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.

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 definition, as well as system specific fault handling definition.

Table of contents

About this document **1**
Table of contents **1**
1 Script Language Overview **3**
1.1 Introduction 3
1.2 Script Development Workflow 3
2 Script Application Examples **4**
2.1 2-Level Speed Selection Interface 4
2.1.1 Speed Selection Interface Requirement 4
2.1.2 Analog Input Pin for Speed Selection Interface 4
2.1.3 Speed Selection State Machine 5
2.1.4 Speed Selection Interface Script Implementation 6
2.2 Low-Pass Filter for DC Bus Voltage 8
2.2.1 DC Bus Voltage Ripple 8
2.2.2 DC Bus Voltage Sensing 8
2.2.3 LPF Design & Implementation 9
2.2.4 LPF Test Results 11
2.3 Target Speed Shaping & Brown-out Protection 13
2.3.1 Target Speed Requirements 13
2.3.2 DC Bus Status State Machine 14
2.3.3 Scaling for Target Speed Shaping Calculation 15
2.3.4 Target Speed Shaping & Brown-out Protection Script Implementation 16
2.3.5 Target Speed Shaping Measurement Results 20
2.4 Dynamic Motor Current Limit Customization 22
2.4.1 Motor Current Limit Requirement 22
2.4.2 Dynamic Motor Current Limit Algorithm Design & Implementation 23
2.4.3 Dynamic Motor Current Limit Measurement Results 29
3 Script Performance Evaluation **32**

Table of Contents

3.1 CPU Load Evaluation 32
3.2 Script Task Timing 33
3.2.1 Script Task Timing Setup 33
3.2.2 Script Task Execution Time Evaluation 34
3.2.3 Script Task Execution Period Evaluation 35
4 Script Guidelines & Limitations 36
5 References 37
Revision history 37

1 Script Language Overview

1.1 Introduction

The latest software release of iMOTION™ MCE includes a script engine, offering users the possibility to customize system level functionalities without affecting the motor and PFC control algorithm. The script engine is a light weight virtual machine that supports reading and writing 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, as well as fault handling.

- 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. It is highly recommended to check actual CPU loading 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 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 higher 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

The typical workflow of script program development starts from using MCEWizard (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].

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 hairdryers. This example detailizes the requirement and script implementation of a 2-level speed selection interface using one of the available ADC pins on 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 OFF state, the voltage range from 1 V to 2 V was defined as LOW SPEED state, and the voltage range from 2 V and above was defined as 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.

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

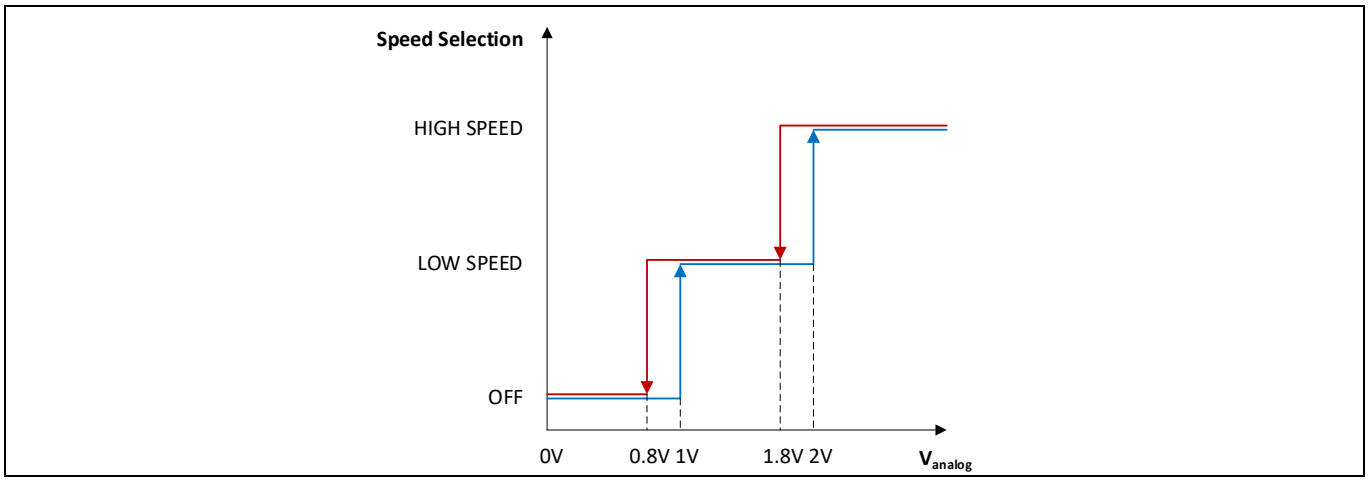


Figure 1 Speed Selection & Analog Voltage Relationship

2.1.2 Analog Input Pin for Speed Selection Interface

This application specific speed selection requirement described in Section 2.1.1 can be conveniently realized by enabling an analog input pin supported by the script engine. For this design, AIN0 pin was chosen to interface with speed selection hardware circuit. Once enabled, this analog input pin is sampled by MCE every 10 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 the following Table 1.

Table 1 Speed Selection Interface Voltage Thresholds

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

2.1.3 Speed Selection State Machine

A state machine was designed to interpret the speed selection input as shown in the following Figure 1. 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 Speed_Mode_OFF state, the target speed is reset to 0, and the motor is stopped. If VSP pin voltage is higher than `VLSStart`, then it shifts to Speed_Mode_LOW_SPEED state. While it is in 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 higher than `VHSStart`, then it shifts to Speed_Mode_HIGH_SPEED state. While it is in Speed_Mode_HIGH_SPEED state, if VSP pin voltage is lower than `VHSStop`, then it shifts to Speed_Mode_LOW_SPEED state. While it is in 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, `Command` parameter can be directly used in the script code without declaration.

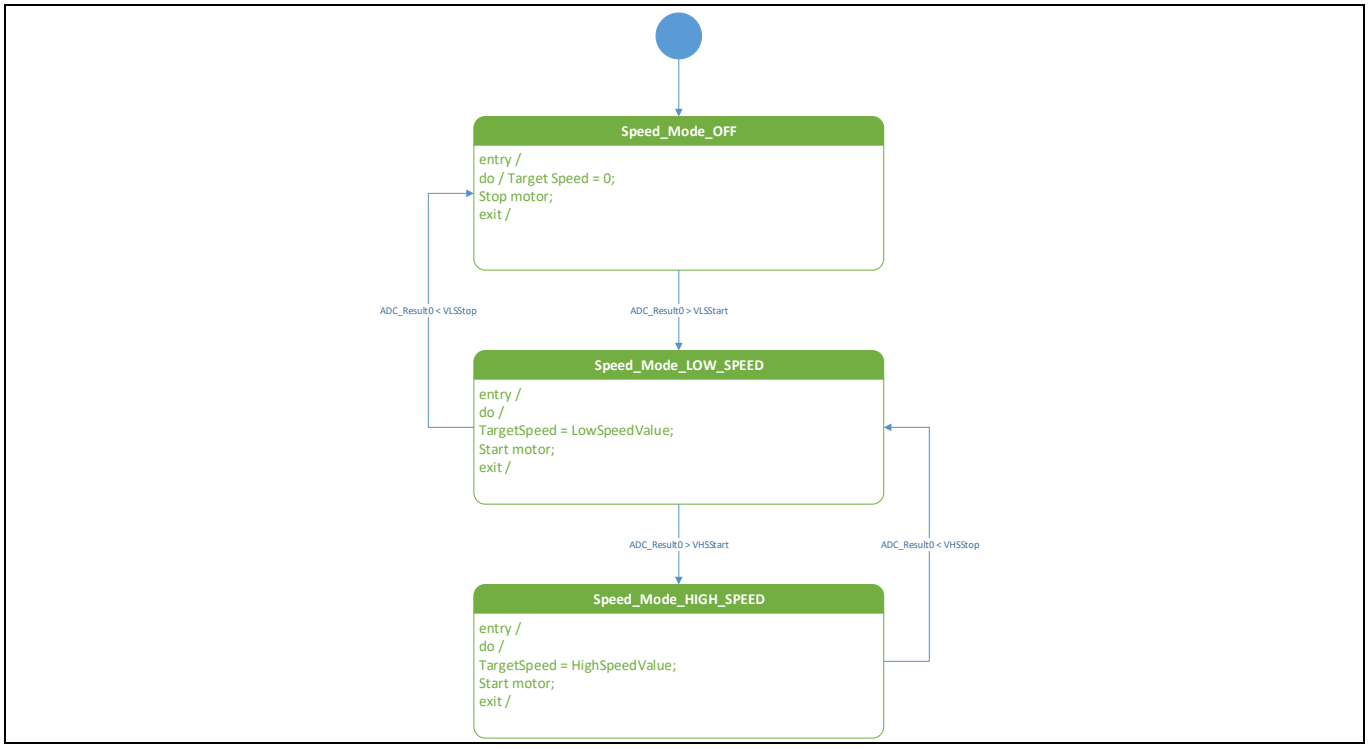


Figure 2 Speed Selection State Machine

2.1.4 Speed Selection Interface Script Implementation

The following Code Listing 1 shows the source code for the 2-level speed selection interface application implemented in Task 1. 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. The following 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 17. So, the execution step for Task 1 should be set to greater than 17 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.

