iMOTION™ Motion Control Engine

Functional Reference Manual

About this document

Scope and purpose

iMOTION ™ is a family of integrated products for the control of permanent magnet (PM) motors that combine industry proven hardware and ready to use software.

These devices are capable of performing sensorless or sensor-based Field-Oriented Control (FOC) across the entire speed range of the motor, ensuring stable control even at deep field weakening speeds. The iMOTION ™ software, deployed onto the hardware devices, is referred to as Motion Control Engine (MCE). A PC tool called the iMOTION ™ Solution Designer (iSD) is available for the configuration, validation, deployment and user script code development. In addition to motor control, the MCE provides a Power Factor Correction Control (PFC) option. Furthermore, the MCE supports a scripting function that allows users to implement system-level functionalities beyond motor control and PFC, thereby expanding the capabilities of the MCE.

This reference manual provides a comprehensive description of the features, protections, and configuration options of the MCE. However, it should be noted that an actual product may only incorporate a subset of these functionalities. For instance, power factor correction is exclusively available in dedicated devices. For detailed information, please consult the relevant data sheet for each specific device.

The electrical, mechanical, timing, and quality parameters of the iMOTION ™ products are outlined in their respective data sheets. These data sheets also specify the specific IO pins for the aforementioned functionalities.

Reference to Parameter Reference Manual (PRM) for parameters details.

Additional documentation on using specific features, application examples or code implementations is made available on the iMOTION ™ web pages.

Intended audience

This document is intended for users of iMOTION ™ devices

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 8
2 Motor Control . . . 8
2.1 State Handling . . . 10
2.2 Variable Scaling . . . 12
2.3 Bootstrap Capacitor Charge . . . 13
2.4 Voltage measurement . . . 14
2.5 Current Sensing Offset Measurement . . . 14
2.6 Current Measurement . . . 15
2.6.1 Leg Shunt Current Measurement . . . 16
2.6.2 Single Shunt Current Measurement . . . 18
2.6.2.1 Limitations of Single Shunt Current Reconstruction . . . 20
2.6.2.2 Phase Shift PWM . . . 21
2.6.2.3 Low Noise Phase Shift PWM . . . 24
2.6.2.4 Peak Current Tracking with No Phase Shift Window . . . 27
2.7 Initial Angle Sensing . . . 29
2.8 Hall Sensor Interface . . . 31
2.8.1 Interface Structure . . . 31
2.8.2 Hall Sample De-Bounce Filter . . . 32
2.8.3 Hall Angle Estimation . . . 32
2.8.3.1 Hall Angle Estimation with PLL . . . 32
2.8.3.2 Hall Angle Estimation without PLL . . . 34
2.8.4 Hall Zero-Speed Check . . . 35
2.8.5 Hall Pattern Validation . . . 35
2.8.6 Hall Angle Offset . . . 36
2.8.7 Atan Angle Calculation . . . 38
2.8.8 Hall Initial Position Estimation . . . 39
2.8.9 Hall Sensor/Sensorless Hybrid Operation . . . 40
2.9 DC Bus Compensation . . . 40
2.10 Motor Current Limit Profile . . . 42
2.11 Over-Modulation . . . 44
2.12 2-Phase Modulation . . . 45
2.13 Torque Compensation . . . 47
2.14 Zero-Vector Braking . . . 49
2.14.1 Parameters . . . 50
2.14.2 State Dependencies . . . 50
2.14.3 Zero-Vector Braking During Faults . . . 51
2.15 Catch Spin . . . 51

2.15.1 Zero Speed Catch Spin . . . 51
2.15.2 Forward Catch Spin . . . 52
2.15.3 Reverse Catch Spin . . . 54
2.16 Control Input . . . 55
2.16.1 UART control . . . 55
2.16.2 Vsp Analog Input . . . 55
2.16.3 Frequency input . . . 57
2.16.4 Duty Cycle Input Control . . . 58
2.16.5 Automatic Restart . . . 59
2.16.6 Forced control input change . . . 59
2.16.7 PG output . . . 59
2.16.8 Control Input Customization . . . 60
2.17 Protection . . . 60
2.17.1 Flux PLL Out-of-Control Protection . . . 61
2.17.2 Rotor Lock Protection . . . 62
2.17.3 Motor Over Current Protection (OCP) . . . 63
2.17.4 Over Temperature Protection . . . 66
2.17.5 DC Bus Over/Under Voltage Protection . . . 66
2.17.6 Phase Loss Protection . . . 67
2.17.7 Current Offset Calibration Protection . . . 67
2.17.8 Staus LED . . . 68
2.17.9 Execution Fault . . . 68

3 Power Factor Correction . . . 69
3.1 PFC Algorithm . . . 69
3.1.1 ADC Measurement . . . 70
3.1.2 Current Control . . . 70
3.1.3 Average Current Calculation . . . 70
3.1.4 Voltage Control . . . 72
3.1.5 Multiplier with Voltage Feed-Forward . . . 72
3.1.6 Notch Filter . . . 73
3.1.7 Zero-Cross Detection . . . 74
3.1.8 Soft Start . . . 75
3.1.9 Vout Ready Monitor . . . 75
3.1.10 Control Modes . . . 76
3.2 State Handling . . . 76
3.3 Scheduling and Timing . . . 80
3.3.1 1:1 Base Rate . . . 80
3.3.2 1:2 Base Rate . . . 81
3.3.3 1:3 Base Rate . . . 82
3.3.4 Co-existence of PFC and Motor Control Algorithm . . . 82
3.4 Protection . . . 82
3.4.1 Over Current Protection . . . 84

3.4.2 VAC Drop-out Protection . . . 87
3.4.3 VAC Over Voltage and Brown-out Protection . . . 88
3.4.4 Input Frequency Protection . . . 90
3.4.5 Vout Over Voltage and Open Loop Protection . . . 90
3.4.6 Current Measurement Offset . . . 91
3.4.7 Execution . . . 92

4 System . . . 92
4.1 Internal Oscillator Calibration . . . 92
4.1.1 Overview . . . 92
4.2 Multiple Parameter Programming . . . 92
4.2.1 Parameter Page Layout . . . 92
4.2.2 Parameter Block Selection . . . 92
4.2.2.1 UART Control . . . 93
4.2.2.2 Analog Input . . . 93
4.2.2.3 GPIO Pins . . . 93
4.2.3 Parameter load fault . . . 95
4.3 Low Power Standby . . . 95
4.3.1 Entry and Exit Conditions . . . 96
4.3.2 Scripting . . . 97
4.3.2.1 Timing . . . 97

5 Communication Interface . . . 97
5.1 User Mode UART . . . 97
5.1.1 Baud Rate . . . 97
5.1.2 Data Frame . . . 97
5.1.3 Node Address . . . 98
5.1.4 Link Break Protection . . . 98
5.1.5 Command . . . 98
5.1.6 Verification . . . 99
5.1.7 UART message . . . 100
5.1.7.1 Read Status: Command = 0x00 . . . 100
5.1.7.2 Clear Fault: Command = 0x01 . . . 101
5.1.7.3 Change Control Input Mode: Command = 0x02 . . . 101
5.1.7.4 Motor Control: Command = 0x03 . . . 101
5.1.7.5 Register Read: Command = 0x05 . . . 101
5.1.7.6 Register Read High-Word: Command = 0x0A . . . 102
5.1.7.7 Register Write: Command = 0x06 . . . 102
5.1.7.8 Register Write Low-Word: Command = 0x08 . . . 103
5.1.7.9 Register Write High-Word: Command = 0x09 . . . 103
5.1.7.10 Load Parameter Set: Command = 0x20 . . . 103
5.1.8 Connecting multiple nodes to same network . . . 104
5.1.9 UART Transmission Delay . . . 104

5.2 JCOM Inter-Chip Communication . . . 104
5.2.1 Operation Mode . . . 104
5.2.1.1 Asynchronous Mode . . . 104
5.2.2 Baud Rate . . . 105
5.2.3 Message Frame Structure . . . 105
5.2.4 Command and Response Protocol . . . 105
5.2.4.1 Message Object: 0 . . . 106
5.2.4.1.1 State Machine Inquiry . . . 106
5.2.4.2 Message Object: 1 . . . 106
5.2.4.2.1 System Configuration Protection . . . 108
5.2.4.2.2 Reset T Core . . . 108
5.2.4.2.3 Access Static Parameter . . . 108
5.2.4.2.4 Set Boot Mode . . . 108
5.2.4.2.5 Set JCOM Baud Rate . . . 108
5.2.4.3 Message Object: 6 . . . 109
5.2.4.3.1 Get Parameter . . . 109
5.2.4.4 Message Object: 7 . . . 109
5.2.4.4.1 Set Parameter . . . 109
5.2.4.5 Message Object: 8 . . . 110
5.2.4.5.1 Get Parameter Request . . . 110

6 Script Engine . . . 110
6.1 Overview . . . 111
6.2 Script Program Structure . . . 111
6.3 Script Program Execution . . . 111
6.3.1 Execution Time Adjustment . . . 112
6.4 Constants . . . 112
6.5 Variable types and scope . . . 112
6.5.1 Script Array define . . . 113
6.5.2 Mapping of Variables to Parameters . . . 113
6.6 MCE Parameter Access . . . 114
6.7 User defined function . . . 114
6.8 Operators . . . 114
6.9 Decision Structures . . . 116
6.10 Loop Structures . . . 117
6.11 Methods . . . 117
6.11.1 Bit access Methods . . . 117
6.11.2 Coherent update methods . . . 118
6.11.3 User GPIOs . . . 118
6.11.3.1 Digital Input and Output Pins . . . 118
6.11.3.2 Analog pins . . . 118
6.12 Plugins . . . 119
6.12.1 Configurable UART . . . 119

6.12.1.1 UART_DriverInit() . . . 119
6.12.1.2 UART_DriverDeinit() . . . 120
6.12.1.3 UART_FifoInit() . . . 121
6.12.1.4 UART_BufferInit() . . . 121
6.12.1.5 UART_GetStatus() . . . 122
6.12.1.6 UART_GetRxDelay() . . . 124
6.12.1.7 UART_Control() . . . 124
6.12.1.8 UART_RxFifo() . . . 126
6.12.1.9 UART_TxFifo() . . . 126
6.12.1.10 UART_RxBuffer() . . . 126
6.12.1.11 UART_TxBuffer() . . . 127
6.12.2 Flash Data Storage . . . 127
6.12.2.1 Flash Data Type . . . 127
6.12.2.2 Flash Data Storage APIs . . . 128
6.12.2.3 Flash_Write() . . . 128
6.12.2.4 Flash_Erase() . . . 128
6.12.2.5 Flash_WriteCount() . . . 129
6.12.2.6 Flash_GetStatus() . . . 129
6.12.2.7 Timing Considerations . . . 129
6.12.2.8 Endurance Considerations . . . 130
6.12.3 Infrared Interface . . . 130
6.12.3.1 Infrared Protocols . . . 130
6.12.3.2 IR Pins . . . 130
6.12.3.3 Infrared Interface APIs . . . 131
6.12.3.4 IR_DriverInit() . . . 131
6.12.3.5 IR_DriverDeinit() . . . 132
6.12.3.6 IR_RxBuffer() . . . 132
6.12.3.7 IR_GetStatus() . . . 133
6.12.3.8 IR_RxCommand() . . . 134
6.12.3.9 IR_RxAddress() . . . 135
6.12.3.10 IR_RxRepeats() . . . 135
6.12.3.11 IR_RxReceived() . . . 135
6.12.3.12 IR_RxRepeating() . . . 136
6.12.3.13 Timing Considerations . . . 136
6.12.3.14 Configuration . . . 136
6.12.4 I2C Interface . . . 136
6.12.4.1 I2C Interface API . . . 137
6.12.4.2 I2C_DriverInit() . . . 137
6.12.4.3 I2C_DriverDeInit . . . 138
6.12.4.4 I2C_MasterStart() . . . 138
6.12.4.5 I2C_MasterRepeatedStart() . . . 138
6.12.4.6 I2C_MasterStop() . . . 139

6.12.4.7 I2C_Transmit() 139
6.12.4.8 I2C_GetDataACK() 139
6.12.4.9 I2C_GetDataNACK() 140
6.12.4.10 I2C_GetRxFifo() 140
6.12.4.11 I2C_GetStatus() 140
6.12.4.12 I2c_Control() 141
6.12.5 TRIAC Control 142
6.12.5.1 TRIAC Pins 145
6.12.5.2 TRIAC Interface APIs 145
6.12.5.3 TRIAC_DriverInit() 146
6.12.5.4 TRIAC_DriverDeInit() 147
6.12.5.5 TRIAC_SetONOFFTimes() 147
6.12.5.6 TRIAC_GetStatus() 148
6.12.5.7 TRIAC_ClearStatus() 148
6.12.5.8 TRIAC_GetOnTime() 149
6.12.5.9 TRIAC_GetOffTime() 149

Revision history 150

Disclaimer 151

1 Introduction

This document describes the motor control, power factor correction, and additional functions available in the iMOTION ™ Motor Control Engine (MCE). The key features of the MCE are:

- Sensorless FOC control: Achieve high-performance sensorless Field Oriented Control (FOC) of Permanent Magnet Synchronous Motors, including surface mounted and interior mount magnet motors. This is accomplished by utilizing fast ADC, integrated op-amps, comparators, and motion peripherals of iMOTION ™ devices
- Hall sensor-based FOC control: Support 2/3 digital Hall sensor and 2 analog Hall sensor configurations.
- Angle sensing for initial rotor angle detection: In combination with direct closed-loop start, the use of angle sensing enhances motor start performance
- Single shunt or leg shunt motor current sensing: Offer unique single shunt and leg shunt current reconstruction capabilities. Integrated op-amps with configurable gain and A/D converter enable direct shunt resistor interface to the iMOTION ™ device, eliminating the need for additional analog/digital circuitry. The single shunt option can utilize either the minimum pulse method or the phase shift method. Phase Shift PWM provides improved startup and low-speed performance in a single shunt configuration
- Support for 3-phase and 2-phase PWM modulation: Enable 2-phase SVPWM (Type-3) to reduce switching losses compared to 3-phase SVPWM, due to symmetrical placement of zero vectors
- Enhanced flux-based control algorithm for quick and smooth start: Attain direct closed-loop control of both torque and stator flux (field weakening) using proportional-integral controllers and space vector modulation with over modulation strategy
- Supports Boost Mode Power Factor Correction (PFC)
- Networking capability with user mode UART: Available in both master and slave modes, supporting up to 15 nodes. Each node possesses its own address, and a broadcast feature is available to update all the slaves simultaneously
- 15 re-programmable parameter blocks: Users can program 15 configuration blocks to save control parameters. Each parameter block has a size of 256 bytes and can be programmed individually or all at once using the Solution Designer
- Multiple motor parameter support: Each parameter block can be assigned to different motors or hardware platforms
- Scripting support to enable users to write system level functionalities above the motor control and PFC

2 Motor Control

The MCE provides an advanced sensorless or Hall sensor based Field Oriented Control (FOC) algorithm to drive Permanent Magnet Synchronous Motor (PMSM) loads, including constant air-gap surface mounted permanent magnet (SPM) motors and interior permanent magnet (IPM) motors with variable-reluctance. A top-level sensorless/Hall sensor based FOC algorithm structure is depicted in Figure 1. The implementation follows the well-established cascaded control structure with an outer speed loop and inner current control loops that adjust the motor winding voltages to drive the motor to the target speed. The field weakening block extends the speed range of the drive.

