How to Use iMOTION™ Script Language

About this document

Scope and purpose

This application note provides a guideline for using the iMOTION™ script language on Motion Control Engine (MCE) platform with typical script examples covering the implementation of Low-Pass Filter (LPF), 2-level speed selection interface, motor target speed shaping based on DC bus voltage with brown-out protection, and dynamic motor current limit customization. Each script example is prepared for both the MCEWizard/MCEDesigner platform and for the iMOTION™ Solution Designer (iSD) platform.

Intended audience

This document is intended for customers who would like to understand how to use the iMOTION™ script language to realize customization of system start-up behavior, specific speed profile design, and system specific fault handling design.

Table of contents

Table of Contents

1 Script Language Overview

1.1 Introduction

The latest software release of iMOTION™ MCE includes a script engine, which offers users the ability to customize system level functionalities without affecting the motor and PFC control algorithm. The script engine is a light weight virtual machine that supports the reading and writing of all the motor control and PFC parameters and variables, allowing users to take advantage of the analog and digital resources that are not used by motor and / or PFC control, and is scalable for any functional extension in future. Typical script use cases include customization of system start-up behavior, specific speed profile definition and parameter configuration, and fault handling.

- The CPU resource is prioritized for the execution of the motor and PFC control algorithms. The spare CPU resource is used for the execution of background tasks as well as the script engine which is used to drive the execution of the script code. The priority of the execution of the script code is lower than that of the motor and PFC control algorithm execution, so that it won't affect the performance of the control algorithms. It is highly recommended to check actual CPU load during the run time to ensure the CPU resource is allocated appropriately.
- The script engine supports 2 independent tasks, namely Task 0 and Task 1, running concurrently. The user script program runs repeatedly on a configurable interval within the Task 0 or Task 1 loop. The shortest possible execution period is 1 ms for Task 0 and 10 ms for Task 1. The execution period for each task can be configured to the multiples of 1 ms for Task 0 or 10 ms for Task 1 in the script code. Task 0 has greater priority than Task 1.
- iMOTION™ script language is a type of interpreted language, for which its implementation compiles a script program into pseudo code (bytecode) first, and then executes instructions directly by a virtual machine running on MCE.

1.2 Script Development Workflow

1.2.1 Using MCEWizard/MCEDesigner

The typical workflow of script program development starts from using MCEWizard [4] (or any other text editors) to write script code and save as script input file with '.mcs' suffix. MCEWizard is used to configure available Analog-to-Digital Converter (ADC) or General-Purpose-Input-Output (GPIO) pins if needed, and MCEWizard is also used to compile the script code to generate a script object file with '.ldf' suffix. The ldf file contains information about the total number of script instructions for Task 0 and Task 1, as well as a list of global variables defined in the script code. Then MCEDesigner [3] is used to download the ldf file to the target MCE, and it also supports monitoring the values of global variables used in the script program. More details about the script language and its development can be found in [2].

1.2.2 Using iMOTION™ Solution Designer

A typical workflow for script program development starts from using the Script Editor in the iSD to create a new script project. The created script project is saved in the "Script" folder of the active iSD project. The created script project has three default template files for script source code files, each with the suffix '.mcs' (Globals.mcs, Script_Task0.mcs and Script_Task1.mcs). Use the Script Editor to edit source code, define the global variables (Globals.mcs), write instructions to be executed in Task 0 (Script_Task0.mcs) and Task 1 (Script_Task1.mcs) respectively, and then save the source code. It is also possible to put all the instructions in one script file, as is the case with MCEWizard/MCEDesigner. In that case, exclude the unused script source code files in the active script project.

The available ADC (Analog-to-Digital Converter) or General-Purpose-Input-Output (GPIO) pins can be configured in the Parameter Configuration Wizard window in the iSD if needed. The Script Editor can also compile the script source code to generate a script bytecode file with the '.ldf' suffix in the “Output” folder, located in the script project directory. The ldf file contains information about the total number of script instructions for Task 0 and Task 1, as well as a list of global variables defined in the script code.

After having successfully built a script project using the Script Editor, program the script bytecode using the iSD (‘Project --> Build Project --> build’ with script, and ‘Tools --> Programmer --> Connect --> Program’ to program the target device). The Script Editor supports a debugging function, and users can monitor the value of global variables, set break points, etc. using the debug session in Script Editor after connecting to the target device with the iSD. Compared to the workflow using MCEWizard/MCEDesigner, the workflow of script program development using the iSD is simpler. More details about the script language and its development can be found in [5] .

1.2.3 Migrating from MCEWizard/MCEDesigner platform to iSD Platform

This section explains how to migrate the script code developed using the MCEWizard/MCEDesigner to the iSD platform. Please refer to the following steps.

1. Prepare an original script source code file developed using the MCEWizard/MCEDesigner’s.
2. Create new project: click “Project” in the Script Editor menu and click “New Project”. Type in the desired new project name in the text box of the “New Project” window, then click “Ok” to generate a new script project. In the example shown in Figure 1, the script project name is given as “mce_script”. By default, there are three *.mcs files included in the script project: “Global.mcs”, “Script_Task0.mcs” and “Script_Task1.mcs”.
3. Copy the original script source code developled using MCEWizard/MCEDesigner to the relevant script source code files in the Script Editor.
    a. Script system parameters, such as execution period and exection steps for Task 0 and Task 1, are defined in the "Properties" window of the Script Editor as shown in Figure 1, and it is not necessary to repeat this information in the source code. Please type in each value of the script system parameters here with reference to the original script source code developed using MCEWizard/MCEDesigner.
    b. Open Globals.mcs file and copy the code of the definition of global variables from the original script source code developed using MCEWizard/MCEDesigner.
    c. Open Script_Task0.mcs file and copy the code of the Task 0 functions, such as Script_Task0_init() and Script_Task0() from the original script source code developed using MCEWizard/MCEDesigner.
    d. Open Script_Task1.mcs file and copy the code of the Task 1 functions, such as Script_Task1_init() and Script_Task1() from the original script source code developed using MCEWizard/MCEDesigner.

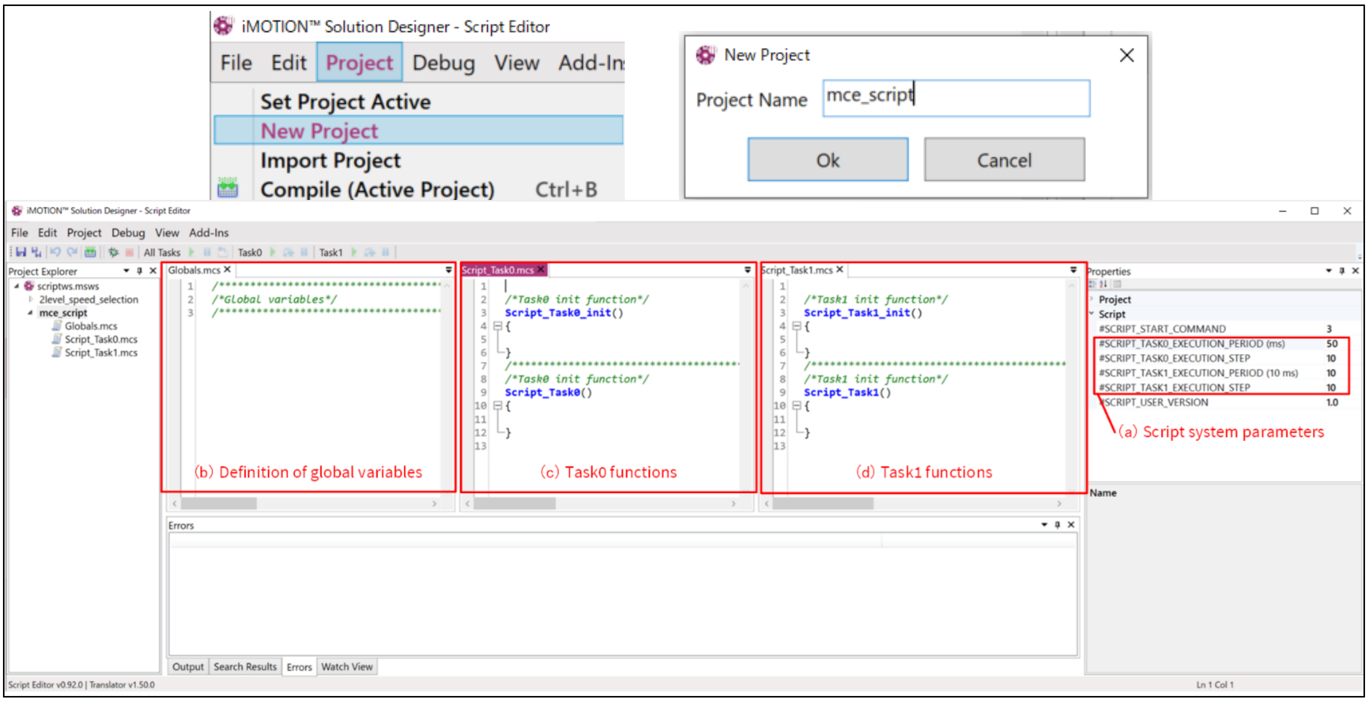


Figure 1 Create new script project on Script Editor in iSD

4. Change the parameter names for the iSD.
   MCE parameter names are different between the MCEDesigner and the iSD. Therefore, it is necessary to modify the parameter names appropriately for the iSD. For example, as shown in Figure 2, you can find the error in the line 17, and the error message in the "Errors" window of the Script Editor is “Error #11: Unknown identifier 'VdcFilt' at mce_script\script_task0.mcs:17”. From this error message, you can find that this variable should be modify for the iSD.

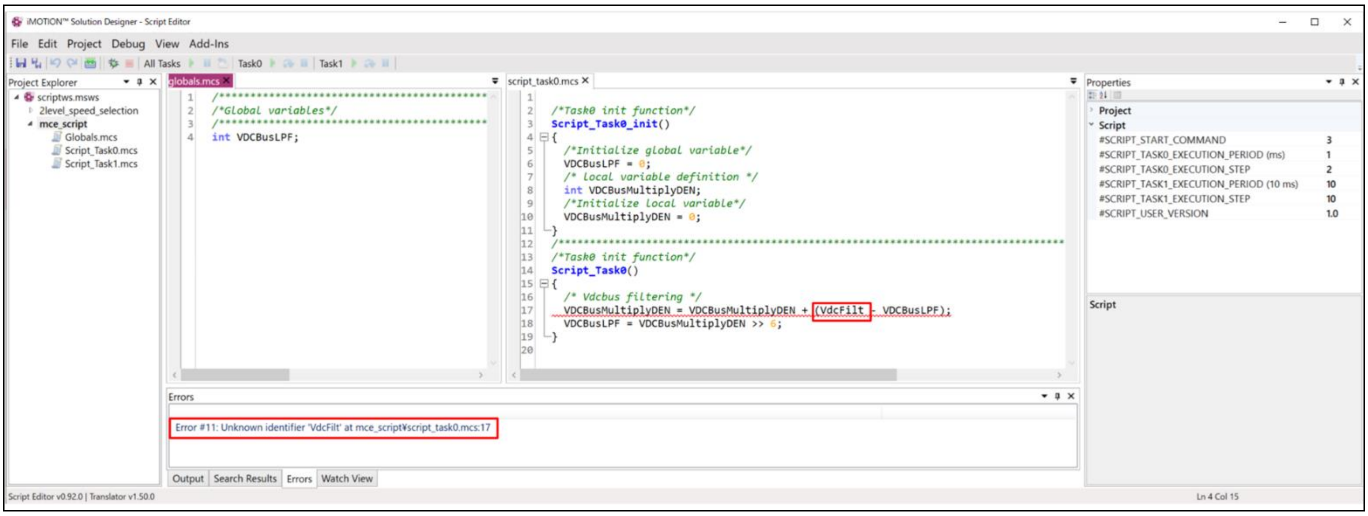


Figure 2 Identify the different name variable between MCEDesigner and iSD

Use the iSD help window to find the correct parameters for the iSD. As shown in Figure 3, click the question mark icon in the iSD and open the help window. In the help window, click on the “Search” Tab, type in parameter name, click the “Search” button, and click the links of the search result, then the parameter information for the iSD will be appeared in the right side of the help window, and you can identify the parameter name for the iSD.

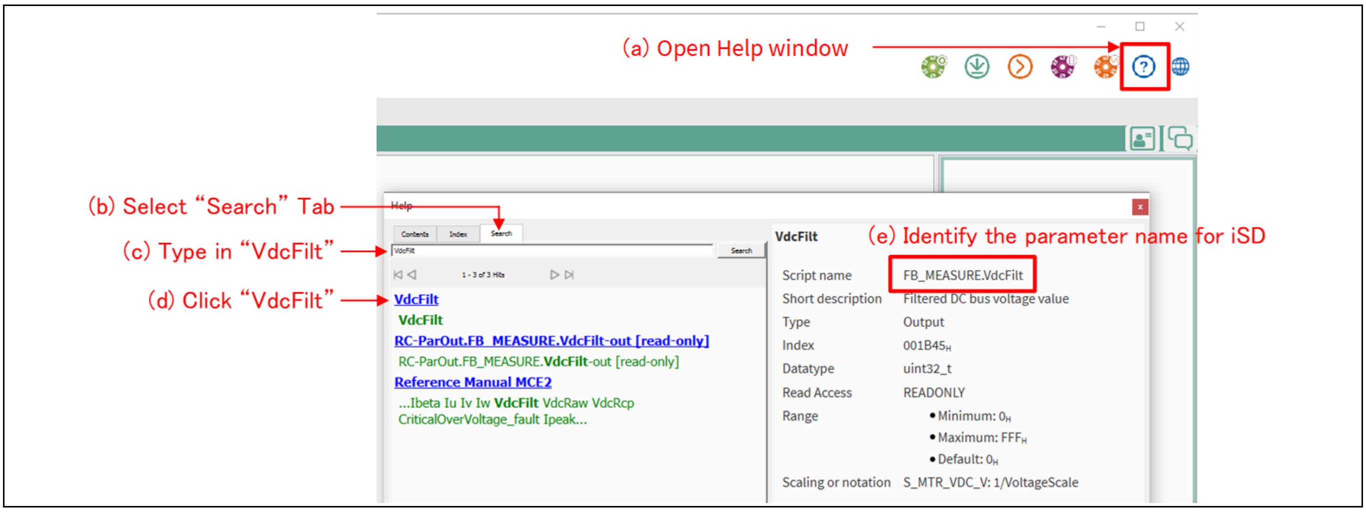


Figure 3 How to identify the parameter name for iSD (using Help window in iSD)

Return to the script and replace the iSD parameter name, resolving the error as shown in Figure 4. With that, the original script source code developed using the MCEWizard/MCEDesigner has been migrated to the iSD. It is ready to build the script project.

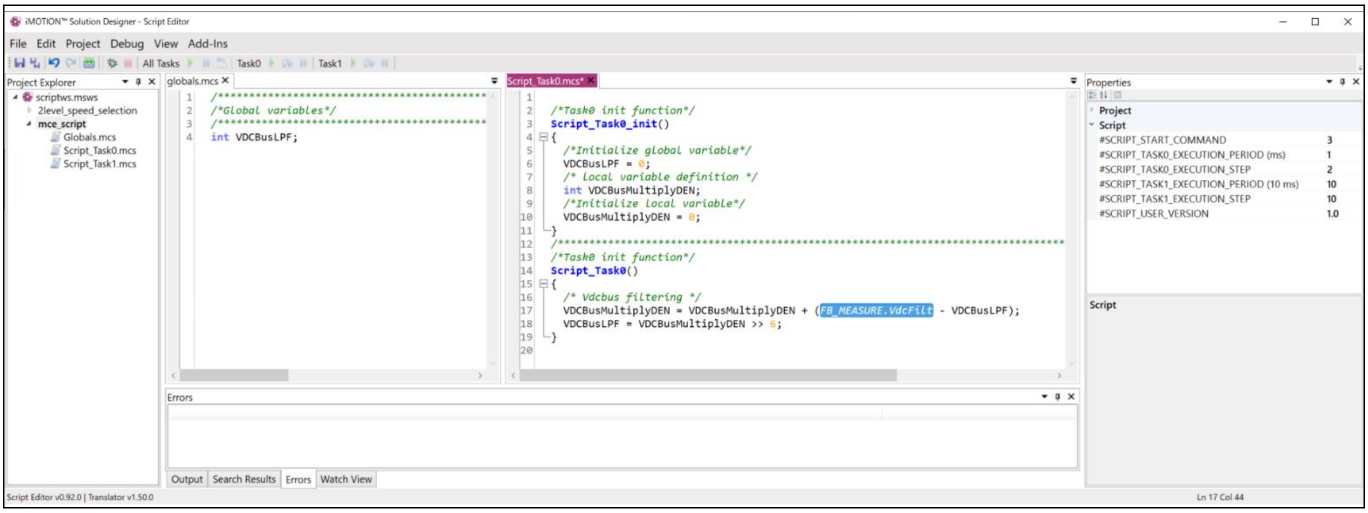


Figure 4 Modified parameter for iSD with NO errors

As an alternative way to change the parameter names for the iSD. There is an auto completion feature in the Script Editor. As shown in Figure 5, at first select the Unknown parameter 'VdcFilt' and next press the "Ctrl+SpaceBar" shortcut keys in the Script Editor, then a complete list of all the available MCE parameters and variables appear inside editor area. Finally select the desired parameter name and press enter to replace. It is ready to build the script project.

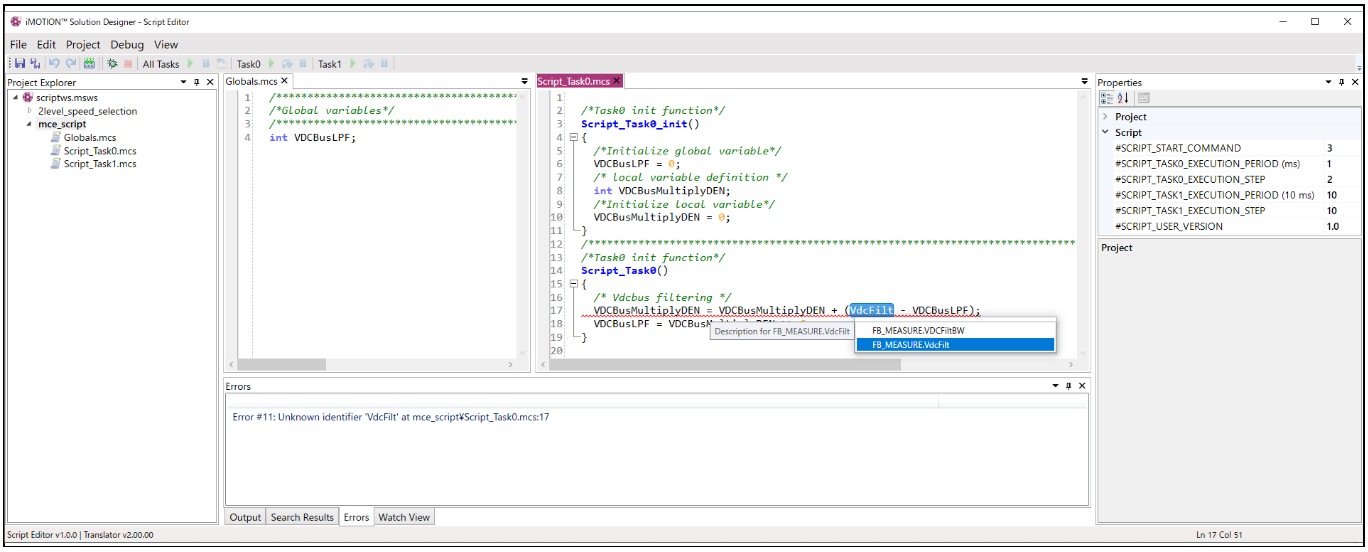


Figure 5 How to identify the parameter name for iSD (using Auto Completion feature)

2 Script Application Examples

2.1 2-Level Speed Selection Interface

2.1.1 Speed Selection Interface Requirement

A multi-level speed selection interface to support different speed levels selected by users is commonly seen in motor control applications such as hair-dryers. This example details the requirement and script implementation of a 2-level speed selection interface using one of the available ADC pins on an IMC101T [1] controller from iMOTION™ MCE series.

Some hardware circuits were designed to translate the position of the speed selection mechanical switch into a corresponding analog voltage level between 0 V and 5 V. Specifically, the voltage range from 0 V to 1 V was defined as the OFF state, the voltage range from 1 V to 2 V was defined as the LOW SPEED state, and the voltage range from 2 V and above was defined as the HIGH SPEED. In order to eliminate potential oscillation when the voltage level is in the vicinity of the boundaries of different speed states, a hysteresis of 0.2 V was introduced.

This application requires an analog voltage sensing interface to sample and translate the analog voltage to the corresponding speed selection levels.

