlock signal. Operations do not start if *interlock* = 1.
servoLink	L	Link to the *Servo* block (see 9.5.13) that describes the servomotor driven by the control algorithm.

Outputs		
busy_find	D	Find Index busy output. Busy flag for Find Index procedure. TRUE while the system is busy to execute the Find index procedure.
Busy_cal	D	Calibrated Angle busy output. Busy flag for Find Index procedure. TRUE while the system is busy to execute the Calibration procedure.
Busy_global	D	Overall busy flag. TRUE while the system is busy to run any of the above procedures.
Failed	D	TRUE if the Find Index procedure failed finding the encoder index track.
elecOffset	A	The value of Calibrated Angle detected by the Calibrated Angle procedure.

Figure 34 Input and Outputs.

4.1.3 Motor driver diagnostic

The Motor Drive Diagnostic block implements several motor diagnostics procedures.

The figure 35 shows the motor load diagnostic block with the test inputs from signal builder

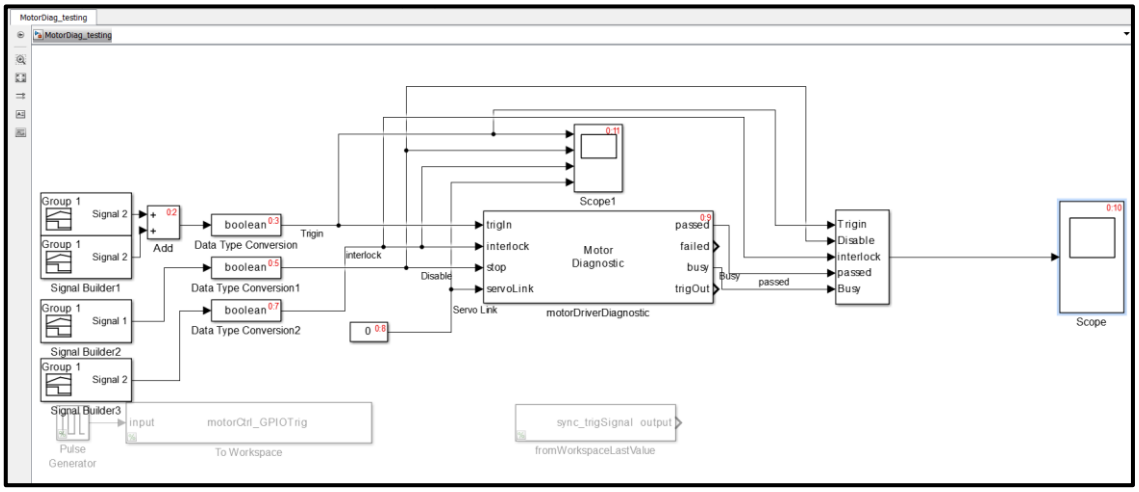

Figure 35 Motor Diagnostic Block.

Figure 36 shows the results obtained from the independent simulation the block.

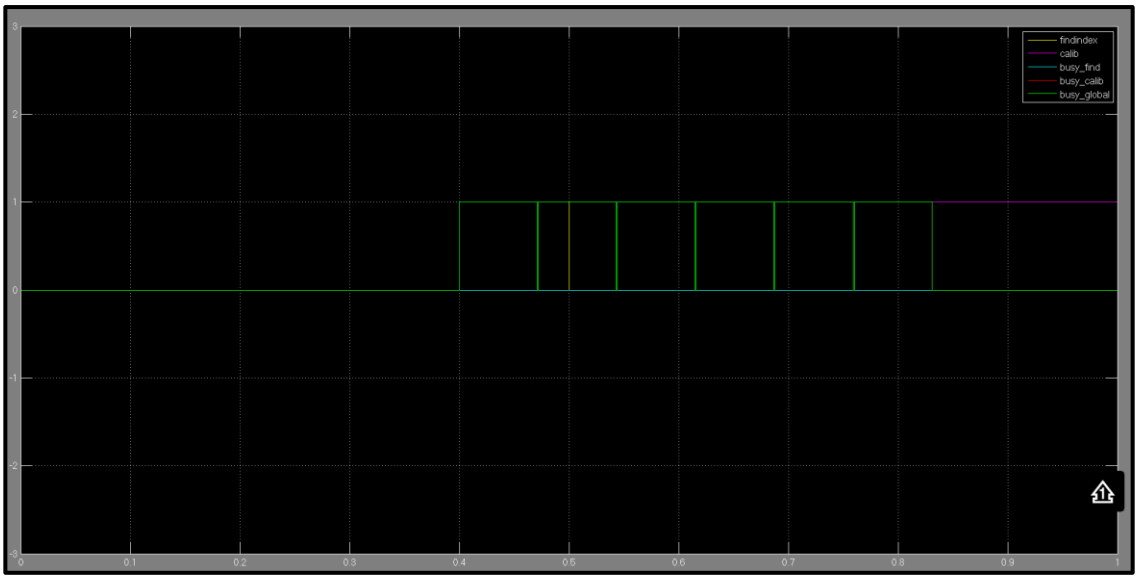

Figure 36 Motor Diagnostic block Simulation results.

Figure 37 shows inputs and outputs for motor diagnostic block.

	Type	Description
Inputs		
trigIn	D	Diagnostics trigger input (1 = start).
disable	D	Disable input (1 = disabled).
stop	D	Stop input. Gracefully stops a running motor diagnostic procedure. The motor diagnostic stops within the same logic execution step.
servoLink	L	Link to the *Servo* block (see 9.5.13) that describes the servomotor driven by the diagnostics algorithm.
Outputs		
passed	D	Diagnostic passed output. TRUE if the diagnostic has been run and successfully passed.
failed	D	Diagnostic failed output. TRUE if the diagnostics has been run and not passed.
busy	D	Busy flag. TRUE while the system is busy to run the diagnostics procedures.
trigOut	D	Trigger output, cleared at the beginning of diagnostics procedures execution and set to 1 at the end of execution, if the diagnostics passed. Can be used to trigger another block when the diagnostics has been successfully completed.

Figure 37 Inputs and Outputs.

4.2 Communication I/O blocks

The I/O blocks of ACU Simulink library related to VD4-AF have been updated for the purpose of Input/Output handling.

4.2.1 CanInDg

The CanInDg Digital block allows receiving 16 digital variables from another ACU unit connected via CAN bus. The transmitting ACU shall write the digital values to the input ports of a CAN Out 16 Digital block using the same Variable Id parameter of the receiver block.

The CAN input for Unit 1, which in term it receives from unit 2 and unit 3 is written by input signals files using CanOutDg as shown in the figure 38

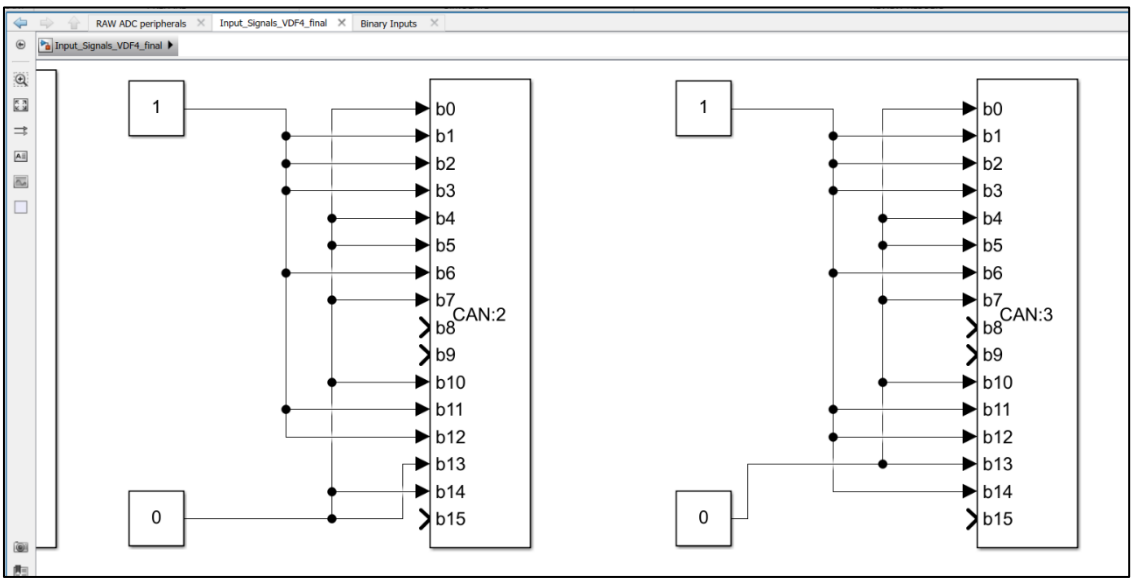

Figure 38 CAN 16 input block.

4.2.2 CanOutDg

The CanOutDg Digital block allows transmitting 16 digital variables to another ACU unit connected via CAN bus. The receiving ACU will read the digital values at the output ports of a CAN In 16 Digital block using the same Variable Id parameter of the transmitter.

The CAN output from Unit 1, which in term is received by unit 2 and unit 3 is written by using CanOutDg as shown in the figure 39.

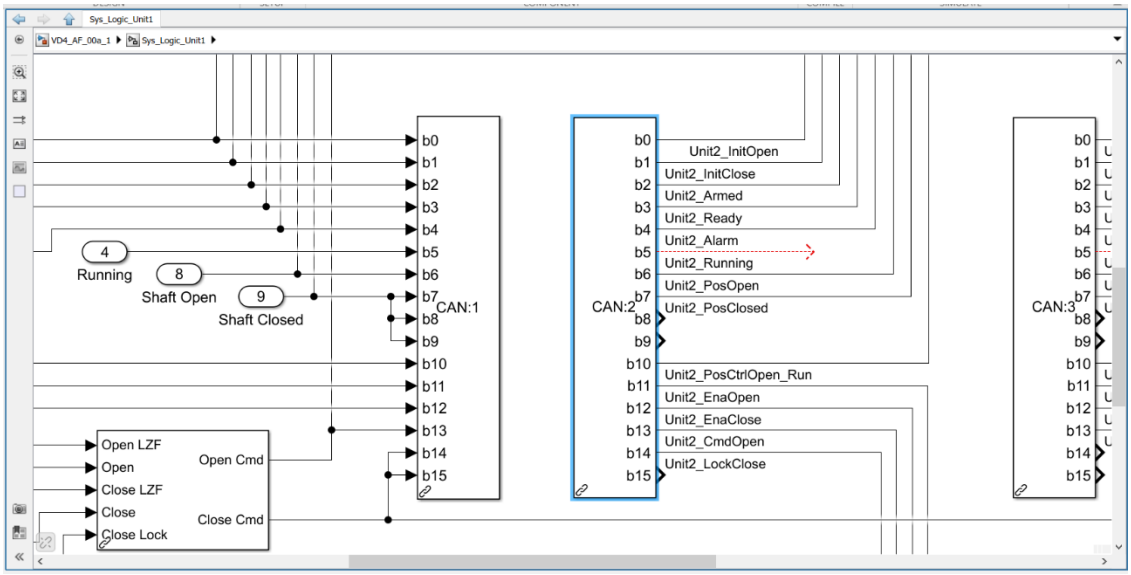

Figure 39 CAN 16 output block.

4.2.3 GPIO Output

The GPIO Output block allows driving an ACU CPU General Purpose I/O. The GPIO is actually driven only if the GPIO configuration of the relevant CPU port/pin mux is correctly set to GPIO output.

The GPIO Output for VD4-AF is written by input signals files using GPIOOutput block as shown in the figure 40.

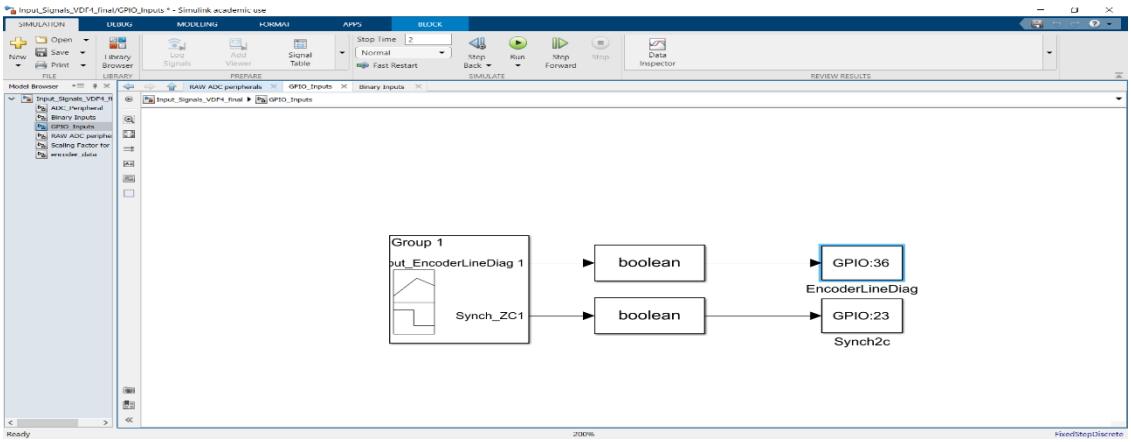

Figure 40 GPIO output block.

4.2.4 GPIO Input

The GPIO Input block allows reading an ACU CPU General Purpose I/O. The GPIO is actually read only if the GPIO configuration of the relevant CPU port/pin mux is correctly set to GPIO input.

The GPIO Input for VD4-AF is read using GPIOInput block as shown in the figure 41

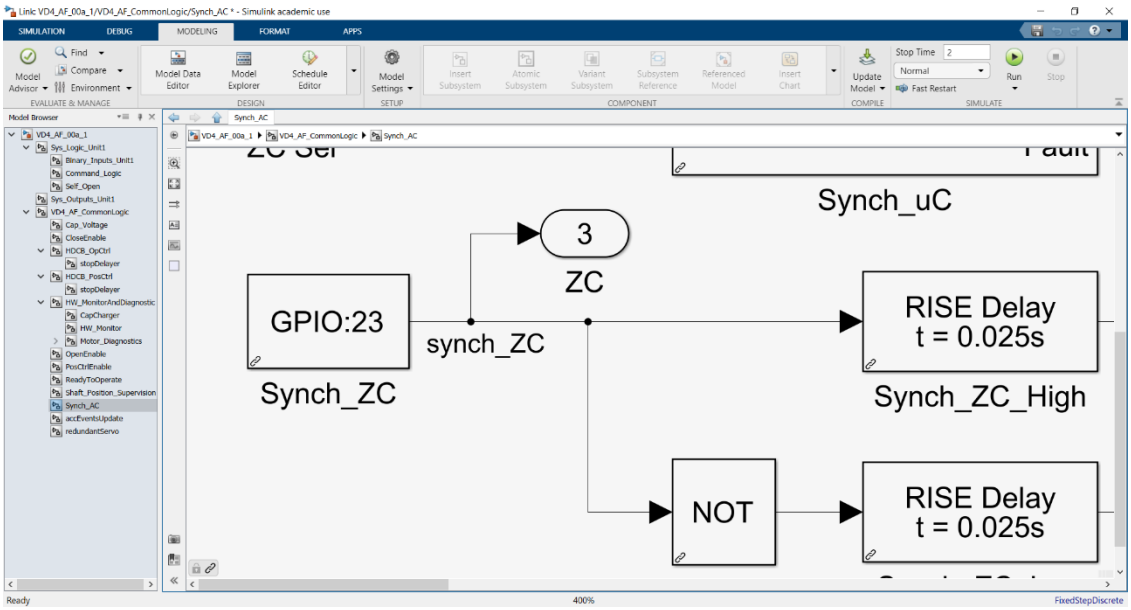

Figure 41 GPIO input block.

4.3 Absolute Encoder and Servo blocks

4.3.1 Absolute Encoder

The Absolute Encoder block implements the interface to the AMS AS5048A (14bit) or AS5055A (12bit) Rotary Sensor devices. AMS devices are magnetic rotary sensors interfaced to the CPU through an SPI line. See: # for details about the specific device.

For Simulation purposes, the shaft position angle read by encoder is assumed as constant for open or close positions. For simulation we have to give the position parameter as shown in the figure 42 that whether the breaker is open or closed initially, than the encoder position is changed according to the respective command for motor, i.e. if the breaker is initially at close and motor gets a open command the encoder will go to open

position after the motor stops running, Similar behavior for the close position is implemented.

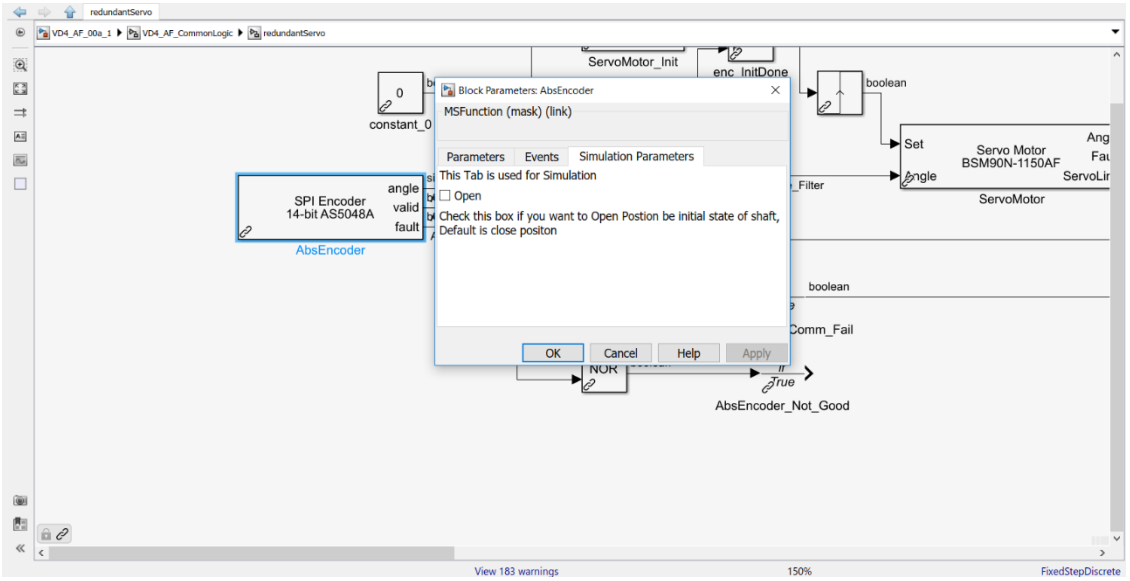

Figure 42 Absolute Encoder block mask.