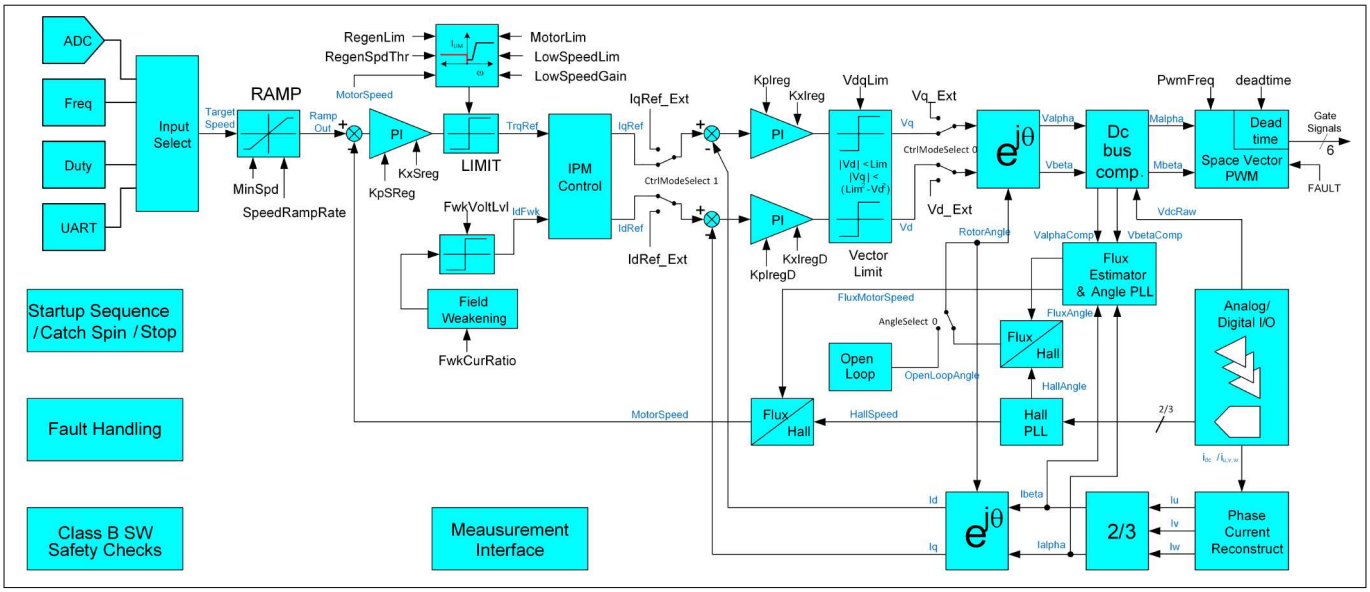


Figure 1 **Top Level Diagram of Speed Control Loop and Sensorless FOC**

The speed controller calculates the motor torque required to follow the target speed. While the current loops drive the motor currents needed to generate this torque. The proportional plus integral (PI) speed loop compensator acts on the error between the target speed and the actual (estimated) speed. The integral term forces the steady state error to zero while the proportional term improves the high frequency response. The PI compensator gains are adjusted depending on the motor and load characteristics to meet the target dynamic performance. The limiting function on the output of the PI compensator prevents integral windup and maintains the motor currents within the motor and drive capability.

The current loops calculate the inverter voltages to drive the motor currents needed to generate the desired torque. Field oriented control (FOC) uses the Clarke transform and a vector rotation to transform the motor winding currents into two quasi dc components, an I_d component that reinforces or weakens the rotor field and an I_q component that generates motor torque.

Two separate regulators control the I_d and I_q currents and a forward vector rotation transforms the current loop output voltages V_d and V_q into the two phase ac components (V_α and V_β). A DC bus compensation function adjusts the modulation index as a function of the dc bus voltage to reject dc bus ripple and improve current loop stability. The Space Vector Pulse Width Modulator (SVPWM) generates the three phase power inverter switching signals based on the V_α and V_β voltage inputs.

Typically, the I_q controller input is the torque reference from the speed controller and the I_d reference current is set to zero. However, above a certain speed, known as the base speed, the inverter output voltage becomes limited by the dc bus voltage. In this situation, the field weakening controller generates a negative I_d to oppose the rotor magnet field that reduces the winding back EMF. This enables operation at higher speeds but at a lower torque output. The controller includes a compensator that adjusts the I_d current to maintain the motor voltage magnitude within the bus voltage limit.

The rotor magnet position estimator consists of a flux estimator and PLL. Flux is calculated based on feedback current, estimated voltages (based on dc bus feedback voltage and modulation index) and motor parameters (inductance and resistance). The output of the flux estimator represents rotor magnet flux in the Alpha-Beta (stationary orthogonal frame, u-phase aligned with Alpha) two-phase components. The angle and frequency phase locked loop (PLL) estimates the flux angle and speed from the rotor magnet flux vector in Alpha-Beta components. The vector rotation calculates the error between the rotor flux angle and the estimated angle. The PI compensator and integrator in the closed loop path force angle and frequency estimate to track the angle and frequency of the rotor flux. The motor speed is derived from the rotor frequency according to the number of rotor poles.

When driving an interior permanent magnet (IPM) motor the rotor saliency can generate a reluctance torque component to augment the torque produced by the rotor magnet. When driving a surface magnet motor, there is zero saliency ($L_d = L_q$) and I_d is set to zero for maximum efficiency. In the case of IPM motor which has saliency ($L_d < L_q$) a negative I_d will produce positive reluctance torque. The most efficient operating point is when the total torque is maximized for a given current magnitude.

2.1 State Handling

The control software has a number of different operating states to support the various transient operating conditions between drive power-up and stable running of the motor under closed loop sensorless control. These include preparation of the drive for starting, running the motor before the flux estimator reaches stable operation, starting a motor that is already running and handling fault conditions. The Motion Control Engine includes a built-in state machine that takes care of all state-handling for starting, stopping and performing start-up. A state machine function is executed periodically (by default, every 1 ms). In total there are 12 states; each state has a value between 0-12, the current state of the sequencer is stored in “Motor_SequencerState” variable.

Table 1 State Description and Transition

State No	Sequence State	State Functionality	Transition Event	Next Sequence State
0	IDLE	After the controller power up, control enters into this state. If there is no valid parameter block, sequencer stays in this state	Parameters are loaded successfully	STOP
1	STOP	Wait for start command. Current and voltage measurement are done for protection	Current Amplifier offset calculation is not done	OFFSETCAL
			Start Command	BTSCHARGE
2	OFFSETCAL	Offset calculation for motor current sensing input	Current offset calculation completed	STOP
3	BTSCHARGE	Boot strap capacitor pre-charge. Current and voltage measurement are done for protection	Bootstrap capacitor charge completed	CATCHSPIN ANGLESENSE PARKING OPENLOOP MOTOR_RUN (Flux/Hall/ Hybrid/ Openloop)
4/ 10/ 11/ 12	MOTOR_RUN (Flux/Hall/ Hybrid/ Openloop)	Normal motor run mode in flux/ hall/hybrid based rotor angle estimation	Stop Command	STOP
5	FAULT	If any fault detected, motor will be stopped (if it was previously running) and enter FAULT state from any other state	In UART control mode, Fault clear command by writing 1 to “FaultClear”variable	STOP

(table continues...)

Table 1 (continued) State Description and Transition

State No	Sequence State	State Functionality	Transition Event	Next Sequence State
			In Frequency/Duty/VSP input control modes, after configured time	STOP
6	CATCHSPIN	Flux estimator and flux PLL are running in order to detect the rotor position and measure the motor speed of free running motor. Speed regulator is disabled and the Id & Iq current commands are set to 0	Measured absolute motor speed is above threshold (“DirectStartThr” parameter)	MOTOR_RUN (Flux)
			Measured absolute motor speed is less than threshold (“DirectStartThr” parameter)	ANGLESENSE PARKING OPENLOOP MOTOR_RUN (Flux)
7	PARKING	Parking state is to align the rotor to a known position by injecting a linearly increased current. The final current amplitude is decided by low speed current limit. Total time duration of this state is configured by “ParkTime” register	Parking completed	OPENLOOP MOTOR_RUN (Flux)
8	OPENLOOP	Move the rotor and accelerate from speed zero to MinSpd by using open loop angle. Flux estimator and flux PLL are executed in this state in order to provide smooth transition to MOTOR_RUN state. Speed acceleration of the open loop angle is configured by “OpenLoopRamp” register	Speed reaches “MinSpd” register value	MOTOR_RUN (Flux/Hall/ Hybrid)
9	ANGLESENSE	Measure the initial rotor angle. The length of each sensing pulse is configured by “IS_Pulses” (in PWM cycles) register	Angle Sensing completed	MOTOR_RUN (Flux)
13	STAND-BY	The MCE lowers standby power consumption by reducing the CPU clock and switching off some of the controller peripherals.	System is in a stopped state, there are no faults and a configured delay time has expired. When fault occurs it goes to FAULT.	STOP FAULT

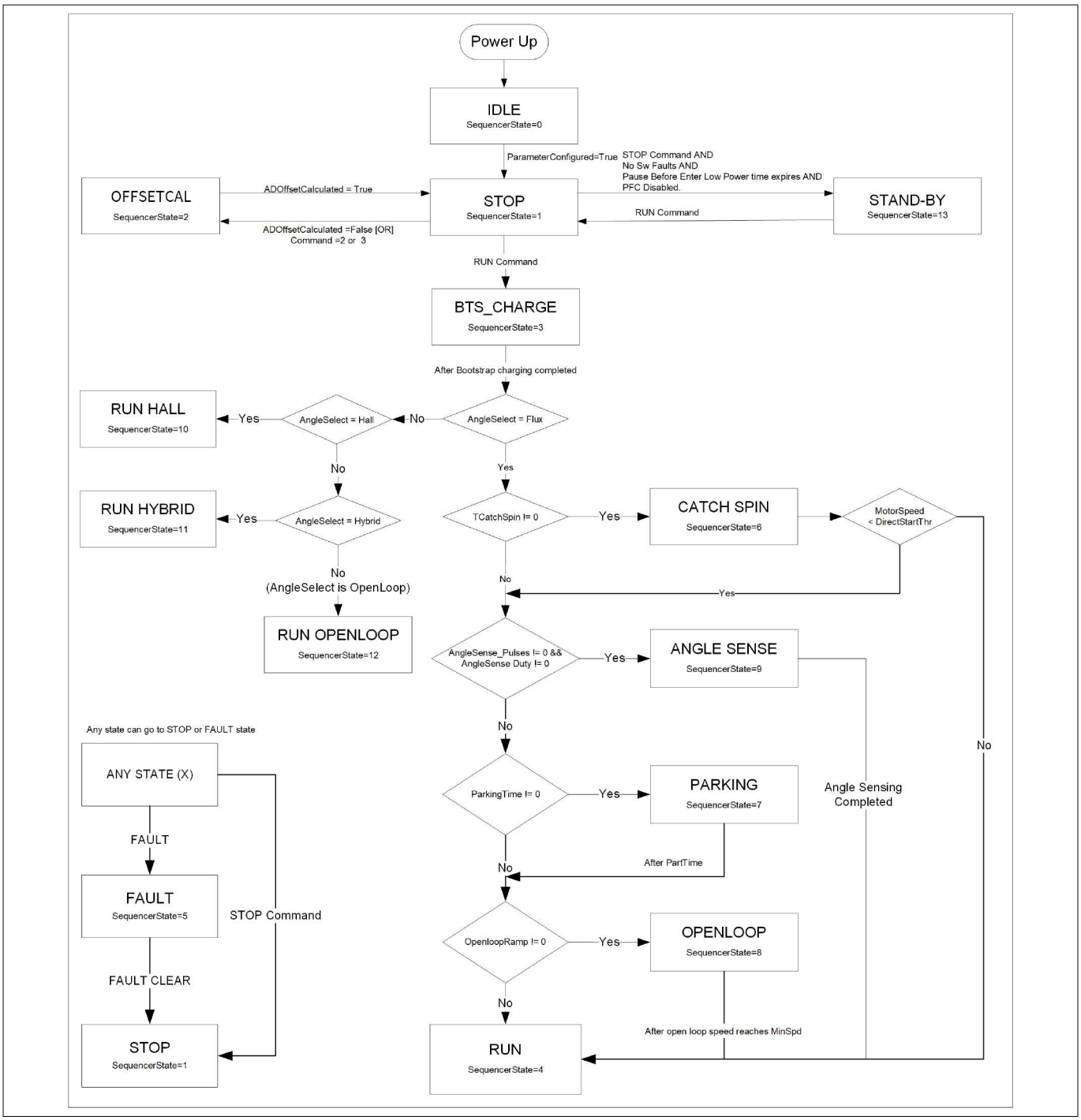


Figure 2 **State Handling and Start Control Flow Chart**

2.2 Variable Scaling

The MCE implements the control algorithms on a fixed point CPU core where physical voltage and current signals are represented by fixed point integers. The MCE algorithm uses appropriate scaling for control parameters and variables to optimize the precision of the motor and PFC control calculations. While all control parameters and variables are stored as integers the MCE Ecosystem tools support display of control variables and parameter settings as real numbers scaled to physical values.

The Figure 3 below describes the scaling used in different domains. In the hardware reference frame, current and voltage measurements are scaled according to the input circuit scaling and the resolution of the ADC. The

α-β and d-q quasi-dc voltages are defined by the PWM modulator resolution and inverter DC bus voltage capability. There is different motor current scaling in the AC and control reference frames. The α-β current scaling is defined by the measurement scaling while the d-q scaling is defined by the motor current ratings. The motor speed scaling is defined by the application requirements. There are three different time scales, the Hardware timing is defined by the IC peripheral clock; the sampling and control timing is set by the PWM frequency while the Application reference frame timing is fixed. All control parameter scaling is derived from the control variable and time scaling for the relevant reference frame.

Reference to iMOTION ™ Solution Designer dashboard parameter tree section Global->Scaling for scaling factor value details.

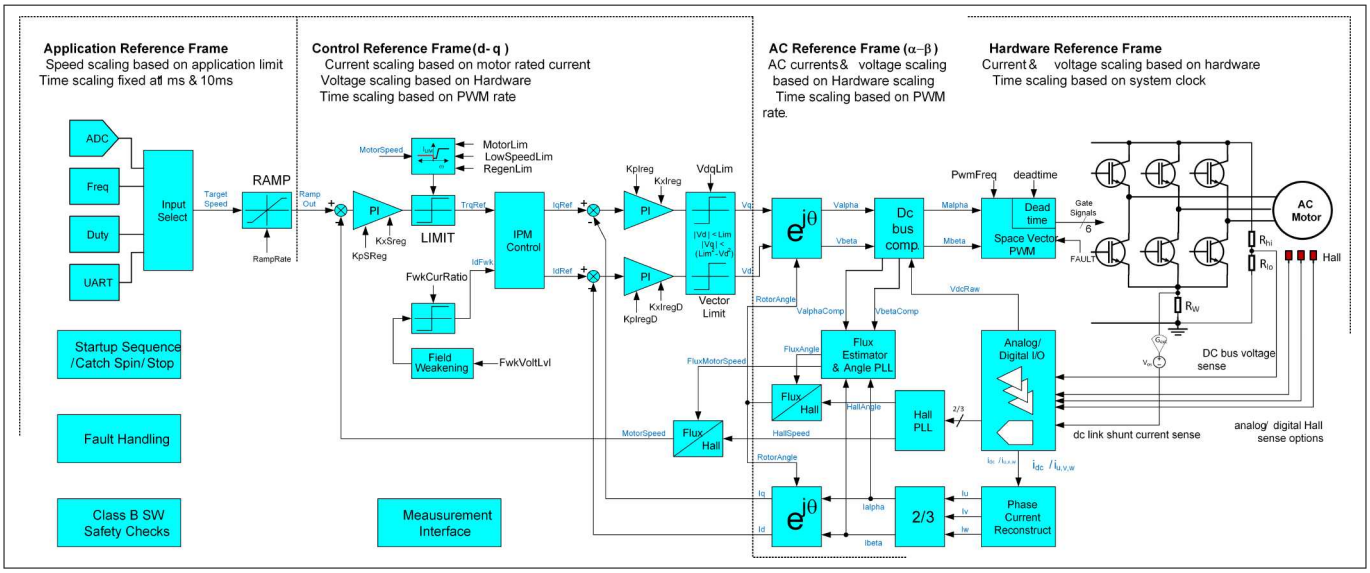


Figure 3 **Scaling Domains for Control Variables and Parameters**

2.3 Bootstrap Capacitor Charge

Bootstrap capacitors are charged by turning on all three low side switches. The charging current is limited by the built-in pre-charge control function.

Instead of charging all low side devices simultaneously, the gate pre-charge control will schedule an alternating (U, V, W phase) charging sequence. Each phase charges the bootstrap capacitor for a duration of 1/3rd of the PWM cycle so each capacitor charge time is 1/3rd of the total pre-charge time.

Figure 4 illustrates the PWM signal during bootstrap capacitor charge state.