Code Listing 1 Speed Selection Interface Script Code

```
001  /*************************************************************/
002  /*Script user version value, should be 255.255*/
003  #SET SCRIPT_USER_VERSION (1.00)
004  #SET SCRIPT_TASK1_EXECUTION_PERIOD (5)
005  /*Defines number of lines to be executed every 10mS in Task1*/
006  #SET SCRIPT_TASK1_EXECUTION_STEP (20)
007  /*************************************************************/
008  /*Task1 init function*/
009  Script_Task1_init()
010  {
011    int SpeedMode;
012    int VLSStart;
013    int VLSStop;
014    int VHSStart;
015    int VHSStop;
016    int LowSpeedValue;
017    int HighSpeedValue;
018    VLSStart = 819;   // 1V => 819 counts
019    VLSStop = 655;    // 0.8V => 655 counts
020    VHSStart = 1638;  // 2V => 1638 counts
021    VHSStop = 1474;   // 1.8V => 1474 counts
022    LowSpeedValue = 5000;
023    HighSpeedValue = 10000;
024  }
025  /*Task1 script function*/
026  Script_Task1()
027  {
028    if (SpeedMode == 0)     // Speed selection is in OFF state.
029    {
030      TargetSpeed = 0;
031      Command = 0;    // Stop the motor.
032      if (ADC_Result0 > VLSStart)
033      {
034        SpeedMode = 1;      // Shift to LOW_SPEED state.
035      }
```

Script Application Examples

Code Listing 1 Speed Selection Interface Script Code

```
036          }
037          if (SpeedMode == 1)     // Speed selection is in LOW_SPEED
   state.
038          {
039            if (ADC_Result0 > VHSStart)
040            {
041               SpeedMode = 2;      // Shift to HIGH_SPEED state.
042            }
043            else
044            {
045              if (ADC_Result0 < VLSStop)
046              {
047                              SpeedMode = 0;    // Shift to OFF state.
048              }
049              else   //Stay in LOW_SPEED state.
050              {
051                TargetSpeed = LowSpeedValue;   // Update TargetSpeed.
052                Command = 1;      // Start motor.
053              }
054            }
055          }
056          if(SpeedMode == 2)      // Speed selection is in HIGH_SPEED
   state.
057          {
058            if(ADC_Result0 < VHSStop)
059            {
060              SpeedMode = 1;      // Shift to LOW_SPEED state.
061            }
062            else     // Stay in HIGH_SPEED state.
063            {
064              TargetSpeed = HighSpeedValue;    // Update TargetSpeed.
065              Command = 1;
066            }
067          }
068        }
```

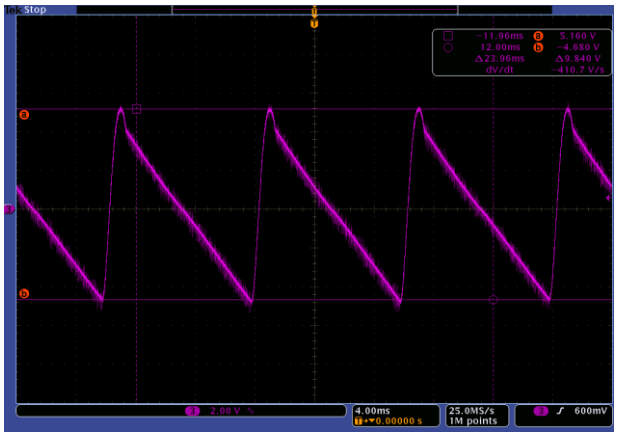
Code Listing 2 Portion of Compiled Script Object File for Speed Selection Interface Script Code

```
001        %--------------------------------------------------------------
002        % Script Object File
003        %--------------------------------------------------------------
004        % SCRIPT_USER_VERSION              : 001.000
005        % DATE & TIME                      : 22.08.2018  22:23:36
006        % SIZE                             : 231 Bytes
007        % Total Number of Lines            : 69
008        % Task0 - Number of Instructions   : 0
009        % Task1 - Number of Instructions   : 17
010        %--------------------------------------------------------------
```

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 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, 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 3 is a screenshot of the AC portion of the actual DC bus voltage waveform with IMC101T controller driving a Permanent Magnet Synchronous Motor (PMSM) running at a speed = 19400 RPM and Vin = 125 VAC / 50Hz. It could be seen that the amplitude of the DC bus voltage ripples was around 9.84 V.



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

Figure 3 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 thanks 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 the following Figure 4 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 757 V.

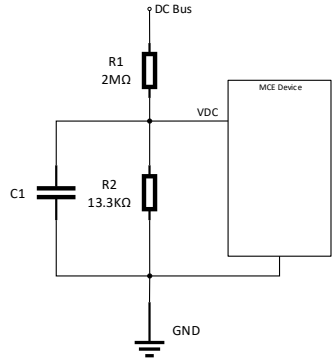


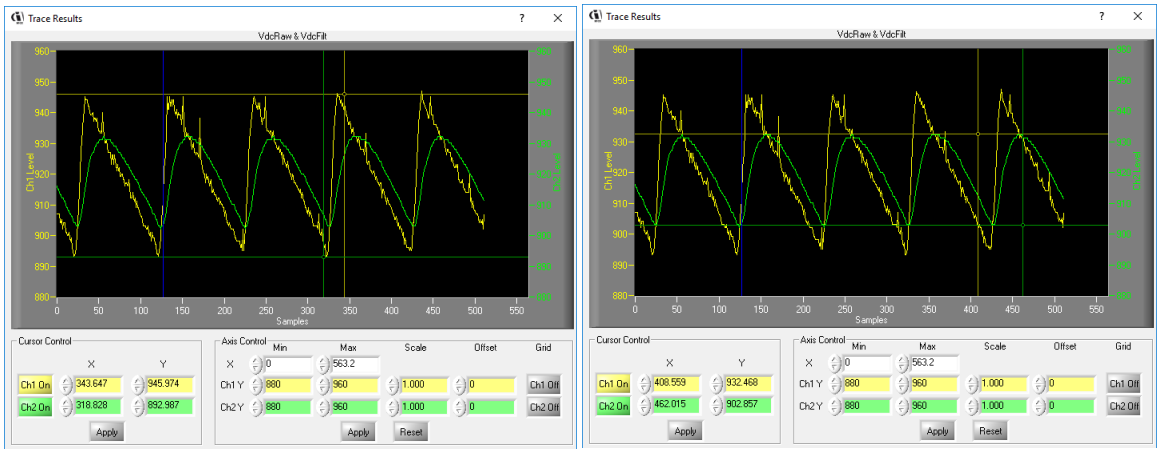
Figure 4 DC Bus Voltage Sensing Interface Circuit Diagram

The DC bus voltage is being 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 5 shows the `VdcRaw` and `VdcFilt` waveforms under the same input / output conditions as in the case of Figure 3 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 5 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}: 20kHz; Sample at PWM frequency divided by 2; Trigger Level: 920; V_{in}: 125VAC / 50Hz; V_{pp_CH1}: 53 ADC counts; V_{pp_CH2}: 30 ADC counts

Figure 5 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 needs to be at least higher than twice of the frequency of interest 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 higher the sampling frequency is, the higher 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 100Hz 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 3.

Code Listing 3 LPF Pseudo Code

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

It is recommended to 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. The following Code Listing 4 shows the script code implementation for the LPF.

Code Listing 4 LPF Script Code

```
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;
}
```

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, 1kHz 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. Using MATLAB, the Bode plot and step response of the implemented LPF were calculated and shown in the following Figure 6 and Figure 7.

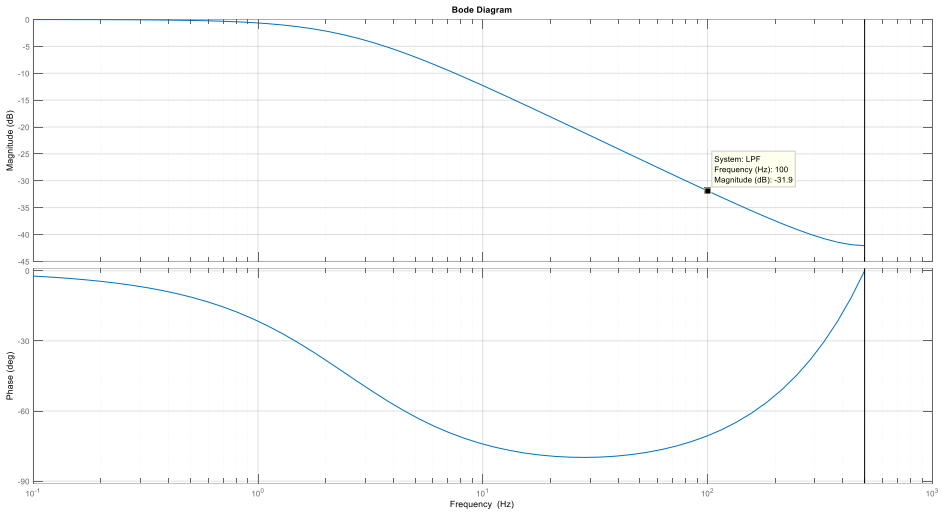


Figure 6 Calculated 1st Order IIR LPF Frequency Response

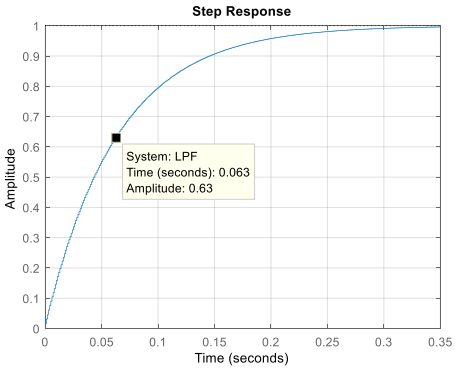


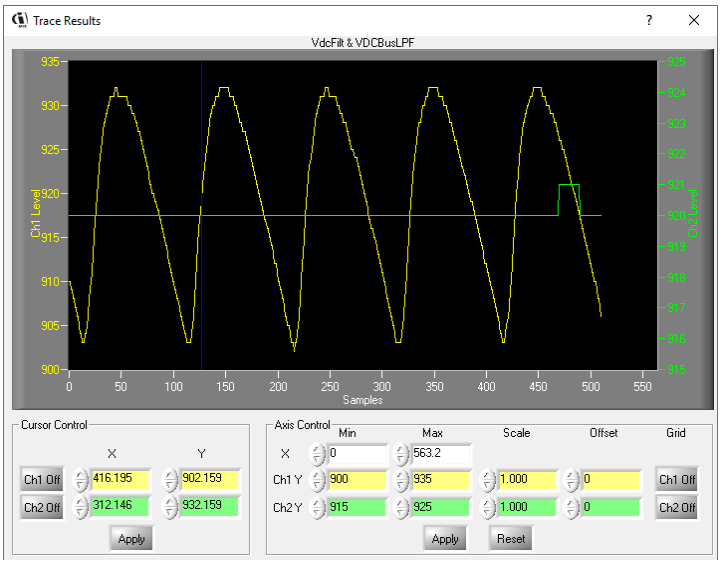
Figure 7 Calculated 1st Order IIR LPF Step Response

2.2.4 LPF Test Results

Figure 8 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.