Figure 6 depicts the relationship between the speed selection and the analog voltage level. The solid line shows that when the speed selection is currently in the OFF state, and if the analog voltage rises above 1 V, then the speed selection shifts from the OFF state to the LOW SPEED state. When the current speed selection is in the LOW SPEED state, and if the analog voltage exceeds 2 V, then the speed selection shifts from the LOW SPEED state to the HIGH SPEED state. The dashed line shows that when the speed selection is currently in the HIGH SPEED state, and if the analog voltage falls below 1.8 V, then the speed selection shifts from the HIGH SPEED state to the LOW SPEED state. When the current speed selection is in the LOW SPEED state, and if the analog voltage falls below 0.8 V, then the speed selection shifts from the LOW SPEED state to the OFF state.

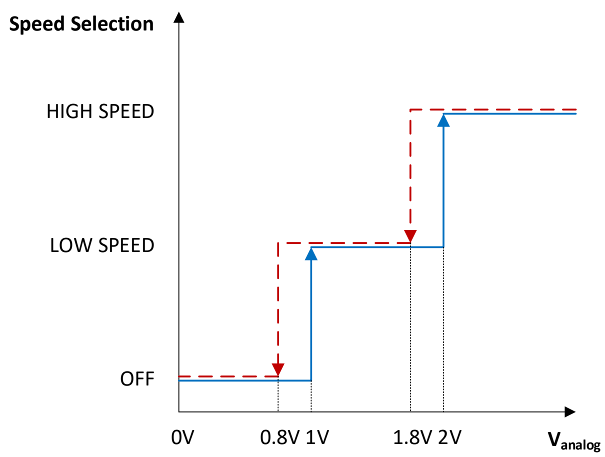


Figure 6 Speed Selection & Analog Voltage Relationship

2.1.2 Analog Input Pin for Speed Selection Interface

This application specific speed selection requirement described in Section 2.1.1 can be achieved by enabling an analog input pin that is supported by the script engine. For this design, an AIN0 pin was chosen to interface with the speed selection hardware circuit. Once enabled, this analog input pin is sampled by MCE every 1 ms [2], and the ADC conversion results can be obtained by reading the variable named `ADC_Result0`.

Given that the resolution of the ADC is 12 bit, the calculation from the voltage at AIN0 pin to the ADC conversion result follows this formula: $\text{ADC\_Result0} = INT(V_{AIN0} \cdot \frac{2^{12}-1}{V_{ref}} + 0.5)$, where V_{ref} is the reference voltage for the ADC. If we choose V_{ref} as 5 V, then those voltage thresholds associated with HIGH SPEED and LOW SPEED levels can be calculated using the abovementioned formula as summarized in Table 1.

Table 1 Speed Selection Interface Voltage Thresholds

Variable Name	Voltage Threshold	ADC Conversion Result
`VLSStart`	1 V	819 (ADC Counts)
`VLSStop`	0.8 V	655 (ADC Counts)
`VHSStart`	2 V	1638 (ADC Counts)
`VHSStop`	1.8 V	1474 (ADC Counts)

2.1.3 Speed Selection State Machine

The speed selection logic can be abstracted using a finite state machine (FSM) model. An FSM can change from one state to another in response to certain inputs. Figure 7 shows a state machine that was designed to interpret the speed selection inputs. It uses a state variable named `SpeedMode` to represent 3 possible states, namely, Speed_Mode_OFF (`SpeedMode = 0`), Speed_Mode_LOW_SPEED (`SpeedMode = 1`), and Speed_Mode_HIGH_SPEED (`SpeedMode = 2`). Starting off in the Speed_Mode_OFF state, the target speed is reset to 0, and the motor is stopped. If VSP pin voltage is greater than `VLSStart`, then it shifts to the Speed_Mode_LOW_SPEED state. While it is in the Speed_Mode_LOW_SPEED state, if VSP pin voltage is lower than `VLSStop`, then it shifts to Speed_Mode_OFF; if VSP pin voltage is greater than `VHSStart`, then it shifts to the Speed_Mode_HIGH_SPEED state. While it is in the Speed_Mode_HIGH_SPEED state, if VSP pin voltage is lower than `VHSStop`, then it shifts to the Speed_Mode_LOW_SPEED state. While it is in the Speed_Mode_HIGH_SPEED or Speed_Mode_LOW_SPEED state, the target speed is set to the pre-defined `HighSpeedValue` or `LowSpeedValue` corresponding to the specific speed selection levels, and the motor is started. The start / stop motor operation can be realized by setting or resetting the motor variable named `Command`. Thanks to the accessibility of the motor parameters enabled by the script engine, the `Command` parameter can be directly used in the script code without declaration.

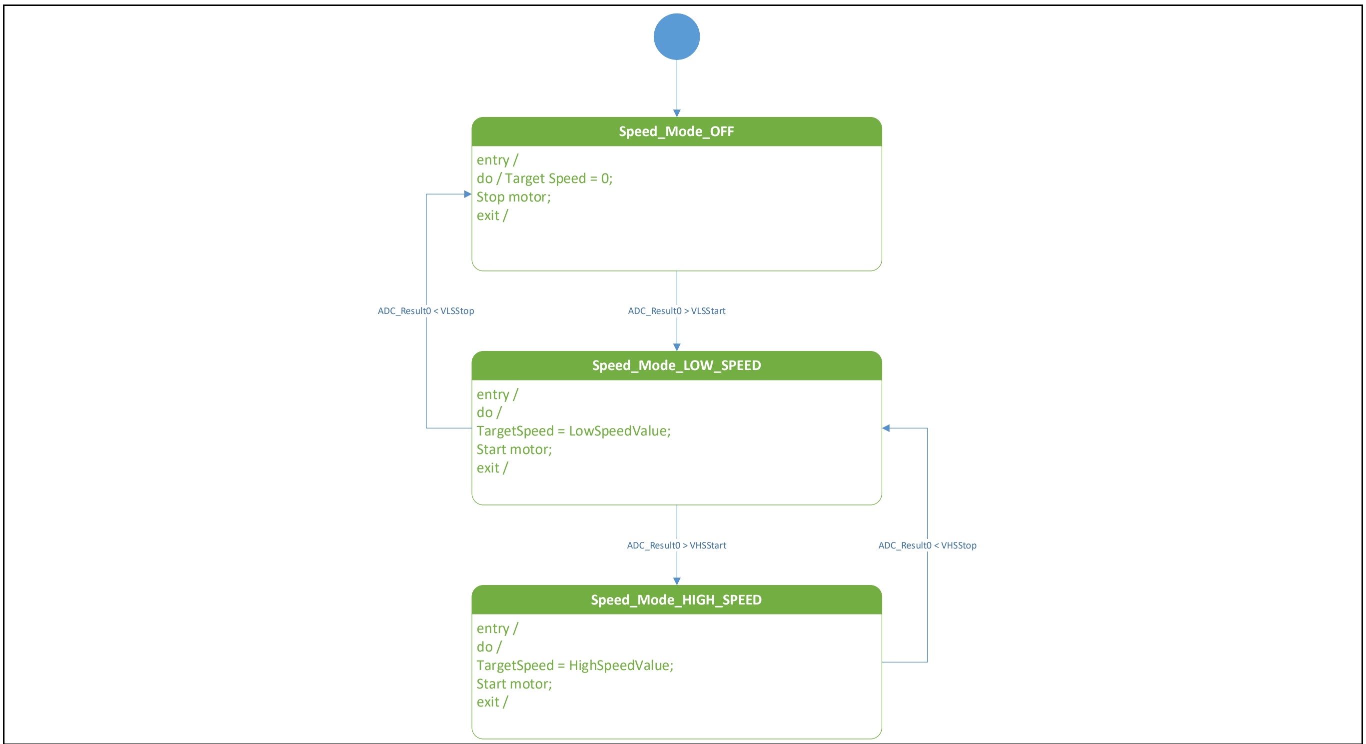


Figure 7 Speed Selection State Machine

2.1.4 Speed Selection Interface Script Implementation

Code Listing 1 shows the source code for the 2-level speed selection interface application implemented in Task 1 using the MCEWizard/MCEDesigner. Since the user speed selection switch position doesn't change frequently, it is recommended to set the loop execution period of Task 1 to be 50 ms. Code Listing 2 shows a portion of the compiled script object file where it shows at line 009 that the number of instructions for Task 1 is 20. With this in mind, the execution step for Task 1 should be set to greater than 20 to ensure that the entire loop of Task 1 is completed during each execution period. In this example, the execution period for Task 1 (`SCRIPT_TASK1_EXECUTION_PERIOD`) was set to 5, and the execution step for Task 1 (`SCRIPT_TASK1_EXECUTION_STEP`) was chosen to be 20 to meet the desired timing requirement.

This example can also be implemented in Task 0, in which case the execution period for Task 0 (`SCRIPT_TASK0_EXECUTION_PERIOD`) should be set to 50 to achieve the same execution period of 50 ms.

2.1.4.1 Script Code for MCEWizard/MCEDesigner

Code Listing 1 Speed Selection Interface Script Code for MCEWizard/MCEDesigner

```
/*************************************************************/
/*Script user version value, should be 255.255*/
#SET SCRIPT_USER_VERSION (1.00)
#SET SCRIPT_TASK1_EXECUTION_PERIOD (5)
/*Defines number of lines to be executed every 10mS in Task1*/
#SET SCRIPT_TASK1_EXECUTION_STEP (20)
/*************************************************************/
/* constant definition */
CONST int VLSStart = 819;  /* 1.0 V =>  819 counts */
CONST int VLSStop = 655;   /* 0.8 V =>  655 counts */
CONST int VHSStart = 1638; /* 2.0 V => 1638 counts */
```

Code Listing 1 Speed Selection Interface Script Code for MCEWizard/MCEDesigner

```
CONST int VHSStop = 1474;  /* 1.8 V => 1474 counts */
CONST int LowSpeedValue = 5000;
CONST int HighSpeedValue = 10000;

/* Task1 init function*/
Script_Task1_init()
{
  /* local variable definition */
  int SpeedMode;

  /* Initialize local variable*/
  SpeedMode = 0;
}
/* Task1 script function*/
Script_Task1()
{
  if (SpeedMode == 0) /* Speed selection is in OFF state. */
    {
      TargetSpeed = 0;
      Command = 0; /* Stop the motor. */
      if (ADC_Result0 > VLSStart)
        {
          SpeedMode = 1; /* Shift to LOW_SPEED state. */
        }
    }
  if (SpeedMode == 1) /* Speed selection is in LOW_SPEED state. */

    {
      if (ADC_Result0 > VHSStart)
        {
          SpeedMode = 2; /* Shift to HIGH_SPEED state. */
        }
      else
        {
          if (ADC_Result0 < VLSStop)
            {
              SpeedMode = 0; /* Shift to OFF state. */
            }
          else /* Stay in LOW_SPEED state. */
            {
              TargetSpeed = LowSpeedValue; /* Update TargetSpeed. */
              Command = 1; /* Start motor. */
            }
        }
    }
  if(SpeedMode == 2) /* Speed selection is in HIGH_SPEED state. */
    {
      if(ADC_Result0 < VHSStop)
        {
          SpeedMode = 1; /* Shift to LOW_SPEED state. */
```

Code Listing 1 Speed Selection Interface Script Code for MCEWizard/MCEDesigner

```
                }
            else /* Stay in HIGH_SPEED state. */
                {
                  TargetSpeed = HighSpeedValue; /* Update TargetSpeed. */
                  Command = 1;
                }
            }
        }
```

Code Listing 2 Portion of Compiled Script Object File for Speed Selection Interface Script Code

```
%-------------------------------------------------------------
% Script Object File
%-------------------------------------------------------------
% SCRIPT_USER_VERSION                : 001.000
% DATE & TIME                        : 26.09.2022  11:22:33
% SIZE                               : 297 Bytes
% Total Number of Lines              : 69
% Task0 - Number of Instructions     : 0
% Task1 - Number of Instructions     : 20
%-------------------------------------------------------------
```

2.1.4.2 Script Code for iSD

Code Listing 3 and Code Listing 4 shows the source code for the 2-level speed selection interface application using the iSD. In order to reuse the source code from the MCEWizard/MCEDesigner in the iSD, it is necessary to modify the names for some variables. The names of variables that need to be modified are listed in Table 2.

In addition, since the iSD sets the Task 0/Task 1 execution period and execution step in the Script Project Properties, these settings are removed from the script code. Click on Script Project in the Script Explorer, then the information of the script execution period and execution steps are appeared in the Property window as shown in Figure 8, then set the appropriate values. This time, with the same settings as in Code Listing 1, the execution period of Task 1 (SCRIPT_TASK1_EXECUTION_PERIOD) is set to 5 and the execution step of Task 1 (SCRIPT_TASK1_EXECUTION_STEP) to 20. As shown in Figure 9, the number of instructions of each task can be confirmed in the Output window as part of the Build result.

Code Listing 3 Speed Selection Interface Script Code for iSD (Global.mcs)

```
/*************************************************************/
/*Global variables*/
/*************************************************************/
/* constant definition */
CONST int VLSStart = 1240; /* 1 V => 1240 counts */
CONST int VLSStop = 992;   /* 0.8 V => 992 counts */
CONST int VHSStart = 2480; /* 2 V => 2480 counts */
CONST int VHSStop = 2232;  /* 1.8 V => 2232 counts */
CONST int LowSpeedValue = 5000;
CONST int HighSpeedValue = 10000;
```

Code Listing 4 Speed Selection Interface Script Code for iSD (Script_Task1.mcs)

```
/**************************************************************/
/*Task1 init function*/
Script_Task1_init()
{
  /* local variable definition */
  uint8_t SpeedMode;

  /*Initialize local variable */
  SpeedMode = 0;
}
/**************************************************************/
/*Task1 script function*/
Script_Task1()
{
  if (SpeedMode == 0) /* Speed selection is in OFF state. */
    {
      APP_MOTOR0.TargetSpeed = 0;
      APP_MOTOR0.Command = 0; /* Stop the motor. */
      if (FB_ADC.adc_result[0] > VLSStart)
        {
          SpeedMode = 1; /* Shift to LOW_SPEED state. */
        }
    }
  if (SpeedMode == 1) /* Speed selection is in LOW_SPEED state. */
    {
      if (FB_ADC.adc_result[0] > VHSStart)
        {
          SpeedMode = 2; /* Shift to HIGH_SPEED state. */
        }
      else
        {
          if (FB_ADC.adc_result[0] < VLSStop)
            {
              SpeedMode = 0; /* Shift to OFF state. */
            }
          else /* Stay in LOW_SPEED state. */
            {
              APP_MOTOR0.TargetSpeed = LowSpeedValue; /* Update APP_MOTOR0.TargetSpeed. */
              APP_MOTOR0.Command = 1; /* Start motor. */
            }
        }
    }
  if(SpeedMode == 2) /* Speed selection is in HIGH_SPEED state. */
    {
      if(FB_ADC.adc_result[0] < VHSStop)
        {
          SpeedMode = 1; /* Shift to LOW_SPEED state. */
        }
      else /* Stay in HIGH_SPEED state. */
```

Code Listing 4 Speed Selection Interface Script Code for iSD (Script_Task1.mcs)

```
                    {
                        APP_MOTOR0.TargetSpeed = HighSpeedValue; /* Update
   APP_MOTOR0.TargetSpeed. */
                        APP_MOTOR0.Command = 1;
                    }
               }
          }
```

Table 2 Parameter name Differences between MCEWizard/MCEDesigner and iSD

MCEWizard/MCEDesigner	iSD
TargetSpeed	APP_MOTOR0.TargetSpeed
Command	APP_MOTOR0.Command
ADC_Result0	FB_ADC.adc_result[0]

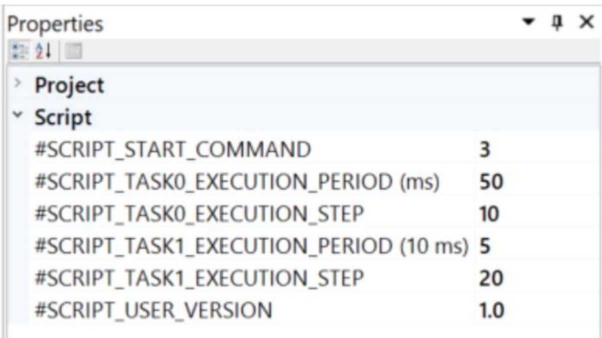


Figure 8 The execution period and step of each tasks

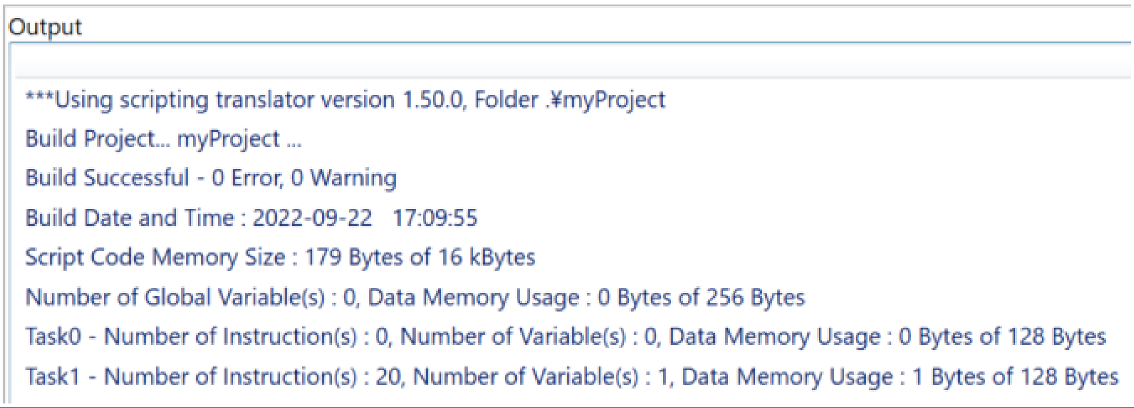


Figure 9 Build Result of the Speed Selection Interface Script Code on iSD

In the iSD, as shown in Figure 10, the “Watch View” window can be used to read or write script variables in debug mode of the Script Editor. It is also possible to read MCE parameters and variables using this window in debug mode.