The motor control **MotorControl_OptCtrl** is used for signalling encoder to opertae, a simulation parameter is added to the mask of motor control as shown in the figure 43, default operation is to not signal any encoder.

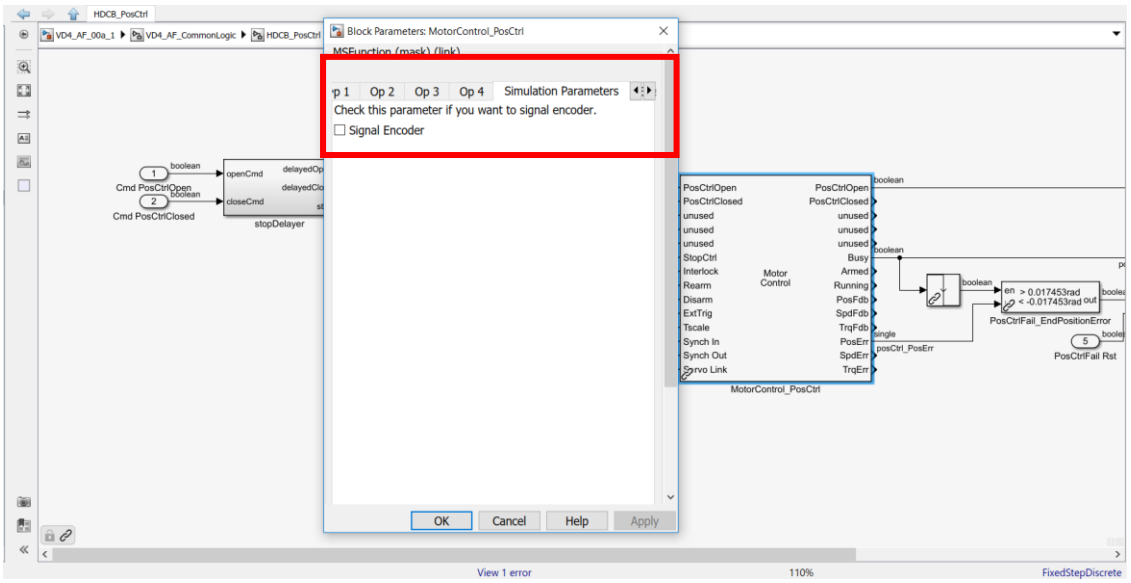

Figure 43 Simulation signal parameter for encoder

4.3.2 Servo Motor

The Servo block implements the data structures related to a servomotor device, where the relevant data are stored in order to be used by the Motor Control block connected using the servoLink output. The Servo block also performs a check of the encoder's index track, to detect invalid index reading or index loss conditions (no index track detected while the encoder rotates).

For Simulation, the servo just passes the angle as shown in figure 44 it gets from encoder so the error can be zero, also the fault is set to zero for all the simulation time, but the error can be added by simulation parameter as shown in figure 45.

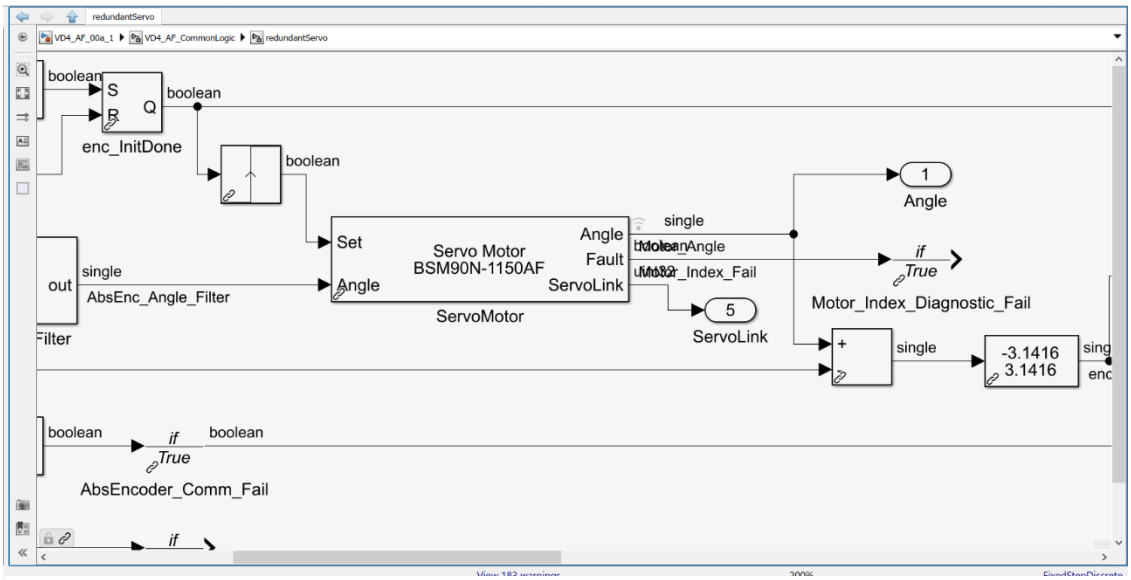

Figure 44 Servo motor block.

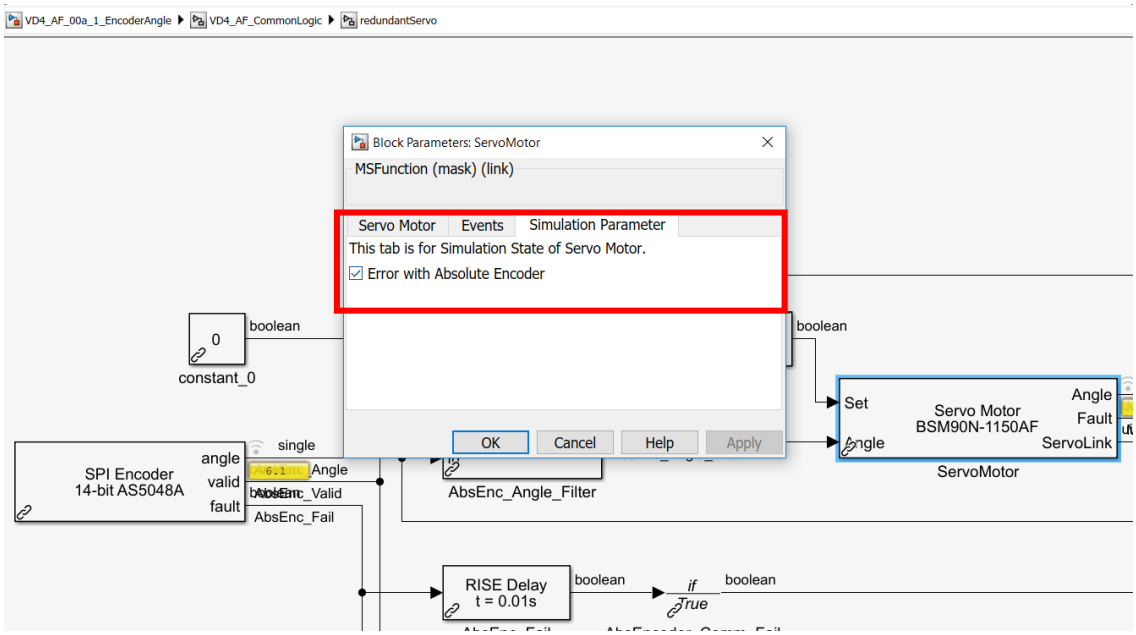

Figure 45 Servo motor error parameter.

Chapter 5

Graphical User Interface for Simulator

Since most of the configuration for different products have already been developed, the purpose is to utilize those configurations to validate the simulator and giver user the option to simulate already available logic configurations. So, a graphical user interface (GUI) for selecting configuration and selecting relevant option has been developed as shown in the figure 44.

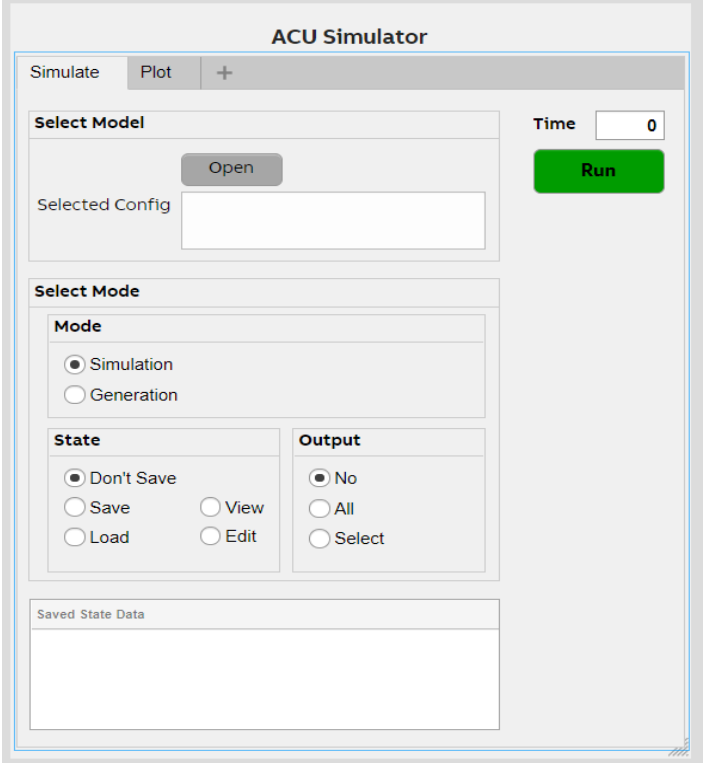

Figure 44 Graphical User Interface for Simulator.

MATALB app designer environment has been used for creating GUI. The design view of app designer is shown in the figure 45 allows us to create professional applications without having to be a professional software developer. To design GUI we simply drag

and drop components, then use the built-in editor to quickly program the behavior, code view is shown in figure 46

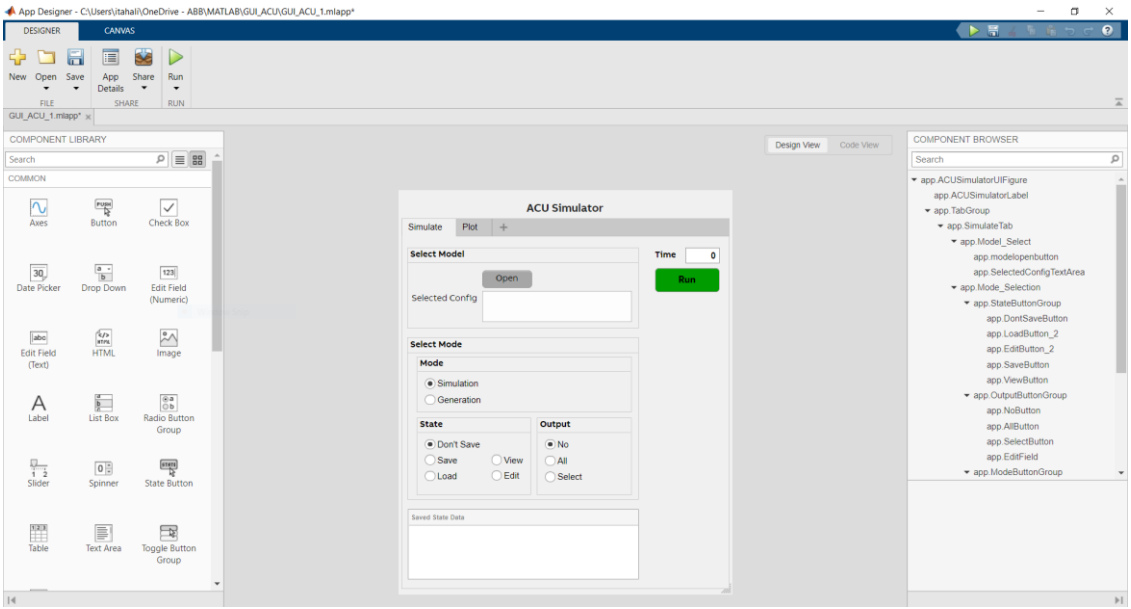

Figure 45 Design view of MATLAB App Designer.

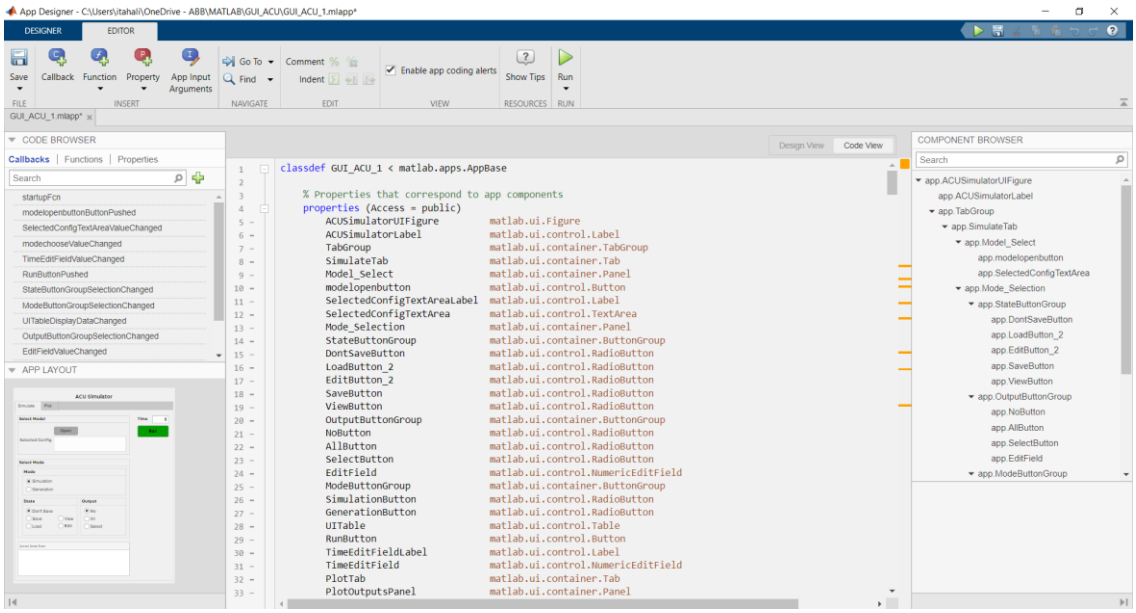

Figure 46 Code view of MATALB App designer.

App Designer provides a large set of components for designing modern, full-featured applications. The tables below list the components that are available in the Component Library:

1. Common Components — Include axes for creating plots, and several components that respond to interactions, such as buttons, sliders, drop-down lists, and trees.

2. Containers and Figure Tools — Include panels and tabs for grouping components, as well as menu bars.
3. Instrumentation — Include gauges and lamps for visualizing status, as well as knobs and switches for selecting input parameters.
4. Toolbox Components— Include toolbox authored UI components. Requires additional toolbox license and installation.

The call for GUI is written in the model callbacks of the logic configuration as shown in the figure 47 (for details about the code refer to appendix 1). The **Bypass_GUI** is used fto bypass GUI call and directly start the simulation, this feature is added for the further tests with Simulink Test Manager where multiple tests are required on the same model.

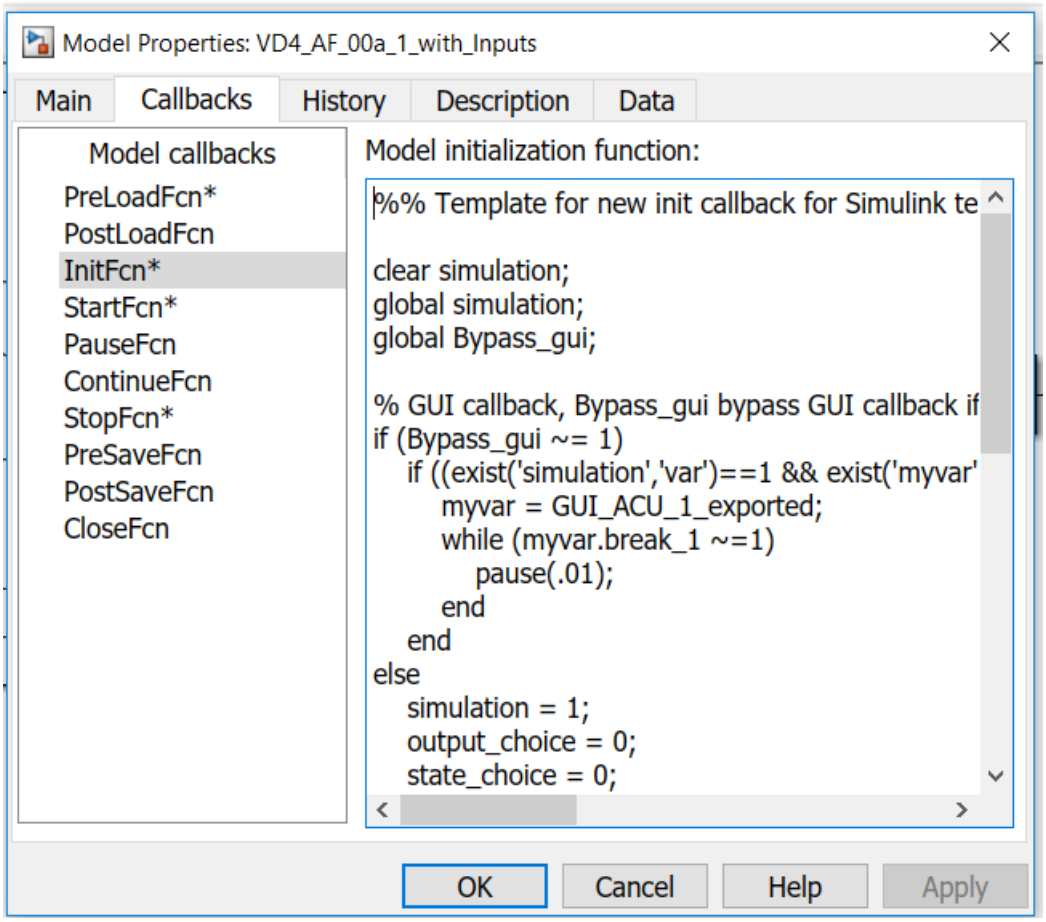

Figure 47 Model Callback for GUI object.