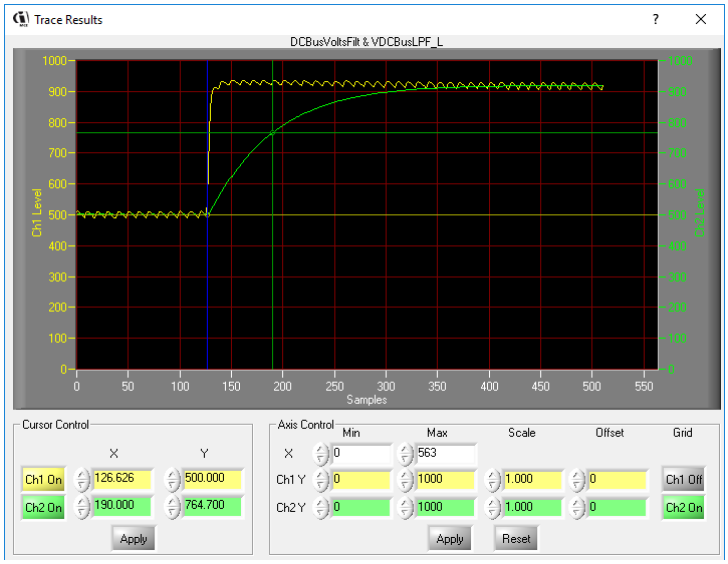
Figure 9 shows the measured step response of the implemented LPF, where V_{in} was increased from 70 VAC to 125 VAC. The initial value of `VdcFilt` was 500 ADC counts, and the steady state value of `VdcFilt` was 919 ADC counts, resulting in a step change of 419 ADC counts. The time it took for `VdcFilt` 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 this 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.



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

Figure 8 `VdcFilt` & `VDCBusLPF_L` Waveform Screenshot



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

Figure 9 Measured 1st Order IIR LPF Step Response Screenshot

2.3 Target Speed Shaping & Brown-out Protection

2.3.1 Target Speed Requirements

Some applications such as hairdryers require setting the motor target speed dynamically based on instantaneous DC bus voltage level. Take an application that uses a 6 pole PMSM 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 the following Table 2.

Table 2 Coefficients of the Quadratic Function for Target Speed & DC Bus Voltage Relationships

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

The calculated target speed using the abovementioned quadratic function needs to be within its corresponding maximum and minimum limits. Table 3 below lists the speed limit requirements for HIGH SPEED and LOW SPEED levels.

Table 3 Max. & Min. Target Speed Limit Definitions

	HIGH SPEED	LOW 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 certain threshold. In order to eliminate potential oscillation when the DC bus voltage is around the brown-out level, a hysteresis of 5V was introduced. The following Table 4 lists the DC bus brown-in and brown-out voltage levels.

Table 4 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 the following Figure 10 and Figure 11. The solid blue 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 red line shows that if the motor is currently running, then it doesn't stop running until the DC bus voltage falls below 85 V.

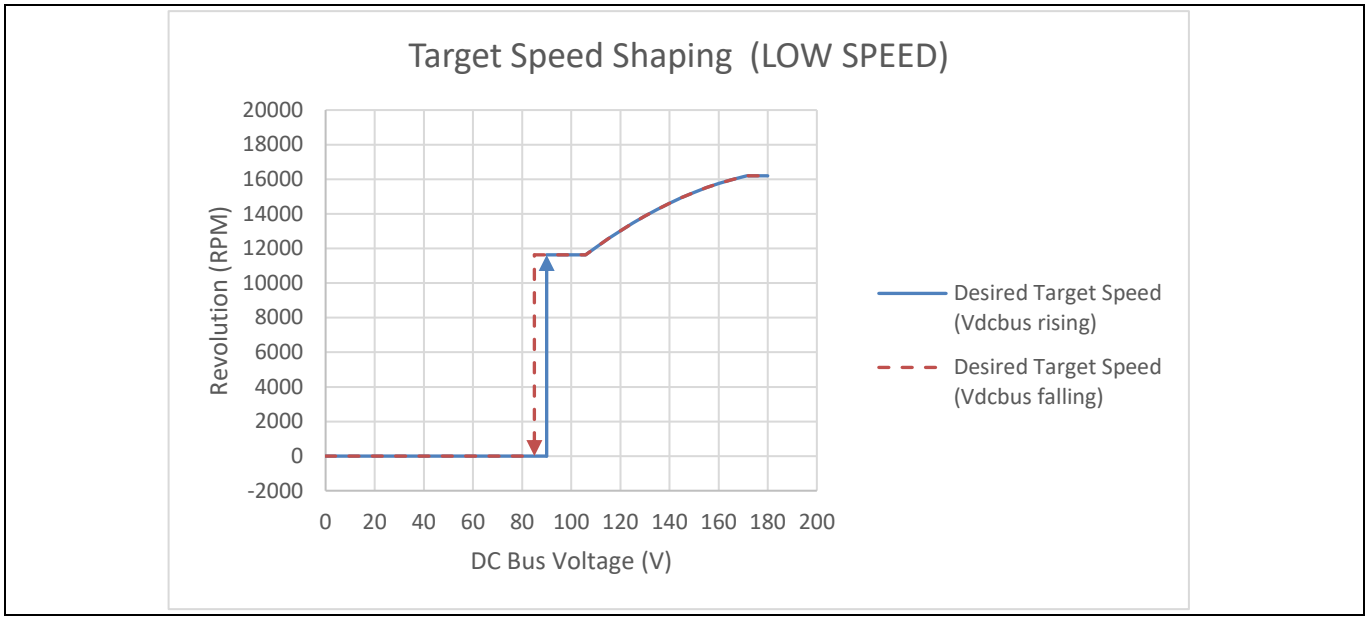


Figure 10 Target Speed Shaping (LOW SPEED)

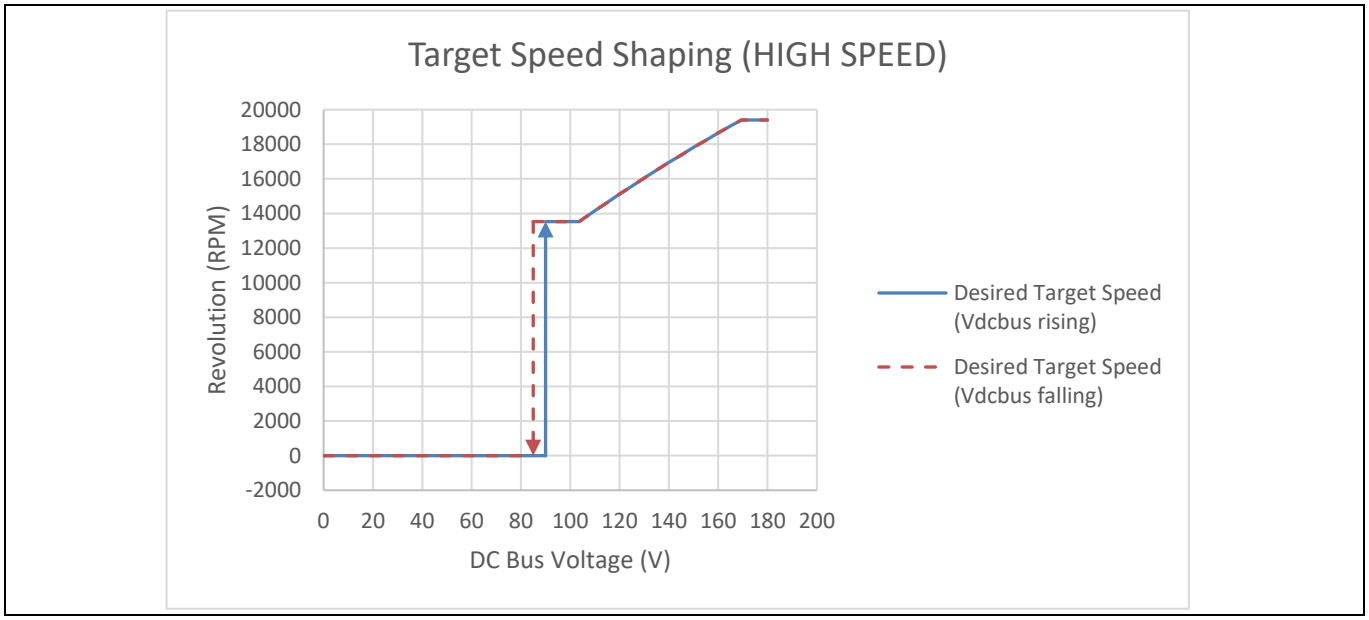


Figure 11 Target Speed Shaping (HIGH SPEED)

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 the following Figure 12. 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 to 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.