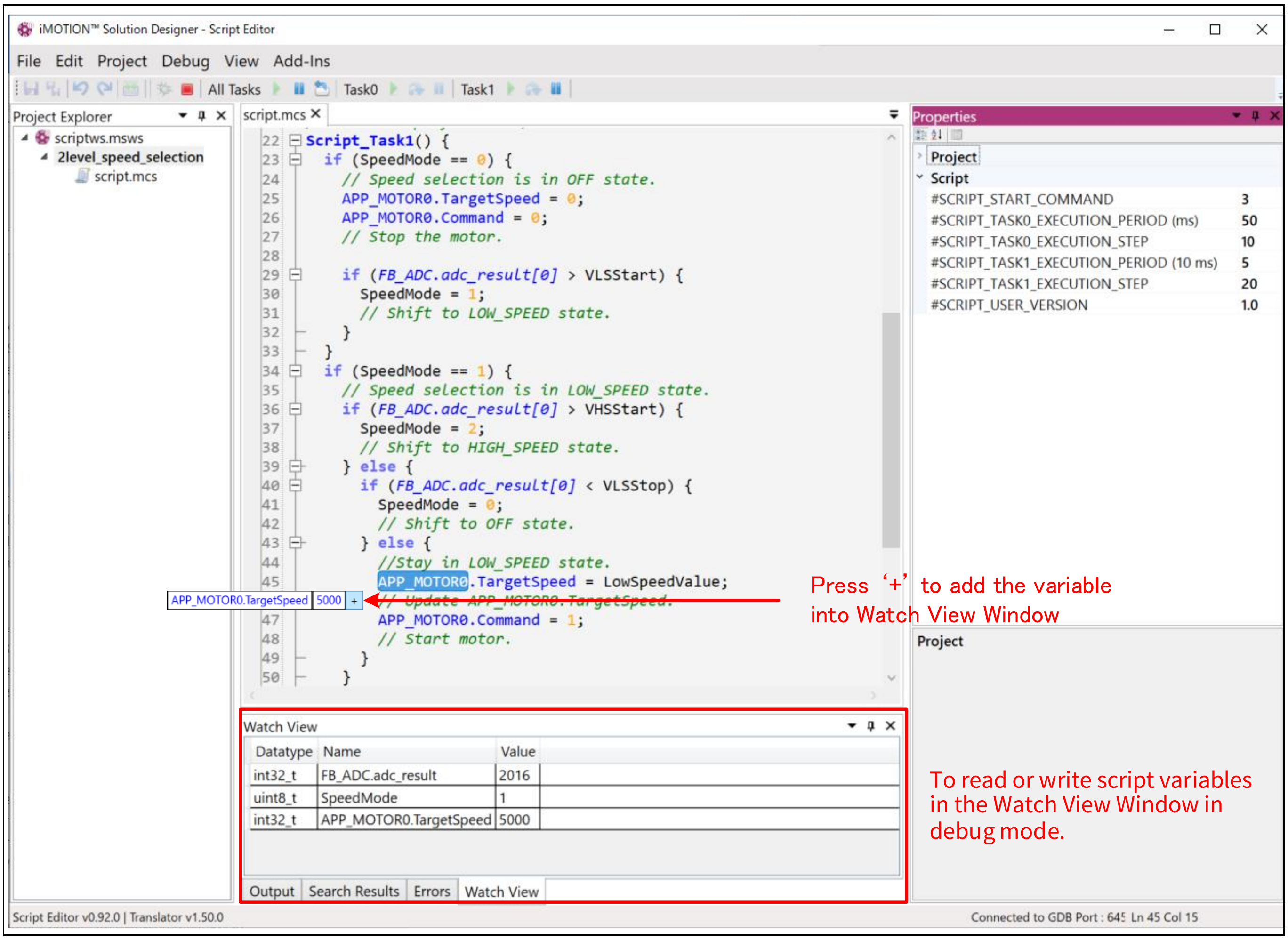
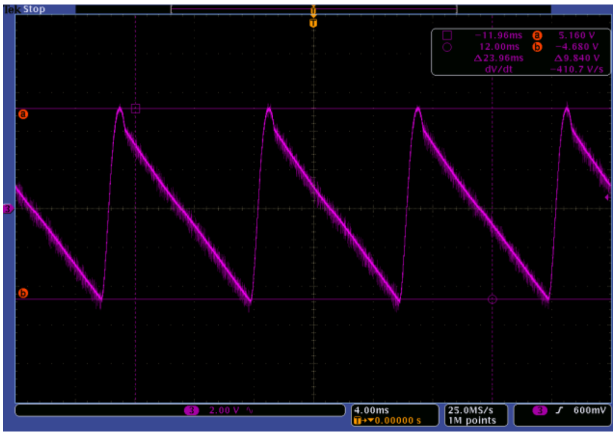


Figure 10 Script Editor: Watch View Window

2.2 Low-Pass Filter for DC Bus Voltage

2.2.1 DC Bus Voltage Ripple

Typically, the AC input front-end stage consists of a bridge rectifier followed by a bulky DC bus capacitor to convert the AC mains voltage to DC voltage whose amplitude tracks the peak of the AC input voltage. DC bus voltage refers to the voltage across the DC bus capacitor. When the motor is running, the DC bus voltage waveform typically contains high frequency switching ripples as well as low frequency ripples due to bus capacitor charge and discharge operation at twice the mains frequency. Figure 11 is a screenshot of the AC portion of the actual DC bus voltage waveform with an IMC101T controller driving a Permanent Magnet Synchronous Motor (PMSM) running at a speed = 19400 RPM and Vin = 125 VAC / 50 Hz. The amplitude of the DC bus voltage ripples is around 9.84 V.



CH3: DC Bus Voltage (AC coupling); V_{pp_CH3}: 9.84 V

Figure 11 DC Bus Voltage Waveform Screenshot

2.2.2 DC Bus Voltage Sensing

The valid input voltage for the ADC of MCE ranges from 0 V to 5 V due to the selection of 5 V as the ADC reference voltage ($V_{ref} = 5V$). Accordingly, the DC bus voltage is scaled down by a voltage divider composed of R1 and R2 as shown in Figure 12 and then connected to VDC pin of MCE. With $R_1 = 2M\Omega$ and $R_2 = 13.3K\Omega$, the DC Bus sensing gain $G_{DCBus\_sensing} = \frac{R2}{R1+R2} = \frac{13.3K\Omega}{2M\Omega+13.3K\Omega} = 0.00661$, and the maximum DC voltage that the VDC pin can sense is up to $\frac{V_{ref}}{G_{DCBus\_sensing}} = \frac{5}{0.00661}$ =757 V.

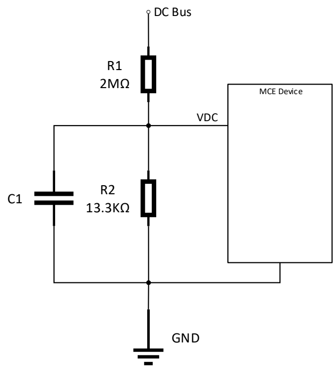


Figure 12 DC Bus Voltage Sensing Interface Circuit Diagram

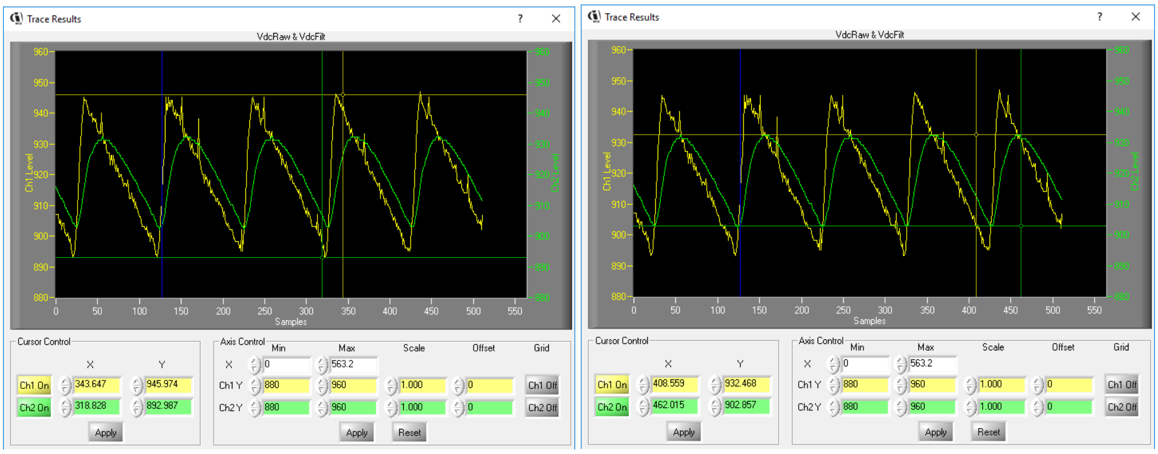
The DC bus voltage is sampled by MCE every motor Pulse Width Modulation (PWM) cycle and is represented by the motor parameter `VdcRaw` whose unit is ADC count [2]. A typical motor PWM cycle value is 50 µs. `VdcRaw` goes through an internal digital LPF stage and the result is stored in `VdcFilt` [2].

Given that the resolution of the ADC is 12 bit, the conversion from DC bus voltage to ADC sampling result follows this formula: $V_{DCBus\_ADC} = INT(V_{DCBus} \cdot G_{DCBus\_sensing} \cdot \frac{2^{12}-1}{V_{ref}} + 0.5)$, where the INT operator means taking the integer portion of a given number.

Thanks to the accessibility of the motor parameters enabled by the script engine, `VdcRaw` and `VdcFilt` parameters can be directly used in the script code without declaration. Figure 13 shows the `VdcRaw` and `VdcFilt` waveforms under the same input / output conditions as in the case of Figure 11 using the tracing window of MCEDesigner [3]. With 9.84 V of DC bus voltage ripple amplitude, `VdcRaw` ripple amplitude should be

53 ADC counts following the abovementioned conversion formula. From Figure 13 it can be seen that the amplitude of `VdcRaw` ripple was about 53 ADC counts. Comparing `VdcFilt` waveform with that of `VdcRaw`, it can be observed that although most of the high frequency ripples seen in `VdcRaw` was attenuated, `VdcFilt` still presented a good amount of low frequency ripples whose amplitude was as high as 30 ADC counts.

In order to obtain an averaged value of DC bus voltage when the system is in steady state, there is a need to implement an additional stage of LPF in the script to attenuate the ripple of `VdcFilt` to no more than 1 ADC count.



Ch1: `VdcRaw`; Ch2: `VdcFilt`; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 2; Trigger Level: 920; V_{in}: 125 VAC / 50 Hz; V_{pp_CH1}: 53 ADC counts; V_{pp_CH2}: 30 ADC counts

Figure 13 VdcRaw & VdcFilt Waveform Screenshot

2.2.3 LPF Design & Implementation

Considering the limited resources supported by the script engine, a 1st order Infinite Impulse Response (IIR) low-pass digital filter algorithm was chosen for this application. Its difference equation is shown as follows: $(n) = \alpha \cdot y(n-1) + (1-\alpha) \cdot x(n)$, where α is a constant between 0 and 1, $x(n)$ is the current input value, $y(n)$ is the current output value, and $y(n-1)$ is the last output value. This filter's z domain transfer function is as follows: $H_{LPF}(z) = \frac{1-\alpha}{1-\alpha \cdot z^{-1}}$. Assuming that the sampling period is represented by T_s, and using $z = e^{s \cdot T_s}$ to replace z, we could obtain the filter's transfer function in s domain: $H_{LPF}(s) = \frac{1-\alpha}{1-\alpha \cdot e^{-sT_s}}$.

The dominant portion of the `VdcFilt` ripples was the twice-of-mains-frequency component at 100 Hz or 120 Hz. According to Nyquist theorem, the sampling frequency must be at least twice of the frequency of interest or greater to realize effective attenuation. Task 1 can support down to 10 ms execution period, which in this case is not enough. Task 0 was chosen to implement this LPF algorithm thanks to its support for down to 1 ms execution period. The greater the sampling frequency is, the greater the frequency of interest that can be attenuated goes. Accordingly, we chose the sampling period $T_s = 1\ ms$ so that the LPF would be effective for the frequency ranging up to 500 Hz.

Taking 100 Hz as the worst case example, attenuation of $\frac{1}{30}$ corresponds to $20 \cdot log_{10}\left(\frac{1}{30}\right) = -29.5dB$. In order to achieve at least $-29.5dB$ at 100 Hz, the desired α needs to be 0.98 based on the calculation of the magnitude of $H_{LPF}(s)$. Unfortunately, the script engine only supports 32-bit signed integer type of variables [2], so that the floating point number 0.98 has to be represented in fractional format. If we define $\alpha = \frac{\alpha_{NUM}}{\alpha_{DEN}}$, then the LPF can be implemented by using the following pseudo code in Code Listing 5.

Code Listing 5 LPF Pseudo Code

```
Y1(n)=Y1(n-1)+(αDEN-αNUM)*(X(n)-Y(n-1));
Y(n)=Y1(n)/αDEN;
```

It is recommended users choose α_{DEN} to be equal to the power of 2, so that the division operation can be realized efficiently by right shift operation. If we choose $\alpha_{DEN} = 64$, then the best integer value with minimum error for $\alpha_{NUM} = 63$, which results in an equivalent $\alpha = 0.984$ with an error of 0.5 %. The division by 64 can be replaced by right shifting 6 bits. Code Listing 6 shows the script code implementation for the LPF using MCEWizard/MCEDesigner, and Code Listing 7 and Figure 16 shows the implementation using the iSD as well.

2.2.3.1 Script Code for MCEWizard/MCEDesigner

Code Listing 6 LPF Script Code for MCEWizard/MCEDesigner

```
001 /*****************************************************************/
002 /*Script execution time for Task0 in ms, maximum value 65535*/
003 #SET SCRIPT_TASK0_EXECUTION_PERIOD (1)
004 /*Defines number of lines to be executed every 1ms in Task0*/
005 #SET SCRIPT_TASK0_EXECUTION_STEP (2)
006 /*****************************************************************/
007 /* Global variable definition */
008 int VDCBusLPF;
009 /*****************************************************************/
010 /*Task0 init function*/
011 Script_Task0_init()
012 {
013   /*Initialize global variable*/
014   VDCBusLPF = 0;
015   /* local variable definition */
016   int VDCBusMultiplyDEN;
017   /*Initialize local variable*/
018   VDCBusMultiplyDEN = 0;
019 }
020
021 /*Task0 script function*/
022 Script_Task0()
023 {
024   /* Vdcbus filtering */
025   VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt - VDCBusLPF);
026   VDCBusLPF = VDCBusMultiplyDEN >> 6;
027 }
```

As can be seen from the code, since there are 2 effective instructions (line 025 and 026) in the LPF implementation, the number of instructions to be executed every 1 ms by Task 0 needs to be set to 2 accordingly (line 005), so that Task 0 loop execution period becomes 1 ms. Thus, 1 kHz sampling frequency for `VdcFilt` is ensured. The effective number of instructions for each Task can be found out in the relevant script object file with a suffix of '.ldf'.

With this implementation, the filter's time constant $\tau = -\frac{T_s}{Ln(\alpha)} = -\frac{1ms}{Ln(0.984)} = 63ms$; the cut-off frequency $f_c = \frac{1}{2\pi\tau} = \frac{1}{2\pi \cdot 63ms} = 2.51Hz$; the gain at 100 Hz is -31.9 dB. The Bode plot and step response of the implemented LPF were calculated and shown in Figure 14 and Figure 15.

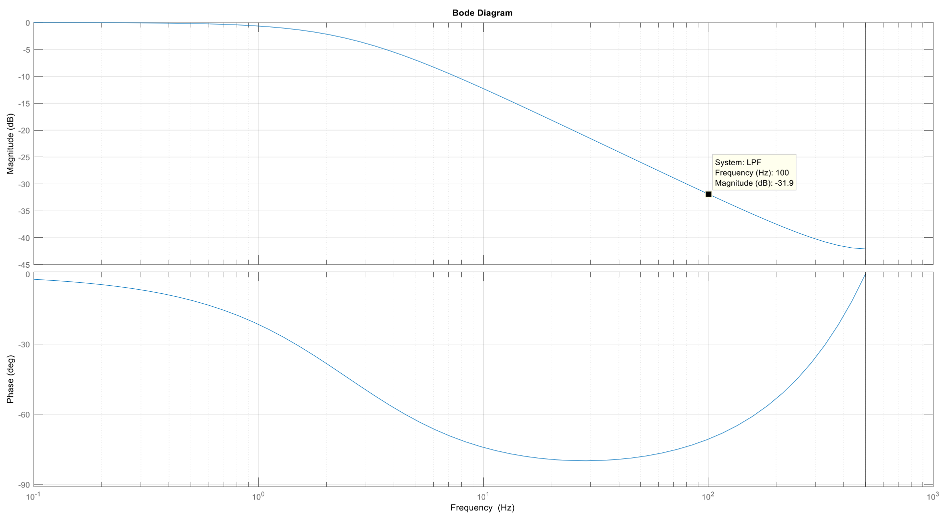


Figure 14 **Calculated 1st Order IIR LPF Frequency Response**

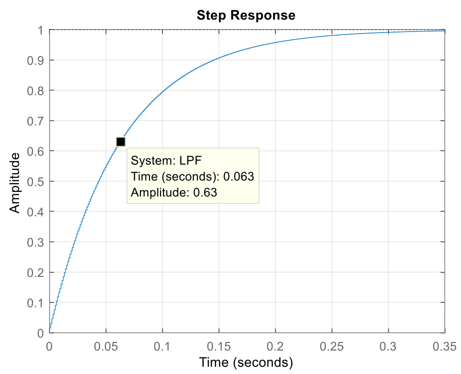


Figure 15 **Calculated 1st Order IIR LPF Step Response**

2.2.3.2 Script Code for iSD

The LPF Script Code for the MCEWizard/MCEDesigner (Code Listing 6), migrated to iSD is shown in Code Listing 7 and Code Listing 8. In order to reuse the source code from MCEWizard/MCEDesigner in iSD, the variable names must be modified as shown in Table 3.

Code Listing 7 **LPF Script Code for iSD (Global.mcs)**

```
/***************************************************************/
/*Global variables*/
/***************************************************************/
/* Global variable definition */
int VDCBusLPF;
```

Code Listing 8 **LPF Script Code for iSD (Script_Task0.mcs)**

```
/***************************************************************/
```

Code Listing 8 LPF Script Code for iSD (Script_Task0.mcs)

```
/*Task0 init function*/
Script_Task0_init()
{
  /*Initialize global variable*/
  VDCBusLPF = 0;
  /* local variable definition */
  int VDCBusMultiplyDEN;
  /*Initialize local variable*/
  VDCBusMultiplyDEN = 0;
}
/************************************************************/
/*Task0 script function*/
Script_Task0()
{
  /* Vdcbus filtering */
  VDCBusMultiplyDEN = VDCBusMultiplyDEN + (FB_MEASURE.VdcFilt - VDCBusLPF);
  VDCBusLPF = VDCBusMultiplyDEN >> 6;
}
```

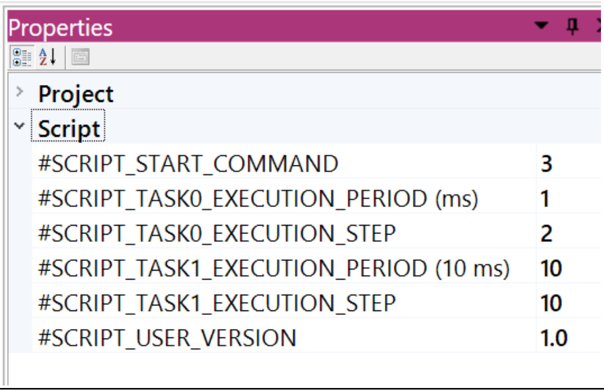


Figure 16 The execution period and the step for LPF Script Code

Table 3 Parameter name Differences between MCEWizard/MCEDesigner and iSD

MCEWizard/MCEDesigner	iSD
VdcFilt	FB_MEASURE.VdcFilt

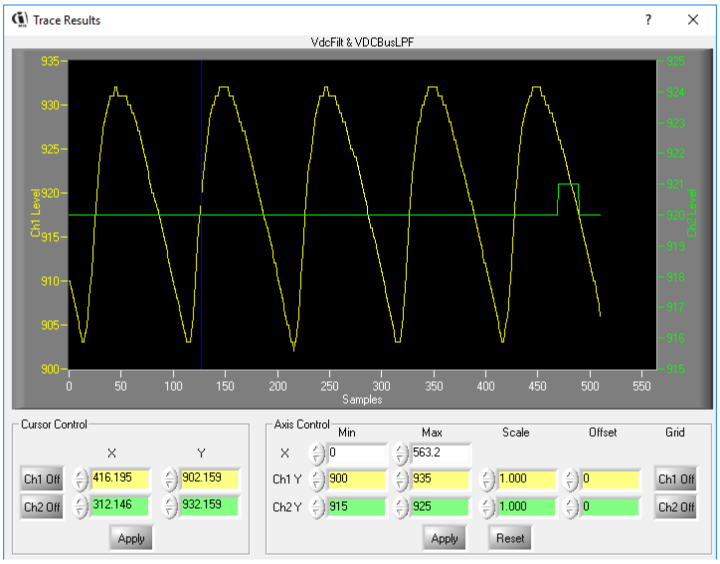
2.2.4 LPF Test Results

Figure 17 shows the waveforms of the filter input represented by `VdcFilt` and the filter output represented by `VDCBusLPF_L` (lower 16 bit of `VDCBusLPF`). It can be seen that the filtered result, `VDCBusLPF_L`, fluctuated by no more than 1 ADC count. With the amplitude of `VdcFilt` being 30 ADC counts, the degree of attenuation achieved was about -30 dB.

Figure 18 shows the measured step response of the implemented LPF, where V_{in} was increased from 70 VAC to 125 VAC. The initial value of `VdcFilt` was 500 ADC counts, and the steady state value of `VdcFilt` was 919 ADC counts, resulting in a step change of 419 ADC counts. The time it took for `VdcFilt` to step up by 265 ADC counts

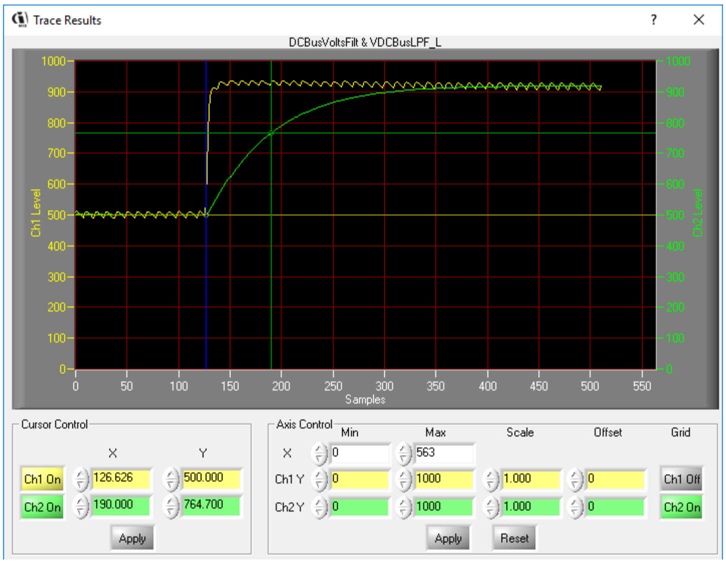
$(419 \cdot (1 - e^{-1}) = 419 \cdot 0.6321 = 265)$ was 63.374 sample counts. Since the motor PWM cycle was 50 µs, and the tracing window screenshot was obtained with a sample rate that was equal to motor PWM frequency divided by 20, the equivalent sample cycle was 1 ms. Accordingly, the measured time constant $\tau_{measured} = 63.374\ sample\ counts \cdot 1ms = 63.374ms$. This result matches the theoretical value very well.