5.1 Graphical User Interface details.

The details about GUI are explained in the following sections.

5.1.1 Model Select

User has the option to select the model if the GUI is started separately, any model can be selected, and the name displays in the text box beneath the select model button as shown in the figure 48. If the GUI is called from the model callback than the name of the model is written to the text box.

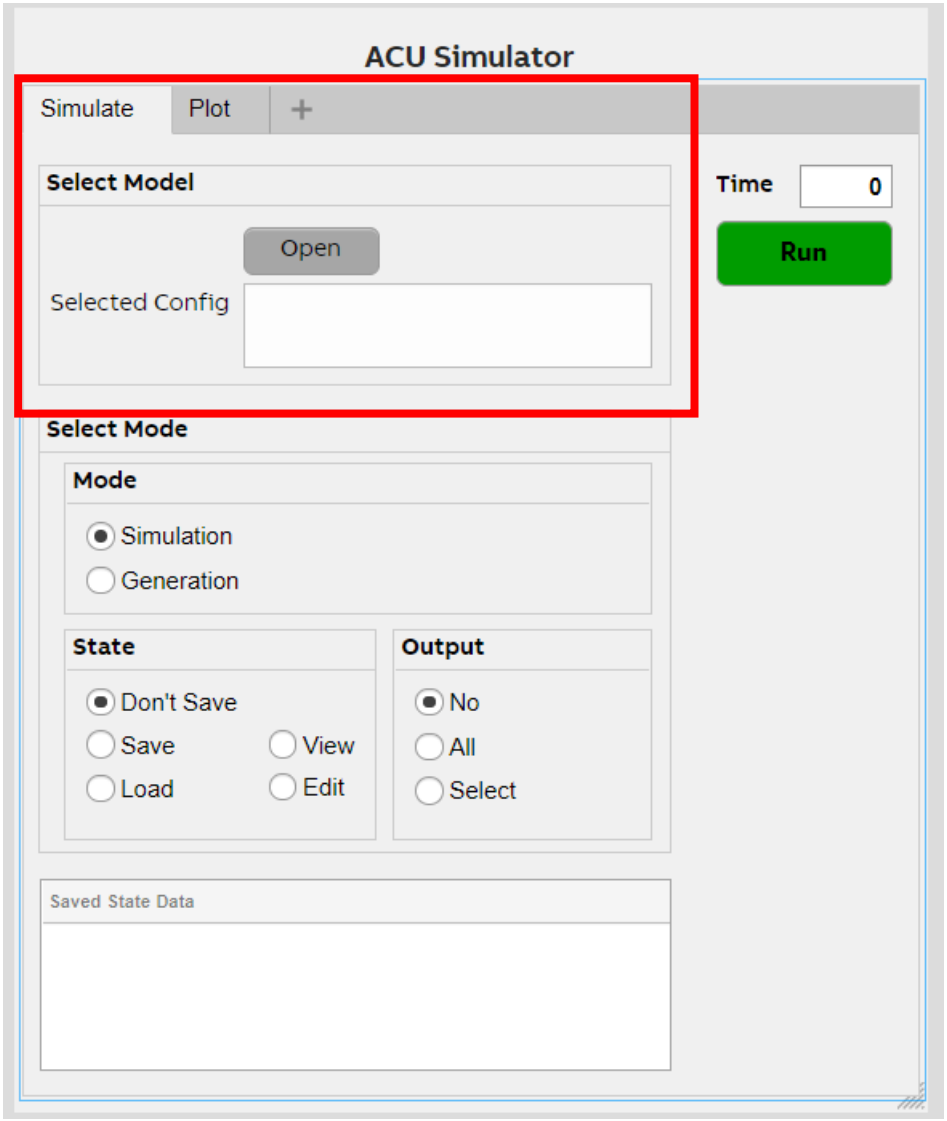

Figure 48 Model select button highlighted.

5.1.2 Simulation Mode

Simulation mode as shown in the figure 49 is used to differentiate between configuration file generation and Simulink simulation. When either of the option is selected a corresponding global variable is created which is used by all the level 2 S-functions for the respective function callbacks. If generation is selected only the start() in level 2 S-function is executed if simulation is selected all other function such as initialization, output terminate are executed.

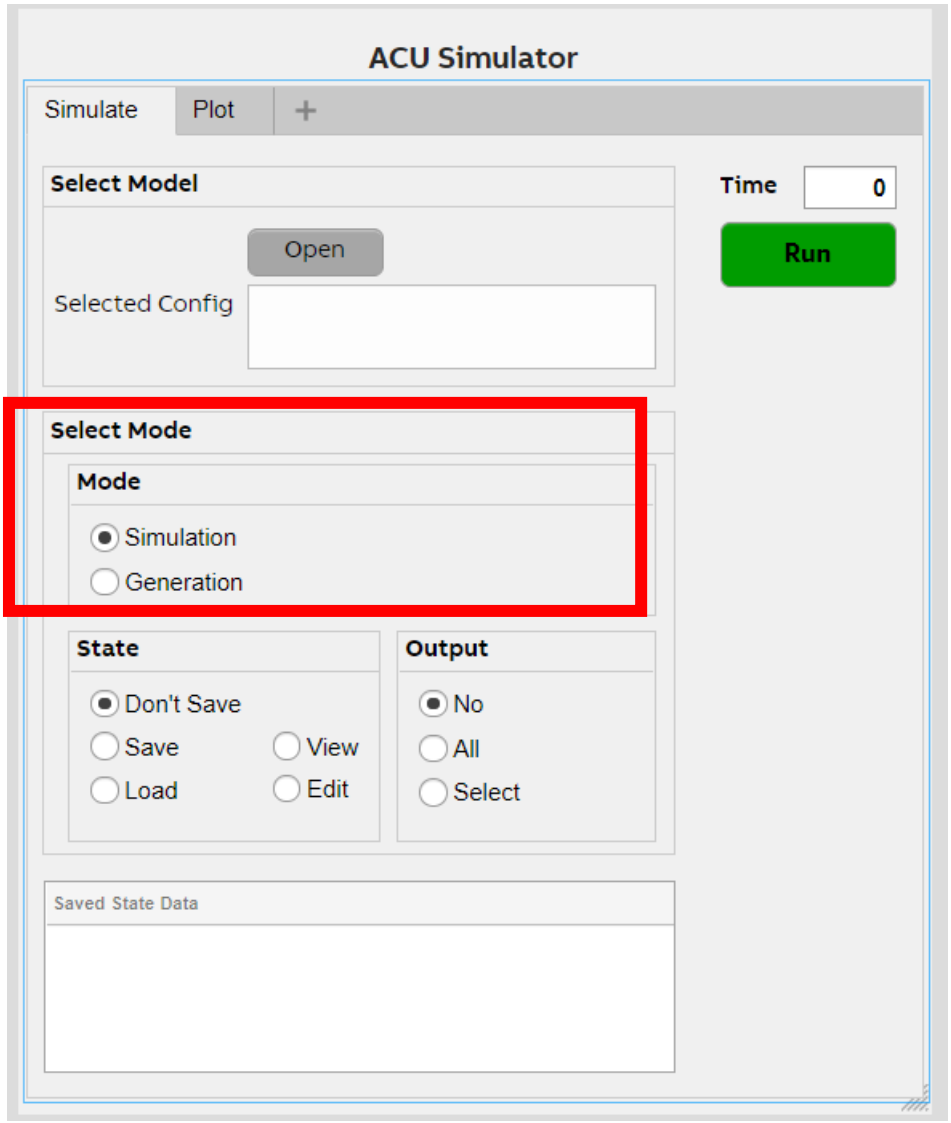

Figure 49 Simulation mode highlighted.

5.1.3 Simulation state

Storing simulation state for time consuming blocks is one of the approaches to save an instance of a simulation, so for the next simulation all the filter, counter and delay blocks have the information about the previous simulation.

User has the options as shown in the figure 50

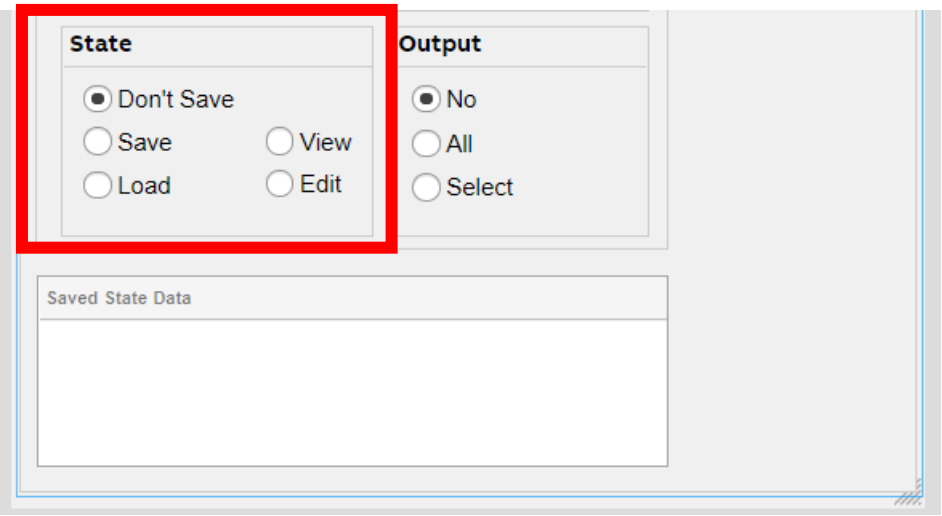

Figure 50 Simulation instance saving options.

If user select **save**, the output of the blocks at last iteration is saved in a binary file in the same MATLAB folder address as shown in the figure 51. If user select option

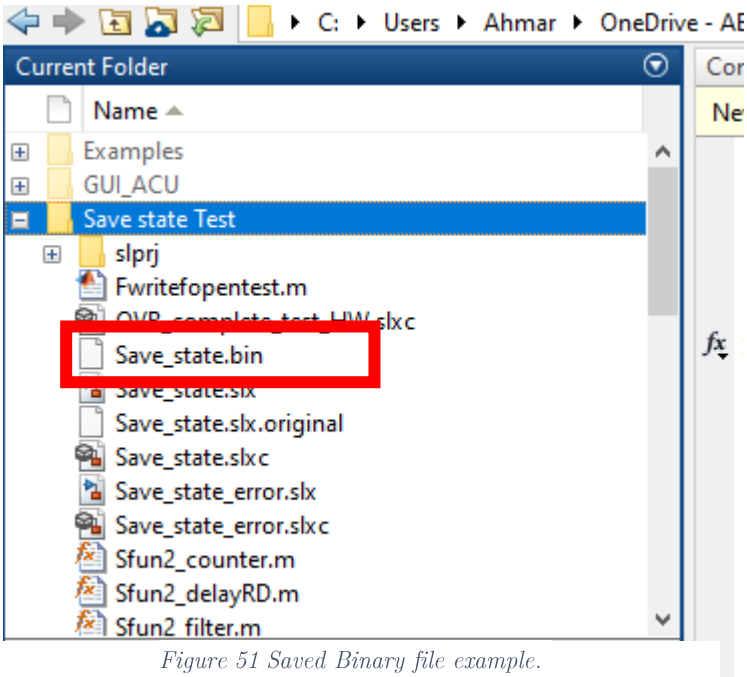

Figure 51 Saved Binary file example.

load, the initial value of the Dwork vectors related to output of the respective blocks are initialized to the data saved in the binary file in the same MATLAB folder address.

The **view** and **edit** option give user to see and modify the data stored in the binary files respectively. User can select any previous stored binary file using Windows standard explorer as shown in the figure 52. If user is only viewing the data the modification functionality is restricted, for modification of data the **edit** option must be selected.

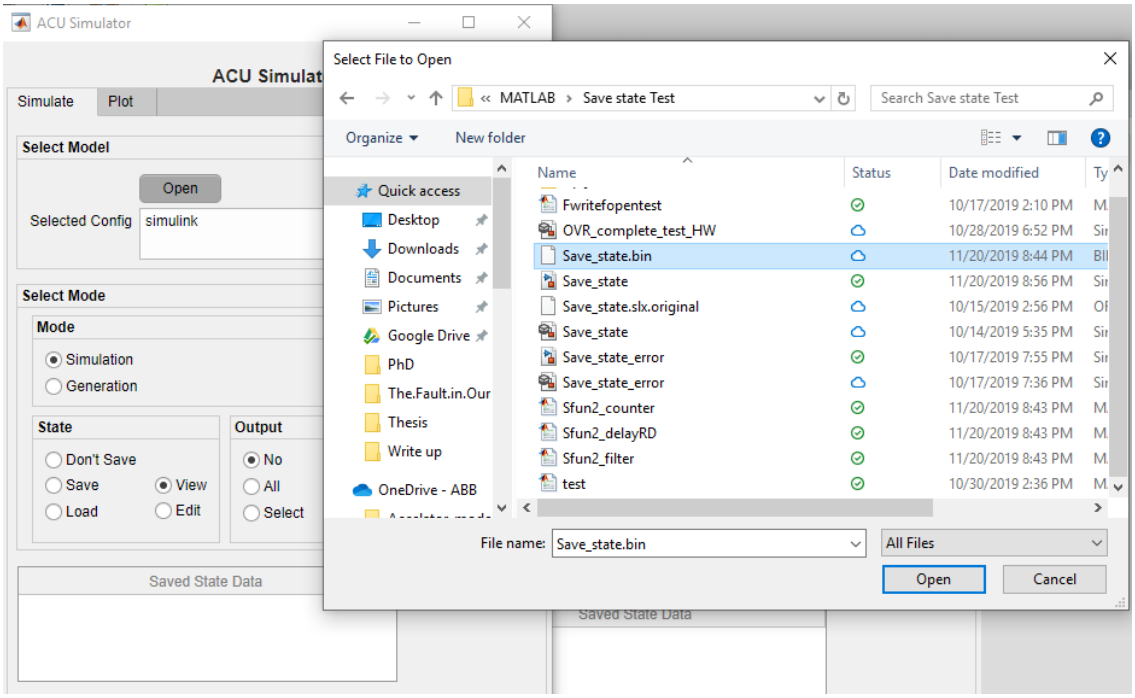

Figure 52 Viewing previous saved Binary File.

The data is displayed in the table below the select state tab as shown in the figure 53

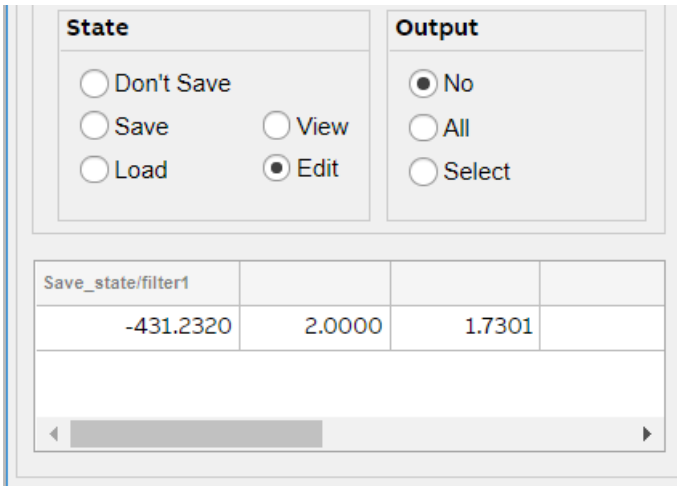

Figure 53 Previous saved state data table.

Instance saving mechanism has been tested by running simulation on a system as shown in figure 54 made up of the bocks which require more time than others like filter, delay, counter blocks. The system is simulated for 2 seconds and the last output is saved

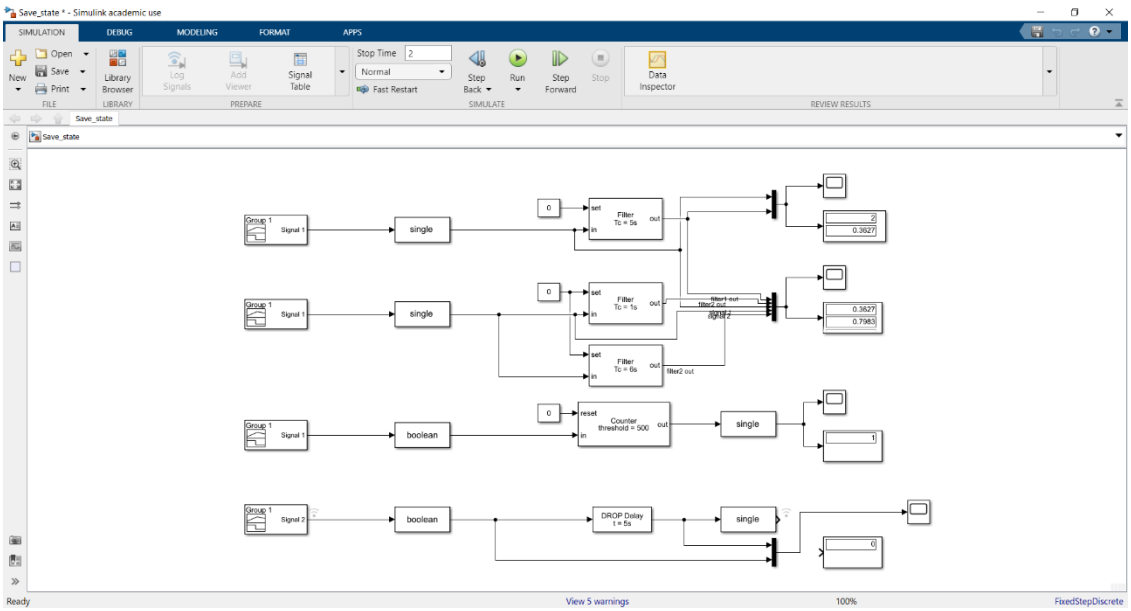

Figure 54 Test Configuration for Saving state.

in a binary file. Then the simulation is again ran, starting from the previous outputs. The outputs of 3 filters used is shown in the figure 55. The output of filter 1 with time constant of 5s starts at .35, filter 2 with time constant of 1s starts at .8. Similarly filter 3 with time constant of 6s starts at .2.