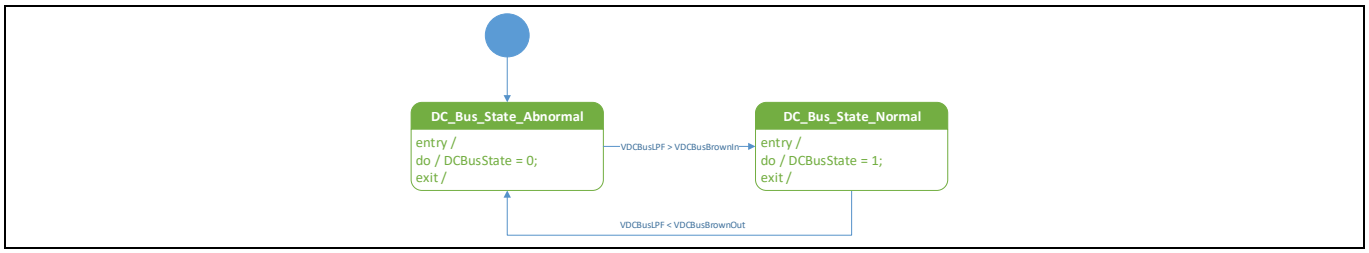


Figure 12 DC Bus Status State Machine Diagram

The calculation of the value of `VDCBusBrownIn` and `VDCBusBrownOut` follows the conversion formula described in Section 2.2.2, and the voltage levels specified in Table 4. The results are shown in Table 5 below.

Table 5

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 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 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 the following Table 6.

Table 6 Coefficients in Floating Point Format for the Quadratic Function for Target Speed & DC Bus Voltage Relationships with Scaling Factors

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

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 script can be realized by using the following pseudo code in Code Listing 5.

Code Listing 5 Target Speed Shaping Calculation Pseudo Code

```
001        Speed_Value = A_NUM * VDCBus * VDCBus + B_NUM * VDCBus +
   C_NUM;
002        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 the following Table 7.

Table 7 Coefficients in Q15.16 Format for the Quadratic Function for Target Speed & DC Bus Voltage Relationships with Scaling Factors

$Denominator$		65536	
HIGH SPEED		LOW SPEED	
A_{h_NUM}	-291	A_{l_NUM}	-1049
B_{h_NUM}	1315558	B_{l_NUM}	2267069
C_{h_NUM}	80227767	C_{l_NUM}	-330352746

2.3.4 Target Speed Shaping & Brown-out Protection Script Implementation

The following Code Listing 6 shows the source code for the target speed shaping with brown-out protection application implemented in Task 1. 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.

Code Listing 6 Target Speed Shaping & Brown-out Protection Script Code

```
001        /*********************************************************/
002        /*Script user version value, should be 255.255*/
003        #SET SCRIPT_USER_VERSION (1.00)
004        /*Script execution time for Task0 in mS, maximum value 65535*/
005        #SET SCRIPT_TASK0_EXECUTION_PERIOD (1)
```

Code Listing 6 Target Speed Shaping & Brown-out Protection Script Code

```
006         /*Defines number of lines to be executed every 1mS in Task0*/
007         #SET SCRIPT_TASK0_EXECUTION_STEP (2)
008         /*Script execution time for Task1 in 10mS, maximum value
    65535*/
009         #SET SCRIPT_TASK1_EXECUTION_PERIOD (5)
010         /*Defines number of lines to be executed every 10mS in Task1*/
011         #SET SCRIPT_TASK1_EXECUTION_STEP (50)
012         /*************************************************************/
013         /* Global variable definition */
014         int VDCBusLPF;
015         int DCBusState;
016         int SpeedMode;
017         int SpeedValue;
018         /*************************************************************/
019         /*Task0 init function*/
020         Script_Task0_init()
021         {
022           /*Initialize global variable*/
023           VDCBusLPF = 0;
024           /* local variable definition */
025           int VDCBusMultiplyDEN;
026           /*Initialize local variable*/
027           VDCBusMultiplyDEN = 0;
028         }
029
030         /*Task0 script function*/
031         Script_Task0()
032         {
033           // Vdcbus filtering
034           VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt -
    VDCBusLPF);
035           VDCBusLPF = VDCBusMultiplyDEN >> 6;
036         }
037
038         /*Task1 init function*/
039         Script_Task1_init()
040         {
041           /* local variable definition */
042           int VDCBusBrownIn;  // Vdcbus_brown_in = 90V => 487 counts
043           int VDCBusBrownOut; // Vdcbus_brown_out = 85V => 460 counts
044
045           int VLSStart;
046           int VLSStop;
047           int VHSStart;
048           int VHSStop;
049
050           int AlNum;
051           int BlNum;
052           int ClNum;
053           int AhNum;
054           int BhNum;
055           int ChNum;
056           int DenShiftBit;
```

Code Listing 6 Target Speed Shaping & Brown-out Protection Script Code

```
057
058          int TspdLSMin;
059          int TspdLSMax;
060          int TspdHSMin;
061          int TspdHSMax;
062
063          /*Initialize global variable*/
064          DCBusState = 0;
065          SpeedMode = 0;
066          SpeedValue = 0;
067
068          /*Initialize local variable*/
069          VDCBusBrownIn = 487;  // Vdcbus_brown_in = 90V => 487 counts
070          VDCBusBrownOut = 460; // Vdcbus_brown_out = 85V => 460
   counts
071
072          VLSStart = 819;   // Vsp_low_spd_start = 1V => 819 counts
073          VLSStop = 655;    // Vsp_low_spd_stop = 0.8V => 655 counts
074          VHSStart = 1638;  // Vsp_high_spd_start = 2V => 1638 counts
075          VHSStop = 1474;   // Vsp_high_spd_stop = 1.8V => 1474 counts
076
077          AlNum = -1049;
078          BlNum = 2267069;
079          ClNum = -330352770;
080          AhNum = -291;
081          BhNum = 1315558;
082          ChNum = 80227700;
083          DenShiftBit = 16; // Denominator = 2^16 = 65536
084
085          TspdLSMin = 9523; // In LOW_SPEED mode, Target Speed min =
   11625 rpm => 9523 counts.
086          TspdLSMax = 13270;  // In LOW_SPEED mode, Target Speed max =
   16200 rpm => 13270 counts.
087          TspdHSMin = 11089;  // In HIGH_SPEED mode, Target Speed min
   = 13537 rpm => 11089 counts.
088          TspdHSMax = 15892;  // In HIGH_SPEED mode, Target Speed max
   = 19400 rpm => 15892 counts.
089        }
090
091        /*Task1 script function*/
092        Script_Task1()
093        {
094          // DC bus state machine
095          if (DCBusState == 0)  // DCBus is abnormal.
096          {
097            if (VDCBusLPF > VDCBusBrownIn)
098            {
099              DCBusState = 1; // Shift to DCBus normal state.
100            }
101          }
102
103          if (DCBusState == 1)  // DCBus is normal.
104          {
```

Code Listing 6 Target Speed Shaping & Brown-out Protection Script Code

```
105             if (VDCBusLPF < VDCBusBrownOut)
106             {
107               DCBusState = 0; // Shift to DCBus abnormal state.
108             }
109           }
110
111           // Speed selection state machine
112           if (SpeedMode == 0) // Speed selection is in OFF state.
113           {
114             TargetSpeed = 0;
115             Command = 0;  // Stop the motor.
116
117             if (ADC_Result0 > VLSStart)
118             {
119               SpeedMode = 1;  // Shift to LOW_SPEED state.
120             }
121           }
122
123           if (SpeedMode == 1) // Speed selection is in LOW_SPEED
    state.
124           {
125             if (ADC_Result0 > VHSStart)
126             {
127               SpeedMode = 2;  // Shift to HIGH_SPEED state.
128             }
129             else
130             {
131               if (ADC_Result0 < VLSStop)
132               {
133                 SpeedMode = 0;  // Shift to OFF state.
134               }
135               else  //Stay in LOW_SPEED state.
136               {
137                 if (DCBusState == 1)  // DC bus voltage is normal.
138                 {
139                   // Calculate target speed. Target speed follows 2nd
    order polynomial curve for LS.
140                   SpeedValue = AlNum * VDCBusLPF * VDCBusLPF + BlNum *
    VDCBusLPF + ClNum;
141                   SpeedValue = SpeedValue >> DenShiftBit;
142                   if (SpeedValue > TspdLSMax) // Upper limit check
143                   {
144                     SpeedValue = TspdLSMax;
145                   }
146                   if (SpeedValue < TspdLSMin) // Lower limit check
147                   {
148                     SpeedValue = TspdLSMin;
149                   }
150                   TargetSpeed = SpeedValue; // Update TargetSpeed.
151                   Command = 1;  // Start motor.
152                 }
153                 else  // DC bus voltage is abnormal.
154                 {
```

Code Listing 6 Target Speed Shaping & Brown-out Protection Script Code