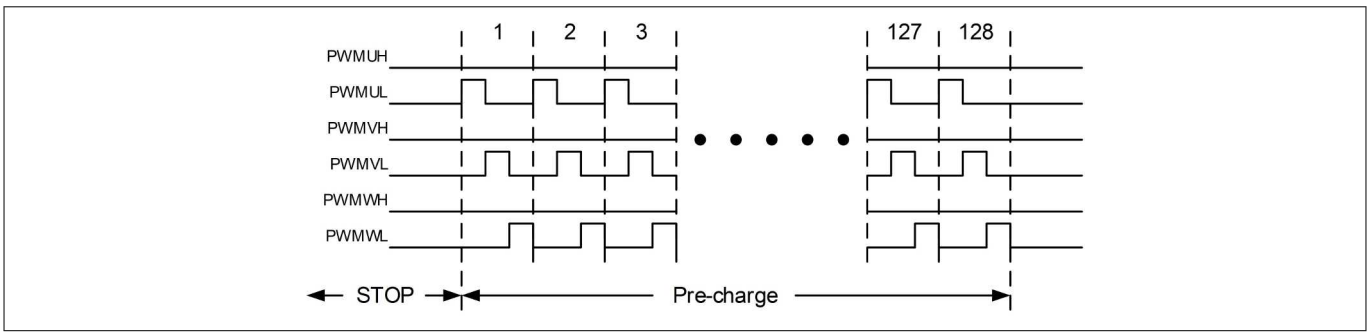


Figure 4 **Bootstrap Capacitor Pre-charge**

Total pre-charge time for each phase can be calculated from: $T_{Charge} = \frac{BtsChargeTime}{3 \times F_{PWM}}$ where the parameter 'BtsChargeTime' is the number of pre-charge PWM cycles.

For example, if PWM frequency is 10 kHz, and BtsChargeTime is 100, then the pre-charge time of each phase will be: $\frac{100}{3 \times 10000} = 3.333\left(ms\right)$.

2.4 Voltage measurement

The measurement of the DC bus voltage of the inverter board is required for voltage protection and DC bus voltage compensation. The voltage is measured at every PWM cycle. DC bus voltage of the inverter is measurement via a voltage divider circuit using 12-bit ADC. Measured DC bus voltage is internally represented in 12 bit format.

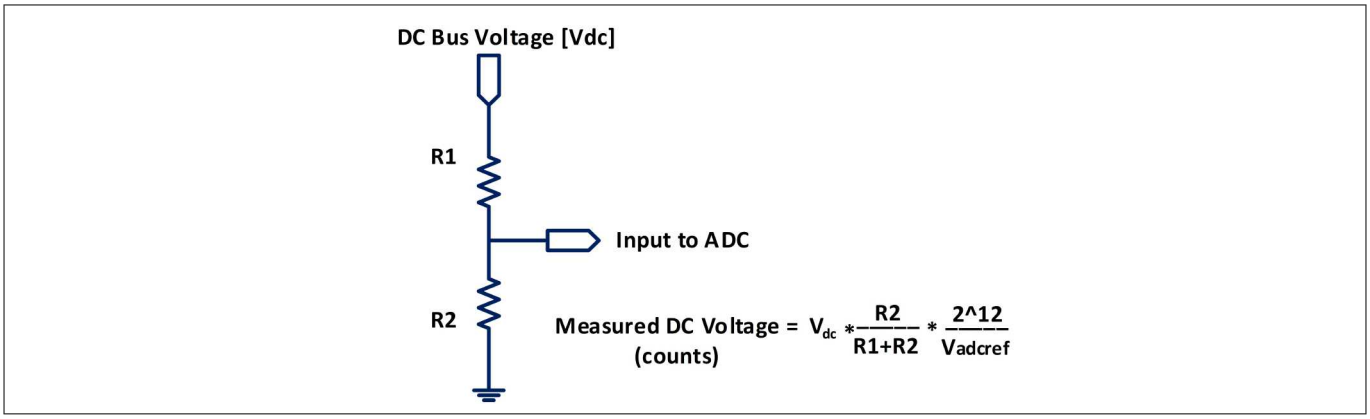


Figure 5 DC Bus voltage feedback signal path

Example: R1 = 2 MΩ, R2 = 13.3 kΩ , Vadcref= 3.3 V and Vdc = 320 V; Measured DC bus voltage = 2623 counts

***Attention**: In Solution Designer R1 and R2 values shall be configured as per actual hardware used. Wrong configuration may lead to wrong under voltage/over voltage/Critical over voltage fault or over voltage/under voltage/critical over voltage conditions may not be detected correctly.*

2.5 Current Sensing Offset Measurement

The current sensing offset is measured by the MCE in the OFFSETCAL state. This is achieved by calculating the average of the configured samples when the inverter is not switching (all six PWM outputs are set to the configured passive level) and when there is no motor phase current flowing through the shunt resistor(s). The number of current input offset samples is determined by the value of the "FB_MEASURE.OffsetSample" parameter. ADC sampling is performed during every base rate period, which is triggered by the PWM unit and occurs at the rate of the PWM period multiplied by the fast control rate.

The number of ADC samples = $2^{\text{FB_MEASURE.OffsetSample}}$; By Default, FB_MEASURE.OffsetSample = 13, so number of samples is 8192

In the event of a single shunt configuration (where "APP_MOTOR0.HwConfig.ShuntType" equals 0), the MCE measures the current sensing offset value at the Iss pin. The current input offset value is sampled independently for both the negative and positive current input ADC channels, and the average value is then stored in the variables "FB_MEASURE.IOffset0" and "FB_MEASURE.IOffset1" respectively.

In the event of a 2-phase leg shunt configuration ("APP_MOTOR0.HwConfig.ShuntType" = 1 and "APP_MOTOR0.HwConfig.LegShuntType" = 0), the MCE (Motor Control Electronics) measures the current sensing offset value at the Iu pin for phase U and at the Iv pin for phase V. These values are then averaged and saved into the variables "FB_MEASURE.IOffset0" and "FB_MEASURE.IOffset1" respectively.

In the event of a 3-phase leg shunt configuration ("APP_MOTOR0.HwConfig.ShuntType" = 1 and "APP_MOTOR0.HwConfig.LegShuntType" = 1), the MCE measures the current sensing offset value at the Iu pin for phase U, at the Iv pin for phase V, and at the Iw pin for phase W. The measured values are then averaged and

saved into the variables "FB_MEASURE.IOffset0," "FB_MEASURE.IOffset1," and "FB_MEASURE.IOffset2," respectively.

The duration of OFFSETCAL state time can be estimated using the following equation:

$T_{Offset_Cal} = \frac{Fast_Control_Rate \times 2^{FB_MESURE.OffsetSample}}{F_{PWM}}$; By Default, FB_MEASURE.OffsetSample =13, so the number of sameples is 8192

Users have the option to enable or disable current offset calculations after a fault is cleared. If enabled, the system will perform current offset calculations again when a fault occurs. If the value of the "APP_MOTOR0.HwConfig.CurrentoffSetComp" bitfield is set to 1, the current offset calculation will no longer take place and will remain as the initial value. Offset Calibration can be triggered by setting the bit 1 of the "APP_MOTOR0.Command" variable while the system is in a stopped state.

2.6 Current Measurement

In order to implement field oriented control, it is crucial to measure the motor winding currents precisely. Motor phase current values are used for the current control and flux estimator. Current is measured at every PWM cycle. The following Table 2 summarizes all the current measurement configurations supported by the MCE. The details of each configuration and its relevant PWM schemes will be described in the following sections.

Table 2 Current Measurement Configurations&PWM Schemes

Current Measurement Configurations		Needed Number of Shunt Resistors	PWM Schemes
Leg shunt -	2 Phase current measurement	2	Center aligned symmetrical PWM
	3 Phase current measurment	3	
Single shunt		1	Center aligned asymmetrical PWM • Phase shift PMW • Low noise phase shift PMW

The internal amplifiers are used for current measurement, so no external op-amp is required. The gain of the internal amplifier can be configured using the Solution Designer.

The following Figure 6 shows the details of the motor phase current feedback signal path.

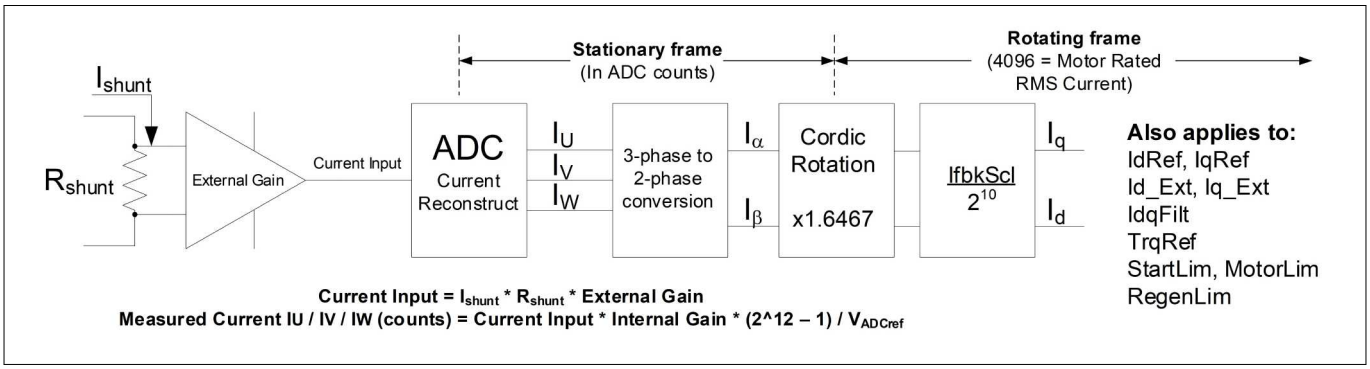


Figure 6 Motor current feedback signal path (TminPhaseShift ≠ 0)

***Attention**: In the Solution Designer current input value shall be configured as per actual hardware used. Wrong configuration may lead to wrong over current fault or over current conditions may not be detected correctly.*

2.6.1 Leg Shunt Current Measurement

Leg-shunt current sensing configuration uses 3 shunt resistors to sense the 3 inverter phases as shown in the following Figure 7. For 2 phase current sensing, the MCE only senses phase U and phase V current, and phase W current is calculated assuming the sum of the three phase current values is zero. For 3 phase current sensing, the MCE senses all three phase currents. The motor phase current would flow through the shunt resistor only when the low-side switch is closed. Accordingly, the MCE chooses to sense the motor phase current values during the zero vector [000] time in the vicinity of the start of a PWM cycle. Accordingly, a minimum duration of zero vector [000] (T_{GB_min}) as shown in Figure 8 is needed to ensure proper sampling of motor phase current values. This minimum duration can be specified by using the parameter 'PwmGuardBand' following this equation. $T_{GB_{\min}} = PwmGuardBand \times 10.417ns$. Thanks to this minimum duration of zero vector [000], the ON time of each phase PWM signal would never be longer than $T_{PWM} - T_{BGmin}$.

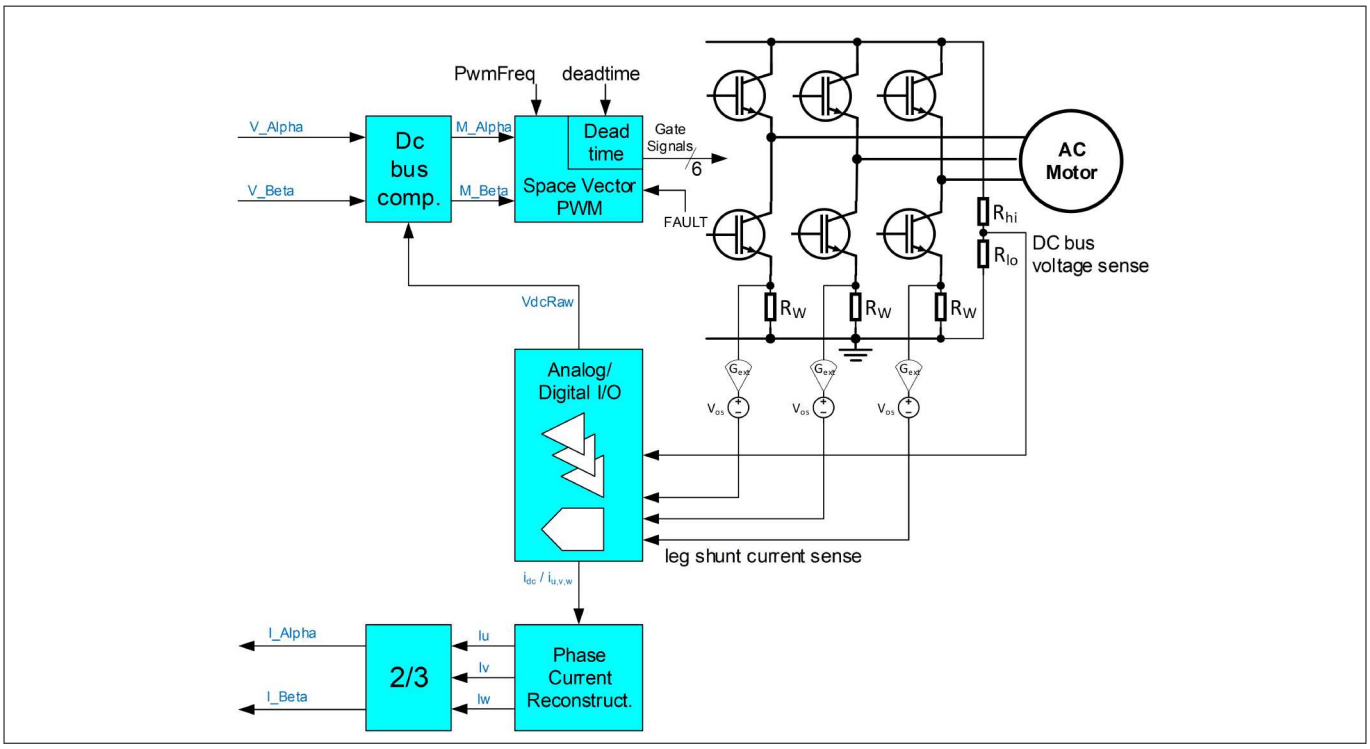


Figure 7 Typical Circuit Diagram for Leg Shunt Current Measurement Configuration

The current sensing timing for 2-phase leg shunt configuration is shown in the following Figure 8. In each PWM cycle, T_a and T_b refer to the 2 active vector time respectively, and 2 x T_0 refers to the total zero vector ([000], [111]) time. The duration of zero vector [111] is the same as that of zero vector [000]. The first current sensing point (CS1) is the time to sense phase U current, and it occurs $T_{SD} + \frac{4}{f_{PCLK}} = T_{SD} + 41.668ns$ after the start of a PWM cycle. T_{SD} is the needed ADC sampling delay time, and it can be positive or negative as required T_{SD}, can be configured by using the parameter 'SHDelay' following this equation $T_{SD} = SHDelay \times 10.417ns$. The ADC sampling time T_{sample} is about, and the ADC conversion time $T_{conversation}$ is about . The second current sensing point (CS2) is the time to sense phase V current. CS2 occurs right after the completion of the CS1 sampling and conversion operation.