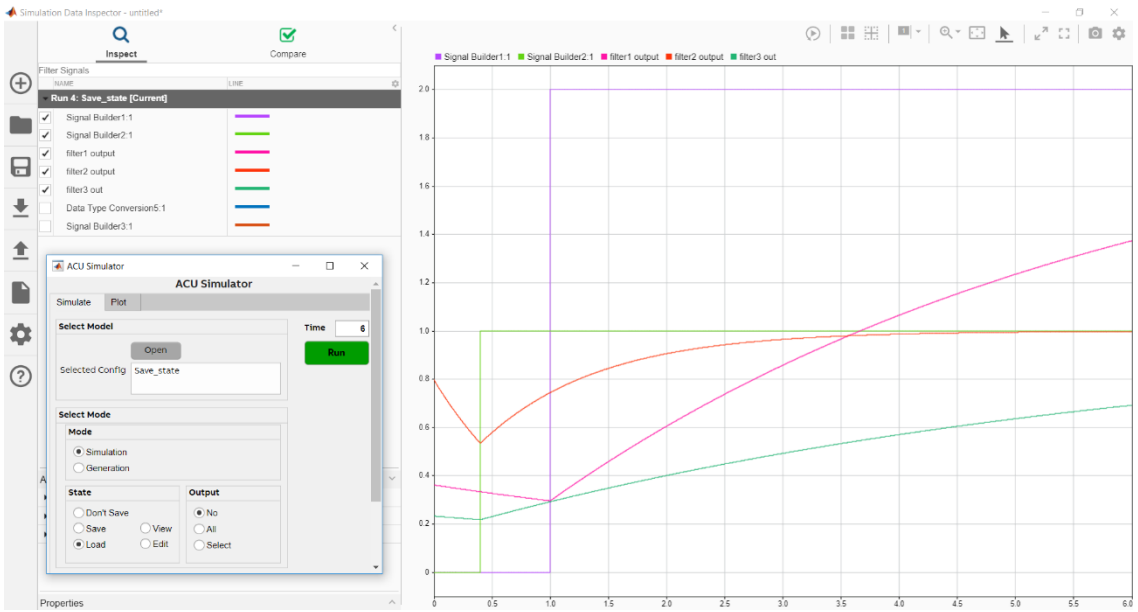

Figure 55 Filter outputs after loading from saved state.

Chapter 6

Software-In-Loop Simulation of VD4_AF

6.1 Testing Configuration VD4_AF

Logic configuration of the VD4-AF is depicted in the figure 56. This is the configuration of unit 1. Since other units are communication via CAN bus, some arbitrary inputs have been passed to the Unit1 by running an input file, other inputs that are required doe the simulation are:

1. Binary close and binary open inputs.
2. ADC peripheral inputs for hardware monitor and diagnostic.
3. GPIO inputs.
4. ADC inputs for capacitor voltage, temperature sensor.
5. Scaling factors for filter and delay blocks.
6. CAN unit 2 and CAN unit 3 inputs.

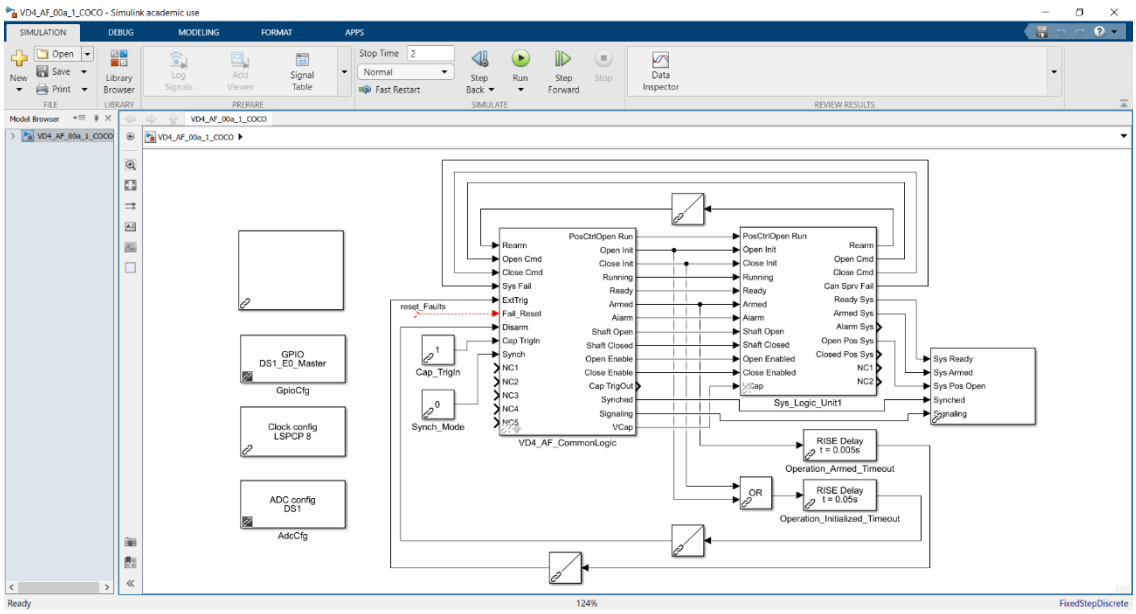

Figure 56 VD4_AF logic configuration.

6.1.1 Inputs for Simulation

Binary close and binary open inputs as shown in the figure 57, required for the system open and close operation when system is ready.

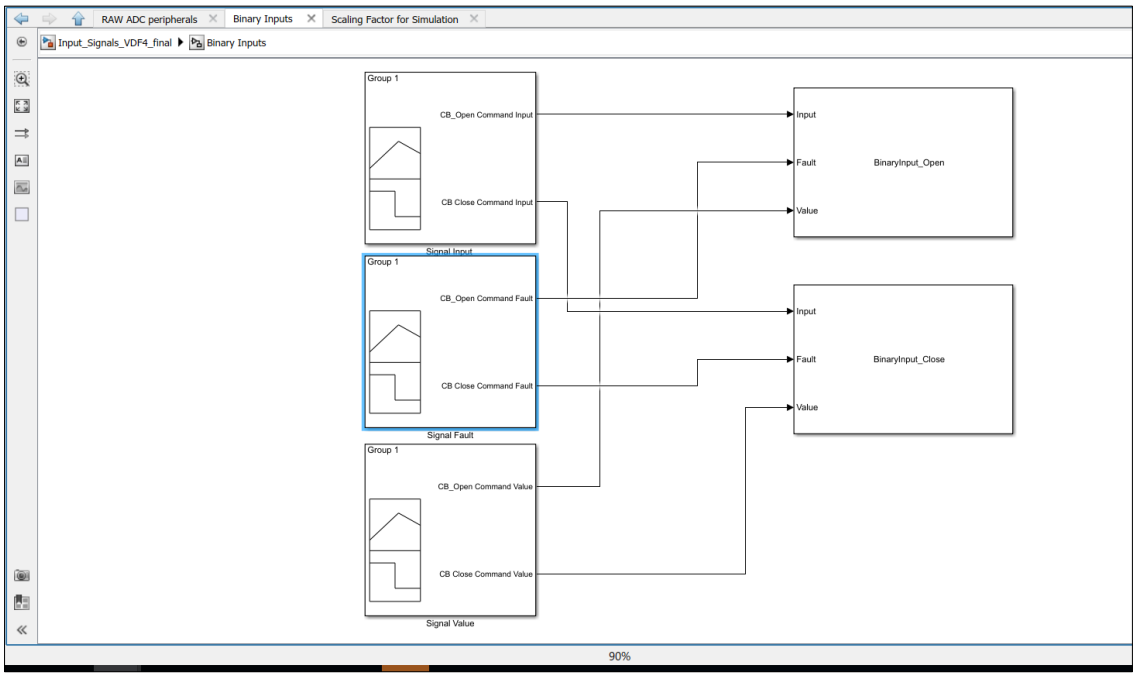

Figure 57 Binary open and close input.

ADC peripheral inputs for hardware monitor and diagnostic, current and voltage sensor inputs, capacitor voltage are shown in the figure 58.

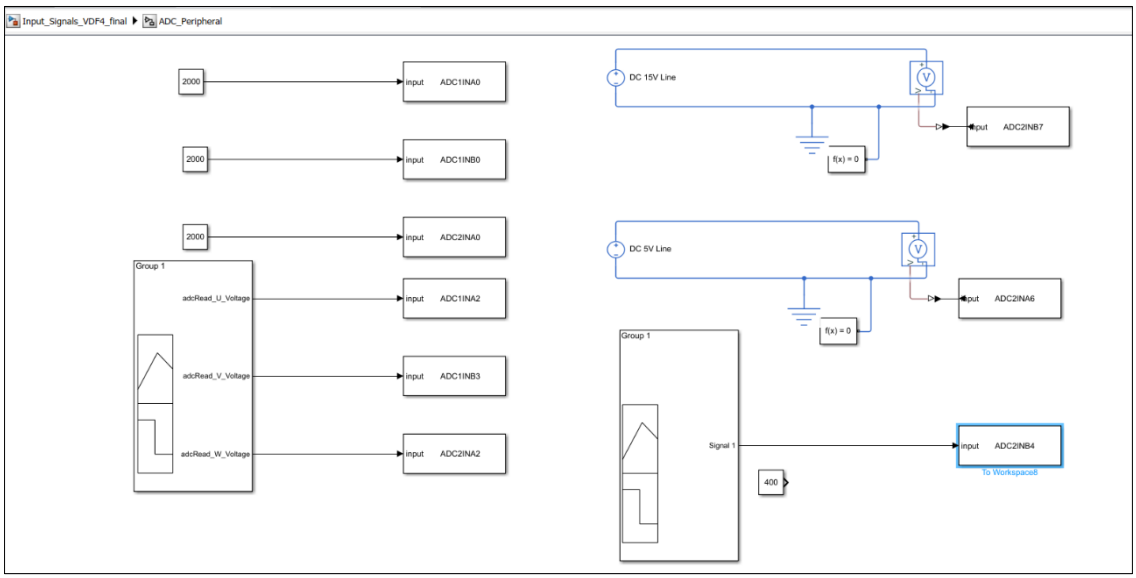

Figure 58 ADC inputs.

GPIO inputs for encoder and Synch blocks are shown in figure 59.

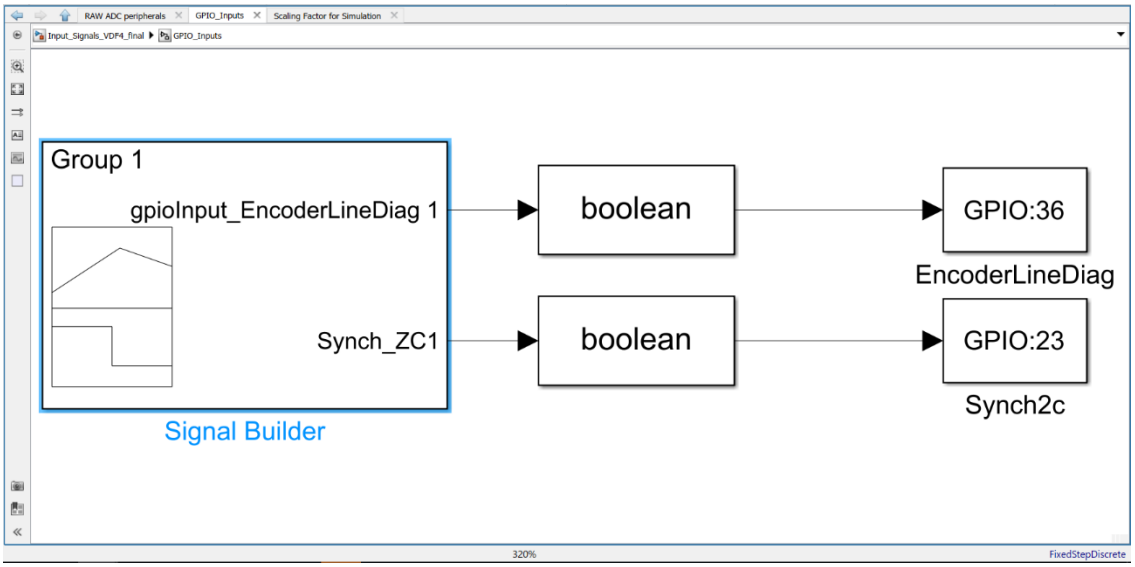

Figure 59 GPIO inputs.

Scaling factors for filter and delay blocks, and encoder flags which are generate by motor control blocks are shown in the following figure 60 and 61 respectively.

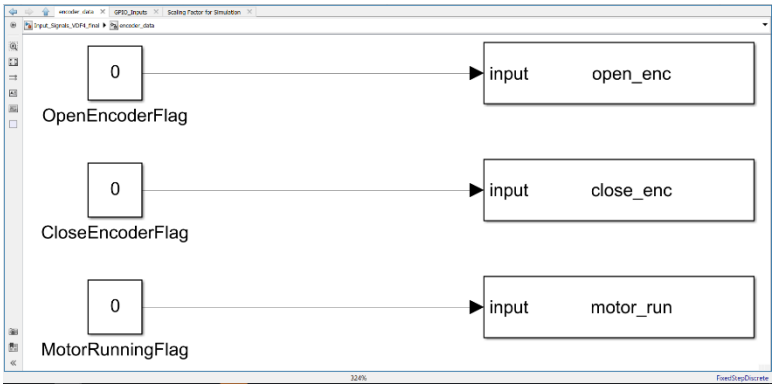

Figure 60 Encoder flags initialized to zero.

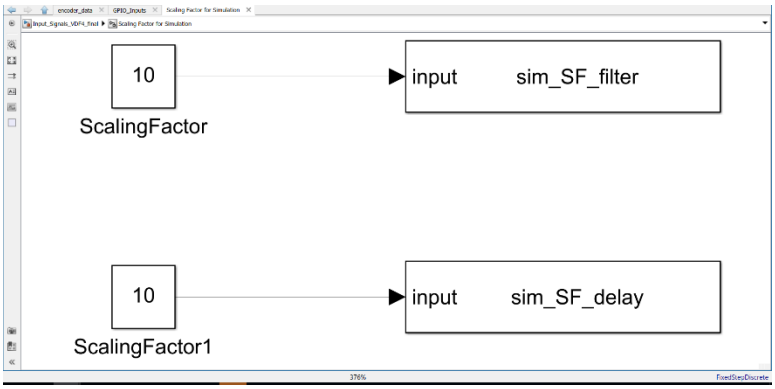

Figure 61 Filter and delay scaling factors.

CAN unit 2 and CAN unit 3 inputs are written to workspace as shown in the figure 62 so that the simulation CAN input block can take input from the workspace.

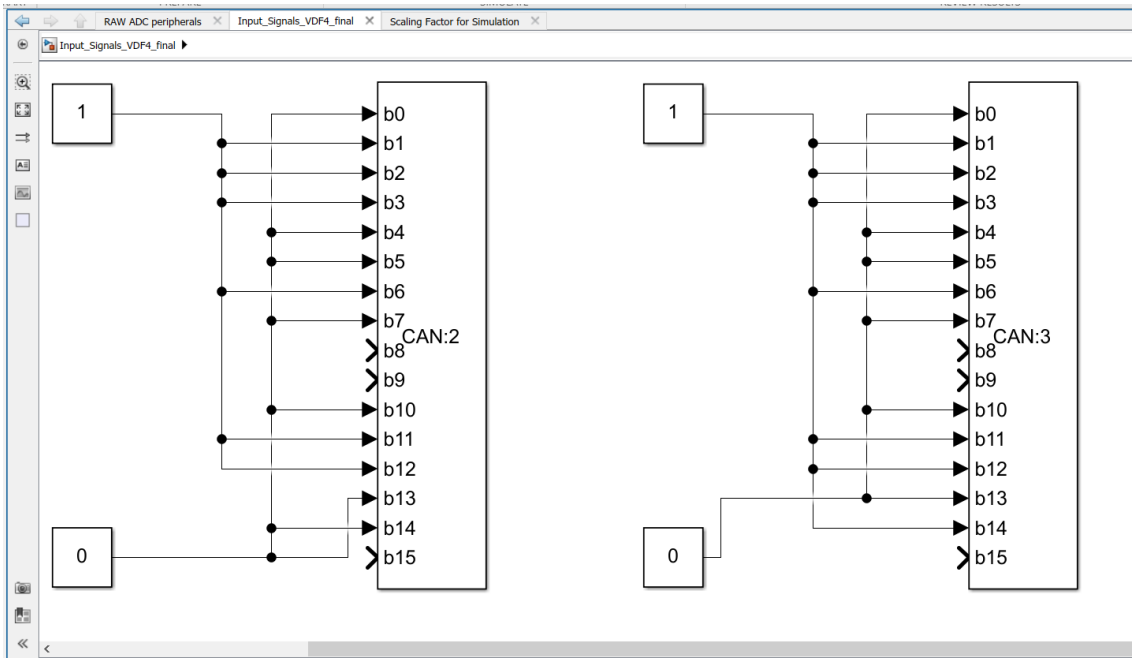

Figure 62 CAN unit 2 and Unit 3 inputs.

6.2 Test Case 1: Open Close test.

The binary inputs as shown in the figure 63, are required to test the simulator working, for this simulation the shaft position of the motor is initialized to close position.