```
                    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.
                }
              }
            }
          }
```

2.3.5 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 LOW SPEED level was shown in the following Table 8 and plotted against the desired target speed shaping curves in Figure 13. As can be seen from the measurement data, 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 LOW SPEED level.

Table 8 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%
130	178.3	16264	16200	0.4%

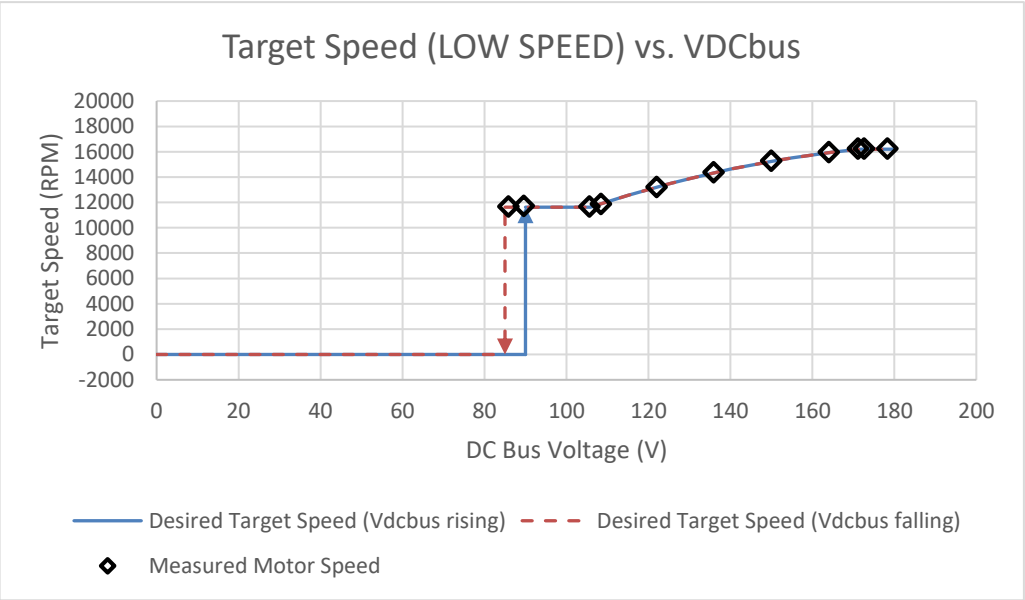


Figure 13 Measurement of Target Speed vs. DC Bus Voltage (LOW SPEED)

Table 9 and Figure 14 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 9 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%

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%
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%

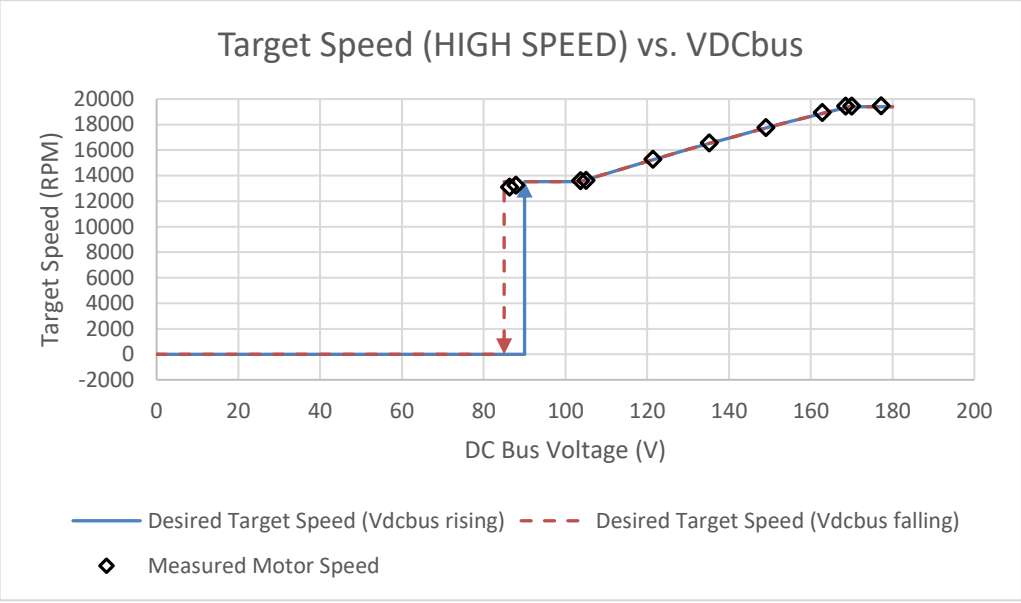


Figure 14 Measurements of Target Speed vs. DC Bus Voltage (HIGH SPEED)

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 customized motor current limit based on the speed selection input to enable tighter torque control. Besides, 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 the following Table 10.

Table 10 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

The following Figure 15 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 steady state at a given speed selection level is updated by 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 which 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 which 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`.

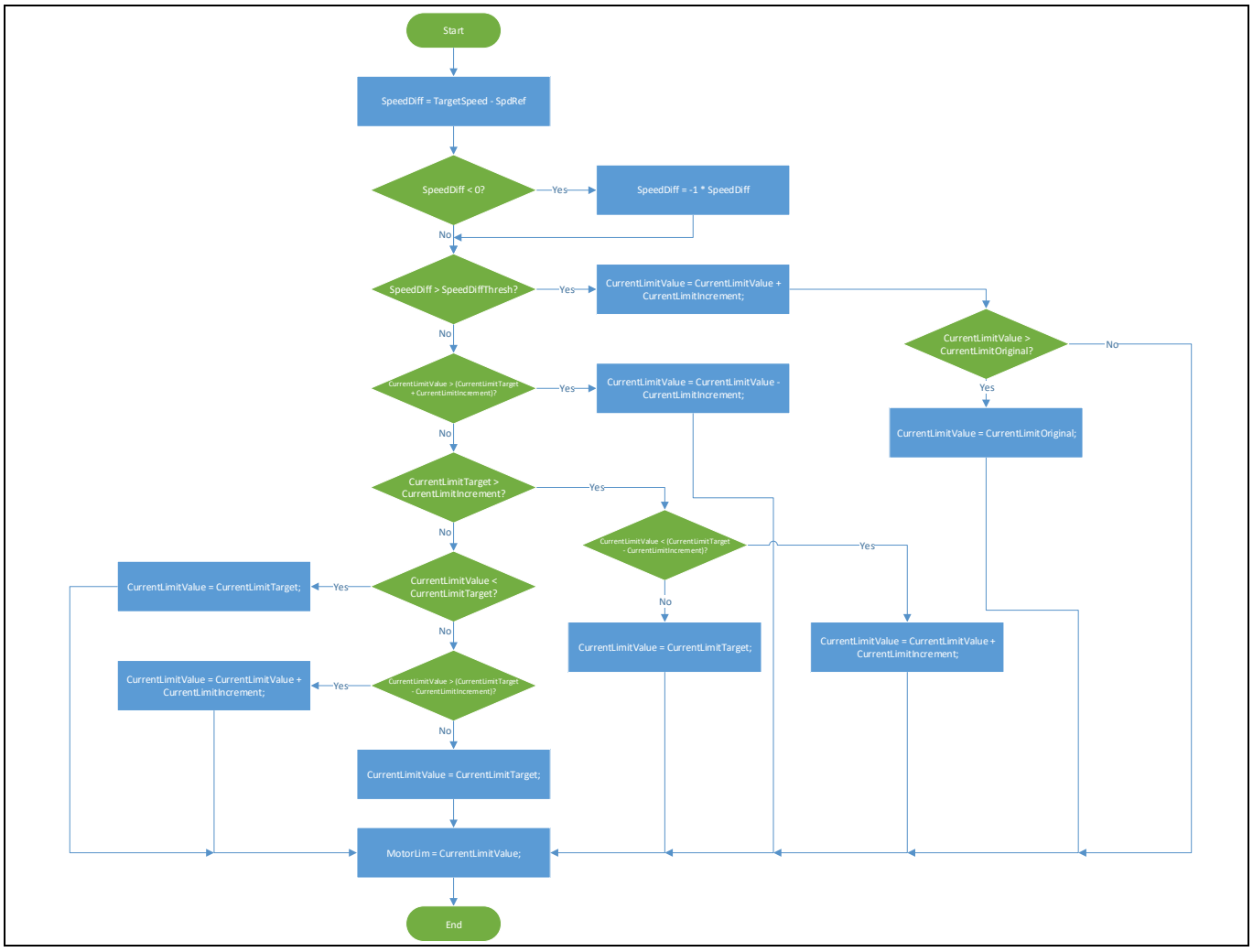


Figure 15 Flowchart of Dynamic Motor Current Limit Algorithm

Script Application Examples

The following Code Listing 7 shows the source code for the dynamic motor current limit customization application implemented in Task 1. 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. So, 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.

Code Listing 7 Dynamic Motor Current Limit Script Code

```
001       /*************************************************************/
002       /*Script user version value, should be 255.255*/
003       #SET SCRIPT_USER_VERSION (1.00)
004       /*Script execution time for Task0 in mS, maximum value 65535*/
005       #SET SCRIPT_TASK0_EXECUTION_PERIOD (1)
006       /*Defines number of lines to be executed every 1mS in Task0*/
007       #SET SCRIPT_TASK0_EXECUTION_STEP (2)
008       /*Script execution time for Task1 in 10mS, maximum value
   65535*/
009       #SET SCRIPT_TASK1_EXECUTION_PERIOD (1)
010       /*Defines number of lines to be executed every 10mS in Task1*/
011       #SET SCRIPT_TASK1_EXECUTION_STEP (60)
012       /*************************************************************/
013       /* Global variable definition */
014       int VDCBusLPF;
015       int DCBusState;
016       int SpeedDiff;
017       int CurrentLimitOriginal;
018       int CurrentLimitValue;
019       int CurrentLimitTarget;
020       int SpeedMode;
021       /*************************************************************/
022       /*Task0 init function*/
023       Script_Task0_init()
024       {
025         /*Initialize global variable*/
026         VDCBusLPF = 0;
027         /* local variable definition */
028         int VDCBusMultiplyDEN;
029         /*Initialize local variable*/
030         VDCBusMultiplyDEN = 0;
031       }
032
033       /*Task0 script function*/
034       Script_Task0()
035       {
036         // Vdcbus filtering
037         VDCBusMultiplyDEN = VDCBusMultiplyDEN + (VdcFilt -
   VDCBusLPF);
038         VDCBusLPF = VDCBusMultiplyDEN >> 6;
```

Code Listing 7 Dynamic Motor Current Limit Script Code