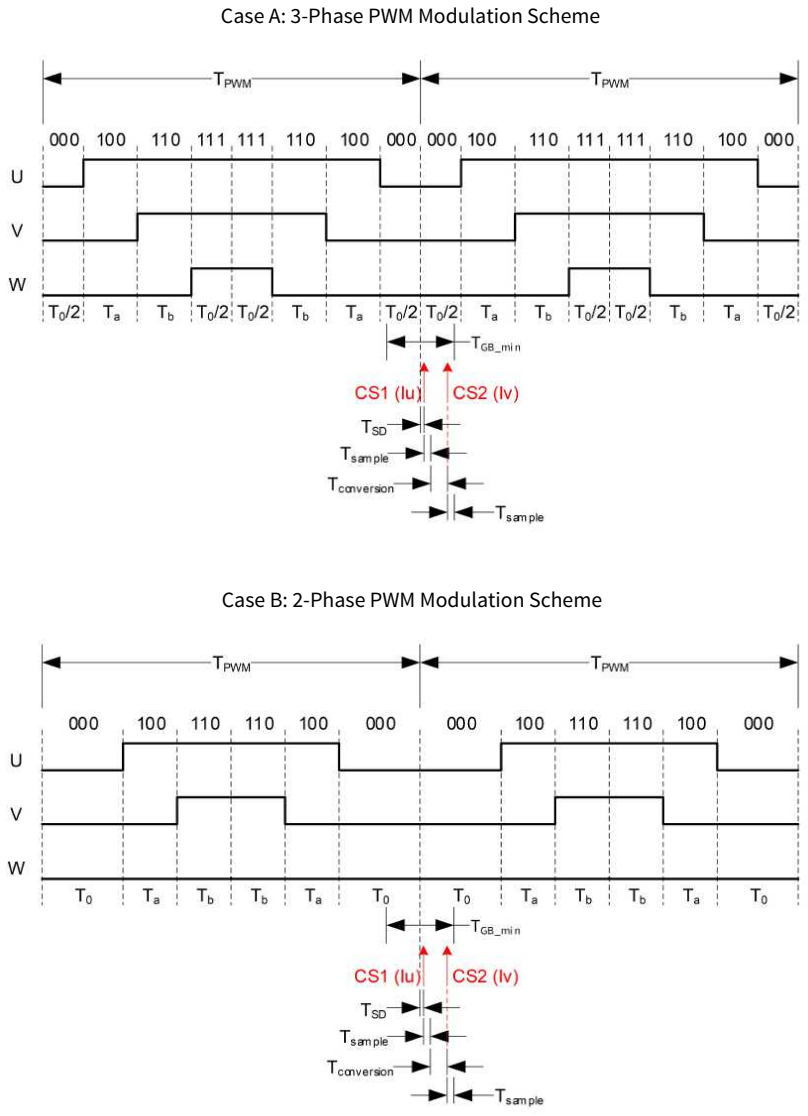


Figure 8 **Leg Shunt Configuration Current Sensing Timing Diagram**

The current sensing timing for the 3-phase leg shunt configuration is shown in the following Figure 9. The first current sensing point (CS1) is the time to sense phase U current. The second current sensing point (CS2) is the time to sense phase V current. CS2 occurs right after the completion of the CS1 sampling and conversion operation. The third current sensing point (CS3) is the time to sense phase W current. CS3 occurs right after the completion of the CS2 sampling and conversion operation.

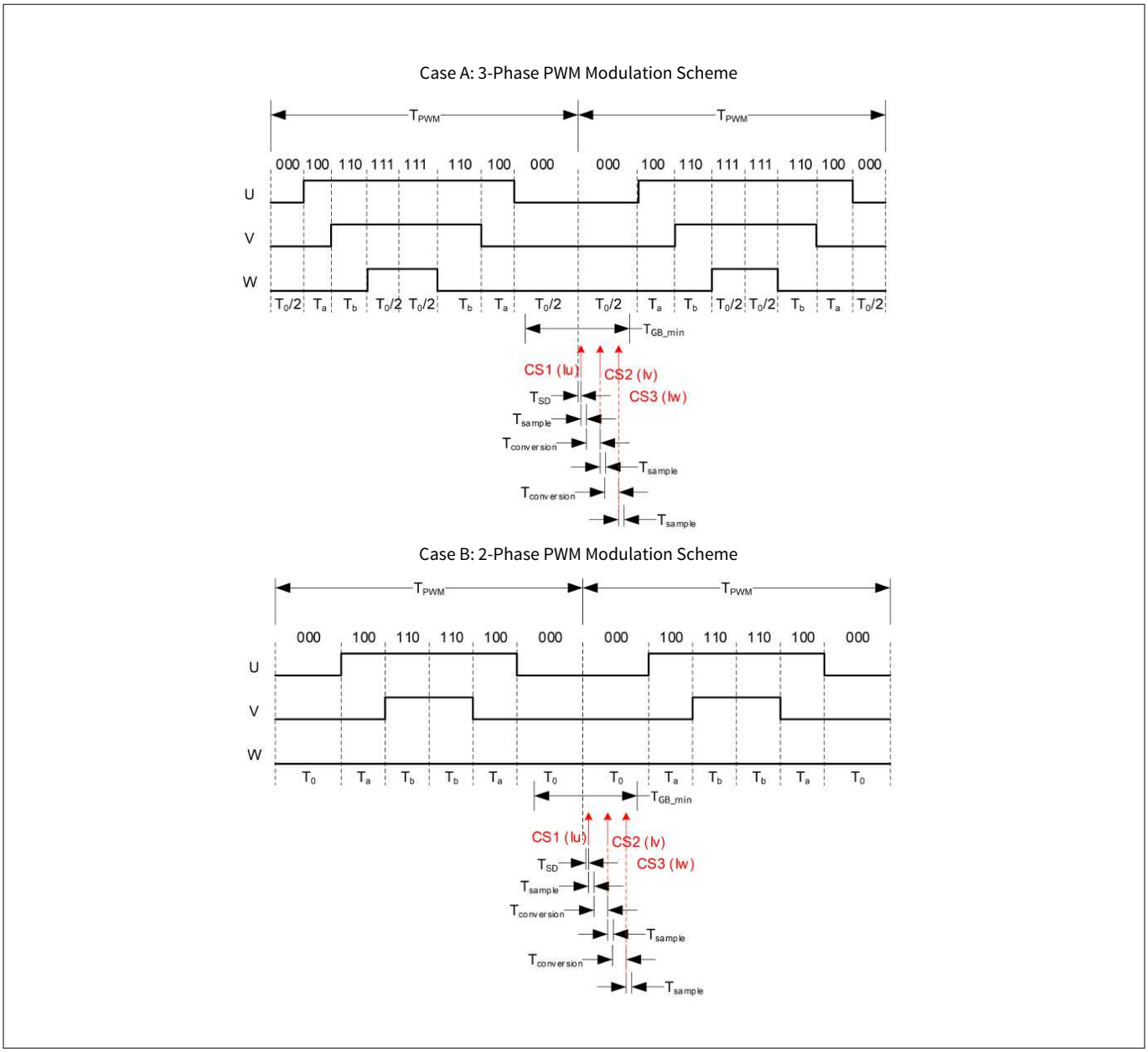


Figure 9 **Leg Shunt Configuration 3-Phase Current Sensing Timing Diagram**

2.6.2 Single Shunt Current Measurement

Single-shunt current sensing configuration uses only one shunt resistor to sense the DC link current as shown in Figure 10. It is often used for the sake of cost advantage. With single-shunt configuration, only the DC link current can be sampled by the MCE. The motor phase currents, needed for control feedback, can be extracted from DC link current when the active (non-zero) vectors are being applied during each PWM cycle. Two different active vectors are applied during each PWM cycle, and the DC link current during each active vector time represents some specific motor phase current depending on sector information. The third motor phase current value can be calculated assuming the sum of the three phase current values is zero.

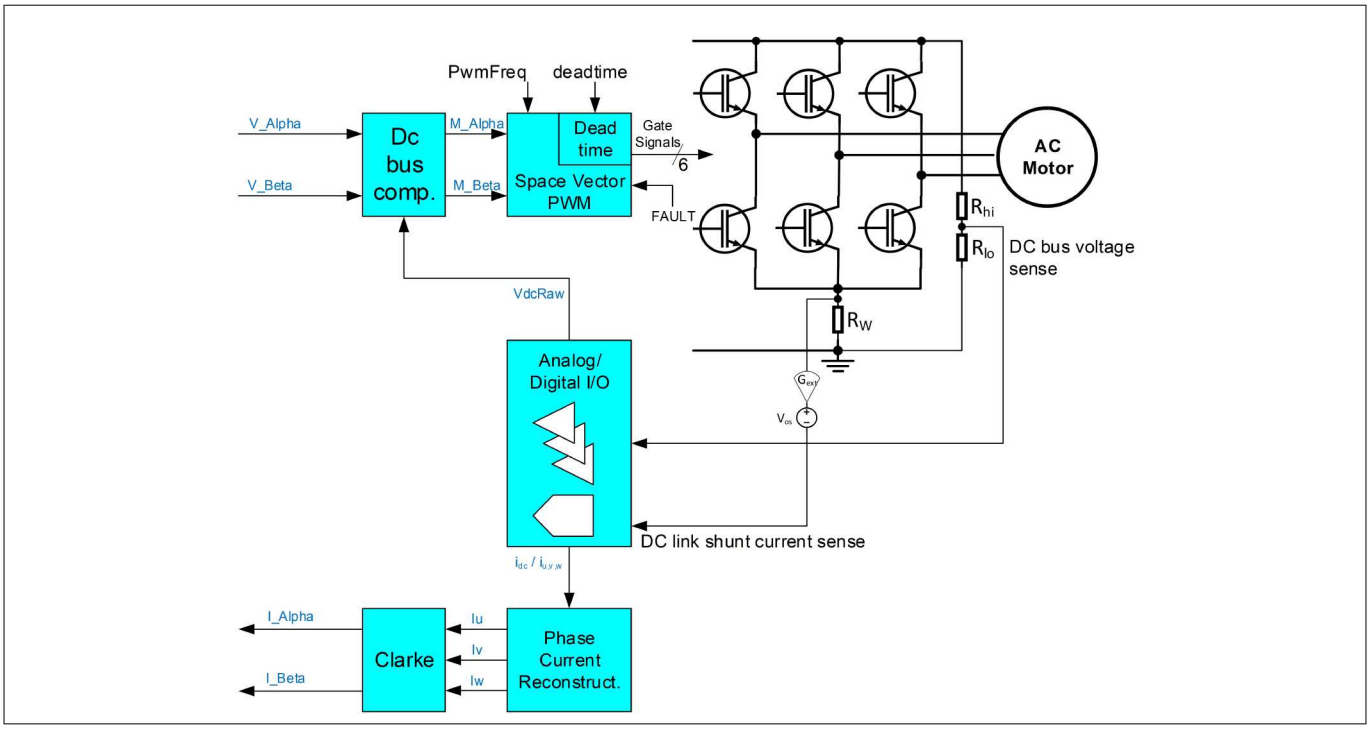


Figure 10 **Typical Circuit Diagram for Single Shunt Current Measurement Configuration**

As mentioned, one phase current can be extracted from the DC-link current during each active vector. With symmetrical PWM, the PWM signals are mirrored around the center point of the cycle meaning both active vectors are applied two times during a PWM cycle. This leaves an opportunity to measure the phase currents in the both the first- and in the second half of the PWM cycle. To minimize control delay, the MCE measures the currents in the second half of the PWM cycle.

The principle behind single shunt current sensing is shown in Figure 11. Case A shows the conventional 3-phase scheme and Case B shows the 2-phase PWM scheme. The first current sensing point (CS1) is for sensing phase current during the first active vector (in the case shown in Figure 11, the sensed current is negative phase W during the active vector [110] time). CS1 occurs $\frac{T_b}{2} + T_{SD}$ after the start of this active vector, where T_b is the active vector on-time and T_{sd} is the sample delay time. The second current sensing point (CS2) is for sensing phase current during the second active vector (in the case shown in Figure 11, the sensed current is phase U during the active vector [100] time). CS2 occurs $\frac{T_a}{2} + T_{SD}$ after the start of this active vector, where T_a is the on-time of the active vector T_{SD} . is the needed ADC sampling delay time, and it can be either positive or negative as required. can be configured by using the parameter 'SHDelay' following the equation $T_{SD} = SHDelay \times 10.417ns$ .Figure 11. Case A shows the conventional 3-phase scheme and Case B shows the 2-phase PWM scheme. The first current sensing point (CS1) is for sensing phase current during the first active vector (in the case shown in , the sensed current is negative phase W during the active vector [110] time). CS1 occurs after the start of this active vector, where T_b is the active vector on-time and T_{sd} is the sample delay time. The second current sensing point (CS2) is for sensing phase current during the second active vector (in the case shown in , the sensed current is phase U during the active vector [100] time). CS2 occurs after the start of this active vector, where T_a is the on-time of the active vector. is the needed ADC sampling delay time, and it can be either positive or negative as required. can be configured by using the parameter 'SHDelay' following the equation .

If the desired CS1 or CS2 points are estimated to occur after the end of the PWM cycle, the actual CS1 or CS2 points are adjusted by the MCE to occur just before the end of this PWM cycle to ensure the latest current sample values are available at the beginning of the following PWM cycle when the FOC calculation begins to execute.

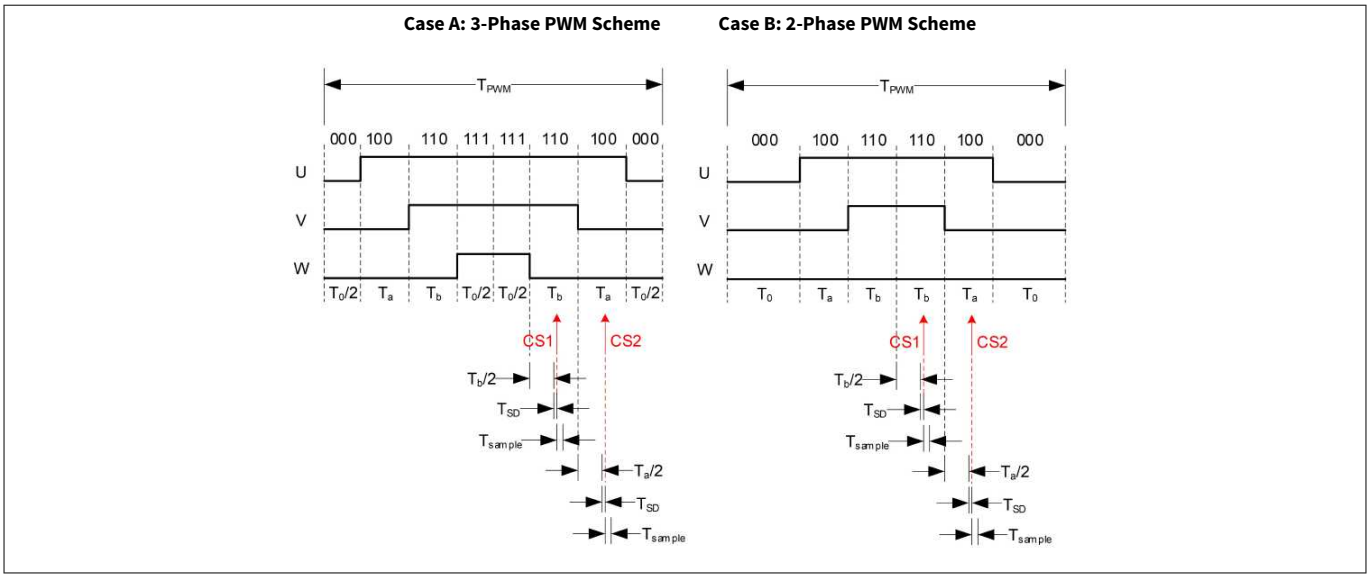


Figure 11 **Principle behind Single Shunt Current Sensing**

2.6.2.1 Limitations of Single Shunt Current Reconstruction

With single shunt reconstruction, the current through one of the phases can be sensed across the shunt resistor during the active vectors. However, when the desired voltage vector is at sector cross-over regions or when the magnitude of the desired voltage vector is low (low modulation index), the duration of one or both active vectors is too short to guarantee reliable sampling of phase current. These operating conditions are shaded in the space vector diagram shown in Figure 12. In the example shown in Figure 12, the active vector [110] time T_b is not long enough to ensure reliable current sensing.