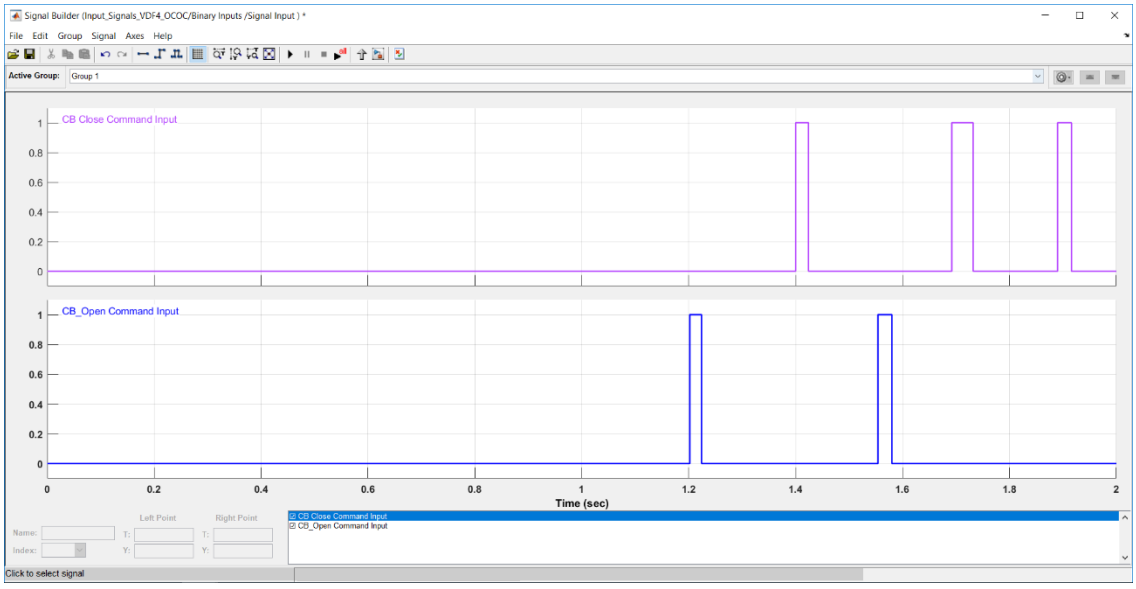

Figure 63 Binary Inputs for Open close test.

The first subplot in the figure 64 shows the binary input close and open received at the input blocks as shown in the figure 64, also the running time of motor as we can see from the 1st subplot the first open input is around 1.2 seconds and then the motor runs for 70 milliseconds and after these 70 milliseconds we can see from the subplot 2 that the motor encoder angle changes from close to open at an angle of 6.1 radians, also the motor close position in red line goes to 0. Again, from the subplot 1 we can see that close input is around 1.4 seconds and similarly the motor runs for 70 milliseconds, from subplot 2 we can see the motor encoder angle changes from open to close which is 3.2 radians and shaft position open goes to 0. This result indicates that simulator performance for closing and opening is normal.

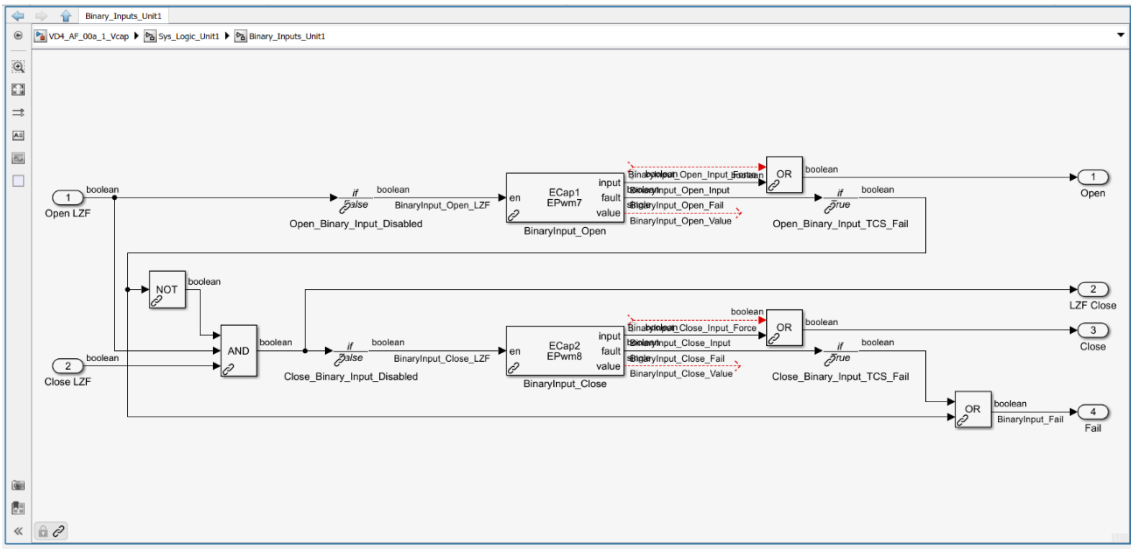

Figure 64 Binary Input open and binary input close blocks.

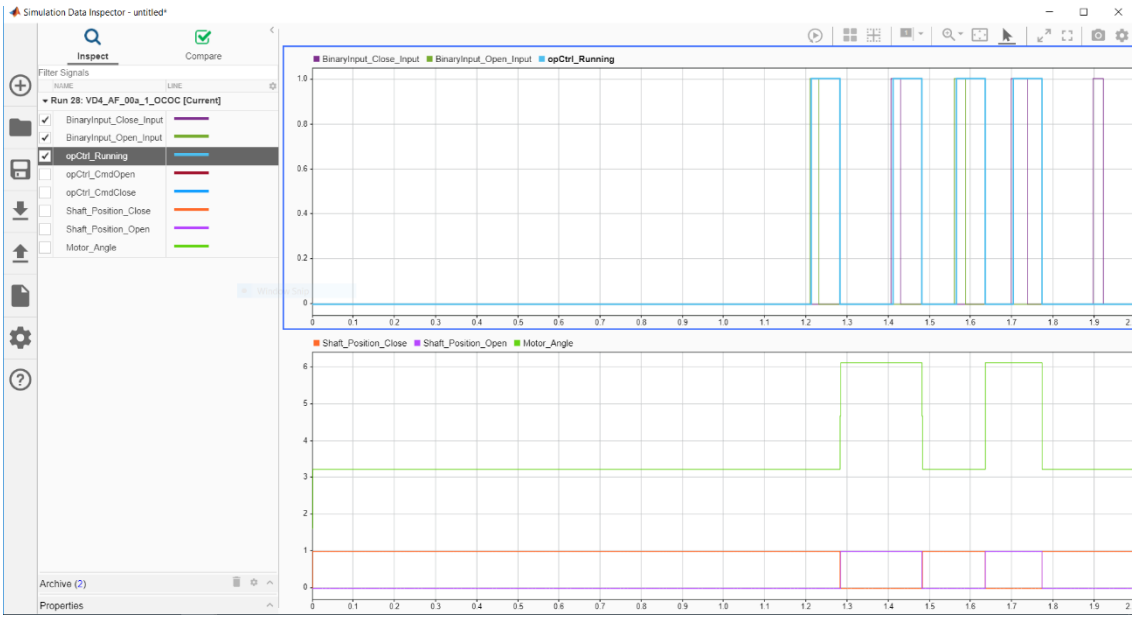

Figure 65 Open close operation simulation results.

6.3 Test Case 2: Close Open test.

The binary inputs as shown in the figure 65, are required to test the simulator working, for this simulation the shaft position of the motor is initialized to open position. From the subplot one we can see the first closing input is at 1.2 seconds, Followed by similar open close inputs.

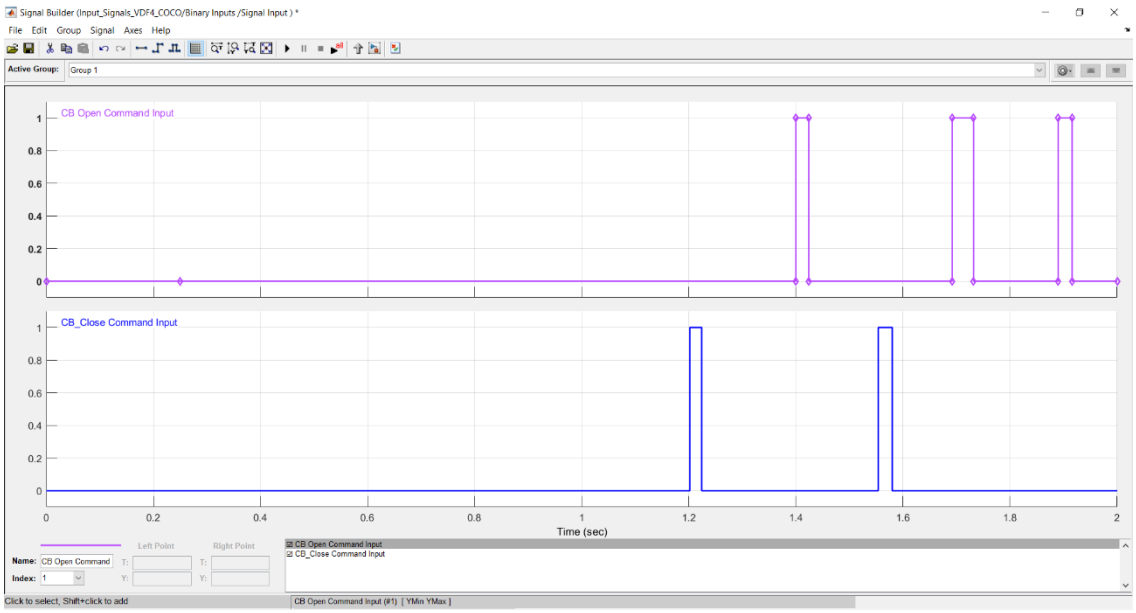

Figure 66 Binary Inputs for Close Open test.

The first subplot in the figure 66 shows the binary input close and open, also the running time of motor as we can see from the 1st subplot the first close input is around 1.2 seconds and then the motor runs for 70 milliseconds and after these 70 milliseconds we can see from the subplot 2 that the motor encoder angle changes from open to close at an angle of 3.2 radians, also the Shaft open position in red line goes to 0. Again, from the subplot 1 we can see that open input is around 1.4 seconds and similarly the motor runs for 70 milliseconds, from subplot 2 we can see the motor encoder angle changes from close to open which is 6.1 radians and shaft position close goes to 0. This result indicates that simulator performance for open and closing is normal.

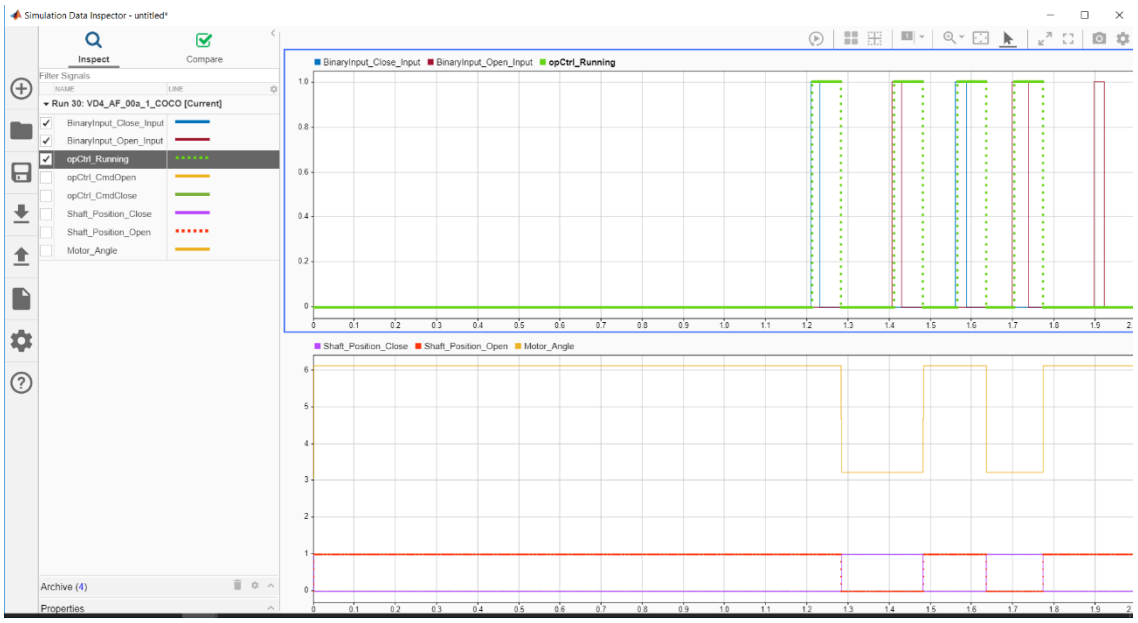

Figure 67 Close open operation simulation results.

6.4 Test Case 3: Capacitor Voltage test.

This test is performed to check the self-open functionality of VD4-AF, the logic block diagram off this functionality is depicted in figure 67, As it is clear from the figure that is the voltage of capacitor goes above 360volts the self-open is rearmed, then if the system is ready and voltage goes below 300volts, the self-open command is given to the motor control.

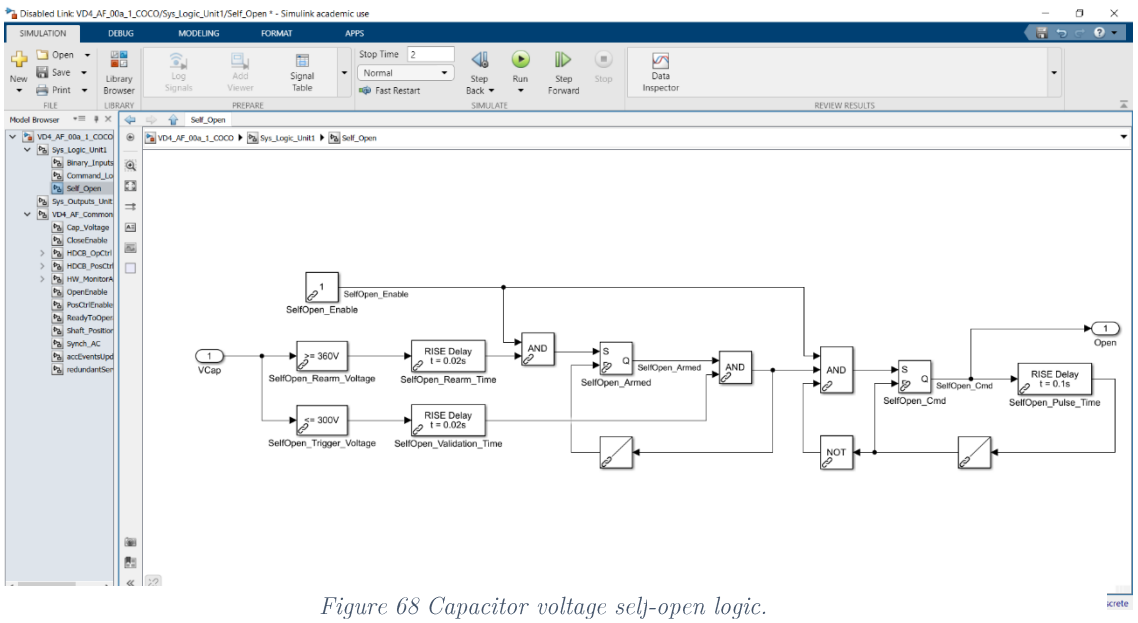

Figure 68 Capacitor voltage self-open logic.

A ramp input is given to the system to check this functionality, voltage is constant i.e. 400volts till 1.2s after that voltage starts to go down as shown in the figure 68.

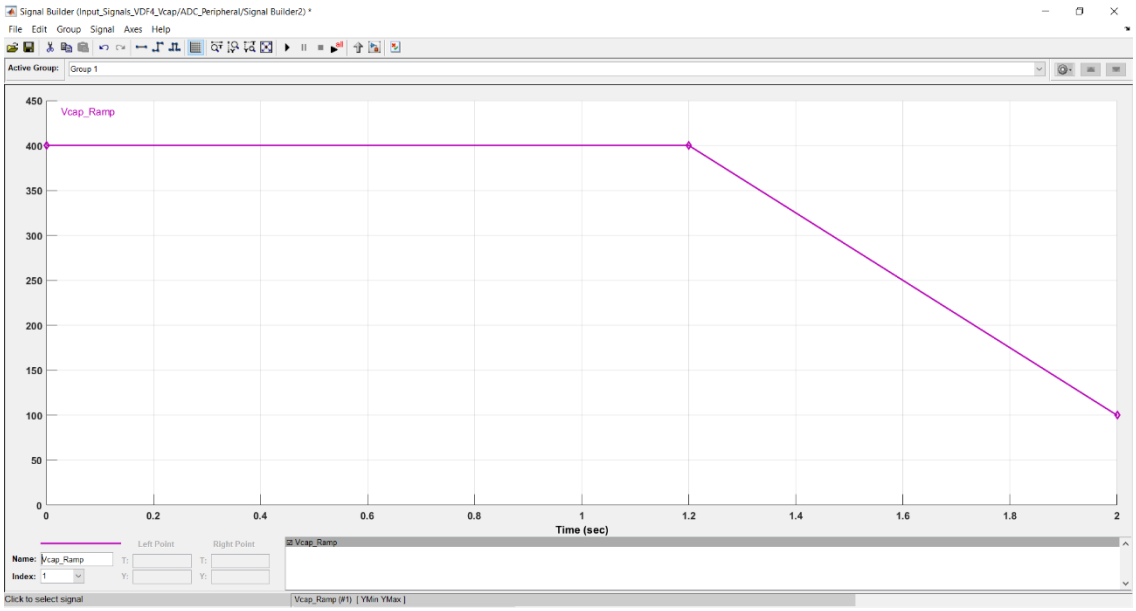

Figure 69 Capacitor Voltage ramp input.

The first subplot in the figure 70 shows the binary capaciotr voltage received at ADC2INB4 (figure 69), also the self-open command to the motor control block in the subplot at time 1.47s is highlted, as we can see from the 2nd subplot after the self-open command, the motor control block runs for 70 milliseconds and after these 70 milliseconds we can see from the subplot 2 that the motor encoder angle changes from close to open at an angle of 6.1 radians, also the Shaft open position in green line goes to 1.

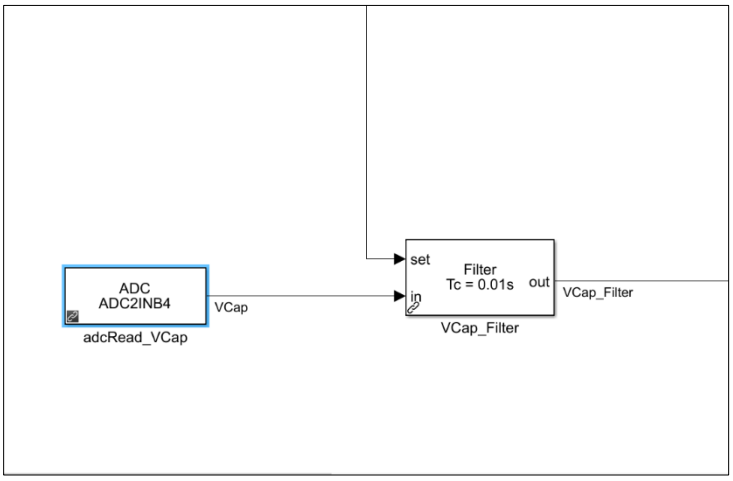

Figure 70 ADC2INB4 Vcap input block.