Note: *In the iSD, the only way to debug the script variables is to use the watch view window in script debugger. Currently it doesn't support reading script variables in dashboard or oscilloscope.*



Ch1: `VdcFilt`; Ch2: `VDCBusLPF_L`; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 2; Trigger Level: 920; Vin: 125 VAC / 50 Hz; V_{pp_CH1}: 30 ADC counts; V_{pp_CH2}: 1 ADC count

Figure 17 **`VdcFilt` & `VDCBusLPF_L` Waveform Screenshot**



Ch1: `VdcFilt`; Ch2: `VDCBusLPF_L`; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 20; Trigger Level: 600; Vin: step up from 70 VAC to 125 VAC; △X: 63.374 sample counts; △Y: 264.700 ADC counts

Figure 18 **Measured 1st Order IIR LPF Step Response Screenshot**

2.3 Target Speed Shaping & Brown-out Protection

In this example, it is shown that the motor target speed can be tailored as a function of the DC bus voltage. Additionally, if the DC bus voltage brown-out occurs, then the motor is stopped. Since both of the requirements are based on the DC bus voltage, these two functions are implemented together in this example.

2.3.1 Target Speed Requirements

Some applications, such as hair-dryers, require setting the motor target speed dynamically based on instantaneous DC bus voltage level. Take an application that uses a 6 pole PMSM type motor whose maximum speed is 20K RPM as an example: given the 2-level speed selection interface described in Section 2.1, the relationship between the target speed and DC bus voltage is defined by a quadratic function with 2 different sets of coefficients for HIGH SPEED and LOW SPEED levels respectively as shown below.

$$TargetSpeed_{HS} = A_h \cdot {V_{DCBus}}^2 + B_h \cdot V_{DCBus} + C_h\,;$$

$$TargetSpeed_{LS} = A_l \cdot {V_{DCBus}}^2 + B_l \cdot V_{DCBus} + C_l\,.$$

The unit of $TargetSpeed$ is RPM, and the unit of V_{DCBus} is Volt. The coefficients of the quadratic function are listed in Table 4.

Table 4 Coefficients of the Quadratic Function for Target Speed & DC Bus Voltage Relationships

LOW SPEED		HIGH SPEED	
$\boldsymbol{A_l}$	-0.572	$\boldsymbol{A_h}$	-0.159
$\boldsymbol{B_l}$	228.480	$\boldsymbol{B_h}$	132.585
$\boldsymbol{C_l}$	-6153.675	$\boldsymbol{C_h}$	1494.450

The calculated target speed using the aforementioned quadratic function needs to be within its corresponding maximum and minimum limits. Table 5 below lists the speed limit requirements for HIGH SPEED and LOW SPEED levels.

Table 5 Max. & Min. Target Speed Limit Definitions

	LOW SPEED	HIGH SPEED
Max. Target Speed Limit	16200 RPM	19400 RPM
Min. Target Speed Limit	11625 RPM	13537 RPM

In addition, DC bus brown-out protection is required to prevent the motor from continuing to operate when the DC bus voltage decreases below a certain threshold. In order to eliminate potential oscillation when the DC bus voltage is around the brown-out level, a hysteresis of 5 V was introduced. Table 6 lists the DC bus brown-in and brown-out voltage levels.

Table 6 DC Bus Brown-In & Brown-Out Voltage Levels

DC Bus Brown-In Voltage	90 V
DC Bus Brown-Out Voltage	85 V

The overall relationships between the target speed and the DC bus voltage for HIGH SPEED and LOW SPEED levels are shown in Figure 19 and Figure 20. The solid line shows that if the DC bus voltage rises from 0 V, the motor won't start to run until the DC bus voltage exceeds 90 V. The dashed line shows that if the motor is currently running, then it doesn't stop running until the DC bus voltage falls below 85 V.

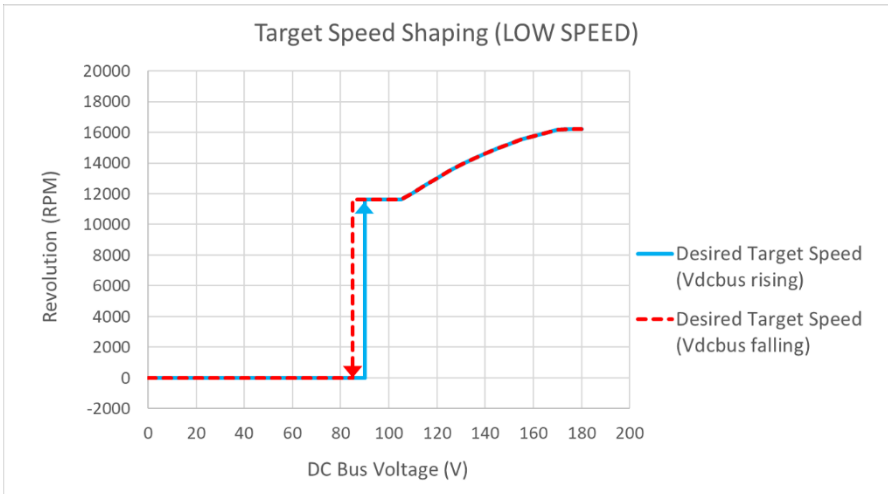


Figure 19 Target Speed Shaping (LOW SPEED)

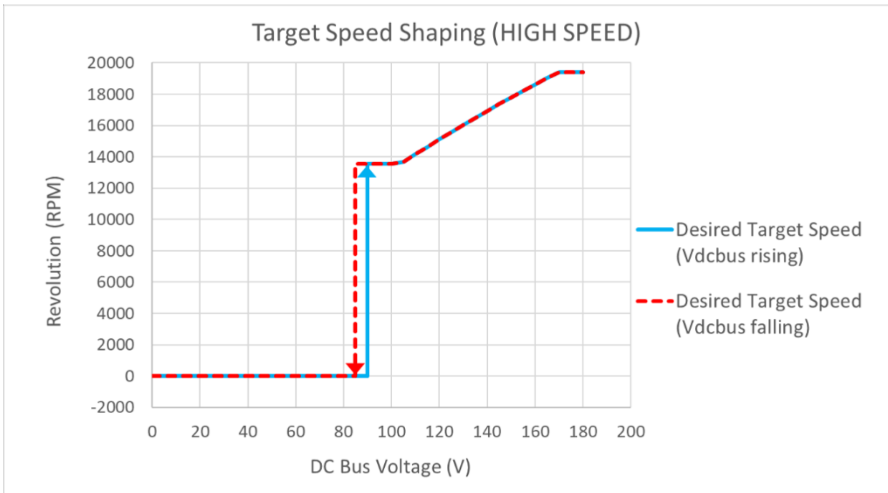


Figure 20 Target Speed Shaping (HIGH SPEED)

2.3.2 DC Bus Status State Machine

A dedicated state machine can be designed to keep track of DC Bus brown-in / brown-out status as shown in Figure 21. The DC bus status state machine uses a state variable `DCBusState` to represent 2 possible states, namely, DC_Bus_State_Abnormal (`DCBusState = 0`), and DC_Bus_State_Normal (`DCBusState = 1`). The input signal for this state machine needs to be an averaged DC bus voltage ADC conversion result to minimize potential oscillation. The LPF described in Section 2.2 can be used to generate the required input signal `VDCBusLPF`. Starting off in DC_Bus_State_Abnormal state, if `VDCBusLPF` is greater than the value of `VDCBusBrownIn`, then the DC bus status state machine shifts to DC_Bus_State_Normal state. While it is in DC_Bus_State_Normal state, if `VDCBusLPF` becomes less than the value of `VDCBusBrownOut`, then the state machine shifts back to DC_Bus_State_Abnormal state.

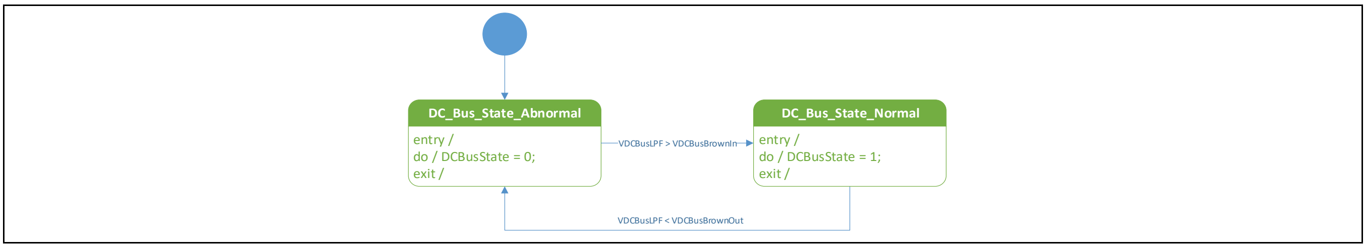


Figure 21 DC Bus Status State Machine Diagram

The calculation of the value of `VDCBusBrownIn` and `VDCBusBrownOut` follows the conversion formula described in Section 2.2.2 (voltage sense range is 757 V), and the voltage levels specified in Table 6. The results are shown in Table 7 below.

Table 7

DC Bus Brown-In Voltage	90 V	`VDCBusBrownIn`	487 (ADC counts)
DC Bus Brown-Out Voltage	85 V	`VDCBusBrownOut`	460 (ADC counts)

2.3.3 Scaling for Target Speed Shaping Calculation

Section 2.3.1 defined the relationship between $TargetSpeed$ and V_{DCBus} for different speed selection levels, where the unit of $TargetSpeed$ was RPM, and the unit of V_{DCBus} was Volt. However, in the MCE software, the target speed is represented by a signed 16-bit integer, where 16383 corresponds to the motor's maximum speed. For this application, `TargetSpeed = 16383` corresponds to the maximum motor speed of 20K RPM. The DC bus voltage in the MCE software is presented by its corresponding ADC sampling result in ADC counts, following the conversion formula described in Section 2.2.2. Thus, the formulas defined in Section 2.3.1 cannot be used directly in script code. In order to obtain the correct calculation result of target speed in script code, those scaling factors need to be taken into consideration as shown in the following formula.

$TargetSpeed_{script} = \left[A \cdot \left(V_{DCBus_{ADC}} \cdot \frac{V_{ref}}{2^{12}-1} \cdot \frac{1}{G_{DCBus_{sensing}}}\right)^2 + B \cdot \left(V_{DCBus_{ADC}} \cdot \frac{V_{ref}}{2^{12}-1} \cdot \frac{1}{G_{DCBus_{sensing}}}\right) + C\right] \cdot \frac{16383}{Speed_{max}}$, where $V_{ref} = 5V$, $G_{DCBus\_sensing} = 0.00661$ (described in Section 2.2.2), $Speed_{max} = 20000$, A, B, and C are the 3 coefficients in the original quadratic function that defines the relationship between the target speed and DC bus voltage for different speed selection levels.

If we define $T_{spd\_factor} = \frac{16383}{Speed_{max}} = 0.819$, and $V_{DCBus\_factor} = \frac{V_{ref}}{2^{12}-1} \cdot \frac{1}{G_{DCBus_{sensing}}} = 0.185$, then we can obtain the following formula:

$$TargetSpeed_{script} = (A \cdot T_{spd\_factor} \cdot V_{DCBus\_factor}{}^2) \cdot V_{DCBus_{ADC}}{}^2 + (B \cdot T_{spd\_factor} \cdot V_{DCBus\_factor}) \cdot V_{DCBus_{ADC}} + (C \cdot T_{spd\_factor}).$$

If we define $A_{script} = A \cdot T_{spd\_factor} \cdot V_{DCBus\_factor}{}^2$, $B_{script} = B \cdot T_{spd\_factor} \cdot V_{DCBus\_factor}$, and $C_{script} = C \cdot T_{spd\_factor} \cdot V_{DCBus\_factor}$, then the above formula can be simplified as follows.

$$TargetSpeed_{script} = A_{script} \cdot V_{DCBus_{ADC}}{}^2 + B_{script} \cdot V_{DCBus_{ADC}} + C_{script}.$$

Using this formula, the relevant coefficients with the inclusion of the scaling factors can be calculated for different speed selection levels as shown in Table 8.

Table 8 Coefficients in Floating Point Format for the Quadratic Function for Target Speed & DC Bus Voltage Relationships with Scaling Factors

LOW SPEED		HIGH SPEED	
A_{l_script}	-0.016	A_{h_script}	-0.004
B_{l_script}	34.593	B_{h_script}	20.074
C_{l_script}	-5040.783	C_{h_script}	1224.179

The script engine only supports 32-bit signed integer type of variables [2], so these floating-point numbers have to be represented in fractional format in the script code. For instance, if we choose a common denominator `DEN`, then the target speed shaping calculation in the script can be realized by using the following pseudo code in Code Listing 9.

Code Listing 9 Target Speed Shaping Calculation Pseudo Code

```
Speed_Value = A_NUM * VDCBus * VDCBus + B_NUM * VDCBus +
  C_NUM;
TargetSpeed = Speed_Value / DEN;
```

Considering the accuracy requirement and overflow limit, we chose a common denominator of 65536 (Q15.16 format), with which the division operation can be replaced by efficient right shifting 16 bits. With that, the numerator value for each coefficient can be calculated accordingly as shown in Table 9.

Table 9 Coefficients in Q15.16 Format for the Quadratic Function for Target Speed & DC Bus Voltage Relationships with Scaling Factors

Denominator		65536	
LOW SPEED		HIGH SPEED	
A_{l_NUM}	-1049	A_{h_NUM}	-291
B_{l_NUM}	2267065	B_{h_NUM}	1315558
C_{l_NUM}	-330352746	C_{h_NUM}	80227776

2.3.4 Target Speed Shaping & Brown-out Protection Script Implementation

Code Listing 10, Code Listing 11, Code Listing 12, and Code Listing 13 show the source code for the target speed shaping with brown-out protection application implemented in Task 1 for the MCEWizard/MCEDesigner and iSD, respectively. Since the target speed doesn't need to be updated too frequently, it is recommended to set the loop execution period of Task 1 to be 50 ms. The compiled script object file shows that the number of instructions for Task 1 is 42. So, the execution step for Task 1 should be set to greater than 42 to ensure that the entire loop of Task 1 is completed during each execution period. In this example, the execution period for Task 1 (`SCRIPT_TASK1_EXECUTION_PERIOD`) was set to 5, and the execution step for Task 1 (`SCRIPT_TASK1_EXECUTION_STEP`) was chosen to be 50 to meet the desired timing requirement.

This example can also be implemented in Task 0, in which case the execution period for Task 0 (`SCRIPT_TASK0_EXECUTION_PERIOD`) should be set to 50 to achieve the same execution period of 50 ms.

Note: When using a negative value for the CONST parameter in the MCEWizard/MCEDesigner, it must be defined in hexadecimal format. If it is defined in decimal format, then, the minus sign will be ignored and the parameter will be recognized as a positive integer. In the iSD, both decimal and hexadecimal format can be used for the negative value CONST parameter.

2.3.4.1 Script Code for MCEWizard/MCEDesigner

Code Listing 10 Target Speed Shaping & Brown-out Protection Script Code for MCEWizard/MCEDesigner

```
/* ******************************************************* */
/* Script user version value, should be 255.255 */
#SET SCRIPT_USER_VERSION (1.00)
/* Script execution time for Task0 in mS, maximum value 65535 */
#SET SCRIPT_TASK0_EXECUTION_PERIOD (1)
/* Defines number of lines to be executed every 1mS in Task0 */
#SET SCRIPT_TASK0_EXECUTION_STEP (2)
/* Script execution time for Task1 in 10mS, maximum value 65535 */
#SET SCRIPT_TASK1_EXECUTION_PERIOD (5)
/* Defines number of lines to be executed every 10mS in Task1 */
#SET SCRIPT_TASK1_EXECUTION_STEP (50)
/* ******************************************************* */
/* constant definition */
CONST int VDCBusBrownIn = 487; /* Vdcbus_brown_in = 90V => 487 counts */
CONST int VDCBusBrownOut = 460; /* Vdcbus_brown_out = 85V => 460 counts */

CONST int VLSStart = 819; /* Vsp_low_spd_start = 1V => 819 counts */
CONST int VLSStop = 655; /* Vsp_low_spd_stop = 0.8V => 655 counts */
CONST int VHSStart = 1638; /* Vsp_high_spd_start = 2V => 1638 counts */
CONST int VHSStop = 1474; /* Vsp_high_spd_stop = 1.8V => 1474 counts */

CONST int AlNum = 0xFFFFFBE7; /* -0.016 * 2^16 = -1049 */
CONST int BlNum = 0x002297B9; /* 34.593 * 2^16 = 2267065 */
CONST int ClNum = 0xEC4F3796; /* -5040.783 * 2^16 = -330352746 */
CONST int AhNum = 0xFFFFFEDD; /* -0.004 * 2^16 = -291 */
CONST int BhNum = 0x001412E6; /* 20.074 * 2^16 = 1315558 */
CONST int ChNum = 0x04C82DC0; /* 1224.179 * 2^16 = 80227776 */
CONST int DenShiftBit = 16; /* Denominator = 2^16 = 65536 */

CONST int TspdLSMin = 9523; /* In LOW_SPEED mode, Target Speed min = 11625 rpm => 9523 counts. */
CONST int TspdLSMax = 13270; /* In LOW_SPEED mode, Target Speed max = 16200 rpm => 13270 counts. */
CONST int TspdHSMin = 11089; /* In HIGH_SPEED mode, Target Speed min = 13537 rpm => 11089 counts. */
CONST int TspdHSMax = 15892; /* In HIGH_SPEED mode, Target Speed max = 19400 rpm => 15892 counts. */

/* Global variable definition */
```

Code Listing 10 Target Speed Shaping & Brown-out Protection Script Code for MCEWizard/MCEDesigner

```
int VDCBusLPF;
int DCBusState;
int SpeedMode;
int SpeedValue;

/* Task0 init function */
Script_Task0_init()
{
  /* Initialize global variable */
  VDCBusLPF = 0;
  /* local variable definition */
  int VDCBusMultiplyDEN;
  /* Initialize local variable */
  VDCBusMultiplyDEN = 0;
}

/* Task0 script function */
Script_Task0()
{
  /* Vdcbus filtering */
  VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt - VDCBusLPF);
  VDCBusLPF = VDCBusMultiplyDEN >> 6;
}

/* Task1 init function */
Script_Task1_init()
{
  /* Initialize global variable */
  DCBusState = 0;
  SpeedMode = 0;
  SpeedValue = 0;

  /* local variable definition */

  /* Initialize local variable */
}

/* Task1 script function */
Script_Task1()
{
  /* DC bus state machine */
  if (DCBusState == 0) /* DCBus is abnormal. */
    {
      if (VDCBusLPF > VDCBusBrownIn)
        {
          DCBusState = 1; /* Shift to DCBus normal state. */
        }
    }

  if (DCBusState == 1) /* DCBus is normal. */
    {
```

Code Listing 10 **Target Speed Shaping & Brown-out Protection Script Code for MCEWizard/MCEDesigner**

```
                if (VDCBusLPF < VDCBusBrownOut)
                  {
                    DCBusState = 0; /* Shift to DCBus abnormal state. */
                  }
              }

            /* Speed selection state machine */
            if (SpeedMode == 0) /* Speed selection is in OFF state.
                                  {
                                  TargetSpeed = 0;
                                  Command = 0; /* Stop the motor. */

              if (ADC_Result0 > VLSStart)
                {
                  SpeedMode = 1; /* Shift to LOW_SPEED state. */
                }
          }

          if (SpeedMode == 1) /* Speed selection is in LOW_SPEED state. */
            {
              if (ADC_Result0 > VHSStart)
                {
                  SpeedMode = 2; /* Shift to HIGH_SPEED state. */
                }
              else
                {
                  if (ADC_Result0 < VLSStop)
                    {
                      SpeedMode = 0; /* Shift to OFF state. */
                    }
                  else /* Stay in LOW_SPEED state. */
                    {
                      if (DCBusState == 1) /* DC bus voltage is normal. */
                        {
                          /* Calculate target speed. Target speed follows 2nd order polynomial curve for LS. */
                          SpeedValue = AlNum * VDCBusLPF * VDCBusLPF + BlNum * VDCBusLPF + ClNum;
                          SpeedValue = SpeedValue >> DenShiftBit;
                          if (SpeedValue > TspdLSMax) /* Upper limit check */
                            {
                              SpeedValue = TspdLSMax;
                            }
                          if (SpeedValue < TspdLSMin) /* Lower limit check */
                            {
                              SpeedValue = TspdLSMin;
                            }
```