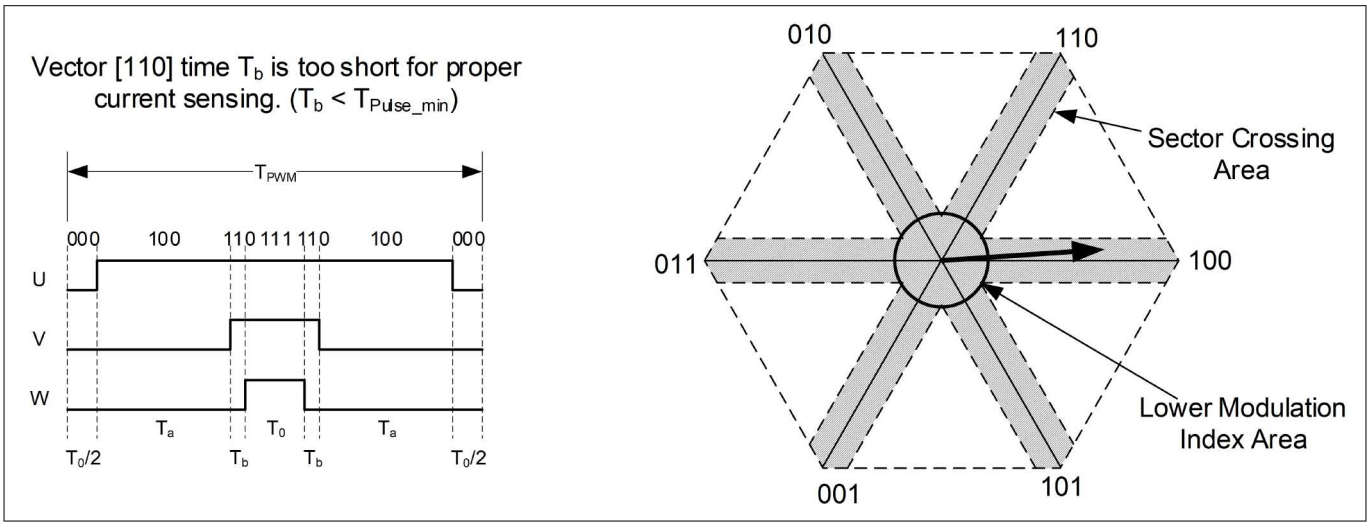


Figure 12 **Narrow pulse limitation of single shunt current sensing**

In order to guarantee reliable sampling of the current, the space vector reference voltage must lie outside the shaded area of the vector diagram. This poses a problem as the motor voltage space vector must rotate smoothly at any modulation index and any deviation between the ideal target output voltage and the actual output voltage leads to voltage distortion. This voltage distortion may cause audible noise and degradation of control performance, especially at lower speed. The shaded regions in the space vector diagram shown in Figure 12 mark the areas where output voltage distortion is introduced.

To enable single shunt current reconstruction a number of techniques exist. They all modify the modulation scheme so that the mimimum active vector required to measure the current is preserved. How this mimimum

measurement time is guaranteed and how much distortion is introduced is highly dependent on the technique used. The MCE offers two advanced schemes which enable optimal control performance and reduced the voltage distortion. The techniques are:

- Phase Shift PWM
- Low Noise Phase Shift PWM

2.6.2.2 Phase Shift PWM

With phase shift PWM scheme, the output of each PWM cycle is not always center aligned. A minimum active vector time (T_{PSmin}) is desired to ensure proper sampling of phase current. T_{PSmin} can be specified by using the parameter 'TminPhaseShift' following this equation $T_{PS_{\min}} = T\min PhaseShift \times 10.417ns$. If the desired active vector time (T_a or T_b) is longer than T_{PSmin}, then the PWM patterns remain intact. If the desired active vector time (T_a or T_b) is less than T_{PSmin}, then the 3 phase PWM patterns are shifted accordingly to ensure that the actual active vector time at the falling edge is no less than the specified minimum active vector time T_{PSmin}. As shown in Figure 13, the active vector [110] time at the falling edge is T_{b2}, and the active vector [100] time at the falling edge is T_{a2}. Given that the desired minimum active vector time $T_{PSmin} = 3 \times T_{a2}$, then T_{a2} is not long enough while T_{b2} is sufficient. Consequently, U phase PWM needs to be shifted right and V phase PWM needs to be shifted left to add enough time for active vector [100] ($T_{a2}' = T_{PSmin}$). It can be observed in Figure 13 case 1 that the PWM phase shift action equivalently adds an additional active vector [010] highlighted in red in Figure 13 that didn't exist originally. However, the impact of this additional vector is mitigated thanks to the extension of vector [100] time and the shrinking of vector [110] time.

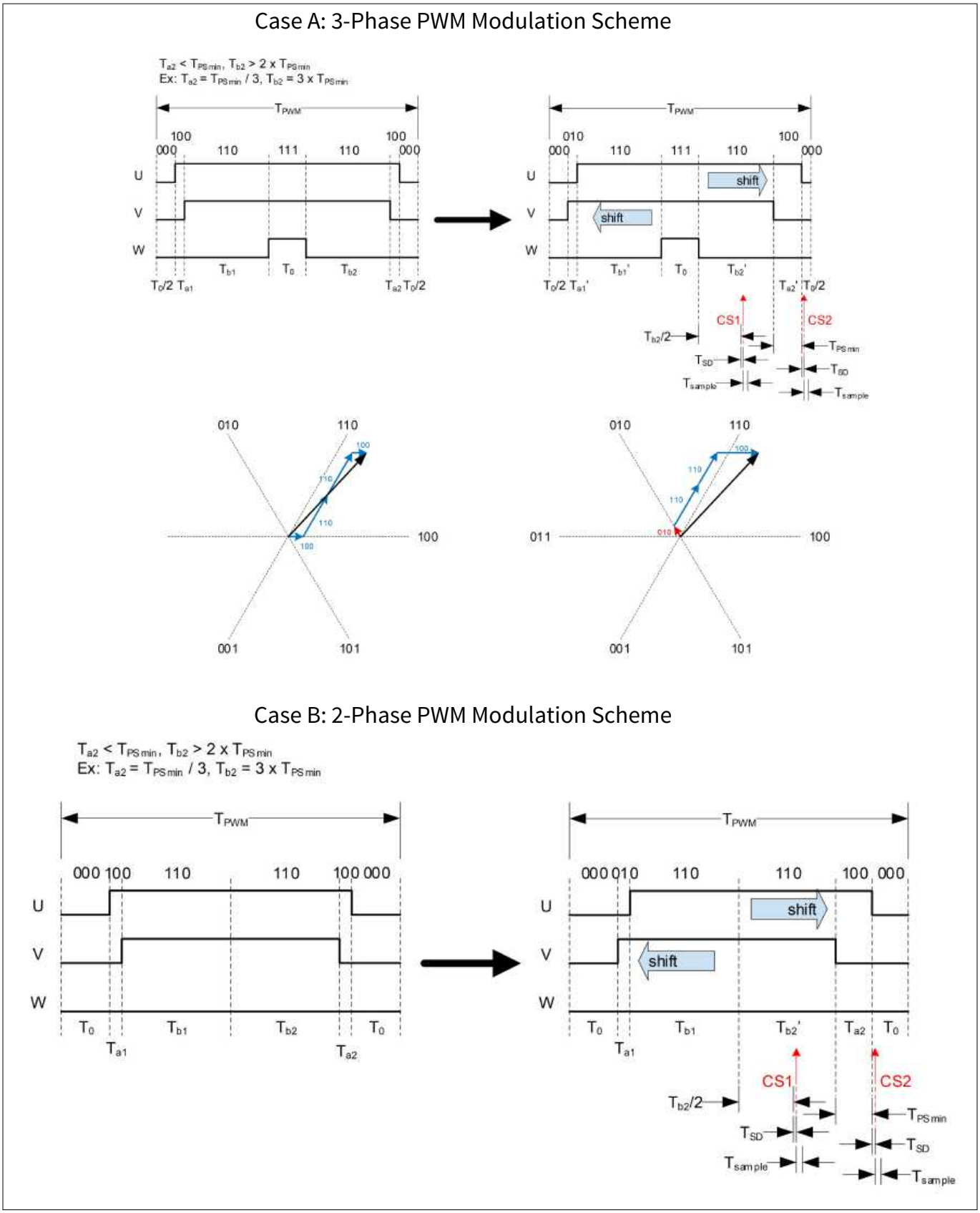


Figure 13 **Single Shunt Configuration with Phase Shift PWM Scheme Current Sensing Timing Diagram (Case 1: $T_{a2} < T_{PSmin}$, $T_{b2} > 2 \times T_{PSmin}$)**

As shown in Figure 14, given that the desired minimum active vector time T_{PSmin} = 3 x T_{b2}, then T_{b2} is not long enough while T_{a2} is sufficient. Consequently, phase W PWM needs to be shifted left to add enough time for active vector [110] (T_{b2}' = T_{PSmin}). It can be observed in Figure 14 the PWM phase shift action equivalently adds an additional active vector [101] highlighted in red in Figure 14 that did not exist originally. However, the impact of this additional vector is mitigated thanks to the extension of vector [110] time and the shrinking of vector [100] time.

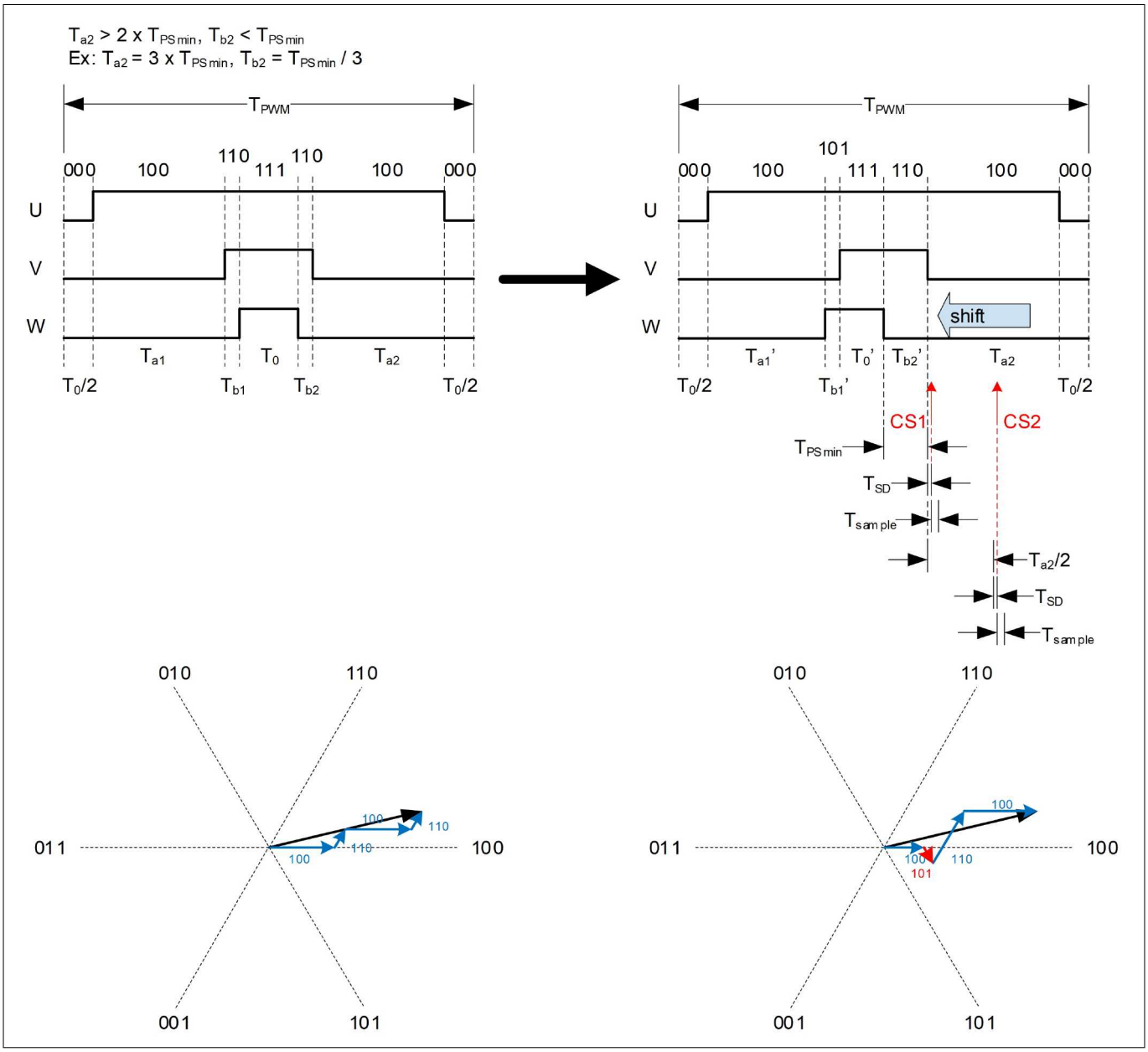


Figure 14 **Single Shunt Configuration with Phase Shift PWM Scheme Current Sensing Timing Diagram (Case 2: T_{a2} > 2 x T_{PSmin}, T_{b2} < T_{PSmin})**

The current sensing timing for single shunt configuration with phase shift PWM scheme depends on the relationship between the active vector time (T_a or T_b) and the desired minimum active vector time T_{PSmin}.

If T_a or T_b is more than 2 times T_{PSmin}, then the corresponding current sensing point occurs at the middle of that active vector time with a sampling delay time T_{SD} . Examples are T_{b2} in Figure 13, and T_{a2} in Figure 14. This is consistent with the current sensing timing described in Figure 11.

If T_a or T_b is within the range from T_{PSmin} to 2 times T_{PSmin}, then the corresponding current sensing point occurs $T_{TS_{\min}} + T_{SD}$ after the start of this active vector time. In the example shown in the following Figure 15, both T_a

and T_b fall between T_{PSmin} and 2 x T_{PSmin}. So, the CS1 occurs $T_{PS_{\min}} + T_{SD}$ after the start of active vector [110] time, and the CS2 occurs $T_{PS_{\min}} + T_{SD}$ after the start of active vector [100] time.

If T_a or T_b is less than T_{PSmin}, then necessary phase shift is applied to ensure desired minimum active vector time T_{PSmin}. Accordingly, the corresponding current sensing point T_{SD} occurs after the end of T_{PSmin}. In Figure 13, T_{a2} is less than T_{PSmin}. So, phase shift is applied to ensure the adjusted T_{a2}' = T_{PSmin}. The corresponding CS2 occurs T_{SD} after the end of T_{a2}'. In Figure 14, T_{b2} is less than T_{PSmin}. So, phase shift is applied to ensure the adjusted T_{b2}' = T_{PSmin}. The corresponding CS1 occurs T_{SD} after the end of T_{b2}'.

If the desired CS1 or CS2 point is estimated to occur after the end of the PWM cycle, then the actual CS1 or CS2 point is adjusted to occur just before the end of this PWM cycle to ensure the latest sampled current values are available at the beginning of the following PWM cycle when the FOC calculation is executed.

By using phase shift scheme, the actual output during each PWM cycle will be exactly the same as target output. Control performance at lower speed can be improved compared to using minimum pulse width PWM scheme. To achieve optimal control performance in this mode, 'TminPhaseShift' and 'SHDelay' parameters need to be tuned appropriately.

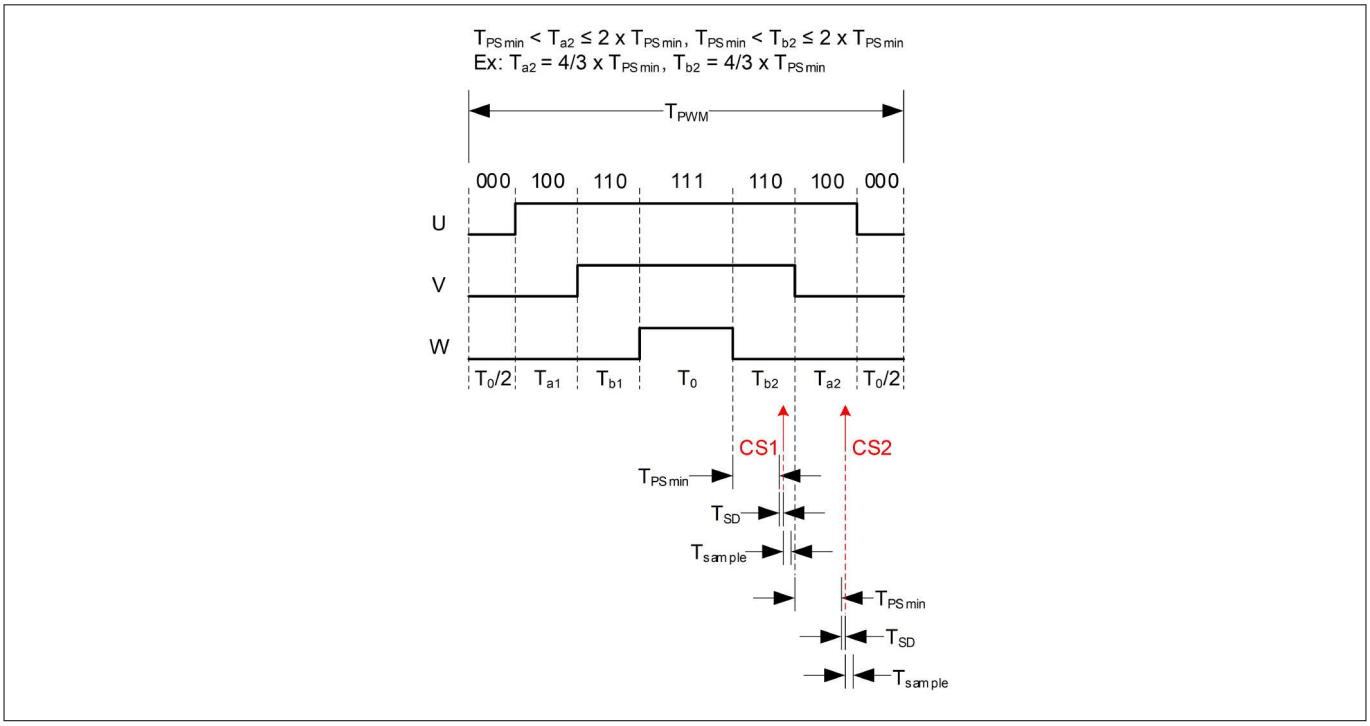


Figure 15 **Single Shunt Configuration with Phase Shift PWM Scheme Current Sensing Timing Diagram (Case 3: $T_{PSmin} \le T_{a2} \le 2 \times T_{PSmin}$, $T_{PSmin} \le T_{b2} \le 2 \times T_{PSmin}$)**

2.6.2.3 Low Noise Phase Shift PWM

One of the drawbacks of the above-mentioned phase shift scheme is that the shifting patterns are different in different sectors, and the change in shifting patterns during the sector-crossing time would still cause some acoustic noise, especially when the motor is running at lower speed.