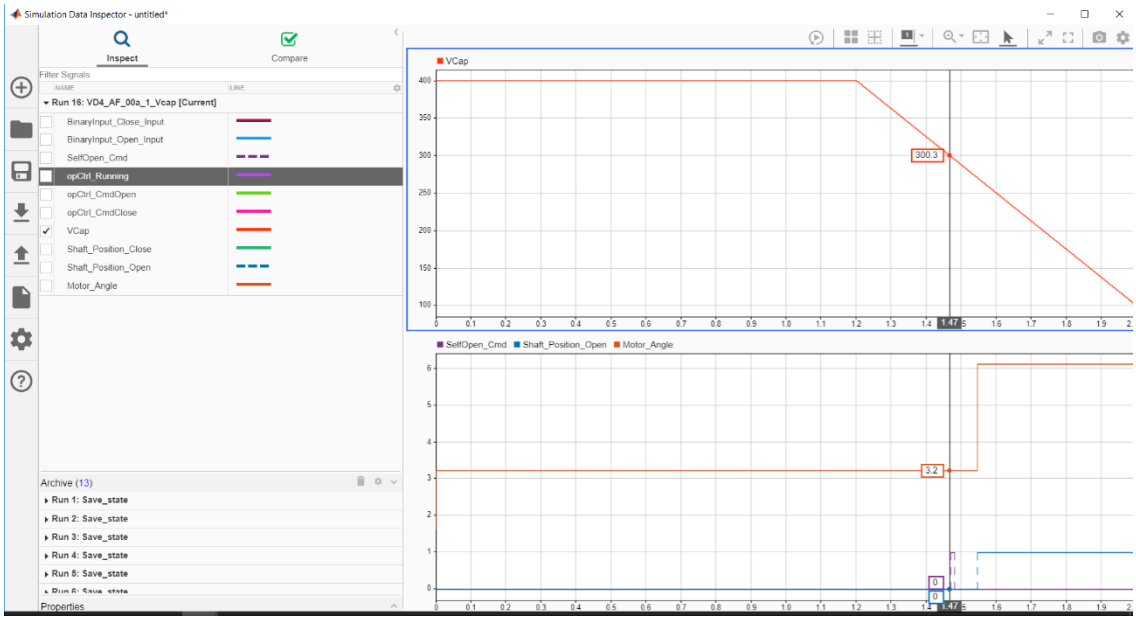

Figure 71 Capacitor Voltage ramp simulation results.

6.5 Test Case 4: Multiple Diagnostics Retries.

This test is performed to check the motor and capacitor diagnostic functionality of VD4-AF, the logic block diagram off this functionality is depicted in figure 72, As it is clear from the figure that is the MotorLoadDiagnostic block retry for diagnostics every 36s if the diagnostic has been passed but if the first diagnostic fails, then it retries after 6s, and the counter MotorLoadDiagnostic_Retries_fail is counting the number of failed attempts as in this case is 5, if the counter counts to 5 it will raise the flag MotorDriverDiagnostic_Retries_LimitReached.

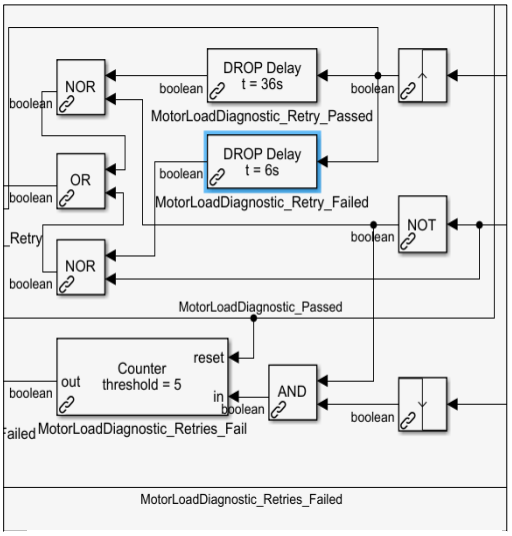

Figure 72 Diagnostic retry mechanism, with filter time constant of 6s.

Since, we are running these simulations for 2s the counter doesn't go up to five so after first failed attempt because it retries after 6s, so to check this mechanism the filter time constant is changed to 1s as shown in the figure 73 and simulation is running for 12 seconds.

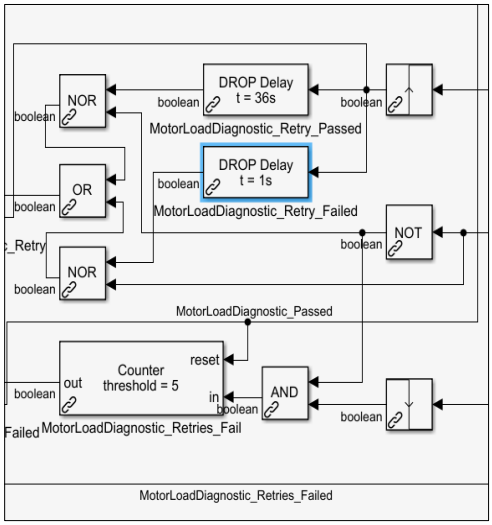

Figure 73 Diagnostic retry mechanism, with filter time constant of 1s.

To fail the diagnostic deliberately two simulation parameters are added to the mask of the diagnostic block, since pass condition are already defined in the mask, we can put fake values using these tabs, these values are assigned to the actual measured parameters in the firmware and after the end of running state the firmware fails the diagnostics. From the left block mask in the figure 74 we can see the pass condition as [1 5] and [3 15]*e-3, in figure 75 fake value for x1 parameter is given i.e. 6.

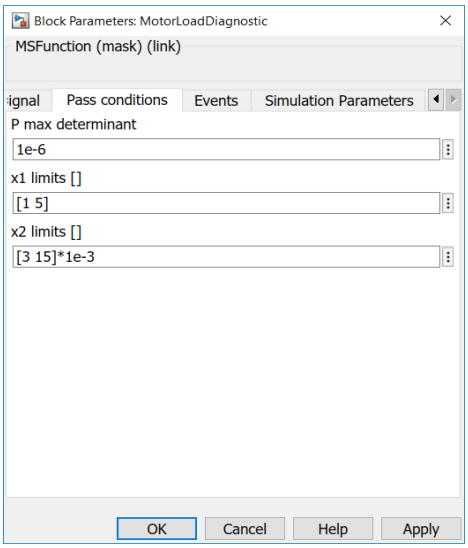

Figure 74 Pass conditions for diagnostic block.

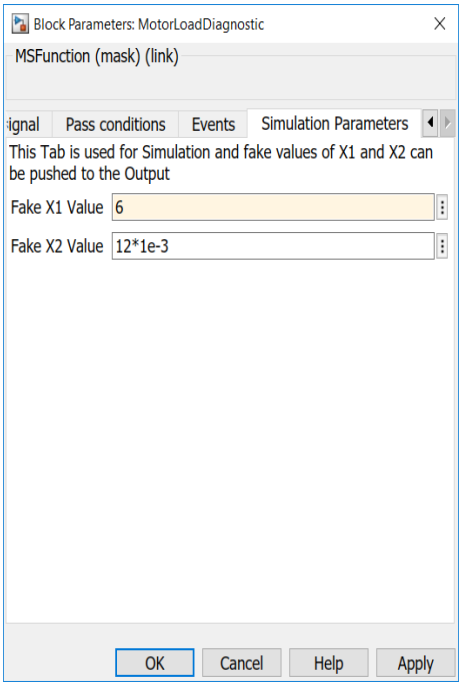

Figure 75 Simulation parameter for diagnostic block.

From the figure 76, in subplot one we can the see that the MotorLoadDiagnostic block first tries the diagnostic at .9s, from subplot two we can see that after 70ms the MotorLoadDiagnostic_Failed signal goes up. And after that every 1s, block is retrying and after 5 failed attempts, MotorLoadDiagnostic_Retries_fail signal goes up around 5.1s.

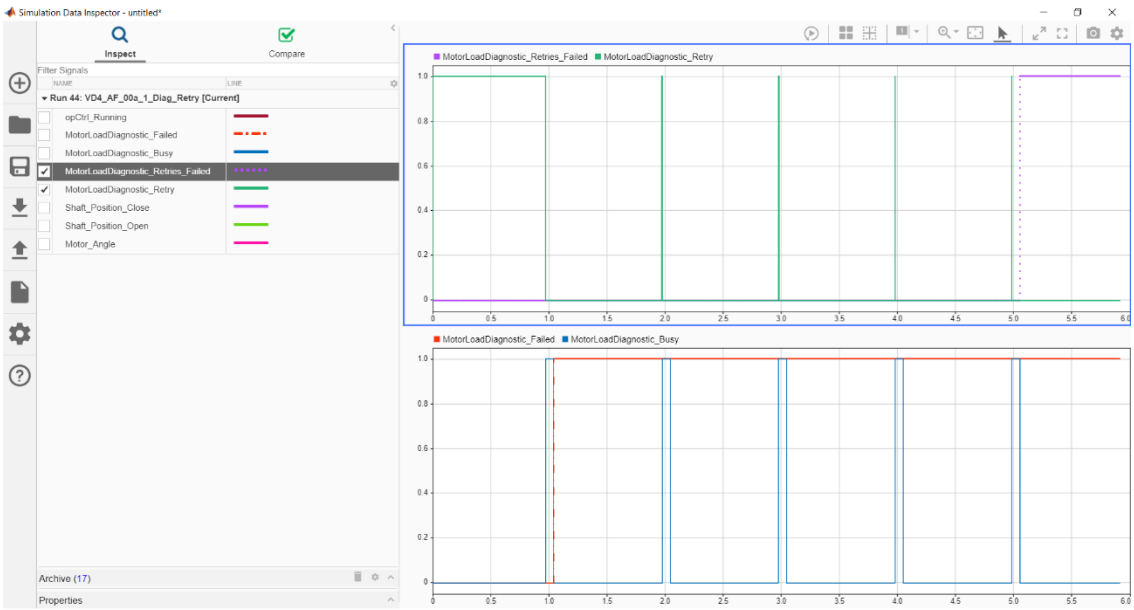

Figure 76 Motor Diagnostic block simulation results.

6.6 Test Case 5: CAN communication failure test.

To check CAN communication two tests have been performed, 1st one is CAN supervisory system, 2nd one is CAN communication failure form other two units.

The CAN Supervisor System block as shown in the figure 77 returns the status of the CAN Supervisor module, i.e. the software module that oversees the CAN communication functionalities and diagnostics. If a failure occurred in CAN communication with other ACU modules, the CAN Supervisor System block output is 0.

Output	Type	Description
out	D	The operational status of the CAN Supervisor module: • 1: CAN communication is operative; • 0: CAN communication failure.

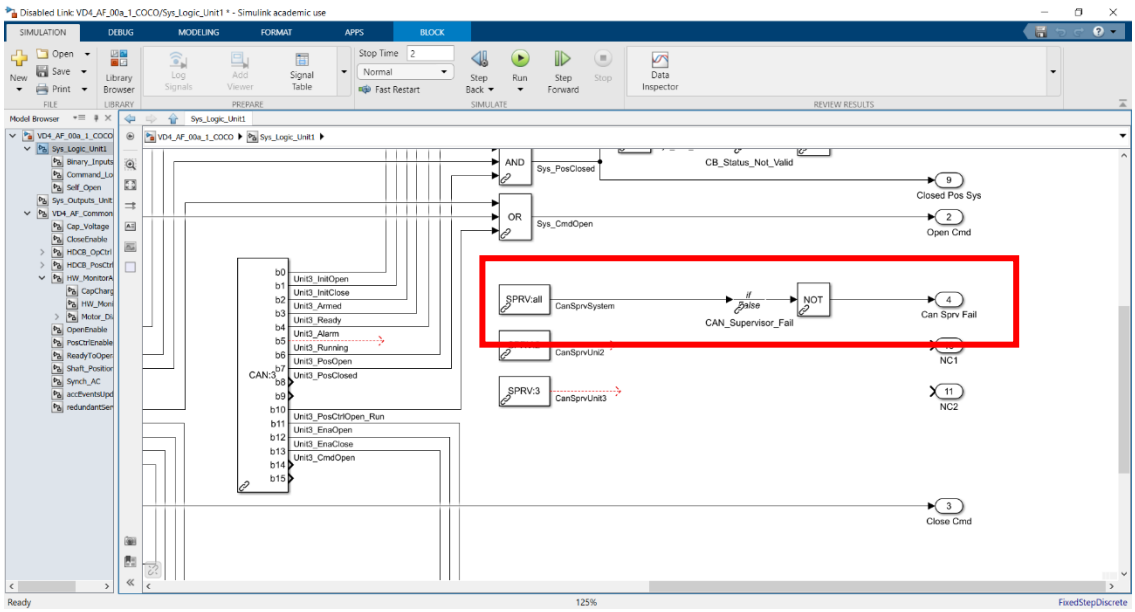

Figure 77 CAN Supervisor system block.

Figure 78 shows the results of CAN Supervisor system block simulation, 1st subplot shows the Close_Enable flag goes to true after the diagnostic has ran at 1.1s which means that system is enabled to close, similarly for the Open_Enable flag from subplot 2, at 1.5s CAN Supervisor system block output goes to false and system Open_Enable and Close_Enable both goes to zero. System will not act upon the open and close triggers now.

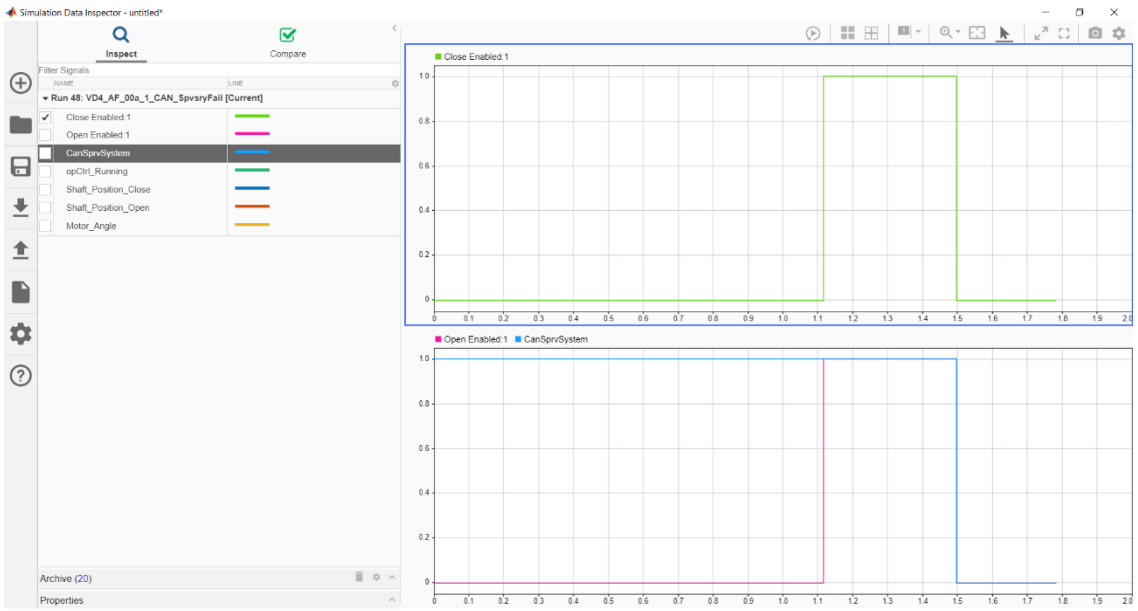

Figure 78 CAN Supervisor system block simulation results.

Figure 79 shows the results of CAN unit 2 and unit 3 failure, CAN Unit2 EnaOpen flag goes to zero for a brief moment at 1.4s and simultaneously unit 1 BinaryInput_Open_LZF flag goes down. CAN Unit3_ EnaClose flag goes to zero for a brief moment at 1.7s as shown in the subplot 4 and simultaneously unit 1 BinaryInput_Close_LZF flag goes down, Now we can see from the results that when Unit2 EnaOpen goes down it affects both BinaryInput_Open_LZF and BinaryInput_Close_LZF of unit 1 but when Unit3_ EnaClose goes down it only affects BinaryInput_Close_LZF but not the BinaryInput_Open_LZF which is still high, that means closing is more restricted operation.

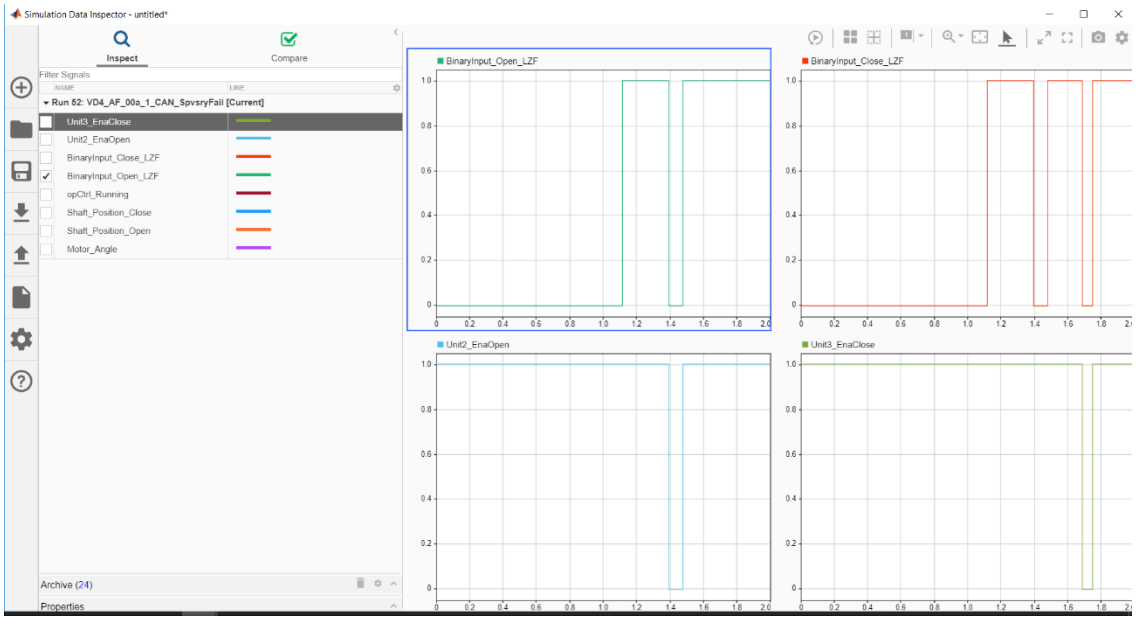

Figure 79 CAN Unit 2 and Unit 3 block simulation results.