Code Listing 10 Target Speed Shaping & Brown-out Protection Script Code for MCEWizard/MCEDesigner

```
                    TargetSpeed = SpeedValue; /* Update TargetSpeed. */
                    Command = 1; /* Start motor. */
                  }
                else /* DC bus voltage is abnormal. */
                  {
                    TargetSpeed = 0; /* Reset TargetSpeed. */
                    Command = 0; /* Stop motor. */
                  }
              }
          }
      }

    if(SpeedMode == 2) /* Speed selection is in HIGH_SPEED state. */
      {
        if(ADC_Result0 < VHSStop)
          {
            SpeedMode = 1; /* Shift to LOW_SPEED state. */
          }
        else /* Stay in HIGH_SPEED state. */
          {
            if (DCBusState == 1) /* DC bus voltage is normal. */
              {
                /* Target speed follows 2nd order polynomial curve for HS. */
                SpeedValue = AhNum * VDCBusLPF * VDCBusLPF + BhNum * VDCBusLPF + ChNum;
                SpeedValue = SpeedValue >> DenShiftBit;
                if (SpeedValue > TspdHSMax) /* Upper limit check */
                  {
                    SpeedValue = TspdHSMax;
                  }
                if (SpeedValue < TspdHSMin) /* Lower limit check */
                  {
                    SpeedValue = TspdHSMin;
                  }
                TargetSpeed = SpeedValue; /* Update TargetSpeed. */
                Command = 1; /* Start motor. */
              }
            else /* DC bus voltage is abnormal. */
              {
                TargetSpeed = 0; /* Reset TargetSpeed. */
                Command = 0; /* Stop motor. */
              }
          }
      }
    }
```

2.3.4.2 Script Code for iSD

Code Listing 11 Target Speed Shaping & Brown-out Protection Script Code for iSD(Global.mcs)

```
/* ******************************************************** */
/* Global variables */
/* ******************************************************** */
/* constant definition */
CONST int VDCBusBrownIn = 487; /* Vdcbus_brown_in = 90V => 487 counts */
CONST int VDCBusBrownOut = 460; /* Vdcbus_brown_out = 85V => 460 counts */

CONST int VLSStart = 819; /* Vsp_low_spd_start = 1V => 819 counts */
CONST int VLSStop = 655; /* Vsp_low_spd_stop = 0.8V => 655 counts */
CONST int VHSStart = 1638; /* Vsp_high_spd_start = 2V => 1638 counts */
CONST int VHSStop = 1474; /* Vsp_high_spd_stop = 1.8V => 1474 counts */

CONST int AlNum = 0xFFFFFBE7; /* -0.016 * 2^16 = -1049 */
CONST int BlNum = 0x002297B9; /* 34.593 * 2^16 = 2267065 */
CONST int ClNum = 0xEC4F3796; /* -5040.783 * 2^16 = -330352746 */
CONST int AhNum = 0xFFFFFEDD; /* -0.004 * 2^16 = -291 */
CONST int BhNum = 0x001412E6; /* 20.074 * 2^16 = 1315558 */
CONST int ChNum = 0x04C82DC0; /* 1224.179 * 2^16 = 80227776 */
CONST int DenShiftBit = 16; /* Denominator = 2^16 = 65536 */

CONST int TspdLSMin = 9523; /* In LOW_SPEED mode, Target Speed min = 11625 rpm => 9523 counts. */
CONST int TspdLSMax = 13270; /* In LOW_SPEED mode, Target Speed max = 16200 rpm => 13270 counts. */
CONST int TspdHSMin = 11089; /* In HIGH_SPEED mode, Target Speed min = 13537 rpm => 11089 counts. */
CONST int TspdHSMax = 15892; /* In HIGH_SPEED mode, Target Speed max = 19400 rpm => 15892 counts. */

/* Global variable definition */
int VDCBusLPF;
int DCBusState;
int SpeedMode;
int SpeedValue;
```

Code Listing 12 Target Speed Shaping & Brown-out Protection Script Code for iSD(Script_Task0.mcs)

```
/* ******************************************************** */
/* Task0 init function */
Script_Task0_init()
{
  /* Initialize global variable */
```

Code Listing 12 Target Speed Shaping & Brown-out Protection Script Code for iSD(Script_Task0.mcs)

```
    VDCBusLPF = 0;
    /* local variable definition */
    int VDCBusMultiplyDEN;
    /* Initialize local variable */
    VDCBusMultiplyDEN = 0;
  }

  /* ********************************************************* */
  /* Task0 script function */
  Script_Task0()
  {
    /* Vdcbus filtering */
    VDCBusMultiplyDEN = VDCBusMultiplyDEN + (FB_MEASURE.VdcFilt - VDCBusLPF);
    VDCBusLPF = VDCBusMultiplyDEN >> 6;
  }
```

Code Listing 13 Target Speed Shaping & Brown-out Protection Script Code for iSD(Script_Task1.mcs)

```
  /* ********************************************************* */
  /* Task1 init function */
  Script_Task1_init()
  {
    /* Initialize global variable */
    DCBusState = 0;
    SpeedMode = 0;
    SpeedValue = 0;

    /* local variable definition */

    /* Initialize local variable */
  }

  /* ********************************************************* */
  /* Task1 script function */
  Script_Task1()
  {
    /* DC bus state machine */
    if (DCBusState == 0) /* DCBus is abnormal. */
      {
        if (VDCBusLPF > VDCBusBrownIn)
          {
            DCBusState = 1; /* Shift to DCBus normal state. */
          }
      }
    if (DCBusState == 1) /* DCBus is normal. */
      {
        if (VDCBusLPF < VDCBusBrownOut)
          {
            DCBusState = 0; /* Shift to DCBus abnormal state. */
          }
```

Code Listing 13 Target Speed Shaping & Brown-out Protection Script Code for iSD(Script_Task1.mcs)

```
          }

        /* Speed selection state machine */
        if (SpeedMode == 0) /* Speed selection is in OFF state. */
          {
            APP_MOTOR0.TargetSpeed = 0;
            APP_MOTOR0.Command = 0; /* Stop the motor. */

            if (FB_ADC.adc_result[0] > VLSStart)
              {
                SpeedMode = 1; /* Shift to LOW_SPEED state. */
              }
          }
        if (SpeedMode == 1) /* Speed selection is in LOW_SPEED state. */
          {
            if (FB_ADC.adc_result[0] > VHSStart)
              {
                SpeedMode = 2; /* Shift to HIGH_SPEED state. */
              }
            else
              {
                if (FB_ADC.adc_result[0] < VLSStop)
                  {
                    SpeedMode = 0; /* Shift to OFF state. */
                  }
                else /* Stay in LOW_SPEED state. */
                  {
                    if (DCBusState == 1) /* DC bus voltage is normal. */
                      {
                        /* Calculate target speed. Target speed follows 2nd order polynomial curve for LS. */
                        SpeedValue = (AlNum * VDCBusLPF * VDCBusLPF) + (BlNum * VDCBusLPF) + ClNum;
                        SpeedValue = SpeedValue >> DenShiftBit;
                        if (SpeedValue > TspdLSMax) /* Upper limit check */
                          {
                            SpeedValue = TspdLSMax;
                          }
                        if (SpeedValue < TspdLSMin) /* Lower limit check */
                          {
                            SpeedValue = TspdLSMin;
                          }
                        APP_MOTOR0.TargetSpeed = SpeedValue; /* Update APP_MOTOR0.TargetSpeed. */
                        APP_MOTOR0.Command = 1; /* Start motor. */
                      }
                    else /* DC bus voltage is abnormal. */
                      {
```

Code Listing 13 Target Speed Shaping & Brown-out Protection Script Code for iSD(Script_Task1.mcs)

```
             APP_MOTOR0.TargetSpeed = 0; /* Reset APP_MOTOR0.TargetSpeed. */
             APP_MOTOR0.Command = 0; /* Stop motor. */
            }
          }
        }
      }
     if(SpeedMode == 2)
      { /* Speed selection is in HIGH_SPEED state. */
       if(FB_ADC.adc_result[0] < VHSStop)
        {
         SpeedMode = 1; /* Shift to LOW_SPEED state. */
        }
       else
        { /* Stay in HIGH_SPEED state. */
         if (DCBusState == 1)/* DC bus voltage is normal. */
          {
           /* Target speed follows 2nd order polynomial curve for HS. */
           SpeedValue = (AhNum * VDCBusLPF * VDCBusLPF) + (BhNum * VDCBusLPF) + ChNum;
           SpeedValue = SpeedValue >> DenShiftBit;
           if (SpeedValue > TspdHSMax) /* Upper limit check */
            {
             SpeedValue = TspdHSMax;
            }
           if (SpeedValue < TspdHSMin) /* Lower limit check */
            {
             SpeedValue = TspdHSMin;
            }
           APP_MOTOR0.TargetSpeed = SpeedValue; /* Update APP_MOTOR0.TargetSpeed. */
           APP_MOTOR0.Command = 1; /* Start motor. */
          }
         else /* DC bus voltage is abnormal. */
          {
           APP_MOTOR0.TargetSpeed = 0; /* Reset APP_MOTOR0.TargetSpeed. */
           APP_MOTOR0.Command = 0; /* Stop motor. */
          }
        }
      }
    }
```

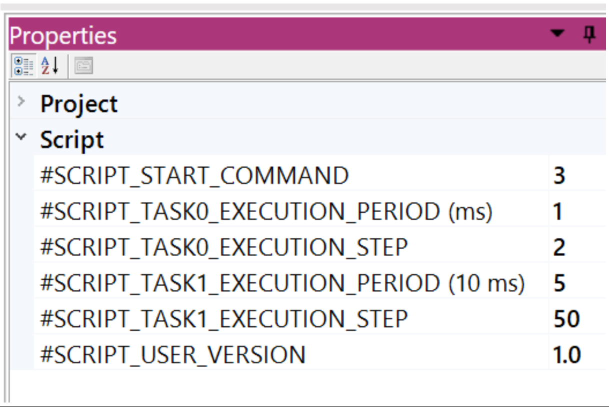


Figure 22 The execution period and the step for Target Speed Shaping Brown-out Protection Script Code

To migrate the code from MCEWizard/MCEDesigner(Code Listing 10) to iSD (Code Listing 11), the variable names should be modified as shown in the Table 10.

Table 10 Parameter name Differences between MCEWizard/MCEDesigner and iSD

MCEWizard/MCEDesigner	iSD
VdcFilt	FB_MEASURE.VdcFilt
TargetSpeed	APP_MOTOR0.TargetSpeed
Command	APP_MOTOR0.Command
ADC_Result0	FB_ADC.adc_result[0]

2.3.5 Target Speed Shaping Measurement Results

The actual motor speed was measured by calculating the frequency of the motor phase current waveforms while the input voltage was swept from 65 VAC to 130 VAC at different speed selection levels. The measurement data for the LOW SPEED level was shown in Table 11 and plotted against the desired target speed shaping curves in Figure 23. As can be seen from the measurement data, the actual motor speed followed the desired target speed calculated as a quadratic function of the DC bus voltage with a tolerance of no more than 1 %. The calculated speed was limited by either the pre-defined minimum or maximum motor speed for the LOW SPEED level.

Table 11 Measurement Data of Target Speed & DC Bus Voltage (LOW SPEED)

V_{in} (Vrms)	V_{DCbus} (Vdc)	Measured Motor Speed (rpm)	Calculated Target Speed (rpm)	Target Speed Error (%)
64	85.8	11680	11625	0.5 %
67	89.6	11740	11625	1.0 %
78	105.6	11680	11590	0.8 %
80	108.4	11900	11890	0.1 %
90	122.0	13216	13204	0.1 %
100	135.9	14380	14329	0.4 %
110	149.9	15300	15238	0.4 %
120	164.0	15980	15926	0.3 %
125	171.1	16260	16188	0.4 %
126	172.6	16264	16200	0.4 %

V_{in} (Vrms)	V_{DCbus} (Vdc)	Measured Motor Speed (rpm)	Calculated Target Speed (rpm)	Target Speed Error (%)
64	85.8	11680	11625	0.5 %
67	89.6	11740	11625	1.0 %
130	178.3	16264	16200	0.4 %

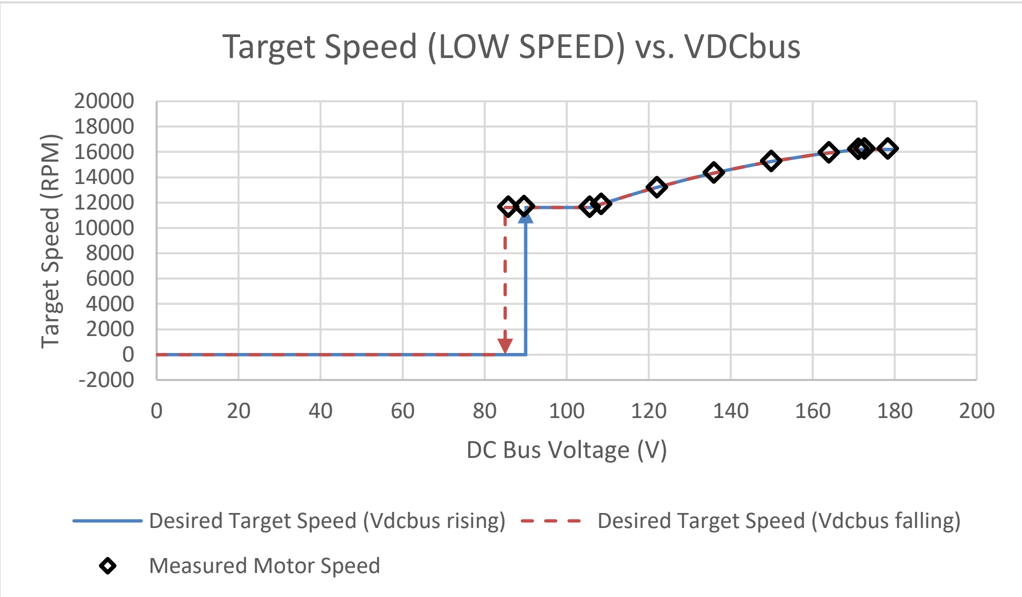


Figure 23 Measurement of Target Speed vs. DC Bus Voltage (LOW SPEED)

Table 12 and Figure 24 show the measurement data for HIGH SPEED level. It can be seen consistently that the actual motor speed followed the desired target speed calculated as a quadratic function of DC bus voltage with tolerance of no more than 1 %. The calculated speed was limited by either the pre-defined minimum or maximum motor speed for HIGH SPEED level.

Table 12 Measurement Data of Target Speed & DC Bus Voltage (HIGH SPEED)

V_{in} (Vrms)	V_{DCbus} (Vdc)	Measured Motor Speed (rpm)	Calculated Target Speed (rpm)	Target Speed Error (%)
65	86.3	13100	13537	-3.2 %
66	87.9	13260	13537	-2.0 %
77	103.7	13600	13537	0.5 %
78	105.0	13628	13667	-0.3 %
90	121.4	15280	15254	0.2 %
100	135.2	16560	16519	0.2 %
110	149.0	17762	17728	0.2 %
120	162.8	18944	18875	0.4 %
124	168.5	19454	19331	0.6 %
125	170.0	19458	19400	0.3 %
130	177.2	19466	19400	0.3 %

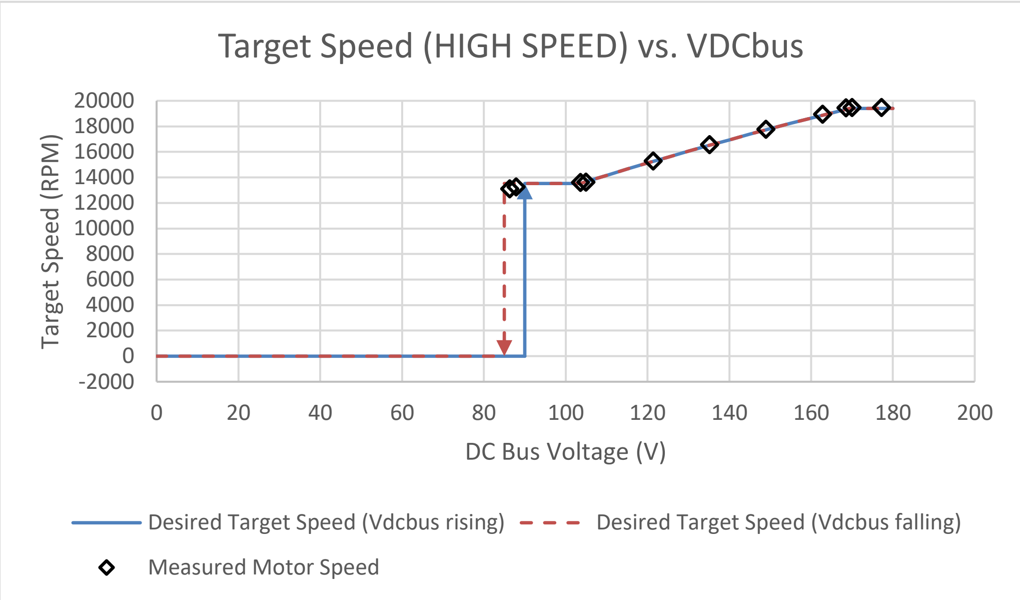


Figure 24 Measurements of Target Speed vs. DC Bus Voltage (HIGH SPEED)

2.4 Dynamic Motor Current Limit Customization

2.4.1 Motor Current Limit Requirement

By default, the motor current limit is set to 100 % of its rated current. Some applications require implementing a customized motor current limit based on the speed selection input to enable tighter torque control. During the motor speed ramp-up or ramp-down period, the motor current limit needs to be loosened up to its original setting (100 % of the rated current) momentarily to allow for quicker response to speed change request. When the motor is stopped, the motor current limit also needs to be restored to its original setting. The detailed motor current limit requirements are listed in Table 13.

Table 13 Motor Current Limit Requirements

	Rated Current	Speed Ramp-up / Ramp-down Period	Speed Selection = OFF	Speed Selection = HIGH SPEED	Speed Selection = LOW SPEED
Motor Current Limit	3 A	3 A	3 A	0.6 A	0.38 A

2.4.2 Dynamic Motor Current Limit Algorithm Design & Implementation

Figure 25 shows the detailed flowchart for dynamic motor current limit calculation algorithm. During the initialization, the original motor current limit value (`MotorLim`) is stored in a variable named `CurrentLimitOriginal`. The customized motor current limit during the steady state at a given speed selection level is updated by the speed selection state machine and is maintained by the variable named `CurrentLimitTarget`. The instantaneous motor current limit (`CurrentLimitValue`) is calculated based on the absolute difference between the `TargetSpeed` and `SpdRef`. The rate of change for `CurrentLimitValue` is set by the variable named `CurrentLimitIncrement`, which was set to 100 (counts / 10 ms) in the script code. Since the rate of change for `SpdRef` is relatively slower than that for `TargetSpeed`, if the absolute difference between `TargetSpeed` and `SpdRef` is greater than the value of `SpeedDiffThresh` (100 counts) then it is determined that the speed transient period is started. During this period the motor current limit is required to increment gradually all the way up to its original value. If the absolute difference between

`TargetSpeed` and `SpdRef` is less than 100 counts, then it implies that the speed steady state is started. During this state the motor current limit is required to decrement gradually down to its customized value represented by `CurrentLimitTarget` for a given speed selection level. The `CurrentLimitValue` calculation is updated every loop execution period (10 ms), and then `MotorLim` value is synchronized to that of `CurrentLimitValue`.