The MCE provides an alternative option of low noise phase shift PWM scheme in order to further reduce the acoustic noise when the motor is running at lower speed. Compared to normal phase shift PWM scheme, the low noise phase shift PWM scheme adopts a fixed shifting pattern in all 6 PWM sectors, so that the acoustic noise caused by shifting pattern change is eliminated.

As shown in Figure 16, a fixed shifting pattern in the order of W→V→U is chosen with which the available vectors for single-shunt current sensing are vector [110] and [100]. With these 2 active vectors, motor current on W

phase and U phase can be sensed consecutively. The duration of these 2 vectors (T_{PSmin}) can be configured by using the parameter 'TMinPhaseShift' following this equation $T_{PS_{\min}} = T\min PhaseShift \times 10.417ns$.

Figure 16 shows 5 typical output voltage vector examples (A, B, C, D, E) that fall within the sector-crossing area (grey area) using low noise phase shift PWM scheme.

In example A, vector [110] and [100] are already available but vector [100] is too short for sensing phase U current properly. With low noise phase shift PWM scheme, V phase PWM and W phase PWM are shifted asymmetrically to extend the period of vector [100] to form an appropriate window for sensing phase U current.

In example B, vector [110] and [100] are already available but vector [110] is too short for sensing phase W current properly. With low noise phase shift PWM scheme, V phase PWM and W phase PWM are shifted asymmetrically to extend the period of vector [110] to form an appropriate window for sensing phase W current.

In example C, vector [100] is already available, but vector [110] is not available. With low noise phase shift PWM scheme, an additional vector [110] is added to form an appropriate window for sensing phase W current by shifting V phase PWM and W phase PWM asymmetrically. The impact of introducing the additional vector [110] is mitigated thanks to the extension of vector [101] and shrinking of vector [100].

In example D, vector [100] is already available, but vector [110] is not available. With low noise phase shift PWM scheme, an additional vector [110] is added to form an appropriate window for sensing phase W current by shifting V phase PWM and W phase PWM asymmetrically. The impact of adding vector [110] is mitigated thanks to the addition of vector [001].

In example E, vector [100] is already available, but vector [110] is not available. With low noise phase shift PWM scheme, an additional vector [110] is added to form an appropriate window for sensing phase W current by shifting V phase PWM and W phase PWM asymmetrically. The impact of adding vector [110] is mitigated thanks to the addition of vector [001].

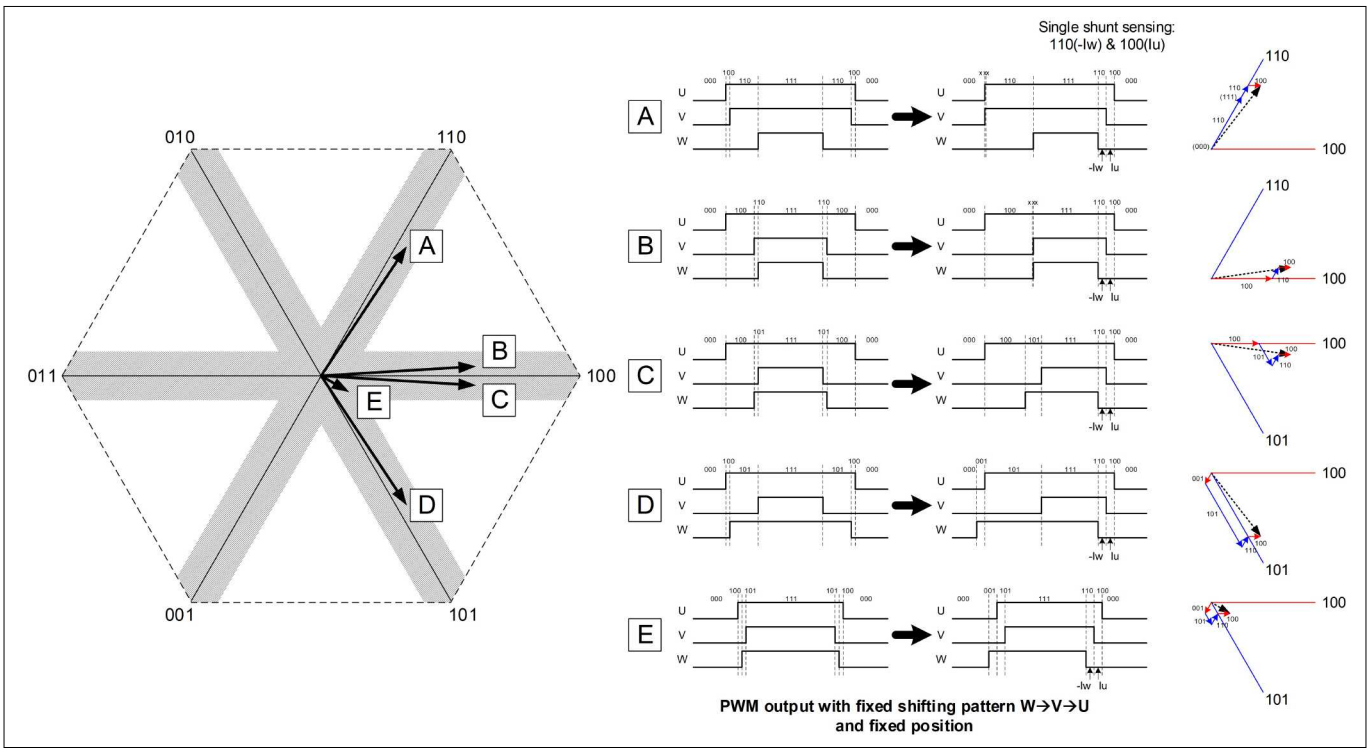


Figure 16 Low Noise Phase Shift PWM Scheme

The current sensing timing for single shunt configuration with low noise phase shift PWM is shown in the following Figure 17. With low noise phase shift PWM scheme, no matter if the active vector time T_{a2} or T_{b2} is sufficient or not compared to the desired minimum active vector time (T_{PSmin}), the phase PWM waveforms are always shifted to include the active vector [110] and [100] time with the duration of T_{PSmin} (T_{a2}' = T_{PSmin}, T_{b2}' =

T_{PSmin}) to satisfy the current sensing requirement. Consequently, the first current sensing point (CS1) occurs after the end of the active vector [110] time T_{b2}'. The second current sensing point (CS2) occurs after the end of the active vector [100] time T_{a2}'.

If the desired CS1 or CS2 point is estimated to occur after the end of the PWM cycle, then the actual CS1 or CS2 point is adjusted to occur just before the end of this PWM cycle to ensure the latest sampled current values are available at the beginning of the following PWM cycle when the FOC calculation is executed.

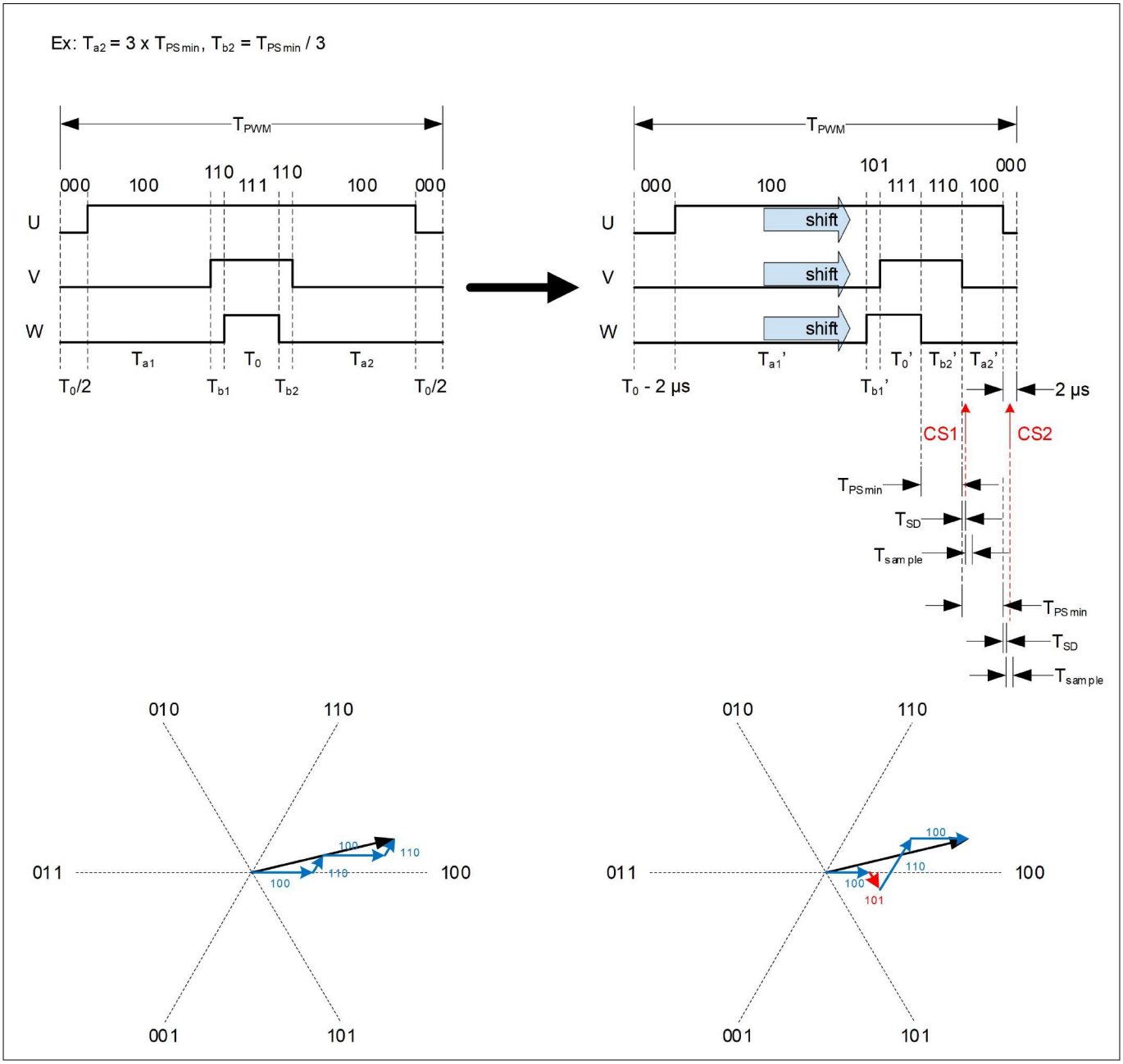


Figure 17 **Single Shunt Configuration with Low Noise Phase Shift PWM Scheme Current Sensing Timing Diagram**

Since the shifting pattern is fixed, low noise phase shift PWM is only applicable to 3-phase PWM modulation type, and the maximum PWM modulation index is limited. When low noise phase shift PWM scheme is enabled, the MCE automatically shifts to normal phase shift PWM scheme if the modulation index increases to more than 50%. If the modulation index is decreased below 35%, the MCE automatically shifts back to low noise phase shift PWM scheme.

With low noise phase shift PWM scheme, the actual output voltage during each PWM cycle is still exactly the same as the target output voltage. As a result, acoustic noise level at low speed and start-up performance is further improved compared to using normal phase shift PWM scheme. To achieve optimal control performance in this mode, 'TminPhaseShift' and 'SHDelay' parameters need to be tuned appropriately.

2.6.2.4 Peak Current Tracking with No Phase Shift Window

Certain AC fan control applications are extremely sensitive to acoustic noise especially in the low speed operating range. In this case, a modulation control mode without a minimum pulse sampling window minimizes sinusoidal voltage modulation distortion and the associated acoustic noise. In the single shunt configuration, the lack of a minimum sampling window restricts inverter current sampling to PWM cycles with active vectors greater than the required minimum pulse width. This discontinuous current sampling does not support AC winding current reconstruction and limits the control to open loop modulation/voltage control. This does not significantly impact drive performance at low speeds but there is a need to limit motor currents in overload conditions. It is still possible to provide overload protection based on the available current samples but a sample rate lower than the PWM frequency. The MCE provides an alternative peak current tracking method to realize peak current limiting function when the phase shift window is fully closed.

When TminPhaseShift = 0 with single shunt configuration, the MCE automatically switches to peak current tracking mode in which it takes 2 consecutive current samplings during each PWM cycle, and the bigger value of the 2 current sample values is assigned to variable 'Ipeak'. Right after each sector change, the 'Ipeak' variable is reset to zero to prepare for the peak current tracking in the new sector. The 'Ipeak' value is then directly assigned to variable 'Iq' per PWM cycle so that the q axis regulator can limit the current. Meanwhile, the 'Id' variable is always set to zero in peak current tracking mode.

The motor phase current feedback signal path with TminPhaseShift = 0 is shown in the following Figure 18. The scaling factor for 'Ipeak' is designed in such a way that 'Ipeak' value is represented in the same way as how the 'Iq' value is represented. Using this peak current tracking method, one can still use the Iq current control loop to monitor and limit the peak current when TminPhaseShift = 0 with single shunt configuration.

When TminPhaseShift ≠ 0, 'Ipeak' variable is reset to zero, and the MCE goes back to normal phase current reconstruction mode with single shunt configuration.

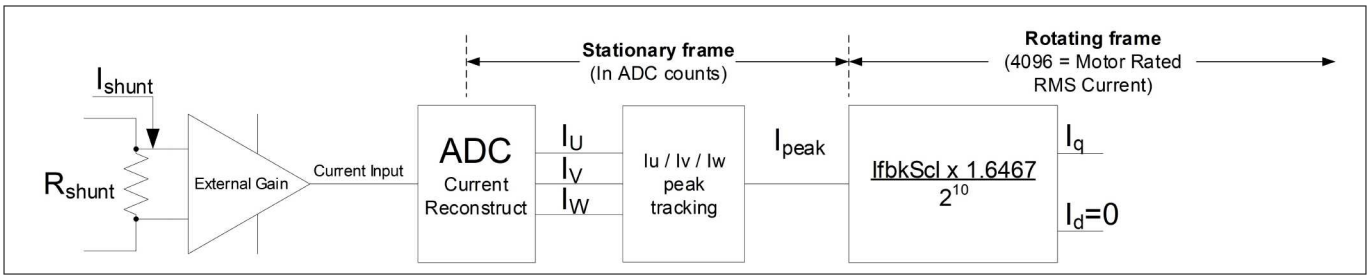


Figure 18 Motor current feedback signal path (TminPhaseShift = 0, single shunt)

In peak current tracking mode, the motor current sensing timing is adjusted as needed.