```
039         }
040
041         /*Task1 init function*/
042         Script_Task1_init()
043         {
044           /* local variable definition */
045           int VDCBusBrownIn;  // Vdcbus_brown_in = 90V => 487 counts
046           int VDCBusBrownOut; // Vdcbus_brown_out = 85V => 460 counts
047           int SpeedDiffThresh;
048           int CurrentLimitIncrement;
049           int CurrentLimitLS;
050           int CurrentLimitHS;
051
052           int VLSStart;
053           int VLSStop;
054           int VHSStart;
055           int VHSStop;
056           int LowSpeedValue;
057           int HighSpeedValue;
058
059           /*Initialize global variable*/
060           DCBusState = 0;
061           SpeedDiff = 0;
062           CurrentLimitOriginal = MotorLim;  // Save the original motor
    current limit set in MCEWizard.
063           CurrentLimitValue = CurrentLimitOriginal; // The initial
    value needs to be synced with the original motor current limit set in
    MCEWizard.
064           CurrentLimitTarget = CurrentLimitOriginal;  // The initial
    value needs to be synced with the original motor current limit set in
    MCEWizard.
065           SpeedMode = 0;
066
067           /*Initialize local variable*/
068           VDCBusBrownIn = 487;  // Vdcbus_brown_in = 90V => 487 counts
069           VDCBusBrownOut = 460; // Vdcbus_brown_out = 85V => 460
    counts
070
071           SpeedDiffThresh = 100;  // Set the speed difference
    threshold to 100 counts.
072           CurrentLimitIncrement = 100;  // Motor current limit ramp
    rate = 100 counts / update interval (10 ms).
073           CurrentLimitLS = 519; // low speed motor current limit =
    0.38A => 519 counts
074           CurrentLimitHS = 819; // high speed motor current limit =
    0.6A => 819 counts
075
076           VLSStart = 819;   // Vsp_low_spd_start = 1V => 819 counts
077           VLSStop = 655;    // Vsp_low_spd_stop = 0.8V => 655 counts
078           VHSStart = 1638;  // Vsp_high_spd_start = 2V => 1638 counts
079           VHSStop = 1474;   // Vsp_high_spd_stop = 1.8V => 1474 counts
080
081           LowSpeedValue = 5000;
```

Code Listing 7 Dynamic Motor Current Limit Script Code

```
082         HighSpeedValue = 10000;
083       }
084
085       /*Task1 script function*/
086       Script_Task1()
087       {
088         // DC bus state machine
089         if (DCBusState == 0)  // DCBus is abnormal.
090         {
091           if (VDCBusLPF > VDCBusBrownIn)
092           {
093             DCBusState = 1; // Shift to DCBus normal state.
094           }
095         }
096
097         if (DCBusState == 1)  // DCBus is normal.
098         {
099           if (VDCBusLPF < VDCBusBrownOut)
100           {
101             DCBusState = 0; // Shift to DCBus abnormal state.
102           }
103         }
104         // Calculate the difference between the target speed and the
    speed reference in preparation for motor current limit calculation.
105         SpeedDiff = TargetSpeed - SpdRef; // Find out the difference
    between the speed reference and the target speed.
106         if(SpeedDiff < 0) // The target speed is lower than the
    speed reference.
107         {
108           SpeedDiff = -1 * SpeedDiff; // Takes the absolute value of
    SpeedDiff.
109         }
110         // Calculate motor current limit based on speed reference
    and target speed.
111         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.
112         {
113           CurrentLimitValue = CurrentLimitValue +
    CurrentLimitIncrement;  // Increase the motor current limit by
    CurrentLimitIncrement until it reaches CurrentLimOriginal.
114           if (CurrentLimitValue > CurrentLimitOriginal) // Upper
    boundary check for CurrentLimitValue.
115           {
116             CurrentLimitValue = CurrentLimitOriginal;
117           }
118         }
119         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.
120         {
```

Code Listing 7 Dynamic Motor Current Limit Script Code

```
121           if(CurrentLimitValue > (CurrentLimitTarget +
   CurrentLimitIncrement))  // The motor current limit value at this
   moment is greater than the specified motor current limit by more than
   CurrentLimitIncrement.
122           {
123             CurrentLimitValue = CurrentLimitValue -
   CurrentLimitIncrement;  // Decrease the motor current limit target by
   CurrentLimitIncrement.
124           }
125           else  // The motor current limit target is no more than
   the specified motor current limit by more than CurrentLimitIncrement.
126           {
127             if (CurrentLimitTarget > CurrentLimitIncrement) //
   CurrentLimitTarget is greater than CurrentLimitIncrement. Boundary
   check needed for the following minus operation.
128             {
129               if (CurrentLimitValue < (CurrentLimitTarget -
   CurrentLimitIncrement)) // The motor current limit value at this
   moment is less than the specified motor current limit by more than
   CurrentLimitIncrement.
130               {
131                 CurrentLimitValue = CurrentLimitValue +
   CurrentLimitIncrement;  // Increase the motor current limit target by
   CurrentLimitIncrement.
132               }
133               else  // The motor current limit target falls between
   CurrentLimitTarget - CurrentLimitIncrement and CurrentLimitTarget +
   CurrentLimitIncrement.
134               {
135                 CurrentLimitValue = CurrentLimitTarget; // Set the
   motor current limit target to the specified motor current limit.
136               }
137             }
138             else  // CurrentLimitTarget is no more than
   CurrentLimitIncrement.
139             {
140               if (CurrentLimitValue < CurrentLimitTarget)
141               {
142                 CurrentLimitValue = CurrentLimitTarget; // Set the
   motor current limit target to the specified LOW_SPEED motor current
   limit.
143               }
144               else  // CurrentLimitValue is greater than
   CurrentLimitTarget
145               {
146                 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.
147
148                 {
```

Code Listing 7 Dynamic Motor Current Limit Script Code

```
149                     CurrentLimitValue = CurrentLimitValue +
   CurrentLimitIncrement;  // Increase the motor current limit target by
   CurrentLimitIncrement.
150                   }
151                   else  //The motor current limit value is within the
   range of CurrentLimitTarget and CurrentTarget - CurrentLimitIncrement.
152                   {
153                     CurrentLimitValue = CurrentLimitTarget; // Set the
   motor current limit target to the specified motor current limit.
154                   }
155                 }
156               }
157             }
158           }
159           MotorLim = CurrentLimitValue; // Update MotorLim.
160
161           // Speed selection state machine
162           if (SpeedMode == 0) // Speed selection is in OFF state.
163           {
164             TargetSpeed = 0;
165             CurrentLimitTarget = CurrentLimitOriginal;
166             Command = 0;  // Stop the motor.
167
168             if (ADC_Result0 > VLSStart)
169             {
170               SpeedMode = 1;  // Shift to LOW_SPEED state.
171             }
172           }
173
174           if (SpeedMode == 1) // Speed selection is in LOW_SPEED
   state.
175           {
176             if (ADC_Result0 > VHSStart)
177             {
178               SpeedMode = 2;  // Shift to HIGH_SPEED state.
179             }
180             else
181             {
182               if (ADC_Result0 < VLSStop)
183               {
184                 SpeedMode = 0;  // Shift to OFF state.
185               }
186               else  //Stay in LOW_SPEED state.
187               {
188                 if (DCBusState == 1)  // DC bus voltage is normal.
189                 {
190                   TargetSpeed = LowSpeedValue;  // Update TargetSpeed.
191                   CurrentLimitTarget = CurrentLimitLS;
192                   Command = 1;  // Start motor.
193                 }
194                 else  // DC bus voltage is abnormal.
195                 {
196                   TargetSpeed = 0;  // Reset TargetSpeed.
```

Code Listing 7 Dynamic Motor Current Limit Script Code