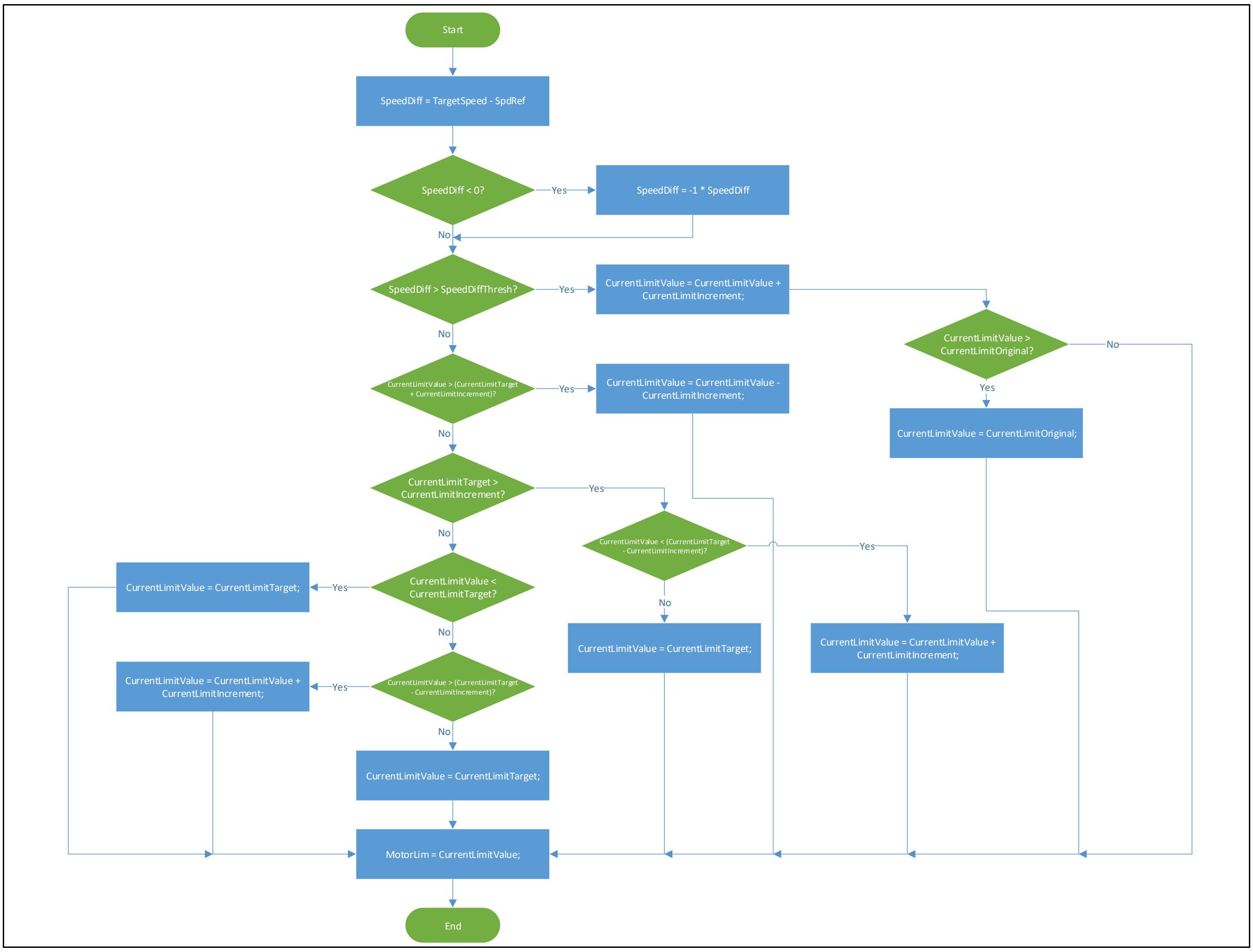


Figure 25 Flowchart of Dynamic Motor Current Limit Algorithm

Code Listing 14 and Code Listing 15 show the source code for the dynamic motor current limit customization application implemented in Task 1 for the MCEWizard/MCEDesigner and iSD, respectively. Since the rate of change for the motor current limit is defined as 100 counts / 10 ms, it is recommended to set the loop execution period of Task 1 to be 10 ms. The compiled script object file shows that the number of instructions for Task 1 is 56. With this in mind, the execution step for Task 1 should be set to greater than 56 to ensure that the entire loop of Task 1 is completed during each execution period. In this example, the execution period for Task 1 (`SCRIPT_TASK1_EXECUTION_PERIOD`) was set to 1, and the execution step for Task 1 (`SCRIPT_TASK1_EXECUTION_STEP`) was chosen to be 60 to meet the desired timing requirement.

This example can also be implemented in Task 0, in which case the execution period for Task 0 (`SCRIPT_TASK0_EXECUTION_PERIOD`) should be set to 10 to achieve the same execution period of 10 ms.

2.4.2.1 Script Code for MCEWizard/MCEDesigner

Code Listing 14 Dynamic Motor Current Limit Script Code

```
/***********************************************************/
/*Script user version value, should be 255.255*/
#SET SCRIPT_USER_VERSION (1.00)
/*Script execution time for Task0 in mS, maximum value 65535*/
#SET SCRIPT_TASK0_EXECUTION_PERIOD (1)
/*Defines number of lines to be executed every 1mS in Task0*/
#SET SCRIPT_TASK0_EXECUTION_STEP (2)
/*Script execution time for Task1 in 10mS, maximum value 65535*/
#SET SCRIPT_TASK1_EXECUTION_PERIOD (1)
/*Defines number of lines to be executed every 10mS in Task1*/
#SET SCRIPT_TASK1_EXECUTION_STEP (60)
/***********************************************************/
/* constant definition */
CONST int VDCBusBrownIn = 487; /* Vdcbus_brown_in = 90V => 487 counts */
CONST int VDCBusBrownOut = 460; /* Vdcbus_brown_out = 85V => 460 counts */
CONST int SpeedDiffThresh = 100; /* Set the speed difference threshold to 100 counts. */
CONST int CurrentLimitIncrement = 100; /* Motor current limit ramp rate = 100 counts / update interval (10 ms). */
CONST int CurrentLimitLS = 519; /* low speed motor current limit = 0.38A => 519 counts */
CONST int CurrentLimitHS = 819; /* high speed motor current limit = 0.6A => 819 counts */
CONST int VLSStart = 819; /* Vsp_low_spd_start = 1V => 819 counts */
CONST int VLSStop = 655; /* Vsp_low_spd_stop = 0.8V => 655 counts */
CONST int VHSStart = 1638; /* Vsp_high_spd_start = 2V => 1638 counts */
CONST int VHSStop = 1474; /* Vsp_high_spd_stop = 1.8V => 1474 counts */
CONST int LowSpeedValue = 5000;
CONST int HighSpeedValue = 10000;
/* Global variable definition */
int VDCBusLPF;
int DCBusState;
int SpeedDiff;
int CurrentLimitOriginal;
int CurrentLimitValue;
int CurrentLimitTarget;
int SpeedMode;
/***********************************************************/
/*Task0 init function*/
```

Code Listing 14 Dynamic Motor Current Limit Script Code

```
Script_Task0_init()
{
  /*Initialize global variable*/
  VDCBusLPF = 0;
  /* local variable definition */
  int VDCBusMultiplyDEN;
  /*Initialize local variable*/
  VDCBusMultiplyDEN = 0;
}

/*Task0 script function*/
Script_Task0()
{
  /* Vdcbus filtering*/
  VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt - VDCBusLPF);
  VDCBusLPF = VDCBusMultiplyDEN >> 6;
}

/*Task1 init function*/
Script_Task1_init()
{
  /*Initialize global variable*/
  DCBusState = 0;
  SpeedDiff = 0;
  CurrentLimitOriginal = MotorLim; /* Save the original motor current limit set in MCEWizard.*/
  CurrentLimitValue = CurrentLimitOriginal; /* The initial value needs to be synced with the original motor current limit set in MCEWizard.*/
  CurrentLimitTarget = CurrentLimitOriginal; /* The initial value needs to be synced with the original motor current limit set in MCEWizard.*/
  SpeedMode = 0;

  /*local variable definition*/

  /*Initialize local variable*/
}

/*Task1 script function*/
Script_Task1()
{
  /* DC bus state machine*/
  if (DCBusState == 0) /* DCBus is abnormal.*/
    {
      if (VDCBusLPF > VDCBusBrownIn)
        {
          DCBusState = 1; /* Shift to DCBus normal state.*/
        }
    }

  if (DCBusState == 1) /* DCBus is normal.*/
```

Code Listing 14 Dynamic Motor Current Limit Script Code

```
        {
          if (VDCBusLPF < VDCBusBrownOut)
            {
              DCBusState = 0; /* Shift to DCBus abnormal state. */
            }
        }
      /* Calculate the difference between the target speed and the speed reference in preparation for motor current limit calculation. */
      SpeedDiff = TargetSpeed - SpdRef; /* Find out the difference between the speed reference and the target speed. */
      if(SpeedDiff < 0) /* The target speed is lower than the speed reference. */
        {
          SpeedDiff = -1 * SpeedDiff; /* Takes the absolute value of SpeedDiff. */
        }
      /* Calculate motor current limit based on speed reference and target speed. */
      if(SpeedDiff > SpeedDiffThresh) /* The speed reference is more than SpeedDiffThresh counts different from the target speed. We need to increase the motor current limit to its original value temperarily. */
        {
          CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement; /* Increase the motor current limit by CurrentLimitIncrement until it reaches CurrentLimOriginal. */
          if (CurrentLimitValue > CurrentLimitOriginal) /* Upper boundary check for CurrentLimitValue. */
            {
              CurrentLimitValue = CurrentLimitOriginal;
            }
        }
      else /* The speed reference is no more than 100 counts different from the target speed. We need to decrease the motor current limit to CurrentLimitTarget. */
        {
          if(CurrentLimitValue > (CurrentLimitTarget + CurrentLimitIncrement)) /* The motor current limit value at this moment is greater than the specified motor current limit by more than CurrentLimitIncrement. */
            {
              CurrentLimitValue = CurrentLimitValue - CurrentLimitIncrement; /* Decrease the motor current limit target by CurrentLimitIncrement. */
            }
          else /* The motor current limit target is no more than the specified motor current limit by more than CurrentLimitIncrement. */
            {
              if (CurrentLimitTarget > CurrentLimitIncrement) /* CurrentLimitTarget is greater than CurrentLimitIncrement. Boundary check needed for the following minus operation. */
                {
```

Code Listing 14 Dynamic Motor Current Limit Script Code

```
                        if (CurrentLimitValue < (CurrentLimitTarget - CurrentLimitIncrement)) /* The motor current limit value at this moment is less than the specified motor current limit by more than CurrentLimitIncrement. */
                        {
                            CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement; /* Increase the motor current limit target by CurrentLimitIncrement. */
                        }
                        else /* The motor current limit target falls between CurrentLimitTarget - CurrentLimitIncrement and CurrentLimitTarget + CurrentLimitIncrement. */
                        {
                            CurrentLimitValue = CurrentLimitTarget; /* Set the motor current limit target to the specified motor current limit. */
                        }
                    }
                    else /* CurrentLimitTarget is no more than CurrentLimitIncrement. */
                    {
                        if (CurrentLimitValue < CurrentLimitTarget)
                        {
                            CurrentLimitValue = CurrentLimitTarget; /* Set the motor current limit target to the specified LOW_SPEED motor current limit. */
                        }
                        else /* CurrentLimitValue is greater than CurrentLimitTarget */
                        {
                            if(CurrentLimitValue > (CurrentLimitTarget - CurrentLimitIncrement)) /* The motor current limit value at this moment is less than the specified LOW_SPEED motor current limit by more than CurrentLimitIncrement. */
                            {
                                CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement; /* Increase the motor current limit target by CurrentLimitIncrement. */
                            }
                            else /* The motor current limit value is within the range of CurrentLimitTarget and CurrentTarget - CurrentLimitIncrement. */
                            {
                                CurrentLimitValue = CurrentLimitTarget; /* Set the motor current limit target to the specified motor current limit. */
                            }
                        }
                    }
                }
            }
        MotorLim = CurrentLimitValue; /* Update MotorLim. */

```

Code Listing 14 Dynamic Motor Current Limit Script Code

```
/* Speed selection state machine  */
if (SpeedMode == 0) /* Speed selection is in OFF state. */
  {
    TargetSpeed = 0;
    CurrentLimitTarget = CurrentLimitOriginal;
    Command = 0; /* Stop the motor. */

    if (ADC_Result0 > VLSStart)
      {
        SpeedMode = 1; /* Shift to LOW_SPEED state. */
      }
  }

if (SpeedMode == 1) /* Speed selection is in LOW_SPEED state. */
  {
    if (ADC_Result0 > VHSStart)
      {
        SpeedMode = 2; /* Shift to HIGH_SPEED state. */
      }
    else
      {
        if (ADC_Result0 < VLSStop)
          {
            SpeedMode = 0; /* Shift to OFF state. */
          }
        else /* Stay in LOW_SPEED state. */
          {
            if (DCBusState == 1) /* DC bus voltage is normal. */
              {
                TargetSpeed = LowSpeedValue; /* Update TargetSpeed. */
                CurrentLimitTarget = CurrentLimitLS;
                Command = 1; /* Start motor. */
              }
            else /* DC bus voltage is abnormal. */
              {
                TargetSpeed = 0; /* Reset TargetSpeed. */
                CurrentLimitTarget = CurrentLimitOriginal; /* When the target speed is zero, motor current limit is restored back to the original limit. */
                Command = 0; /* Stop motor. */
              }
          }
      }
  }

if(SpeedMode == 2) /* Speed selection is in HIGH_SPEED state. */
  {
    if(ADC_Result0 < VHSStop)
      {
```

Code Listing 14 Dynamic Motor Current Limit Script Code

```
                SpeedMode = 1; /* Shift to LOW_SPEED state. */
              }
            else /* Stay in HIGH_SPEED state. */
              {
                if (DCBusState == 1) /* DC bus voltage is normal. */
                  {
                    /* Target speed follows 2nd order polynomial curve for HS. */
                    TargetSpeed = HighSpeedValue; /* Update TargetSpeed. */
                    CurrentLimitTarget = CurrentLimitHS;
                    Command = 1; /* Start motor. */
                  }
                else /* DC bus voltage is abnormal. */
                  {
                    TargetSpeed = 0; /* Reset TargetSpeed. */
                    CurrentLimitTarget = CurrentLimitOriginal; /* When the target speed is zero, motor current limit is restored back to the original limit. */
                    Command = 0; /* Stop motor. */
                  }
              }
          }
      }
```

2.4.2.2 Script Code for iSD

Code Listing 15 Dynamic Motor Current Limit Script Code (Global.mcs)

```
/*************************************************************/
/*Global variables*/
/*************************************************************/
/* constant definition */
CONST int VDCBusBrownIn = 737; /* Vdcbus_brown_in = 90V => 737 counts */
CONST int VDCBusBrownOut = 696; /* Vdcbus_brown_out = 85V => 696 counts */

CONST int SpeedDiffThresh = 100; /* Set the speed difference threshold to 100 counts. */
CONST int CurrentLimitIncrement = 100; /* Motor current limit ramp rate = 100 counts / update interval (10 ms). */
CONST int CurrentLimitLS = 519; /* low speed motor current limit = 0.05A => 519 counts (12.5% of the maximum value) */
CONST int CurrentLimitHS = 819; /* high speed motor current limit = 0.08A => 819 counts (20% of the maximum value) */

CONST int VLSStart = 1240; /* 1V => 1240 counts */
CONST int VLSStop = 992; /* 0.8V => 992 counts */
CONST int VHSStart = 2480; /* 2V => 2480 counts */
CONST int VHSStop = 2232; /* 1.8V => 2232 counts */

```

Code Listing 15 Dynamic Motor Current Limit Script Code (Global.mcs)

```
CONST int LowSpeedValue = 5000; /* TargetSpeed = 5000 / 16383 * MaxSpeed(1000) = 305 rpm (31% of the maximum speed) */
CONST int HighSpeedValue = 10000; /* TargetSpeed = 10000 / 16383 * MaxSpeed(1000) = 610 rpm (61% of the maximum speed) */

/* Global variable definition */
int VDCBusLPF;
int DCBusState;
int SpeedDiff;
int CurrentLimitOriginal;
int CurrentLimitValue;
int CurrentLimitTarget;
int SpeedMode;
```

Code Listing 16 Dynamic Motor Current Limit Script Code (Script_Task0.mcs)

```
/***************************************************************/
/*Task0 init function */
Script_Task0_init()
{
  /*Initialize global variable */
  VDCBusLPF = 0;
  /* local variable definition */
  int VDCBusMultiplyDEN;
  /*Initialize local variable */
  VDCBusMultiplyDEN = 0;
}

/*Task0 script function */
Script_Task0()
{
  /* Vdcbus filtering */
  VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt - VDCBusLPF);
  VDCBusLPF = VDCBusMultiplyDEN >> 6;
}
```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
/***************************************************************/
/* Task1 init function */
Script_Task1_init() {
  /* Initialize global variable */
  DCBusState = 0;
  SpeedDiff = 0;
  CurrentLimitOriginal = FB_LIMIT_SPEED.MotorLim; /* Save the original motor current limit set in MCEWizard. */
```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
        CurrentLimitValue = CurrentLimitOriginal; /* The initial value needs to be synced with the original motor current limit set in MCEWizard. */
        CurrentLimitTarget = CurrentLimitOriginal; /* The initial value needs to be synced with the original motor current limit set in MCEWizard. */
        SpeedMode = 0;

        /* local variable definition */

        /* Initialize local variable */
      }

      /*************************************************************/
      /*Task1 script function*/
      Script_Task1() {

        /* DC bus state machine */
        /* DCBus is abnormal. */
        if (DCBusState == 0)
          {
            if (VDCBusLPF > VDCBusBrownIn)
              {
                /* Shift to DCBus normal state. */
                DCBusState = 1;
              }
          }
        /* DCBus is normal. */
        if (DCBusState == 1)
          {
            if (VDCBusLPF < VDCBusBrownOut)
              {
                /* Shift to DCBus abnormal state. */
                DCBusState = 0;
              }
          }
        /* Calculate the difference between the target speed and the speed reference in preparation for motor current limit calculation. */
        /* Find out the difference between the speed reference and the target speed. */
        SpeedDiff = APP_MOTOR0.TargetSpeed - FB_SPEEDREGULATOR.SpeedRef;

        /* The target speed is lower than the speed reference. */
        if(SpeedDiff < 0)
          {
            /* Takes the absolute value of SpeedDiff. */
            SpeedDiff = -1 * SpeedDiff;
          }
        /* Calculate motor current limit based on speed reference and target speed. */
        if(SpeedDiff > SpeedDiffThresh)
```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
           /* The speed reference is more than SpeedDiffThresh counts different from the target speed. We need to increase the motor current limit to its original value temperarily. */
           {
             /* Increase the motor current limit by CurrentLimitIncrement until it reaches CurrentLimOriginal. */
             CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement;
             /* Upper boundary check for CurrentLimitValue. */
             if (CurrentLimitValue > CurrentLimitOriginal)
               {
                 CurrentLimitValue = CurrentLimitOriginal;
               }
           }
         /* The speed reference is no more than 100 counts different from the target speed. We need to decrease the motor current limit to CurrentLimitTarget. */
         else
           {
             /* The motor current limit value at this moment is greater than the specified motor current limit by more than CurrentLimitIncrement. */
             if(CurrentLimitValue > (CurrentLimitTarget + CurrentLimitIncrement))

               {
                 /* Decrease the motor current limit target by CurrentLimitIncrement. */
                 CurrentLimitValue = CurrentLimitValue - CurrentLimitIncrement;
               }
             /* The motor current limit target is no more than the specified motor current limit by more than CurrentLimitIncrement. */
             else
               {
                 /* CurrentLimitTarget is greater than CurrentLimitIncrement. Boundary check needed for the following minus operation. */
                 if (CurrentLimitTarget > CurrentLimitIncrement)
                   {
                     /* The motor current limit value at this moment is less than the specified motor current limit by more than CurrentLimitIncrement. */
                     if (CurrentLimitValue < (CurrentLimitTarget - CurrentLimitIncrement))
                       {
                         /* Increase the motor current limit target by CurrentLimitIncrement. */
                         CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement;
                       }
```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
                    /* The motor current limit target falls between CurrentLimitTarget - CurrentLimitIncrement and CurrentLimitTarget + CurrentLimitIncrement. */
                    else
                      {
                        /* Set the motor current limit target to the specified motor current limit. */
                        CurrentLimitValue = CurrentLimitTarget;
                      }
                  }
                /* CurrentLimitTarget is no more than CurrentLimitIncrement. */
                else
                  {
                    if (CurrentLimitValue < CurrentLimitTarget)
                      {
                        /* Set the motor current limit target to the specified LOW_SPEED motor current limit. */
                        CurrentLimitValue = CurrentLimitTarget;
                      }
                    /* CurrentLimitValue is greater than CurrentLimitTarget */
                    else
                      {
                        /* The motor current limit value at this moment is less than the specified LOW_SPEED motor current limit by more than CurrentLimitIncrement. */
                        if(CurrentLimitValue > (CurrentLimitTarget - CurrentLimitIncrement))

                          {
                            /* Increase the motor current limit target by CurrentLimitIncrement. */
                            CurrentLimitValue = CurrentLimitValue + CurrentLimitIncrement;
                          }
                        /* The motor current limit value is within the range of CurrentLimitTarget and CurrentTarget - CurrentLimitIncrement. */
                        else
                          {
                            /* Set the motor current limit target to the specified motor current limit. */
                            CurrentLimitValue = CurrentLimitTarget;

                          }
                      }
                  }
              }
          }
        FB_LIMIT_SPEED.MotorLim = CurrentLimitValue;
        /* Update MotorLim. */