For normal phase shift PWM scheme in peak current tracking mode as shown in the following Figure 19, if both active vector time (T_a and T_b) are longer than 1 µs (Case 1), then the first current sensing point (CS1) occurs $T_{DT} + T_{SD}$ after half of the active vector time T_b. The second current sensing point (CS2) occurs $T_{DT} + T_{SD}$ after half of the active vector time T_a.

If the active vector time T_a is shorter than 1 µs (Case 2, Case 4), then CS2 point is relocated to 0.5 µs before the end of the current PWM cycle to avoid getting invalid current sensing value.

If the active vector time T_b is shorter than 1 µs (Case 3, Case 4), then CS1 point is relocated to 0.5 µs after the start of the current PWM cycle to avoid getting invalid current sensing value

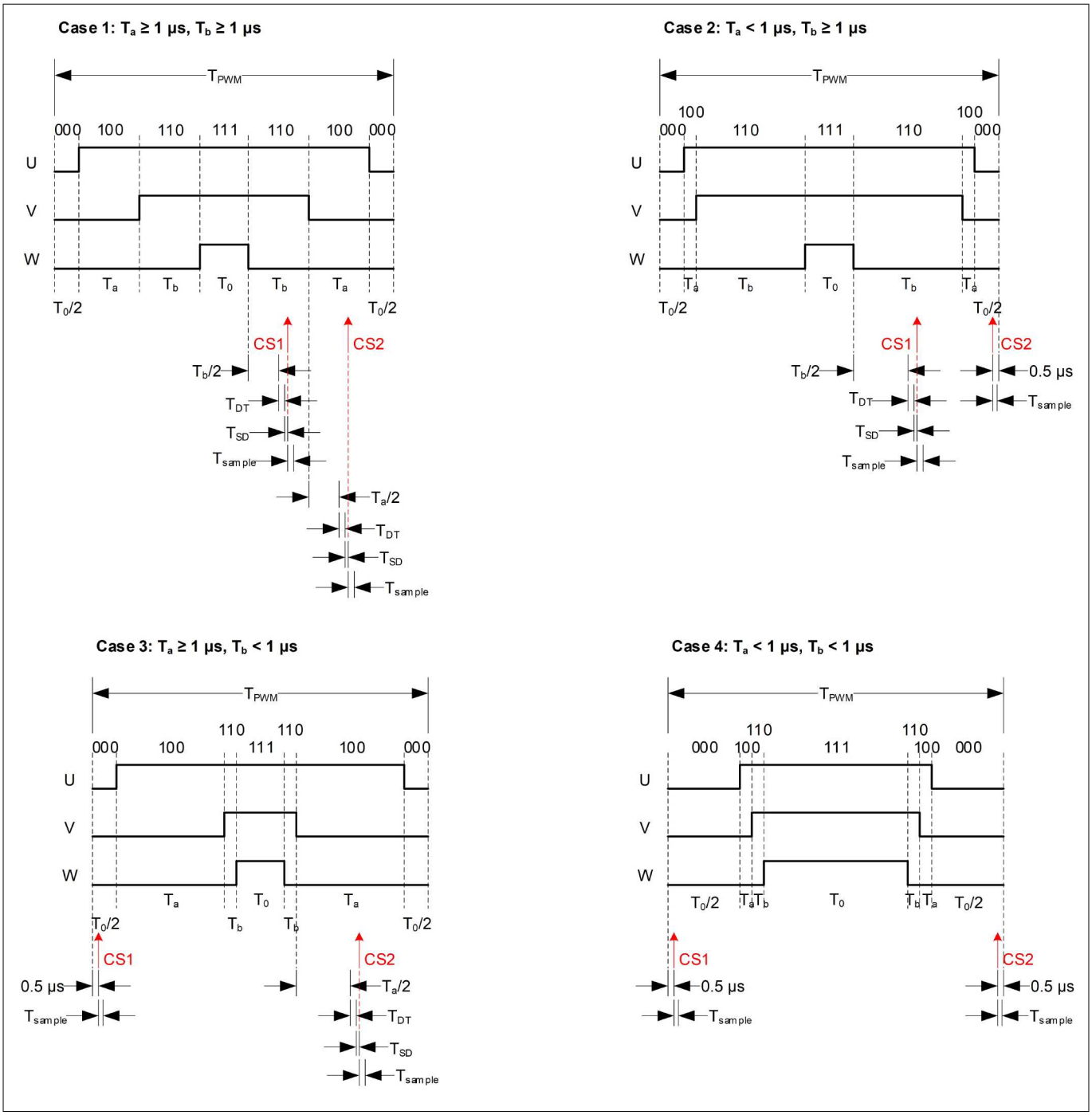


Figure 19 **Single Shunt Configuration with Phase Shift PWM Scheme in Peak Current Tracking Mode Current Sensing Timing Diagram**

For low noise phase shift PWM scheme in peak current tracking mode as shown in the following Figure 20, assuming the total active vector time is 2 x T_a and 2 x T_b respectively, if both T_a and T_b are longer than 1 µs (Case 1), then the first current sensing point (CS1) occurs $T_a + T_{DT} + T_{SD} + 1\mu s$ after the start of the active vector time 2 x T_a. The second current sensing point (CS2) occurs $T_a + T_{DT} + T_{SD} + 1\mu s$ after the start of the active vector time 2 x T_b.

If T_a is shorter than 1 µs (Case 2, Case 4), then CS1 point is relocated to 1 µs before the start of the active vector time 2 x T_a to avoid getting invalid current sensing value. If the desired CS1 point is estimated to occur before the start of the PWM cycle, then the actual CS1 point is adjusted to occur just after the start of this PWM cycle to avoid getting invalid current sensing value.

If T_b is shorter than 1 µs (Case 3, Case 4), then CS2 point is relocated to 4 µs after the start of the zero vector [111] time T_0 to avoid getting invalid current sensing value.

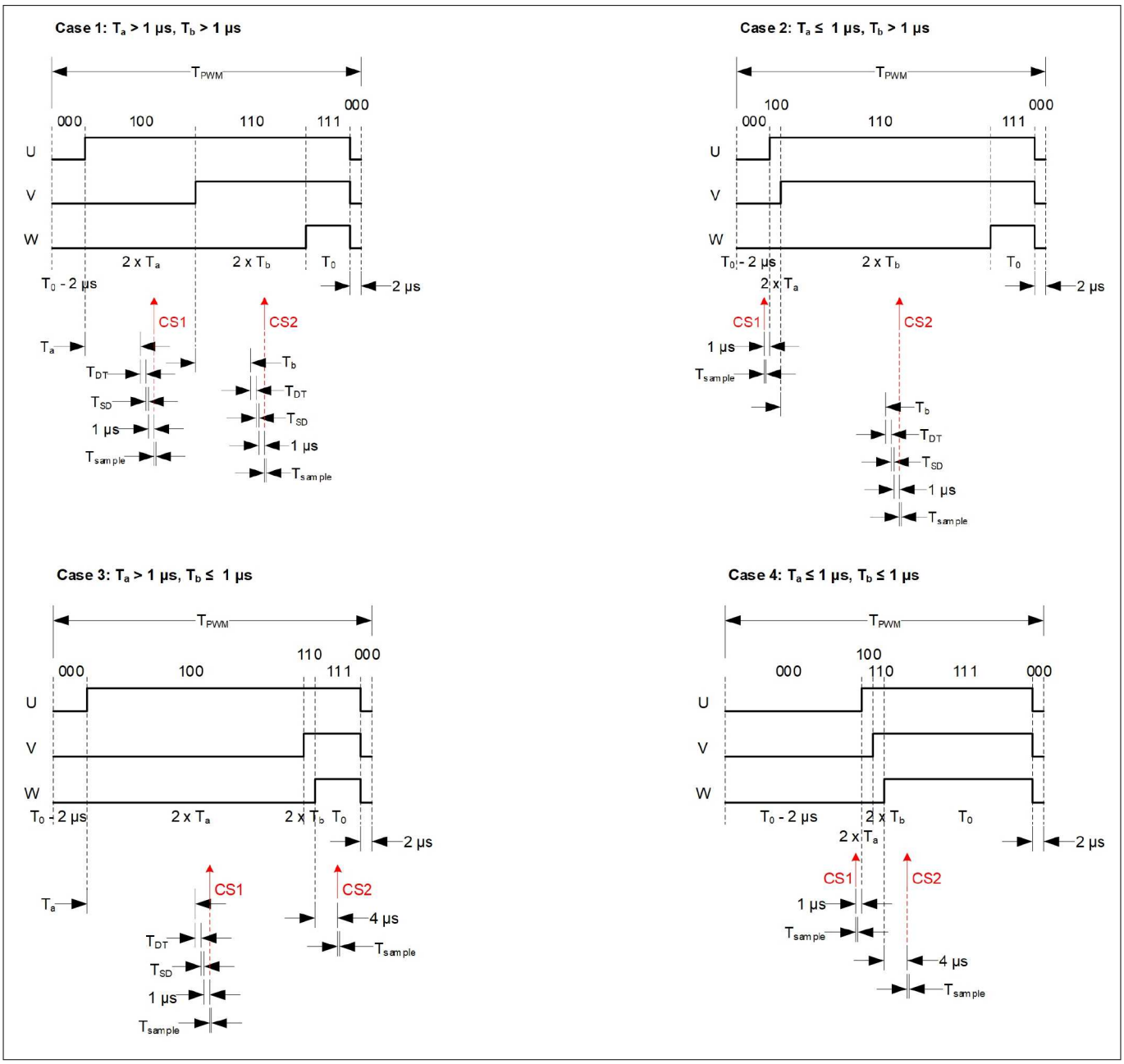


Figure 20 **Single Shunt Configuration with Low Noise Phase Shift PWM Scheme in Peak Current Tracking Mode Current Sensing Timing Diagram**

2.7 Initial Angle Sensing

Some fan applications requires starting up motors in the right direction reliably without reverse motion. Using the traditional parking + open-loop method would cause undesired reverse motion in some cases. Using direct start method sometimes might fail due to insufficient Back-EMF at low motor speed range.

MCE offers a patented initial angle sensing function that estimates the rotor angle by injecting six current pulses at different angles for a duration of a few milliseconds before starting. The initial angle is then calculated based on the current amplitude of those sensing pulses. After ANGLE_SENSING state is completed, the motor state machine would shift to MOTOR_RUN state to run the closed loop FOC control directly.

Figure 21 shows the inject current pulses sequence of phase A. Two injection signals will generate two opposite magnetic fields.

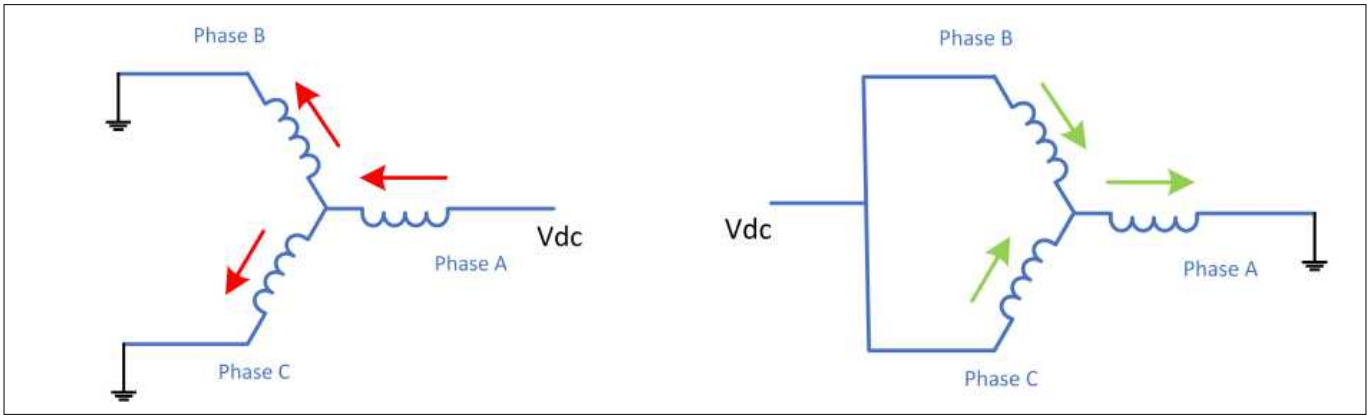


Figure 21 Inject current pulses

Every two opposite injection signals will generate two current signals. The current signal with a higher peak value represents the direction of the magnetic field closer to the direction of the injected signal. Inject the remaining 4 sets of current pulse signals in the same way. All the 6 pulse signals sequence is 0°,180°,60°,240°,120°,300°.Measured and compared all the current and the direction of the magnetic field will be limited to a range of 60 degrees finally. Figure 22 shows the current measured and the final calculated magnetic field direction.

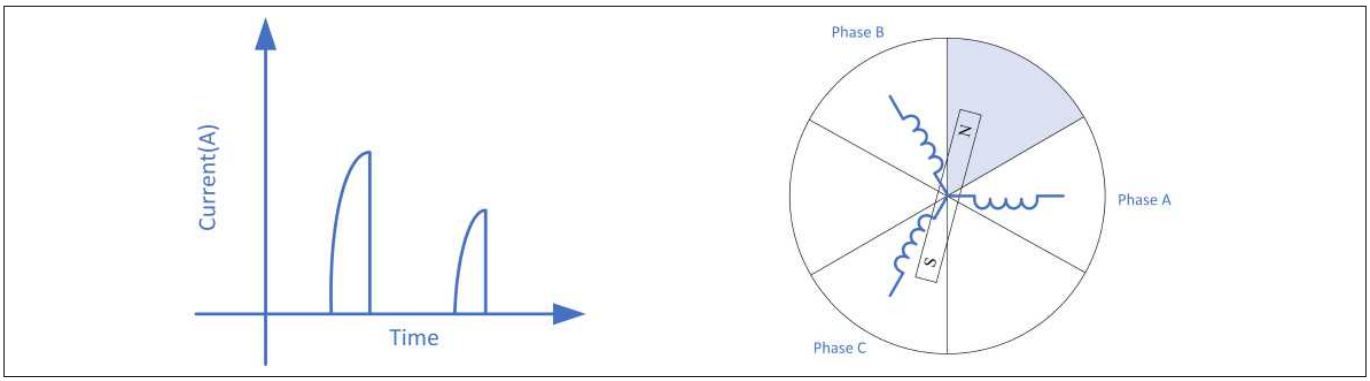


Figure 22 Current pulses and rotor position

Using the initial angle sensing function can always starts the motor in the right direction and avoids potential reverse motion during parking when used in sensorless FOC control. The initial angle estimation relies on rotor magnetic saliency and performs better when the motor L_d to L_q ratio is less than 95% and the average inductance is greater than 0.1 mH.

The relevant control parameters (IS_Pulses, IS_Duty, IS_IqInit) are automatically calculated by Solution Designer based on the L_d and L_q motor parameters entered. Due to the calculation of Ton also involves the rated current of the motor, accurate Irated parameters are also necessary.

This method only takes care of the initial angle measurement so tuning the flux estimator may be required when driving high inertia loads. If the motor speed is not zero at the start-up, then the detected angle might not be accurate. It is then recommended to use catch-spin function in that scenario.

When speed area between 0 ~5%, user still needs some tuning before flux PLL can work properly. When speed is above ~10%, this method is not accurate.

2.8 Hall Sensor Interface

The MCE Hall angle extraction algorithm estimates rotor angle and velocity signals per motor PWM cycle from the four times (2 Hall sensors) or six times (3 Hall sensors) per electrical cycle digital Hall input transition events. The optional Atan angle algorithm extracts rotor angle and velocity signals per motor PWM cycle from the two analog Hall sensor signals.

The MCE Hall sensor interface supports the following Hall sensor configurations as shown in Table 3.

Table 3 Supported Hall Sensor Configurations

Interface Type	Supported Configuration	Sensor Displacement (Electrical Angle)
Digital	2/3 digital Hall sensors	120°
Analog	2 analog Hall sensors	120°

2.8.1 Interface Structure

As shown in the following Figure 23, the analog Hall sensor positive and negative outputs are connected to non-inverting and inverting inputs of the internal comparators with configurable hysteresis (bit field [7:6] of parameter 'HallConfig') respectively. During every Hall zero-crossing event between AHALLx+ and AHALLx- (x = 1, 2), the relevant comparator output toggles accordingly. The internal comparator outputs are connected via a multiplexer to H1 and H2 inputs of the Hall Event Capture block. The analog Hall sensor outputs are also connected to the four internal ADC channels through an equivalent gain stage of 1 for the purpose of sampling analog Hall sensor output voltage values, which are used to calculate Atan angle when Hall Atan angle calculation method is enabled.