```
197                   CurrentLimitTarget = CurrentLimitOriginal;  // When
   the target speed is zero, motor current limit is restored back to the
   original limit.
198                   Command = 0;  // Stop motor.
199                 }
200               }
201             }
202           }
203
204           if(SpeedMode == 2)  // Speed selection is in HIGH_SPEED
   state.
205           {
206             if(ADC_Result0 < VHSStop)
207             {
208               SpeedMode = 1;  // Shift to LOW_SPEED state.
209             }
210             else  // Stay in HIGH_SPEED state.
211             {
212               if (DCBusState == 1)  // DC bus voltage is normal.
213               {
214                 // Target speed follows 2nd order polynomial curve for
   HS.
215                 TargetSpeed = HighSpeedValue; // Update TargetSpeed.
216                 CurrentLimitTarget = CurrentLimitHS;
217                 Command = 1;  // Start motor.
218               }
219               else  // DC bus voltage is abnormal.
220               {
221                 TargetSpeed = 0;  // Reset TargetSpeed.
222                 CurrentLimitTarget = CurrentLimitOriginal;  // When
   the target speed is zero, motor current limit is restored back to the
   original limit.
223                 Command = 0;  // Stop motor.
224               }
225             }
226           }
227         }
```

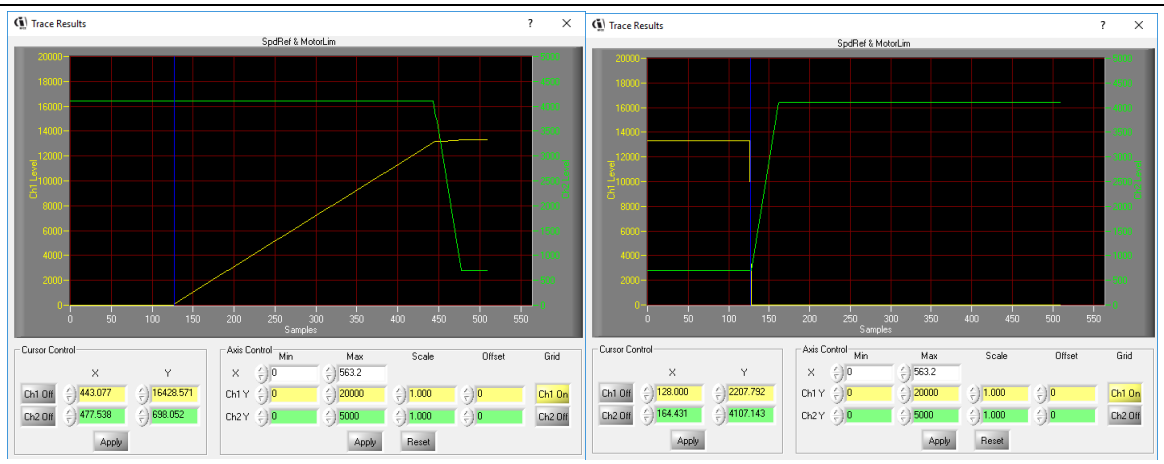
2.4.3 Dynamic Motor Current Limit Measurement Results

The following Figure 166 shows how the motor current limit was dynamically changed when the speed selection changed between OFF state and HIGH SPEED state. When the speed selection changed from OFF to 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 HIGH SPEED level (`CurrentLimitHS = 819`). When the speed selection changed from HIGH SPEED to 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.

Script Application Examples

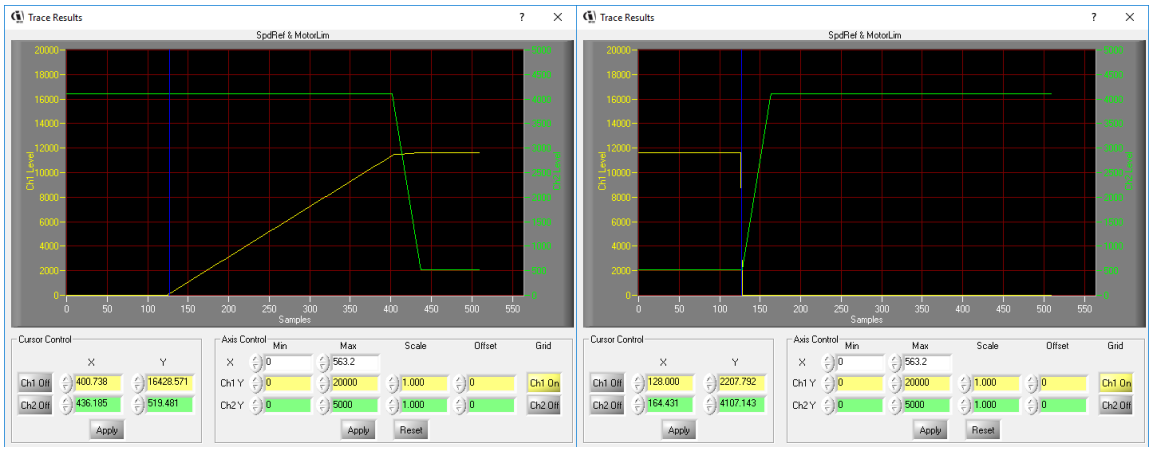
The following Figure 17 shows how the motor current limit was dynamically changed when the speed selection changed between OFF state and LOW SPEED state. When the speed selection changed from OFF to LOW SPEED state, the motor started to spin with its current limit `MotorLim` set to its original value 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 LOW SPEED level (`CurrentLimitLS = 519`). When the speed selection changed from LOW SPEED to 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.

The following Figure 18 shows how the motor current limit was dynamically changed when the speed selection changed between LOW SPEED state and HIGH SPEED state. When the speed selection changed from LOW SPEED to 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 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 HIGH SPEED level (`CurrentLimitHS = 819`). When the speed selection changed from HIGH SPEED to 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 LOW SPEED level (`CurrentLimitLS = 519`).



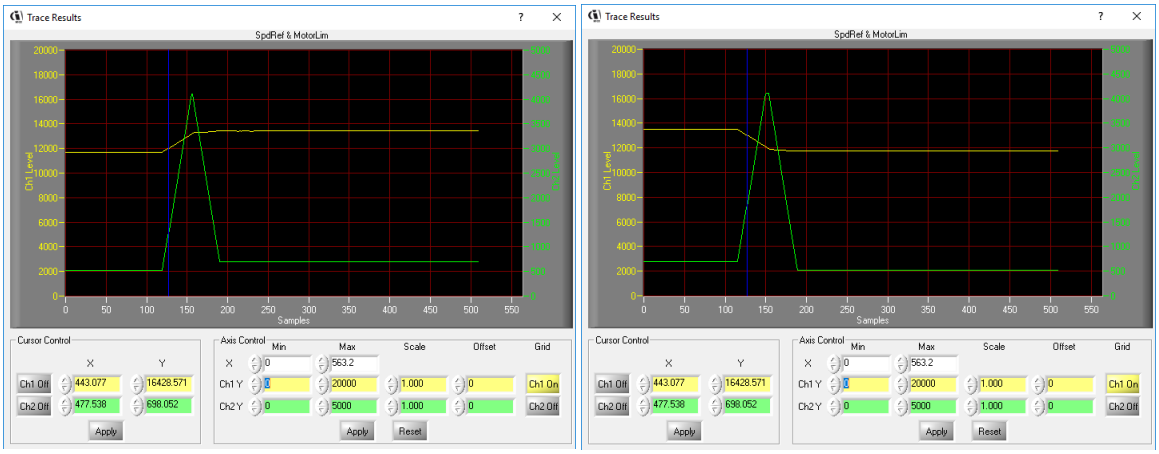
Ch1: `SpdRef`; Ch2: `MotorLim`; F_{PWM}: 20kHz; Sample at PWM frequency divided by 200; Trigger Level: 100; Speed Selection: OFF to HIGH SPEED (left) / HIGH SPEED to OFF (right); Vin: 100VAC

Figure 16 Motor Current Limit Screenshots (OFF <-> HIGH SPEED)



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

Figure 17 Motor Current Limit Screenshots (OFF <-> LOW SPEED)



Ch1: `SpdRef`; Ch2: `MotorLim`; F_{PWM}: 20kHz; 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: 100VAC

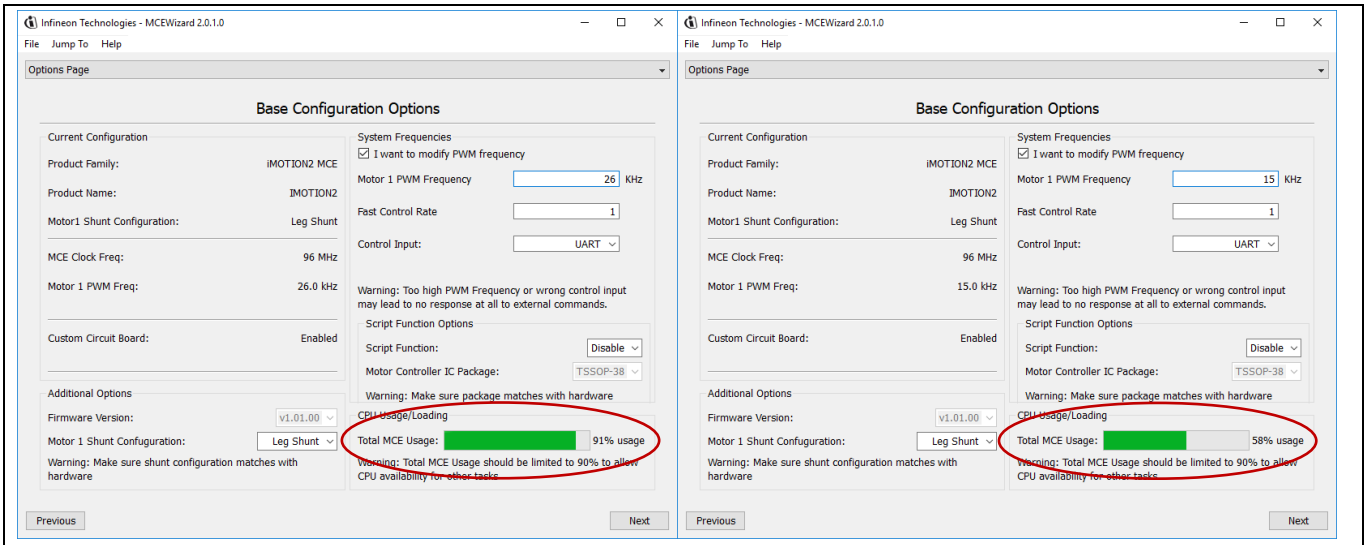
Figure 18 Motor Current Limit Screenshots (LOW SPEED <-> HIGH SPEED)

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.

The estimated CPU usage varies depending on the configuration of the motor or PFC PWM frequency as well as the safety functions. MCEWizard can be used to estimate the CPU usage. If the CPU usage estimation is higher than 90% as shown in the left screenshot of the following Figure 19, then enabling script function is likely to overload the CPU. It is highly recommended to keep the CPU usage estimation to no more than 90% when the users intend to enable the script function.

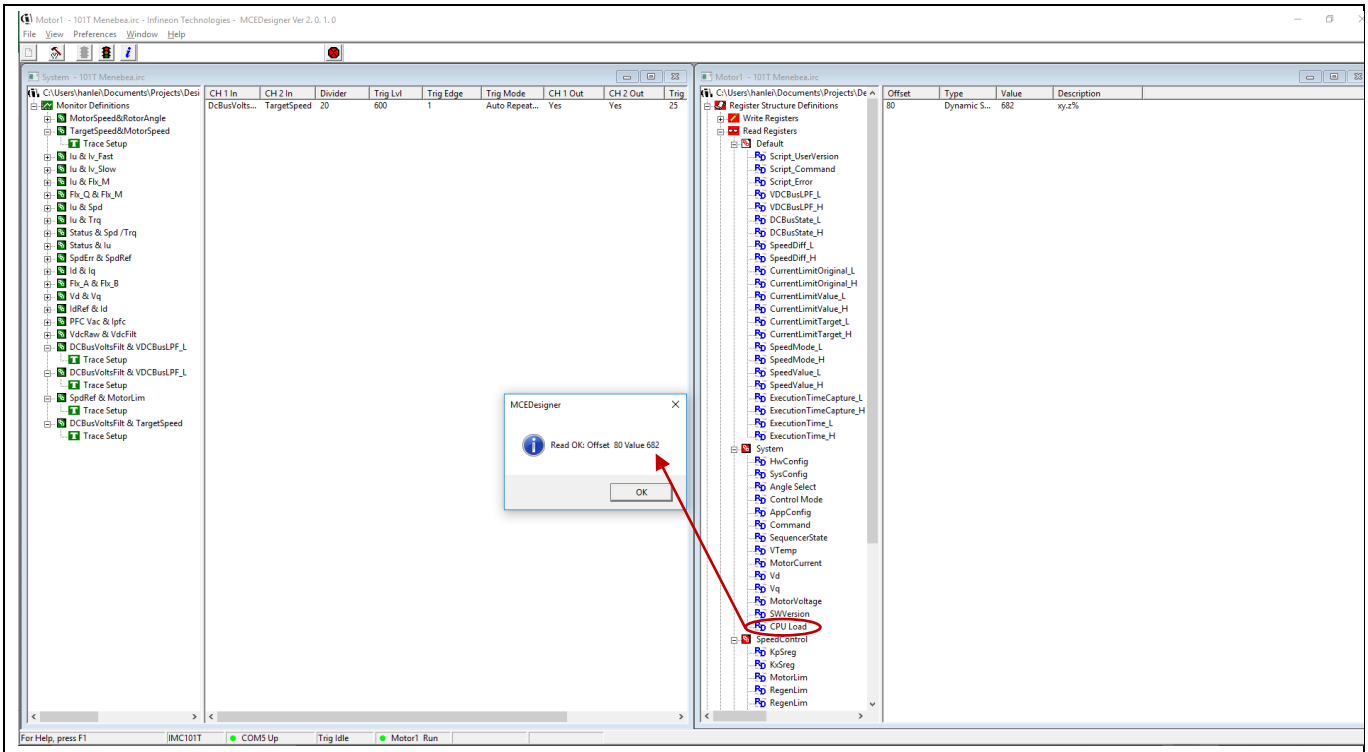


Left: CPU Usage = 91%; Right: CPU Usage = 58%

Figure 19 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]. The following Figure 20 shows that CPU load was 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 higher CPU load it would demand.



Speed Selection: LOW SPEED; Vin: 100VAC

Figure 20 Reading 'CPU Load' Parameter Using MCEDesigner When Script Is Running

3.2 Script Task Timing

3.2.1 Script Task Timing Setup

The script engine supports 2 independent tasks, namely, 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 higher 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

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 an example shown in the following Code Listing 8, 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.

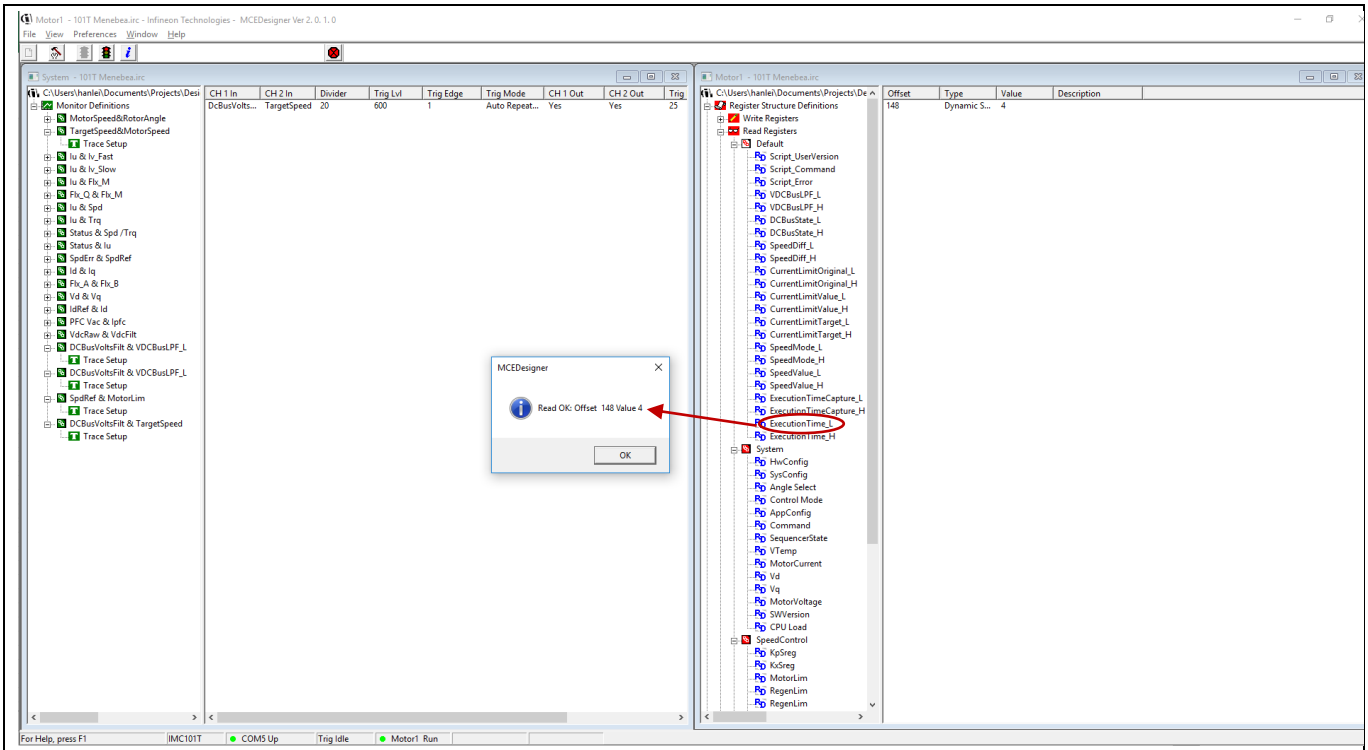
The script program described in Section 2.4 was used as an example to evaluate execution time for Task 1, whose execution period was set to 10 mS. The following Figure 21 shows the value of `ExecutionTime_L` (lower 16 bit of `ExecutionTime`) was 4 with the script enabled while the motor was running with 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 8 Execution Time Measurement for Task 1 Using `RunTimeCounter` in Script Code