```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
/* Speed selection state machine */
/* Speed selection is in OFF state. */
if (SpeedMode == 0)
  {
    APP_MOTOR0.TargetSpeed = 0;
    CurrentLimitTarget = CurrentLimitOriginal;
    /* Stop the motor. */
    APP_MOTOR0.Command = 0;
    if (FB_ADC.adc_result[0] > VLSStart)
      {
        /* Shift to LOW_SPEED state. */
        SpeedMode = 1;
      }
  }
/* Speed selection is in LOW_SPEED state. */
if (SpeedMode == 1)
  {
    if (FB_ADC.adc_result[0] > VHSStart)
      {
        /* Shift to HIGH_SPEED state. */
        SpeedMode = 2;

      }
    else
      {
        if (FB_ADC.adc_result[0] < VLSStop)
          {
            /* Shift to OFF state. */
            SpeedMode = 0;
          }
        else
          /* Stay in LOW_SPEED state. */
          {
            /* DC bus voltage is normal. */
            if (DCBusState == 1)
              {
                /* Update APP_MOTOR0.TargetSpeed. */
                APP_MOTOR0.TargetSpeed = LowSpeedValue;
                CurrentLimitTarget = CurrentLimitLS;
                /* Start motor. */
                APP_MOTOR0.Command = 1;
              }
            /* DC bus voltage is abnormal. */
            else
              {
                /* Reset APP_MOTOR0.TargetSpeed. */
                APP_MOTOR0.TargetSpeed = 0;
                /* When the target speed is zero, motor current limit is restored back to the original limit. */
                CurrentLimitTarget = CurrentLimitOriginal;
                /* Stop motor. */
                APP_MOTOR0.Command = 0;
              }
```

Code Listing 17 Dynamic Motor Current Limit Script Code (Script_Task1.mcs)

```
      }
    }
  }
 /* Speed selection is in HIGH_SPEED state. */
 if(SpeedMode == 2)
  {
   if(FB_ADC.adc_result[0] < VHSStop)
    {
     /* Shift to LOW_SPEED state. */
     SpeedMode = 1;
    }
   /* Stay in HIGH_SPEED state. */
   else
    {
     /* DC bus voltage is normal. */
     if (DCBusState == 1)
      {
       /* Target speed follows 2nd order polynomial curve for HS. */
       /* Update APP_MOTOR0.TargetSpeed. */
       APP_MOTOR0.TargetSpeed = HighSpeedValue;
       CurrentLimitTarget = CurrentLimitHS;
       /* Start motor. */
       APP_MOTOR0.Command = 1;

      }
     /* DC bus voltage is abnormal. */
     else
      {
       /* Reset APP_MOTOR0.TargetSpeed. */
       APP_MOTOR0.TargetSpeed = 0;
       /* When the target speed is zero, motor current limit is restored back to the original limit. */
       CurrentLimitTarget = CurrentLimitOriginal;
       /* Stop motor. */
       APP_MOTOR0.Command = 0;
      }
    }
  }
}
```

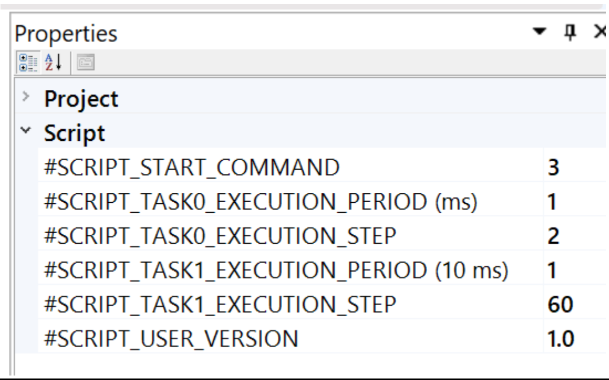


Figure 26 The execution period and the step for Dynamic Motor Current Limit Script Code

To migrate the code from MCEWizard/MCEDesigner (Code Listing 14) to iSD (Code Listing 15, Code Listing 16, and Code Listing 17), the variable names should be modified as shown in the Table 14.

Table 14 Parameter name Differences between MCEWizard/MCEDesigner and iSD

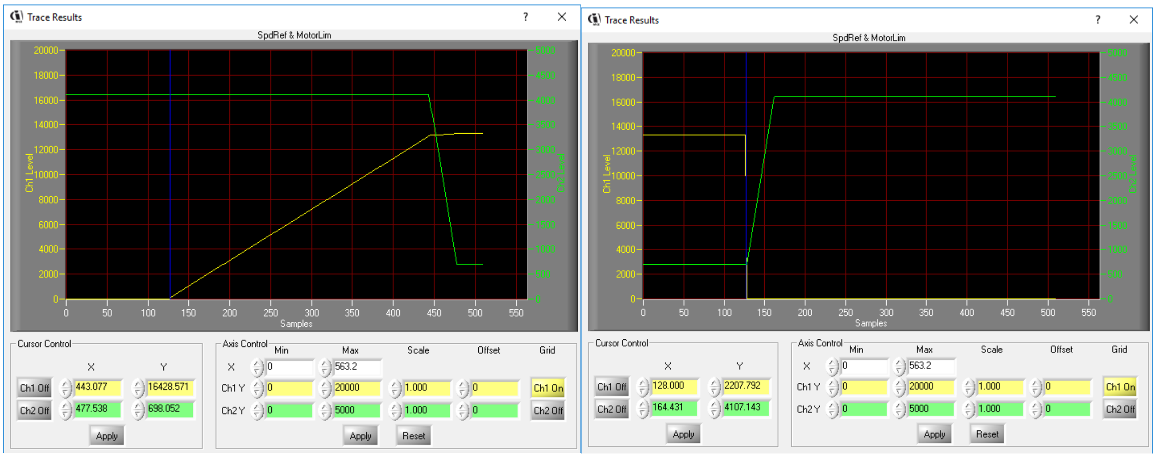
MCEWizard/MCEDesigner	iSD
VdcFilt	FB_MEASURE.VdcFilt
TargetSpeed	APP_MOTOR0.TargetSpeed
Command	APP_MOTOR0.Command
ADC_Result0	FB_ADC.adc_result[0]
MotorLim	FB_LIMIT_SPEED.MotorLim

2.4.3 Dynamic Motor Current Limit Measurement Results

Figure 27 shows how the motor current limit was dynamically changed when the speed selection changed between the OFF state and the HIGH SPEED state. When the speed selection changed from the OFF state to the HIGH SPEED state, the motor started to spin with its current limit `MotorLim` set to its original value saved in `CurrentLimitOriginal`. As the motor speed reference `SpdRef` approached its HIGH SPEED steady state target speed, the motor current limit `MotorLim` started to decrease with a rate of 100 counts / 10 ms. After about 330 ms, it reached its customized limit for the HIGH SPEED level (`CurrentLimitHS = 819`). When the speed selection changed from the HIGH SPEED state to the OFF state, the motor speed reference `SpdRef` was instantly reset, while the motor limit `MotorLim` started to ramp up with a rate of 100 counts / 10 ms, and stabilized at its original value saved in `CurrentLimitOriginal` after about 330 ms.

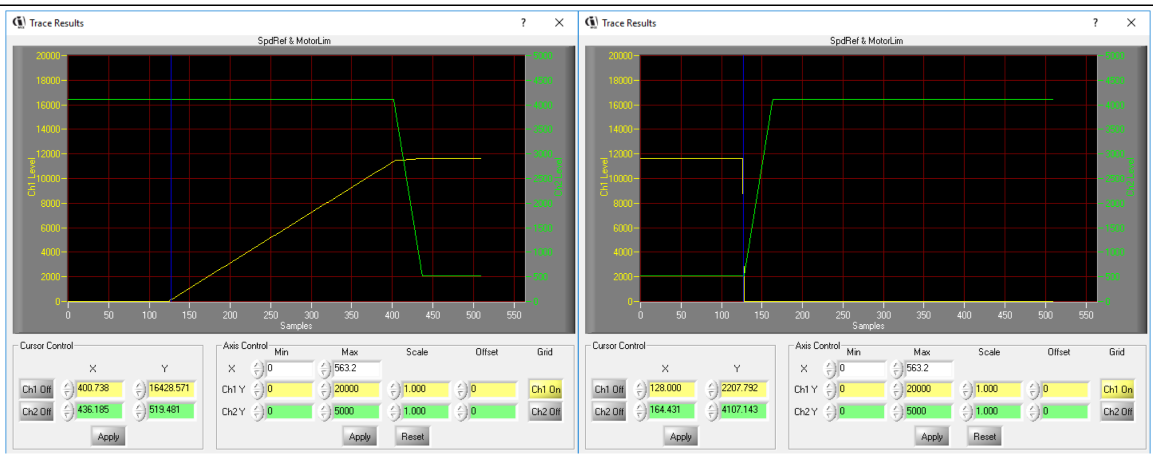
Figure 28 shows how the motor current limit was dynamically changed when the speed selection changed between the OFF state and the LOW SPEED state. When the speed selection changed from the OFF state to the LOW SPEED state, the motor started to spin with its current limit `MotorLim` set to its original value saved in `CurrentLimitOriginal`. As the motor speed reference `SpdRef` approached its LOW SPEED steady state target speed, the motor current limit `MotorLim` started to decrease with a rate of 100 counts / 10 ms. After about 360 ms, it reached its customized limit for the LOW SPEED level (`CurrentLimitLS = 519`). When the speed selection changed from the LOW SPEED state to the OFF state, the motor speed reference `SpdRef` was instantly reset, while the motor limit `MotorLim` started to ramp up with a rate of 100 counts / 10 ms, and stabilized at its original value saved in `CurrentLimitOriginal` after about 360 ms.

Figure 29 shows how the motor current limit was dynamically changed when the speed selection changed between the LOW SPEED state and the HIGH SPEED state. When the speed selection changed from the LOW SPEED state to the HIGH SPEED state, the motor speed reference `SpdRef` started to ramp up, while the motor current limit `MotorLim` started to ramp up with a rate of 100 counts / 10 ms from its customized limit for the LOW SPEED level (`CurrentLimitLS = 519`). It finally reached its original value saved in `CurrentLimitOriginal`. As soon as `SpdRef` approached its steady state HIGH SPEED target speed, the motor current limit `MotorLim` started to decrease with the same ramp rate, and eventually it was stabilized at its customized limit for the HIGH SPEED level (`CurrentLimitHS = 819`). When the speed selection changed from the HIGH SPEED state to the LOW SPEED state, the motor speed reference `SpdRef` started to ramp down, while the motor current limit `MotorLim` started to ramp up with a rate of 100 counts / 10 ms from its customized limit for HIGH SPEED level (`CurrentLimitHS = 819`). It finally reached its original value saved in `CurrentLimitOriginal`. As soon as `SpdRef` approached its steady state LOW SPEED target speed, the motor current limit `MotorLim` started to decrease with the same ramp rate, and eventually it was stabilized at its customized limit for the LOW SPEED level (`CurrentLimitLS = 519`).



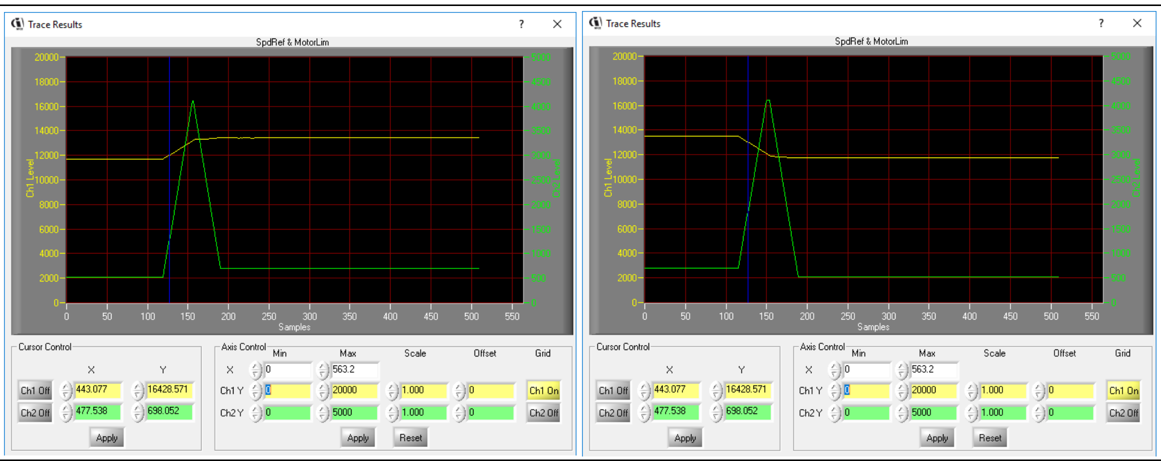
Ch1: `SpdRef`; Ch2: `MotorLim`; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 200; Trigger Level: 100; Speed Selection: OFF to HIGH SPEED (left) / HIGH SPEED to OFF (right); Vin: 100 VAC

Figure 27 Motor Current Limit Screenshots (OFF <-> HIGH SPEED)



Ch1: `SpdRef`; Ch2: `MotorLim`; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 200; Trigger Level: 100; Speed Selection: OFF to LOW SPEED (left) / LOW SPEED to OFF (right); Vin: 100 VAC

Figure 28 Motor Current Limit Screenshots (OFF <-> LOW SPEED)



Ch1: SpdRef; Ch2: MotorLim; F_{PWM}: 20 kHz; Sample at PWM frequency divided by 200; Trigger Level: 12000; Speed Selection: LOW SPEED to HIGH SPEED (left) / LOW SPEED to HIGH SPEED (right); Vin: 100 VAC

Figure 29 Motor Current Limit Screenshots (LOW SPEED <-> HIGH SPEED)

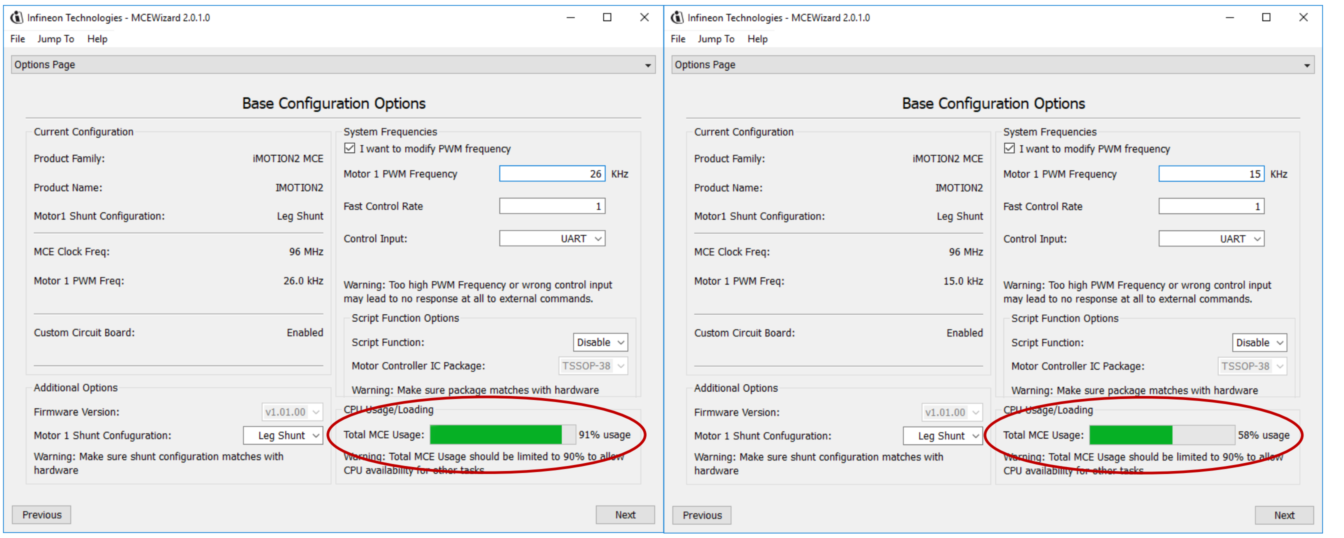
3 Script Performance Evaluation

3.1 CPU Load Evaluation

The CPU resource is prioritized for the implementation of the motor and PFC control algorithm. The script engine is designed to take advantage of the spare CPU resource for the execution of the script program. The priority of the execution of the script program is lower than that of the motor and PFC control algorithm, so that it won't affect the performance of the control algorithm. However, CPU usage needs to be carefully evaluated before enabling the script function.

3.1.1 CPU Load Evaluation Using MCEWizard/MCEDesigner

The estimated CPU usage varies depending on the configuration of the motor or PFC PWM frequency as well as the safety functions. The MCEWizard can be used to estimate the CPU usage. If the CPU usage estimation is greater than 90 %, as shown in the left screenshot of Figure 30, then enabling the script function is likely to overload the CPU. It is highly recommended to keep the CPU usage estimation at no more than 90 % when the users plan to enable the script function.

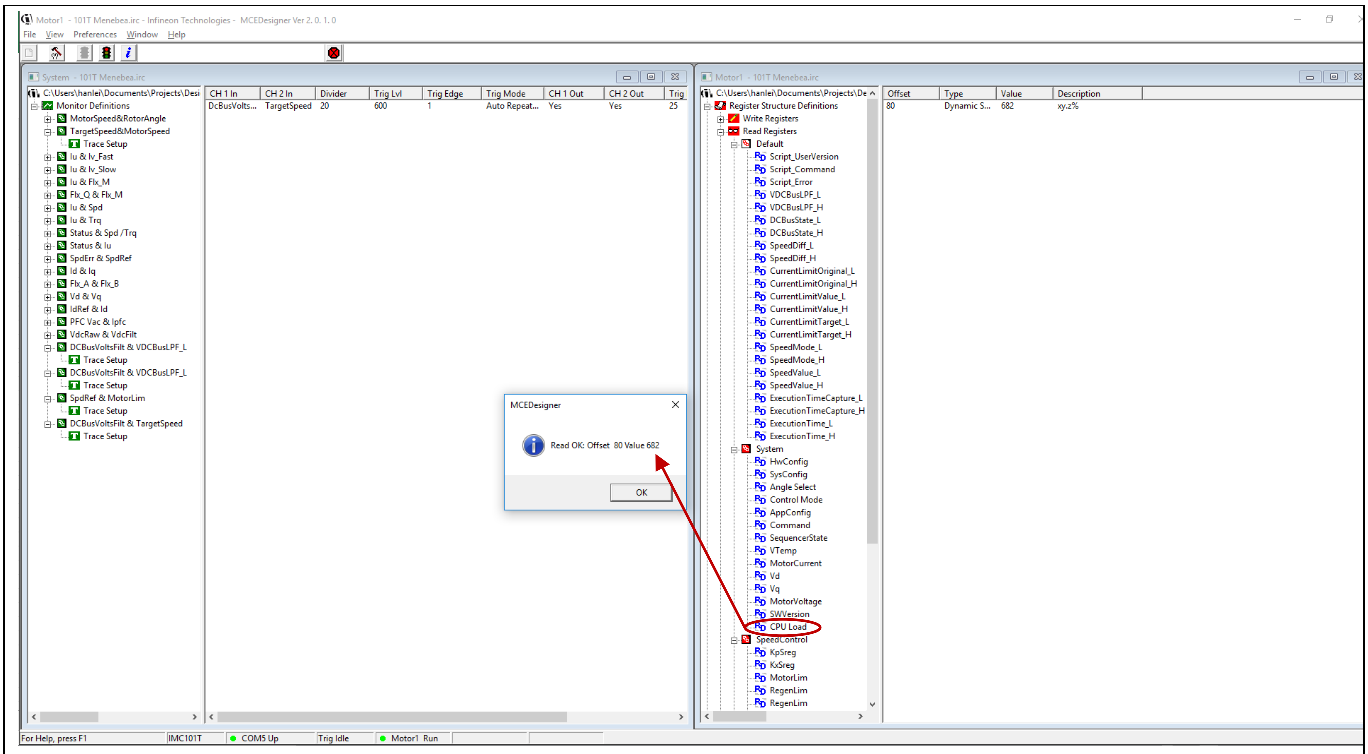


Left: CPU Usage = 91 %; Right: CPU Usage = 58 %

Figure 30 CPU Usage Estimation Using MCEWizard

The execution of the script program, depending on the complexity of the code and the configuration of the execution period and the execution step for each task, would have an impact on the CPU loading. It is recommended to evaluate the CPU load during run time with the script program enabled to ensure that the MCE is not overloaded.

The CPU load status can be obtained by reading the system parameter 'CPU Load' [2] using MCEDesigner [3]. The CPU load is represented in 0.1 % [2]. Figure 31 shows that CPU load was at 68.2 %, with the script described in Section 2.3 enabled, while the motor was running with speed selection set to LOW SPEED level. The more complicated the script code becomes, the greater CPU load it would demand.

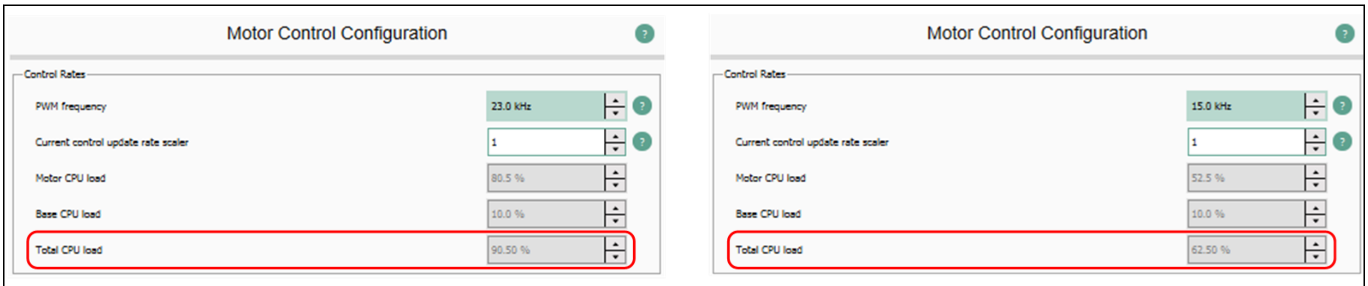


Speed Selection: LOW SPEED; Vin: 100VAC

Figure 31 Reading 'CPU Load' Parameter Using MCEDesigner When Script Is Running

3.1.2 CPU Load Evaluation Using iSD

The estimated CPU usage varies depending on the configuration of the motor or PFC PWM frequency as well as the safety functions. The Configuration Wizard in the iSD can be used to estimate the CPU usage. If the CPU usage estimation is greater than 90 % as shown in the left screenshot of Figure 32, then enabling the script function is likely to overload the CPU. It is highly recommended to keep the CPU usage estimation at no more than 90 % when the users plan to enable the script function.

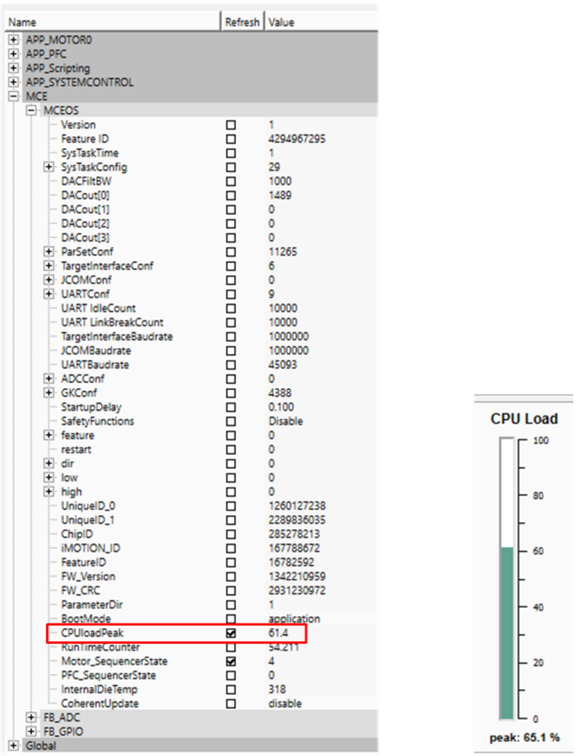


Left: CPU Usage = 90.5 %, fPWM = 23 kHz; Right: CPU Usage = 62.5 %, fPWM = 15 kHz

Figure 32 CPU Usage Estimation Using iSD

The execution of the script program, depending on the complexity of the code and the configuration of the execution period and the execution step for each task, would have an impact on the CPU loading. It is recommended to evaluate the CPU load during run time with the script program enabled to ensure that the MCE is not overloaded.

The CPU load status can be obtained by reading the system parameter 'CPUloadPeak' [5] as shown in the left screenshot of Figure 33 or the CPU load progress bar as shown in the right screenshot of Figure 33 using the iSD. The CPU load is represented in 0.1 % [5]. Figure 33 shows that 'CPUloadPeak' was at 61.4 %, with the script described in Section 2.3 enabled and with the debug mode in the iSD Script Editor, while the motor was running with speed selection set to LOW SPEED level. The more complicated the script code becomes, the greater CPU load it would demand.



Speed Selection: LOW SPEED, CPU Usage = 61.4 %, fPWM = 15 kHz

Figure 33 Reading 'CPU Load' Parameter Using iSD When Script Is Running

3.2 Script Task Timing

3.2.1 Script Task Timing Setup

The script engine supports 2 independent tasks, Task 0 and Task 1, running concurrently. Task 0 is scheduled to be executed in the system tick interrupt, which typically occurs every 1 ms. Task 1 is scheduled to be executed in the background loop task. Task 0 has greater priority than Task 1.

The user script program runs repeatedly on a configurable interval within Task 0 or Task 1 loop. The execution period of Task 0 is configurable in the script code by setting the parameter named `SCRIPT_TASK0_EXECUTION_PERIOD`. The granularity of the configurable execution period for Task 0 is 1 ms. For example, setting `SCRIPT_TASK0_EXECUTION_PERIOD` to `5` results in an execution period of $5 \cdot 1mS = 5mS$ for Task 0. The execution period of Task 1 is also configurable in the script code by setting the parameter named `SCRIPT_TASK1_EXECUTION_PERIOD`. The granularity of the configurable execution period for Task 1 is 10 ms. For example, setting `SCRIPT_TASK1_EXECUTION_PERIOD` to `5` results in an execution period of $5 \cdot 10mS = 50mS$ for Task 1.

The number of script instructions that gets executed by each task during every execution period can be configured in the script code by setting the parameter named `SCRIPT_TASK0_EXECUTION_STEP` for Task 0 or `SCRIPT_TASK1_EXECUTION_STEP` for Task 1 accordingly [2].

The actual timing setup for each script task needs to be adjusted according to the specific application requirements.

3.2.2 Script Task Execution Time Evaluation

3.2.2.1 Execution Time Evaluation Using MCEWizard/MCEDesigner

The execution time of Task 0 or Task 1 can be measured by taking advantage of the variable named `RunTimeCounter` provided by the MCE software. `RunTimeCounter` is a free running timer with 1 ms resolution that is accessible from within the script code. As shown in Code Listing 18, one can capture the value of `RunTimeCounter` at the beginning of Task 1 and save it in a variable named `ExecutionTimeCapture`. At the end of Task 1, the value of `RunTimeCounter` gets captured again and then subtracted with the value of `ExecutionTimeCapture` to obtain the execution time for Task 1 which is saved in the variable named `ExecutionTime`. As a global variable, the value of `ExecutionTime` can be read using MCEDesigner during run time.

The script program described in Section 2.4 was used as an example to evaluate execution time for Task 1, whose execution period was set to 10 ms. Figure 34 shows the value of `ExecutionTime_L` (lower 16 bit of `ExecutionTime`) was 4 with the script enabled while the motor was running with speed selection set to LOW SPEED level. This shows the loop execution time of Task 1 was about 4 ms while the motor was running. Since the actual execution time for Task 1 was shorter than the specified execution period, it indicates that Task 1 didn't overrun.

The more complicated the script code in each task becomes, the longer loop execution time it would result in. As long as the loop execution time for a script task doesn't exceed the specified loop execution period, the script task wouldn't overrun and the timing requirements can always be guaranteed. If the loop execution time for a script task exceeds the specified loop execution period, then the desired timing for this script task cannot be guaranteed. In that case, the script task will continue to finish up the on-going loop execution and then immediately start a new loop execution, in which case the actual loop execution period for this script task is determined by the loop execution time.

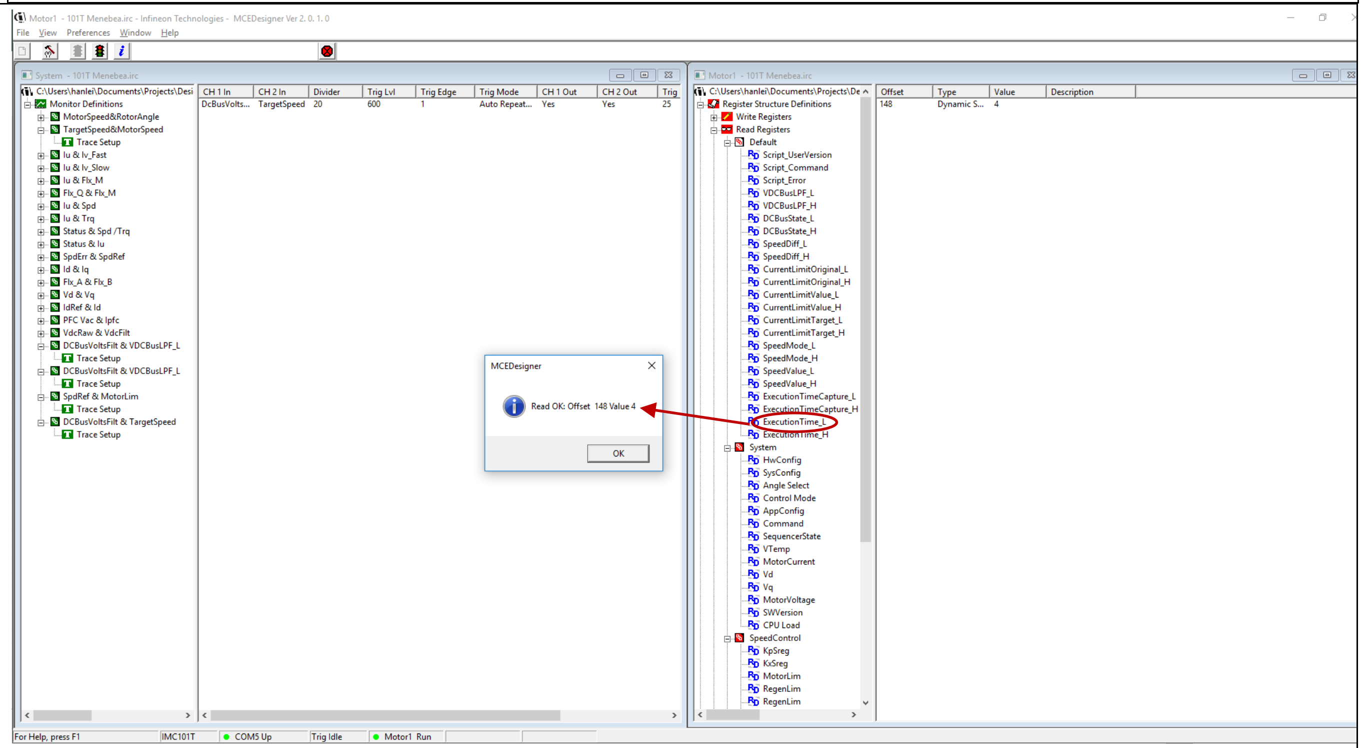
If the execution period of Task 0 is set to 1 ms, then it is not possible to use `RunTimeCounter` to estimate the execution time of Task 0 due to the resolution limit. In that case, the CPU load can be checked to indirectly estimate the execution status of Task 0. As long as the actual CPU load doesn't exceed 95 %, the specified number of instructions for Task 0 can be guaranteed to be executed within 1 ms period without over-run situation. If Task 0 hasn't finished up executing the specified number of instructions by the end of the 1 ms period, then it would overload the CPU. In that case, an execution fault would be registered by asserting the 10th bit of the variable `FaultFlags` [2], and cause the system to go into fault state when the safety functions are disabled, or going into failsafe mode when the safety functions are enabled.

Code Listing 18 Execution Time Measurement for Task 1 Using `RunTimeCounter` in Script Code for MCEWizard/MCEDesigner