6.7 Test Case 6: Absolute Encoder angle mismatch with Servo Motor Incremental encoder.

This test is performed to check the motor position control functionality of VD4-AF, the logic block diagram off this functionality is depicted in figure 80, As it is clear from the figure that ServoMotor block has input for Absolute encoder angle, if the Servo Motor's incremental encoder angle and Absolute Encoder angle mismatch as shown in the figure 81 the redundantServo.Fail flag is generated.

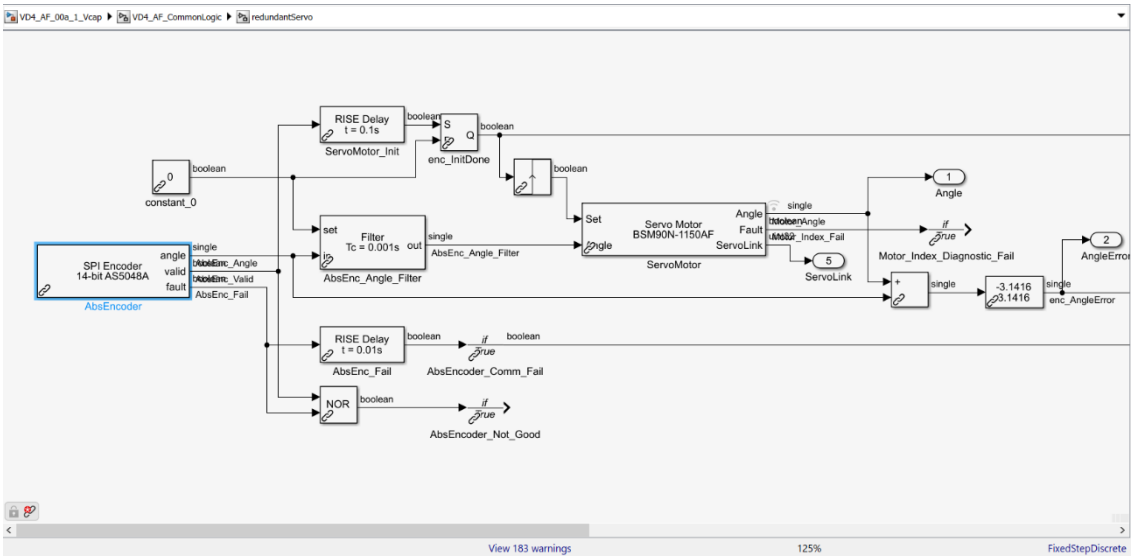

Figure 80 Absolute Encoder and Servo Motor angle logic.

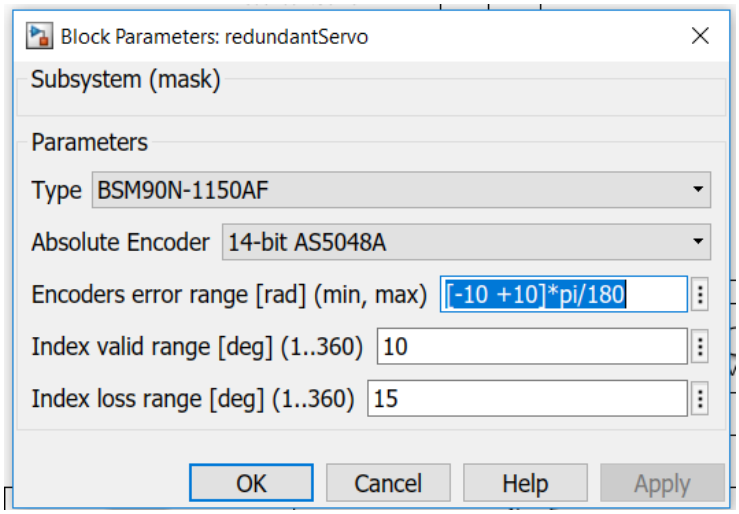

Figure 81 Encoders error range.

To test redundantServo.Fail flag, shaft is started from close position and an binary open command is given at 1.2s as shown in the figure 82. The mask simulation parameter

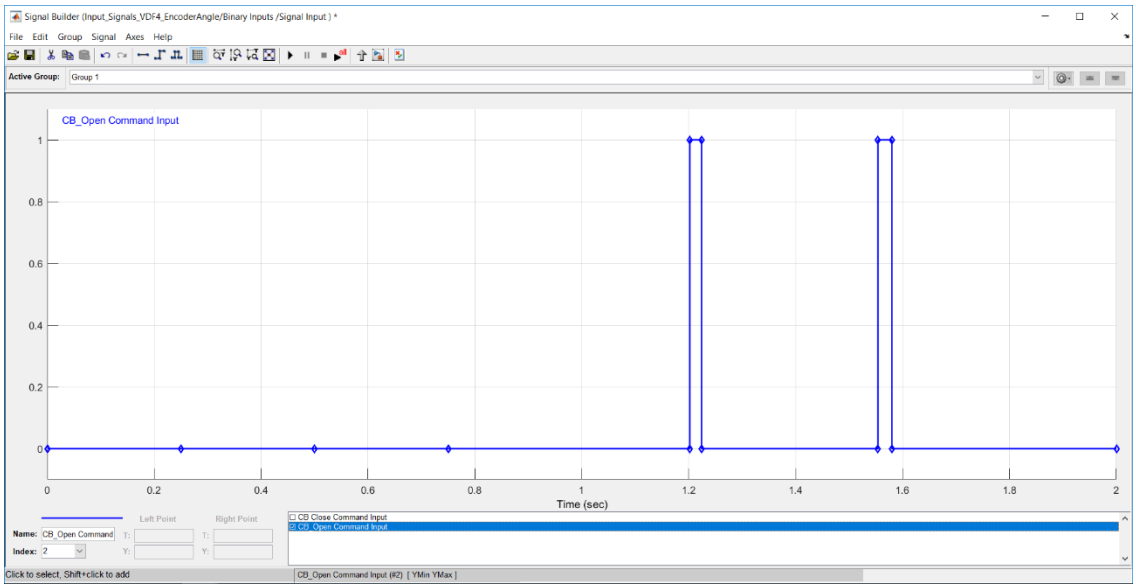

Figure 82 Binary Open Input for Encoder angle mismatch test.

is also added to servo motor block mask with an error of 2 radians shown the figure 83, this error starts when motor control block (MotorControl_OpCtrl) runs the motor from close to open.

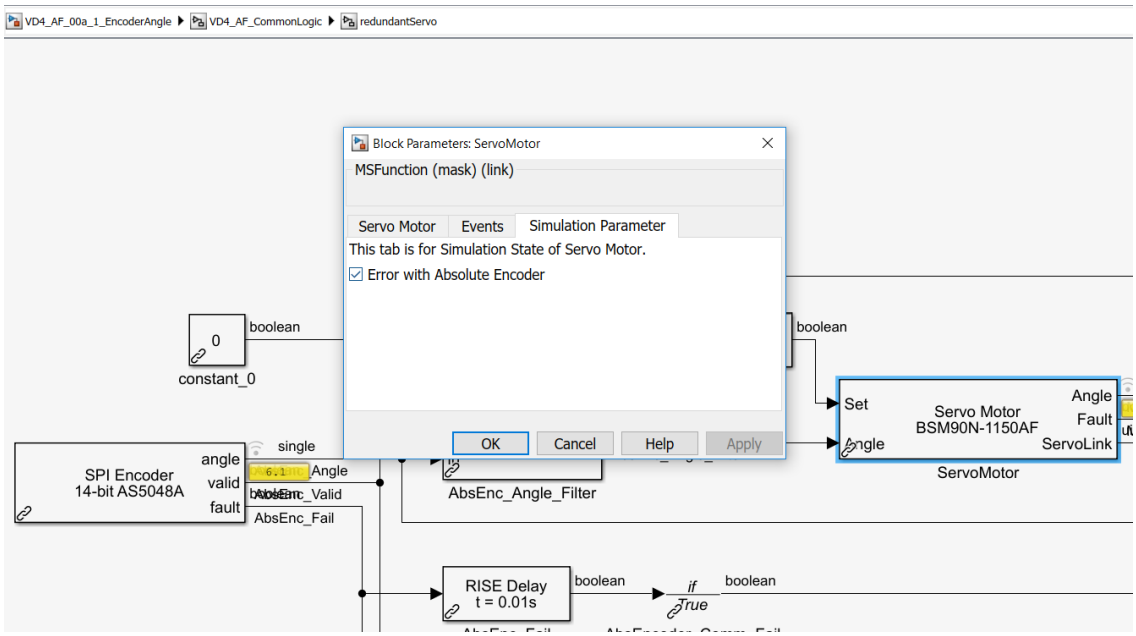

Figure 83 Servo Motor mask.

The result of the simulation with encoder angles mismatch are shown in the figure 84, subplot one sows the angle of Absolute encoder and servo motor at 3.2radians (Motor shaft close position), after the binary open input at 1.2 seconds a constant error of 2 radians is also visible, the motor control block (MotorControl_OpCtrl) runs for 70ms as shown in subplot 2, the redundantServo.Fail flag goes up at 1.38s we can see from subplot 1. The system is not ready after the redundantServo.Fail flag goes up, as shown in subplot 3.

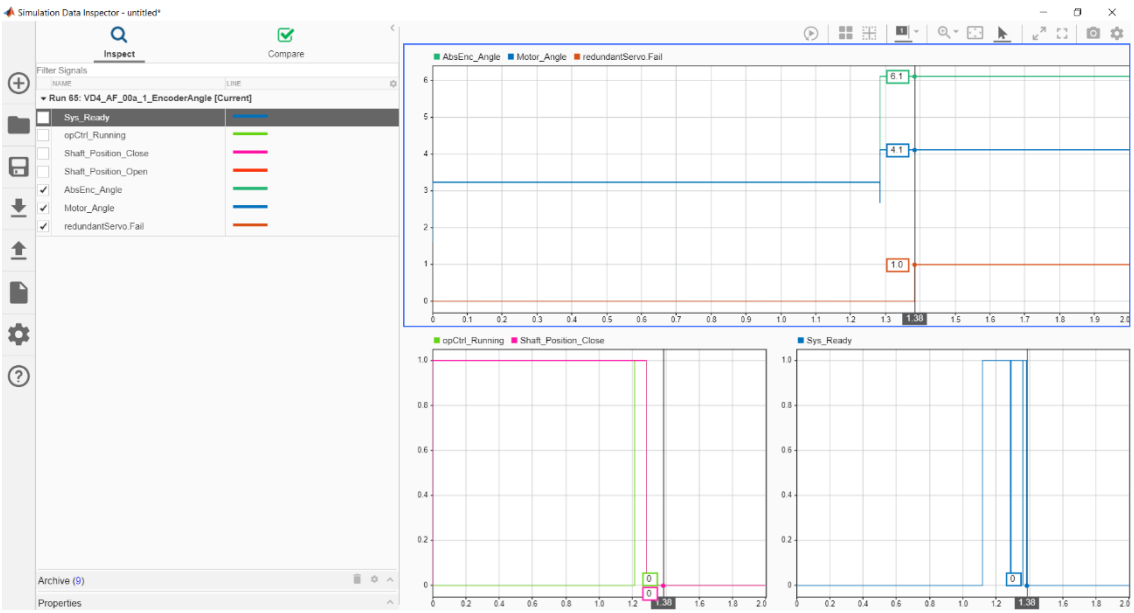

Figure 84 Encoder angle mismatch simulation results.

6.8 Test Case 7: Low Capacitor Voltage, Close Open enable test.

This test is performed to check the low capacitor voltage functionality of VD4-AF, the logic block diagram off this functionality is depicted in figure 85, As it is clear from the figure that Close_enable is low when the voltage is below 330volts, Similarly if the voltage is below 270volts, Open_enable is low. To test this logic Capacitor voltage is given as shown in the figure 85, and the binary open and binary close commands as shown in the figure 86.

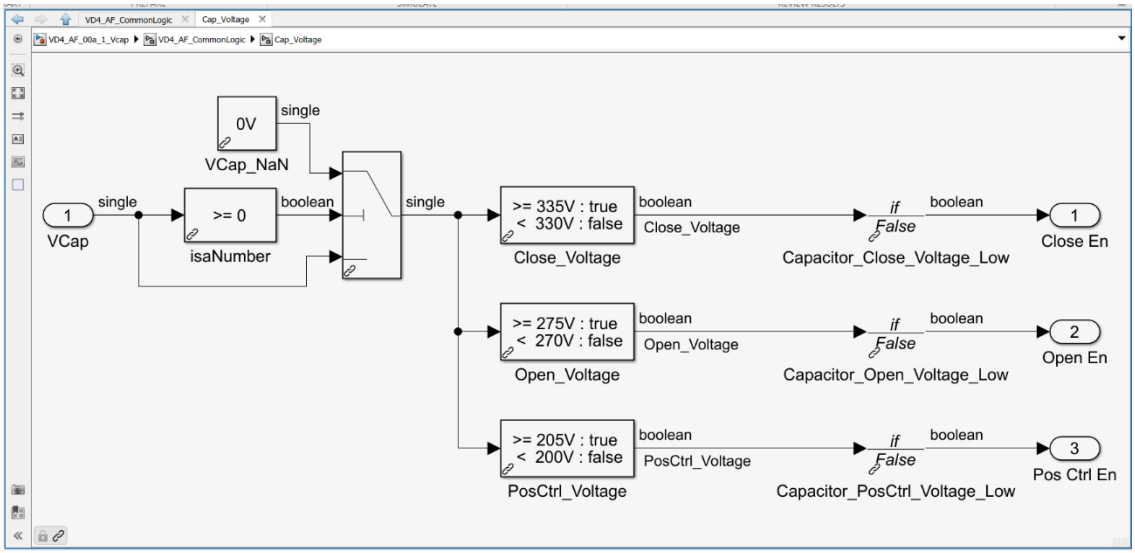

Figure 85 Low Capacitor Voltage Logic.

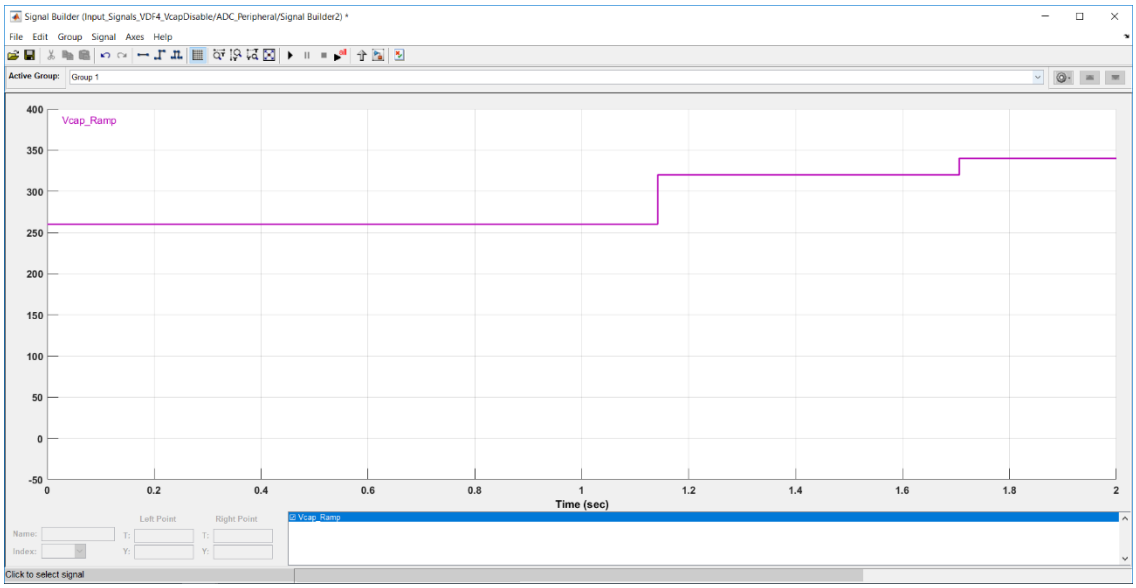

Figure 86 Capacitor Voltage input.

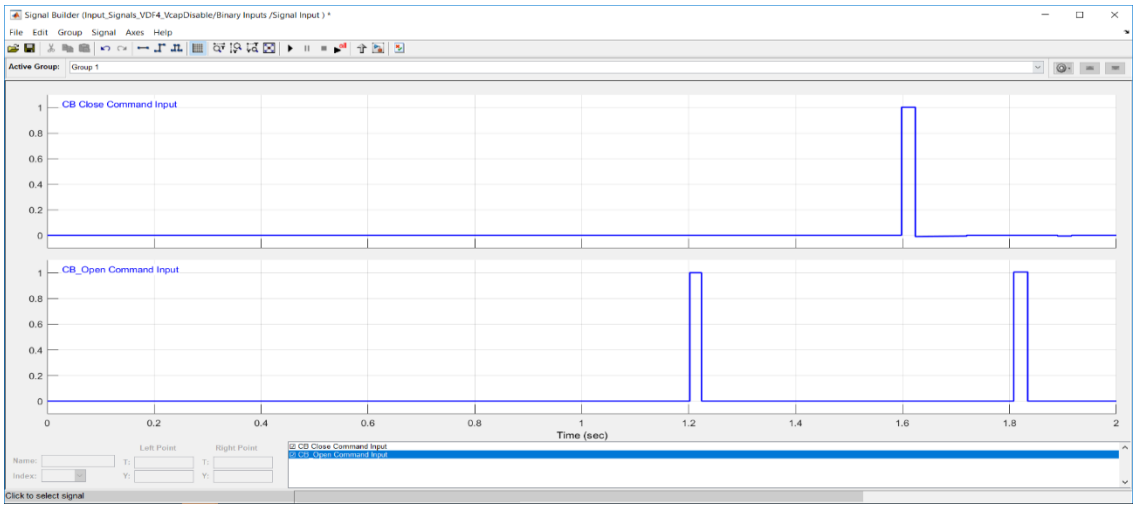

Figure 87 Binary Open and binary close inputs.