```
001     /*************************************************************/
002     /* Global variable definition */
003     int ExecutionTimeCapture;
004     int ExecutionTime;
005     /*************************************************************/
006     /*Task1 script function*/
007     Script_Task1()
008     {
009           ExecutionTimeCapture = RunTimeCounter;
010     …
011     …
012     …
013           ExecutionTime = RunTimeCounter - ExecutionTimeCapture;
014     }
```



Speed Selection: LOW SPEED; Vin: 100VAC

Figure 21 Reading 'ExecutionTime_L' Variable Used in Script Code Using MCEDesigner

3.2.3 Script Task Execution Period Evaluation

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. The following Code Listing 9 shows an example of using `RunTimeCounter` to measure the loop execution period of Task 1.

Code Listing 9 Loop Execution Period Measurement for Task 1 Using `RunTimeCounter` in Script Code

```
001        /*************************************************************/
002        /* Global variable definition */
003        int LoopExecutionPeriodCapture;
004        int LoopExecutionPeriod;
005        /*************************************************************/
006        /*Task1 script function*/
007        Script_Task1()
008        {
009          LoopExecutionPeriod = RunTimeCounter -
   LoopExecutionPeriodCapture;
010          LoopExecutionPeriodCapture = RunTimeCounter;
011          …
012          …
013          …
014        }
```

4 Script Guidelines & Limitations

- The maximum number of global variables supported by the script engine is 30. The maximum number of local variables for each task is 24. The intercommunication between Task 0 and Task 1 can be realized by using global variables. Only global variables are accessible from MCEDesigner or user UART interface. It is recommended to define a variable as the global type if users intend to read its value during run time using MCEDesigner [3].
- The maximum allowed script code size is 16kB. This is equivalent to approximately 1500 lines of script code. The actual object code size is reported in the compiled script object file. An example is shown at line 006 in Code Listing 2.
- The script engine only supports 32 bit signed integer type of variables, so that the floating point type of variables or constants 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.
- 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 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 higher 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 MCE every 10 mS. According to Nyquist theorem, if the input analog signal frequency is higher than 50 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 higher than 50 Hz to minimize the aliasing effect.
- The GPIO pins are sampled and updated by MCE every 10 mS. Any GPIO input changes that occur faster than 10 mS will not be properly captured by the sampling method of MCE script engine. Similarly, any GPIO output changes that happen faster than 10 mS cannot be realized by using the script program. The fastest possible GPIO toggling frequency by using script is 50 Hz.
- It is recommended to change a specific GPIO pin value only once within the Task 0 or Task 1's loop. If there are more than one instances 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 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 then it gets reset at the end of Task 0, the actual GPIO will not toggle as expected. Instead, it will remain in reset state after the execution of Task 0 loop.
- For those time critical functions, it is recommended to be implemented in 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 to set the execution period of the script task to 50 mS typically.
- 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 higher than 500 Hz cannot be properly sampled and processed.
- The script language doesn't support the implementation of infra-red communication.
- The script language doesn't support customized UART communication protocols.
- The script language doesn't support TRIAC phase control interface due to the limitation of GPIO update rate.
- The script language can support momentary switch interfaces.
- The script language can support Programmable Logic Controller (PLC) as long as the minimum timing requirement is no less than 10 mS.
- The script language doesn't support the implementation of digital Hall Effect sensors.

5 References

[1] iMOTION™ IMC100 High Performance Motor Control IC Series Datasheet (REV 1.1).

[2] iMOTION™ Motor Control Engine Software Reference Manual (REV 1.1).

[3] MCEDesigner User Guide (REV 2.0.1.0).

Revision history

Document version	Date of release	Description of changes
1.0	9/5/2018	Initial release.

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

Edition 9/5/2018

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2018 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.