```
/*************************************************************/
/* Global variable definition */
int ExecutionTimeCapture;
int ExecutionTime;
/*************************************************************/
/*Task1 script function*/
Script_Task1()
{
        ExecutionTimeCapture = RunTimeCounter;
```

Code Listing 18 **Execution Time Measurement for Task 1 Using `RunTimeCounter` in Script Code for MCEWizard/MCEDesigner**

```
…
…
…
        ExecutionTime = RunTimeCounter - ExecutionTimeCapture;
}
```



Speed Selection: LOW SPEED; Vin: 100VAC

Figure 34 **Reading 'ExecutionTime_L' Variable Used in Script Code Using MCEDesigner**

3.2.2.2 Execution Time Evaluation Using iSD

Code Listing 19 and Code Listing 20 shows the code for the iSD that is migrated from the code for the MCEWizard/MCEDesigner (Code Listing 18). In this migration, the variable names should be modified as shown in Table 15.

Code Listing 19 **Execution Time Measurement for Task 1 Using RunTimeCounter in Script Code for iSD (Global.mcs)**

```
/*************************************************************/
/* Global variable definition */
int ExecutionTimeCapture;
int ExecutionTime;
```

Code Listing 20 Execution Time Measurement for Task 1 Using RunTimeCounter in Script Code for iSD (Script_Task1.mcs)

```
/*************************************************************/
/*Task1 script function*/
Script_Task1()
{
  ExecutionTimeCapture = MCEOS.RunTimeCounter;
…
…
…
  ExecutionTime = MCEOS.RunTimeCounter- ExecutionTimeCapture;
}
```

Table 15 Parameter name Differences between MCEWizard/MCEDesigner and iSD

MCEWizard/MCEDesigner	iSD
RunTimeCounter	MCEOS.RunTimeCounter

3.2.3 Script Task Execution Period Evaluation

3.2.3.1 Execution Period Evaluation Using MCEWizard/MCEDesigner

The variable `RunTimeCounter` can also be used to measure the loop execution period of Task 0 or Task 1. `RunTimeCounter` is a free running timer with 1 ms resolution that is accessible from within the script code. Code Listing 21 shows an example of using `RunTimeCounter` to measure the loop execution period of Task 1 for MCEWizard/MCEDesigner.

Code Listing 21 Loop Execution Period Measurement for Task 1 Using `RunTimeCounter` in Script Code for MCEWizard/MCEDesigner

```
/*************************************************************/
/* Global variable definition */
int LoopExecutionPeriodCapture;
int LoopExecutionPeriod;
/*************************************************************/
/*Task1 script function*/
Script_Task1()
{
  LoopExecutionPeriod = RunTimeCounter –
LoopExecutionPeriodCapture;
  LoopExecutionPeriodCapture = RunTimeCounter;
  …
  …
  …
}
```

3.2.3.2 Execution Period Evaluation Using iSD

Code Listing 22 and Code Listing 23 shows the code for the iSD that is migrated from the code for the MCEWizard/MCEDesigner (Code Listing 21). In this migration, the variable names should be modified as shown in Table 16

Code Listing 22 Loop Execution Period Measurement for Task 1 Using `RunTimeCounter` in Script Code for iSD (Global.mcs)

```
/**************************************************************/
/* Global variable definition */
int LoopExecutionPeriodCapture;
int LoopExecutionPeriod;
```

Code Listing 23 Loop Execution Period Measurement for Task 1 Using `RunTimeCounter` in Script Code for iSD (Script_Task1.mcs)

```
/**************************************************************/
/*Task1 script function*/
Script_Task1()
{
  LoopExecutionPeriod = MCEOS.RunTimeCounter – LoopExecutionPeriodCapture;
  LoopExecutionPeriodCapture = MCEOS.RunTimeCounter;
  …
  …
  …
}
```

Table 16 Parameter name Differences between MCEWizard/MCEDesigner and iSD

MCEWizard/MCEDesigner	iSD
RunTimeCounter	MCEOS.RunTimeCounter

4 Script Guidelines & Limitations

- With FW REV 1.03.03, the maximum number of global variables supported by the script engine is 30, and the maximum number of local variables for each task is 24. With FW REV 5.X.X and its support of several different variable types, 256 byte of data memory is allocated for global variables and 128 byte of data memory is allocated for local variables in each task separately. The maximum number of global variables and local variables depend on the variable type being used. The intercommunication between Task 0 and Task 1 can be realized by using global variables. Only global variables are accessible from the MCEDesigner, iSD, or user UART interface. It is recommended to define a variable as the global type if users plan to read its value during run time using the MCEDesigner [3] or iSD [5].
- The maximum allowed script code size is 16 kB. This is equivalent to approximately 1500 lines of script code. The actual object code size is reported in the compiled script bytecode file. An example is shown at line 008 and line 009 in Code Listing 2 for the MCEDesigner, and shown in Figure 9 for the iSD.
- The script engine only supports integer type variables, so the floating-point type variables or constants will need to be converted to Q format for proper processing in the script code. An example of Q format conversion can be found in Section 2.3.3. And each script engine in the MCEDesigner [2] and iSD [5] support different types of variables as shown below Table 17 and Table 18 respectively.

Table 17 Script Variable Types in MCEDesigner

Type	Storage Size	Value range	Description
int	4 bytes	-2,147,483,648 to 2,147,483,647	integer

Table 18 Script Variable Types in iSD

Type	Storage Size	Value range	Description
uint8_t	1 bytes	0 to 255	Byte length unsigned integer
int8_t	1 bytes	-128 to 127	Byte length integer
uint16_t	2 bytes	0 to 65,535	Short unsigned integer
int16_t	2 bytes	-32,768 to 32,767	Short integer
int32_t	4 bytes	-2,147,483,648 to 2,147,483,647	integer
int	4 bytes	-2,147,483,648 to 2,147,483,647	integer

- The script engine supports 2 independent tasks, Task 0 and Task 1, running concurrently. The user script program runs repeatedly on a configurable interval within Task 0 or Task 1 loop. The shortest possible execution period is 1 ms for Task 0, and 10 ms for Task 1. The execution period for each task can be configured to the multiples of 1 ms for Task 0 or 10 ms for Task 1 in the script code. Task 0 has greater priority than Task 1. The actual timing setup for each script task needs to be adjusted according to the specific application requirements.
- The analog input pins are sampled by the MCE every 1 ms. According to Nyquist theorem, if the input analog signal frequency is greater than 500 Hz, then it cannot be properly represented by the sampling method of MCE script engine. It is highly recommended that an analog LPF should be used to attenuate the input analog signal frequency that is greater than 500 Hz to minimize the aliasing effect.
- The GPIO pins are sampled and updated by MCE every 1 ms. Any GPIO input changes that occur faster than 1 ms will not be properly captured by the sampling method of MCE script engine. Similarly, any GPIO output

changes that happen faster than 1 ms cannot be realized by using the script program. The fastest possible frequency generated by toggling an GPIO pin using script is 500 Hz.

- It is recommended to change a specific GPIO pin value only once within the Task 0 or Task 1's loop. If there is more than one instance of GPIO manipulation within Task 0 or Task 1's loop, only the last operation would take effect due to the unique GPIO update mechanism in the MCE software. For example, given that a specific GPIO pin is originally reset to logic low level, if this GPIO pin is set to logic high level at the beginning of Task 0, and is then reset at the end of Task 0, the actual GPIO will not toggle as expected. Instead, it will remain in a reset state after the execution of Task 0 loop.
- For those time critical functions, it is recommended users utilize Task 0, whose minimum execution period can be set to 1 ms. For those functions that are not time critical, either Task 0 or Task 1 can be used. In that case, it is recommended users set the execution period of the script task to 50 ms.
- Digital filter implementation using the script can be realized in Task 0 with sampling frequency up to 1 kHz due to the minimum execution period limit of Task 0. As a result, signal frequency greater than 500 Hz cannot be properly sampled and processed.
- The script language in the MCEWizard/MCEDesigner doesn't support the implementation of infra-red communication. It's available from FW5.1 in the iSD.
- The script language can support Programmable Logic Controller (PLC) as long as the minimum timing requirement is no less than 1 ms.
- The script language doesn't support the implementation of digital Hall Effect sensors.
- In the iSD, the only way to debug the script variables is to use the watch the view window in the script debugger. The iSD doesn't support reading script variables in dashboard or plotting out script variables using 'oscilloscope' function.

5 References

[1] iMOTION™ IMC100 High Performance Motor Control IC Series Datasheet (REV 1.6).

[2] iMOTION™ Motor Control Engine Software Reference Manual (REV 1.3).

[3] MCEDesigner User Guide (REV 2.3.0.1).

[4] MCEWizard 2.0 User Guide (REV 2.3.0.1).

[5] iMOTION™ Motor Control Engine Functional Reference Manual (REV 1.0).

Revision history

Document version	Date of release	Description of changes
1.0	9/5/2018	Initial release.
1.1	6/5/2020	Analog input sampling rate and GPIO update rate are revised. Example code in Section 2.1.4, 2.3.4, and 2.4.2 revised to use CONST keyword to define constants.
1.2	10/27/2022	A comment of the iSD is added in the Scope and purpose and Script Guideline & Limitations. Workflow for the iSD is added in the Script Development Workflow. Example code in Section 2.1.4, 2.2.3, 2.3.4, 2.4.2 and 3.2.2 revised to add the code for the iSD, and to show each code separately. References is updated to the latest version.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 12/15/2022

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AN2018-27

IMPORTANT NOTICE
The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.