The result of the simulation with low capacitor voltage is shown in the figure 88, subplot one shows the capacitor input voltage, after the Vcap goes to 300volts the Open_voltage is high as we can see from the subplot 2 at 1.1 seconds, after Vcap goes to 340 volts, Close_voltage is high as we can see from the subplot 2 at 1.7 seconds.

Since the system is not ready because of the low capacitor voltage the Binary open and binary close inputs doesn't operate the breaker.

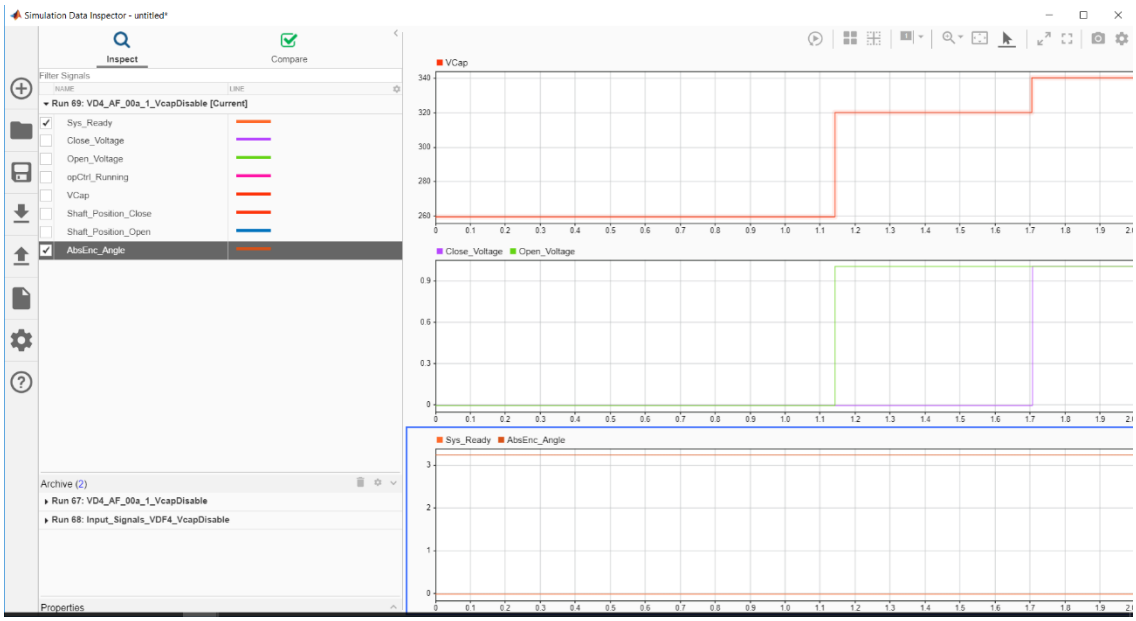

Figure 88 Low capacitor voltage test results.

Conclusions

Outcome of the activity described in this work is the Development of Simulator of Logic Control Blocks for Electromagnetic Actuators for Medium Voltage apparatus, simulator was test on different product configuration. The simulator has been designed and successfully tested. Advanced features of our design include:

- Graphical User interface for simulation mode selection, output saving choice for simulation performance improvements.
- Saving the instance of simulation to start from previous state.

A concept system was built and tested using a Software in loop test on ABB steel furnace breaker VD4-AF, simulator was tested for multiple test cases and the simulator were verified according to the logic. This simulator gives us the ability to perform preliminary testing on the development of new product, it can also serve as the first test of hardware.

The simulator gives the incentive of development of new generation of smart products as the designing and development becomes easier with the simulation process, the development of hardware and software can go in parallel with this tool in hand.

Appendix 1

Select_output script:

```
% function Select_output
clear handles
clear str_list s_max s_min handles;
lineOption = {'-.','-','--','.',':'};

POLLING_ON_GCB = 1;
POLLING_ON_SELECTED = 2;
pollingState = POLLING_ON_SELECTED;

%% Input Dialog: select the number of plot in the figure
prompt = {'Enter number of blocks you want to save output for:'};
dlgtitle = 'Select blocks';
dims = [1 35];
definput = {'5'};
options.Resize='on';
answer = inputdlg(prompt,dlgtitle,dims,definput,options);

%% Deselect current block
if(strcmp(get_param(gcb,'Selected'),'on')) %clears any selected block gcb gives
name of current block; 'selected' is common block properties
    set_param(gcbh,'Selected','off');
end

%% Block selection
if(~isempty(answer))

    gcb_old = gcb; %gcb old=vacp filter
    gcs_old = gcs; %main system
    i = 1;

    while (i<=str2num(answer{1})) % i 1-5
        pause(0.1);      % with the pause, it is possible to select a block in
the meanwhile, and read the update gcb
        gcb_temp = gcb;
        gcs_temp = gcs;

        switch pollingState
            case POLLING_ON_GCB
                if (~strcmp(gcb_old,gcb_temp))  % new block slected

                    if(strcmp(gcs_old,gcs_temp))
                        % still in the same subsystem
                        if(strcmp(get_param(gcbh,'BlockType'),'M-S-Function'))
% detect elementary blocks and not susbsystem: a gcb is returned also for
subsystem. To be discarded
                            str_list{i} = strcat(strrep(gcb_temp,'/','.'));
                            gcb_old = gcb_temp;
                            handles{i}=gcbh;
                            disp(['Selected block: ',str_list{i}]);
                            disp(['Selected    block    unique    handle:    ',
handles(i)]);
```

```
                            i = i+1;
                        end
                    else
                        % Subsystem is changed
                        set_param(gcbh,'Selected','off');
                        pollingState = POLLING_ON_SELECTED;
                        gcs_old = gcs;
                    end
                end

            case POLLING_ON_SELECTED
                if(strcmp(get_param(gcb,'Selected'),'on'))
                    % New block selected

                    if(strcmp(gcs_old,gcs_temp))
                        % still in the same subsystem
                        if(strcmp(get_param(gcbh,'BlockType'),'M-S-Function'))
% detect elementary blocks and not susbsystem: a gcb is returned also for
subsystem. To be discarded
                                str_list{i}                                    =
strcat(strrep(gcb_temp,'/','.'));
                                gcb_old = gcb_temp;
                                pollingState = POLLING_ON_GCB;
                                  handles{i}=gcbh ;
                                disp(['Selected block: ',str_list{i}]);
                                disp(['Selected  block  unique  handle:  ',
handles(i)]);

                                i = i+1;
                        end
                    else
                        % Subsystem is changed
                        set_param(gcbh,'Selected','off');
                        pollingState = POLLING_ON_SELECTED;
                        gcs_old = gcs;
                    end
                end
        end
    end

    clear POLLING_ON_GCB POLLING_ON_SELECTED str_list gcs_temp gcs_old gcb_temp
pollingState prompt dlgtitle dims answer  i lineOption definput options gcb_old
end
```

Initialization model callback code:

```
%% Template for new init callback for Simulink test harness models

clear simulation;
global simulation;
global Bypass_gui;

% GUI callback, Bypass_gui bypass GUI callback if it is ruuning from
simulink test manager
if (Bypass_gui ~= 1)
    if ((exist('simulation','var')==1 && exist('myvar')==0 ))
        myvar = GUI_ACU_1_exported;
%   myvar is new GUI object
        while (myvar.break_1 ~=1)
%   If run button is pressed myvar.break_1 goes to 1 and loop breaks
and simulation starts
            pause(.01);
        end
```

```
    end
else
%   If bypassed from simulink test
    simulation = 1;
    output_choice = 0;
%   output_choice variable if GUI bypassed, 0 is ignoring all the
outputs
    state_choice = 0;
%   state_choice  variable if GUI bypassed, 0 is ignoring not saving
any end state
end

if ((exist('simulation','var')==1) & simulation == 1)
    disp('simulation is starting now');
    disp(simulation)

    clear global BUSY_STATUS
    clear Events_data
    clear Events_log
    clear Events_table
    global BUSY_STATUS;
else
    disp('Model Callback: InitFcn');
    model_init;
    productId    = uint32(1);
    numCtrlUnit = uint32(1);
    configType  = uint32(0);
    sysName     = 'Troubleshooting';
end
```

Mex code Template:

```
//%#include <stdio.h>
#include "mex.h"
    //C:\Projects\ACU\Release_1.0\C28_Control\Source\Include
    // includes the logic file where functions are declared
    #include "pl_logic.h"
    #include "pl_controls.h"
    #include "pl_motor_diag.h"
    #include "matrix.h"
    //*********MEMBERS COUNTER BLOCKS ARE DECLARED
HERE******************

    PL_SECTION           pl;                      //member of type
PL_SECTION is defined
    PL_MOTOR_DIAG_BLOCK  pBlock;                 //member of type
PL_MOTOR_DIAG_BLOCK is defined
    Uint16 synchTrigger;
    Uint16 GPIOTrigger;

    //**********MEX FUNCTION IS DEFINED HERE**********************

    void mexFunction( int nlhs, mxArray *plhs[],
            int nrhs, const mxArray *prhs[])
    {
        //Input Block Variables
        bool di_Trig;
        bool di_Interlock;
        bool di_StopCtrl;
```

```
        LK_VAR li_ServoLink;

        //Output Block Variables
        bool do_passed;
        bool do_failed;
        bool do_busy;
        bool do_trig;
        bool do_Armed;
        bool do_Running;

        //Local Variables
        float local_pBlock_lv_Start             ;
        float local_pBlock_lv_FailCnt           ;
        float local_commons_lv_Interrupted      ;
        float local_commons_lv_Synch            ;
        float local_commons_lv_ActiveOperation  ;

        float control_header_ticker             ;
        float control_header_state              ;
        float control_header_lastStep           ;

        int init_status;

        //Outputs are defined here
        double *out0,*out1,*out2,*out3,*out4,*out5;
// Physical outputs of Block
        double
*lv_out0,*lv_out1,*lv_out2,*lv_out3,*lv_out4,*lv_out5,*lv_out6,*lv_out
7,*lv_out8;      // local variable

        //input Values from Sfunction
        di_Trig                 =       (bool)mxGetScalar(prhs[0]);
        di_Interlock            =       (bool)mxGetScalar(prhs[1]);
        di_StopCtrl             =       (bool)mxGetScalar(prhs[2]);
        li_ServoLink            =       (float)0;

        //Output Values from Sfunction (Since all outputs are not
updating at every iteration)
        do_passed       =       (bool)mxGetScalar(prhs[4]);
        do_failed       =       (bool)mxGetScalar(prhs[5]);
        do_busy         =       (bool)mxGetScalar(prhs[6]);
        do_trig         =       (bool)mxGetScalar(prhs[7]);
        do_Armed        =       (bool)mxGetScalar(prhs[8]);
        do_Running      =       (bool)mxGetScalar(prhs[9]);

        //Local Variables updated values from Sfunction
        local_pBlock_lv_Start               =
(float)mxGetScalar(prhs[10]);
        local_pBlock_lv_FailCnt             =
(float)mxGetScalar(prhs[11]);
        local_commons_lv_Interrupted        =
(float)mxGetScalar(prhs[12]);
        local_commons_lv_Synch              =
(float)mxGetScalar(prhs[13]);
        local_commons_lv_ActiveOperation    =
(float)mxGetScalar(prhs[14]);
        control_header_ticker               =
(float)mxGetScalar(prhs[15]);
        control_header_state                =
(float)mxGetScalar(prhs[16]);
```

```
        control_header_lastStep             =
(float)mxGetScalar(prhs[17]);
        init_status                         =
(int)mxGetScalar(prhs[18]);

        synchTrigger = (Uint16)mxGetScalar(prhs[19]);
        /* GPIO input trigger: generated externally and read by the
motor control
         * function. It is not an input of the simulink block, but the
firmware
         * reads this trigger from a GPIO */
        GPIOTrigger = (Uint16)mxGetScalar(prhs[20]);


        /* check for proper number of arguments */
        if(nrhs!=21) {
            mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","21 inputs
required.");
        }
        if(nlhs!=15) {
            mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","15
outputs required.");
        }
        /* make sure the first input argument is scalar */

        // INPUTS INDEXES
        pBlock.commons.di_Trig                      =
(DG_INDEX)0;
        pBlock.commons.di_Interlock                 =
(DG_INDEX)1;
        pBlock.commons.di_StopCtrl                  =
(DG_INDEX)2;
        pBlock.li_ServoLink                         =
(AN_INDEX)0;

        // INPUT VALUES
        pl.dgVar[pBlock.commons.di_Trig]            =
(DG_VAR)di_Trig;
        pl.dgVar[pBlock.commons.di_Interlock]       =
(DG_VAR)di_Interlock;
        pl.dgVar[pBlock.commons.di_StopCtrl]        =
(DG_VAR)di_StopCtrl;
        pl.lkVar[pBlock.li_ServoLink]               =
(LK_VAR)li_ServoLink;


        // OUTPUT INDEXES
        pBlock.do_Passed                    = (DG_INDEX)3;
        pBlock.do_Failed                    = (DG_INDEX)4;
        pBlock.commons.do_Busy              = (DG_INDEX)5;
        pBlock.do_Trig                      = (DG_INDEX)6;
        pBlock.commons.do_Armed             = (DG_INDEX)7;
        pBlock.commons.do_Running           = (DG_INDEX)8;

        //OUTPUT Values
        pl.dgVar[pBlock.do_Passed]          = (DG_VAR)do_passed;
        pl.dgVar[pBlock.do_Failed]          = (DG_VAR)do_failed;
        pl.dgVar[pBlock.commons.do_Busy]    = (DG_VAR)do_busy;
        pl.dgVar[pBlock.do_Trig]            = (DG_VAR)do_trig;
        pl.dgVar[pBlock.commons.do_Armed]   = (DG_VAR)do_Armed;
        pl.dgVar[pBlock.commons.do_Running] = (DG_VAR)do_Running;
```

```
        // LOCAL VARIABLES - SAVED IN DWORK
        pBlock.lv_Start                     = local_pBlock_lv_Start;
        pBlock.lv_FailCnt                   = local_pBlock_lv_FailCnt;
        pBlock.commons.lv_Interrupted       =
local_commons_lv_Interrupted;
        pBlock.commons.lv_Synch             = local_commons_lv_Synch
;
        pBlock.commons.lv_ActiveOperation   =
local_commons_lv_ActiveOperation  ;

        pBlock.control.header.ticker        = control_header_ticker;
        pBlock.control.header.state         = control_header_state;
        pBlock.control.header.lastStep      = control_header_lastStep;

        // create the output matrix
        plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[1] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[2] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[3] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[4] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[5] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[6] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[7] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[8] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[9] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[10] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[11] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[12] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[13] = mxCreateDoubleMatrix(1,1,mxREAL);
        plhs[14] = mxCreateDoubleMatrix(1,1,mxREAL);
        // get a pointer to the real data in the output matrix
        out0 = mxGetPr(plhs[0]);
        out1 = mxGetPr(plhs[1]);
        out2 = mxGetPr(plhs[2]);
        out3 = mxGetPr(plhs[3]);
        out4 = mxGetPr(plhs[4]);
        out5 = mxGetPr(plhs[5]);

        // Local variables
        lv_out0  = mxGetPr(plhs[6]);
        lv_out1  = mxGetPr(plhs[7]);
        lv_out2  = mxGetPr(plhs[8]);
        lv_out3  = mxGetPr(plhs[9]);
        lv_out4  = mxGetPr(plhs[10]);
        lv_out5  = mxGetPr(plhs[11]);
        lv_out6  = mxGetPr(plhs[12]);
        lv_out7  = mxGetPr(plhs[13]);
        lv_out8  = mxGetPr(plhs[14]);

        if (!init_status)
        {
            motorDiagBlockInit(&pBlock);
            init_status = 1;
        }


        motorDiag_simulationCalbackInit(&pBlock);
        motorDiagBlockRun(&pBlock);             //Call to the block
function
```

```
    *out0  = pl.dgVar[pBlock.do_Passed];
    *out1  = pl.dgVar[pBlock.do_Failed];
    *out2  = pl.dgVar[pBlock.commons.do_Busy];
    *out3  = pl.dgVar[pBlock.do_Trig];
    *out4  = pl.dgVar[pBlock.commons.do_Armed];
    *out5  = pl.dgVar[pBlock.commons.do_Running];

    *lv_out0 = pBlock.lv_Start                      ;
    *lv_out1 = pBlock.lv_FailCnt                    ;
    *lv_out2 = pBlock.commons.lv_Interrupted        ;
    *lv_out3 = pBlock.commons.lv_Synch              ;
    *lv_out4 = pBlock.commons.lv_ActiveOperation    ;

    *lv_out5 = pBlock.control.header.ticker         ;
    *lv_out6 = pBlock.control.header.state          ;
    *lv_out7 = pBlock.control.header.lastStep       ;
    *lv_out8 = init_status                          ;

}
```

Bibliography

- http://ams.com/eng/Products/Position-Sensors/Magnetic-Rotary-Position-Sensors/AS5048A
- http://ams.com/eng/Products/Position-Sensors/Magnetic-Rotary-Position-Sensors/AS5055A
- https://it.mathworks.com/products/matlab/app-designer.html
- https://it.mathworks.com/help/simulink/ug/consult-the-performance-advisor.html
- “ACU R1.0 Software Technical Requirements Specification", ABB Technology Ltd. – BU PPMV, 04 June 2018,
- “ACU PL LOGIC LIBRARY”, ABB Technology Ltd. – BU PPMV, 1VCD600102 04 June 2018,