The digital Hall sensor outputs are directly connected via the multiplexer to the corresponding H1, H2, and H3 inputs of the Hall Event Capture block, whose outputs are Hall event timing information and the Hall pattern that are used by Hall PLL block to estimate Hall angle and Hall speed.

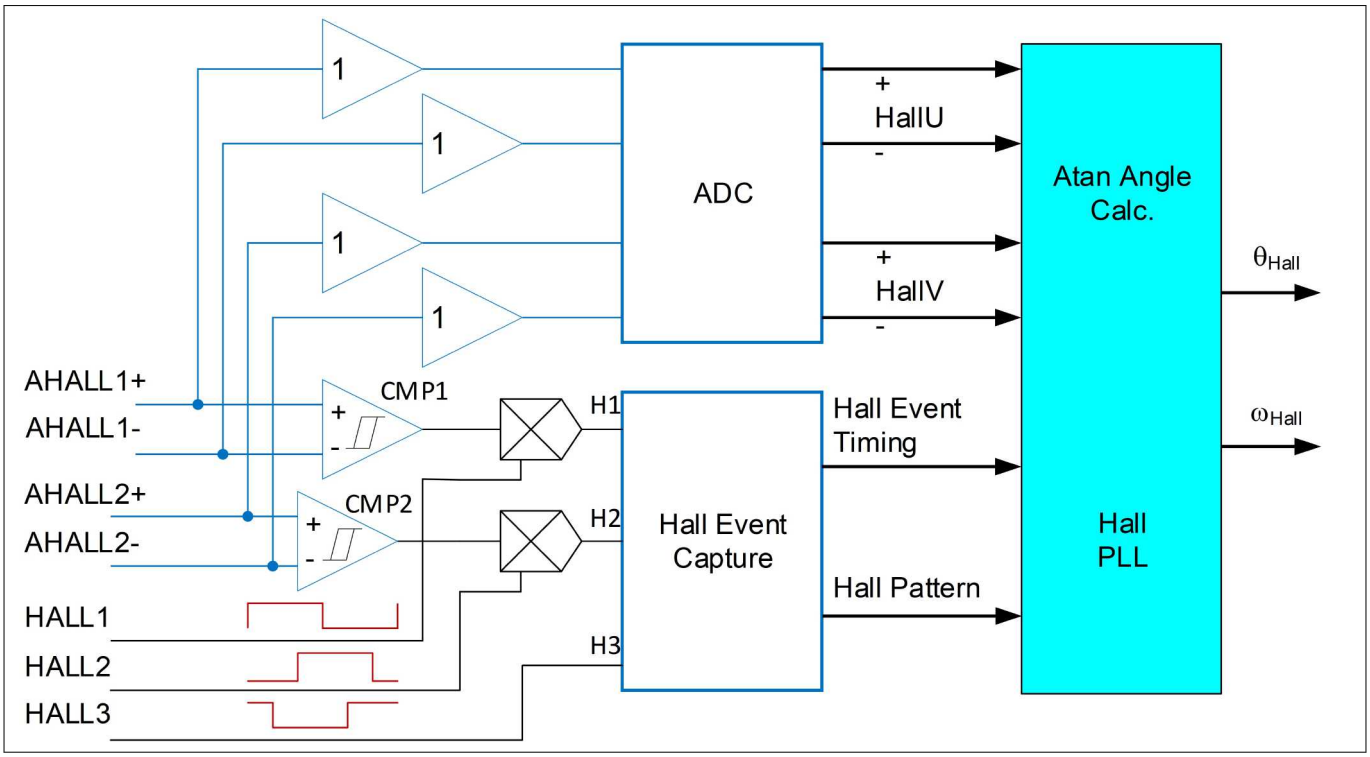


Figure 23 Hall Sensor Interface High-Level Structure Overview

2.8.2 Hall Sample De-Bounce Filter

A hardware noise filter is included in the Hall event capture block to provide de-bounce check mechanism before sampling Hall inputs. The noise filter timing mechanism is shown in the following Figure 24 . Whenever there comes a transition detected at H1, H2 or H3 input, its status is not sampled until after a configurable de-bounce time (T_{DB}) has elapsed. If there comes another transition before T_{DB} has elapsed, then the scheduled following sampling operation is cancelled and the de-bounce time counting starts over. This de-bounce time can be configured by using the parameter 'SampleFilter' following this equation
$T_{DB} = SampleFilter \times 10.417ns$.

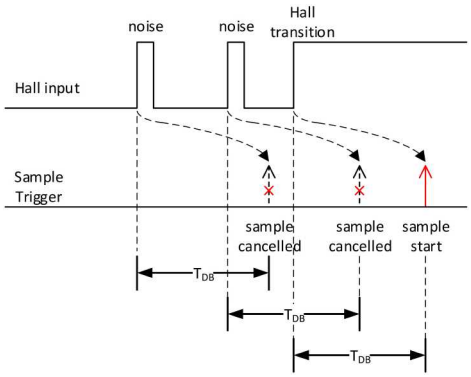


Figure 24 Hall Sensor Noise Filter Timing Diagram

2.8.3 Hall Angle Estimation

Digital Hall sensors or comparator based analog Hall sensor interface provide discrete angle inputs at each Hall transition event. For 3 digital Hall sensor configuration, a Hall transition event occurs every 60° electrical angle.

For 2 digital Hall sensor configuration, a Hall input transition event occurs every 60° electrical angle (normal sector) or 120° electrical angle (wide sector) alternately. For 2 analog Hall sensor configuration, the two internal comparators are used to detect zero-crossing events, and the corresponding Hall transition event occurs the same way as in the case of 2 digital Hall sensor configuration.

The MCE's Hall angle estimation algorithm estimates Hall angle between sequential hall transition events by integrating a Hall frequency estimate. It takes advantage of a PLL loop to keep track of the actual Hall frequency and correct angle estimation error by subtracting a compensation term to the Hall frequency integrator over the next Hall transition event cycle.

The status of the digital inputs of the Hall Event Capture block is sampled by the MCE's hardware peripheral when a Hall transition event occurs. The sampled Hall inputs form certain Hall pattern as described in Chapter 2.8.3.1.

The MCE's Hall angle estimation routine is executed during each motor PWM cycle. Details of the Hall angle estimation process is described in the following two sub-sections.

2.8.3.1 Hall Angle Estimation with PLL

When Hall PLL is enabled (KpHallPLL > 0), the Hall angle estimation algorithm is depicted in the following Figure 25.

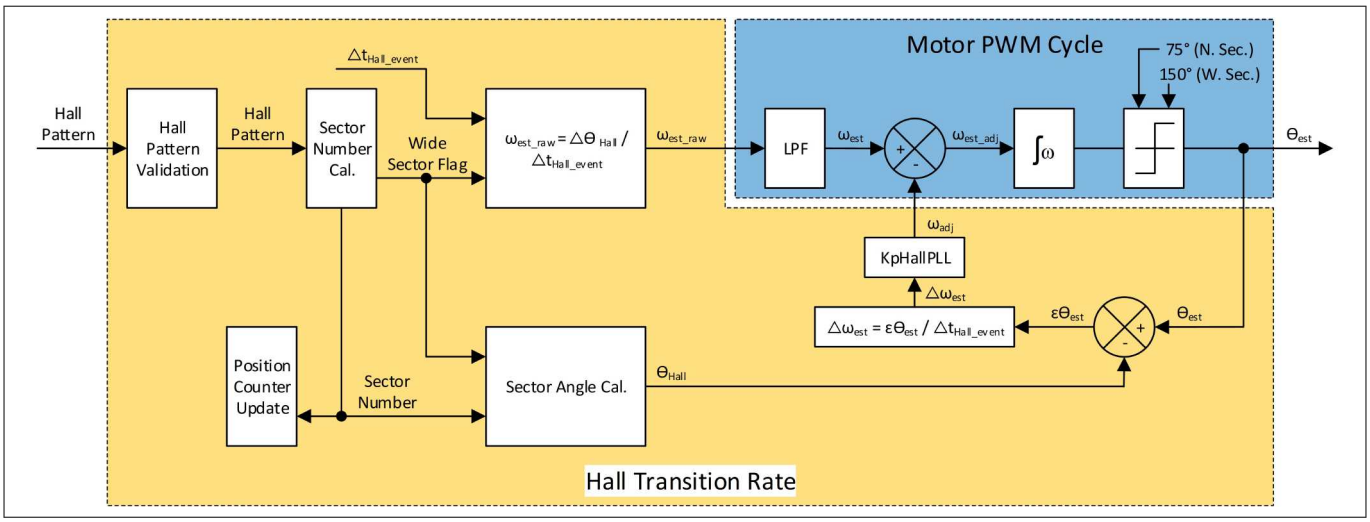


Figure 25 Hall Angle Estimation Algorithm Diagram (Hall PLL Enabled)

During each motor PWM cycle, the MCE's Hall angle estimation routine integrates the difference ω_{est_adj} between the low-pass filtered Hall frequency estimate ω_{est} (variable 'HallFreq') and a compensation term ω_{adj} to generate the estimated Hall angle θ_{est} as shown in the blue block in Figure 25. If the estimated Hall angle increment is accumulated up to 75° (normal sector) or 150° (wide sector) since last Hall transition event, no further integration is performed and θ_{est} stays flat until next Hall transition event occurs.

The MCE also checks if there occurs a new Hall transition event since last check during each motor PWM cycle. If there exists a new Hall transition event, the following steps are performed as shown in the orange block in Figure 25.

The newly sampled Hall pattern is first validated against an expected pattern based on rotating direction.

If it is validated successfully, then a corresponding new sector number is calculated based on a mapping relationship between Hall patterns and sector numbers as shown in Figure 27.

Next, raw Hall frequency estimate ω_{est_raw}, which represents the amount of angle change per PWM cycle, is calculated as the result of the division of angle difference between the two sequential Hall transition events $\Delta\theta_{Hall}$ and the time interval Δt_{Hall_event} between the two sequential Hall transition events ($\omega_{est\ raw} = \frac{\Delta\theta_{Hall}}{\Delta t_{Hall\ event}}$). If the time interval between the two sequential Hall transition events Δt_{Hall_event} is longer than 4096 PWM cycles, then it is considered timed out and ω_{est_raw} is rest to zero. The updated raw Hall frequency estimate is low-pass filtered with a configurable time constant T_{decay}. To achieve a desired bandwidth $\omega_c = \frac{1}{T_{decay}}$ for this low-pass filter, please follow this equation to calculate the value for the variable 'FrequencyBW': $FrequencyBW = 2^{16} \times \left(1 - e^{\frac{\omega_c \times Fast_Control_Rate}{F_{PWM}}}\right)$. The filtered Hall frequency estimate ω_{es} (variable 'HallFreq') is available to be used for Hall angle estimation.

Next, the actual Hall angle θ_{Hall} is calculated based on the updated sector number and the rotating direction and wide sector flag. The estimated Hall angle θ_{est} is not adjusted immediately at each Hall transition event. The Hall angle estimation error $\varepsilon\theta_{est}$ is corrected by adding a compensation term ω_{adj} to the Hall frequency integrator ove the next Hall transition event cycle. The frequency compensation term ω_{adj} is calculated as the product of a proportional factor (parameter 'KpHallPLL') and the division of the angle estimation error $\varepsilon\theta_{est}$ by the time interval between the two sequential Hall transiton events ($\omega_{adj} = KpHallPLL \times \frac{\varepsilon\theta_{est}}{\Delta t_{Hall\ event}}$). If the angle estimation error $\varepsilon\theta_{est}$ is greater than 15° (normal sector) or 30° (wide sector), then the estimated Hall angle θ_{est} is reset to the value of the actual Hall angle θ_{Hall}, and the compensation term ω_{adj} is reset to 0.

Finally, the variable 'HallAngle' is updated following this equation: $HallAngle = \theta_{est} + HallAngleOffset$. The configuration of the parameter 'HallAngleOffset' is described in Chapter 2.8.6. The parameter 'HallSpeed' is updated from the product of the low-pass filtered Hall frequency estimate ω_{est} with coresponding scaling factors.

With Hall PLL enabled, the angle estimation error $\varepsilon\theta_{est}$ is corrected over the following Hall event cycles, so that the estimated Hall angle θ_{est} would not jump abruptly at each Hall transition event. Thus, the motor is expected to run relatively more smoothly when it is accelerating or decelerating. It is recommended to take advantage of Hall PLL by selecting a value for parameter 'KpHallPLL' between 0 and 4096 for better performance when using Hall sensors.

Users are advised to select the value of the parameter 'KpHallPLL' with the consideration of the trade-offs between torque/speed dynamics and operational smoothness depending on different application requirements. For example, door opener applications may prefer higher dynamics of torque, while fan applications may favor operational smoothness over torque dynamics. Higher 'KpHallPLL' value provides quicker speed/torque response with the compromise of operational smoothness due to sudden change of estimated Hall speed and angle. Lower 'KpHallPLL' value provides smoother torque/speed change while sacrificing dynamics.

2.8.3.2 Hall Angle Estimation without PLL

When Hall PLL is disabled (KpHallPLL = 0), the Hall angle estimation algorithm is depicted in the following Figure 26.

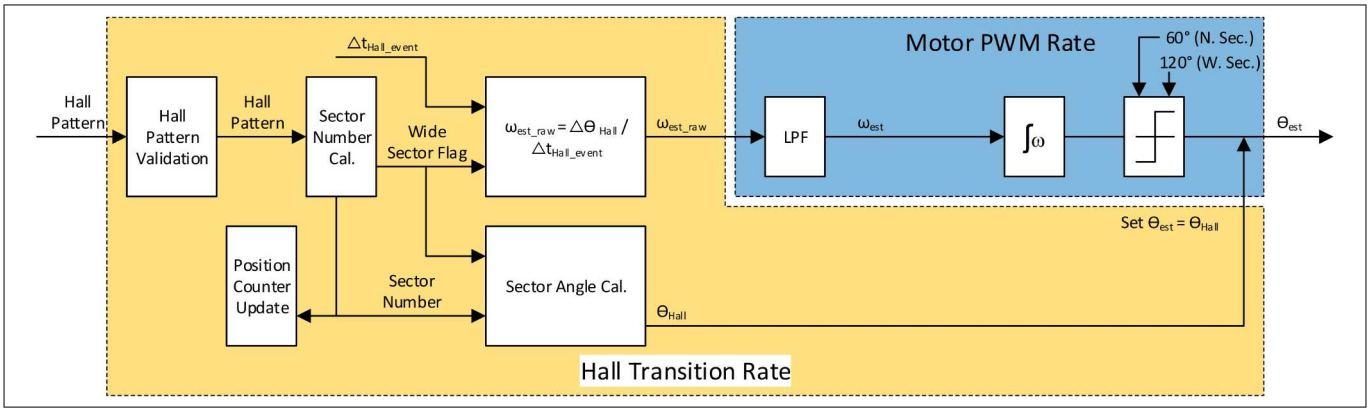


Figure 26 Hall Angle Estimation Algorithm Diagram (Hall PLL Disabled)

During each motor PWM cycle, the MCE's Hall angle estimation routine integrates the low-pass filtered Hall frequency estimate ω_{est} to generate the estimated Hall angle θ_{est} as shown in the blue block in Figure 26. If the estimated Hall angle increment is accumulated up to 60° (normal sector) or 120° (wide sector) since last Hall transition event, no further integration is performed and θ_{est} stays flat until next Hall transition event occurs.

The MCE also checks if there occurs a new Hall transition event since last check during each motor PWM cycle. If there exists a new Hall transition event, the MCE performs a similar set of steps compared to the scenario with PLL enabled as shown in the orange block in Figure 26 .

Hall pattern validation, sector number calculation, Hall frequency estimate calculation, actual Hall angle calculation, the variable 'HallAngle', and 'HallSpeed' update steps are the same as those in the scenario with PLL enabled.

The step that differs is the angle estimation error correction. The angle estimation error $\varepsilon\theta_{est}$ is corrected by adding the latest angle estimation error $\varepsilon\theta_{est}$ to the estimated Hall angle θ_{est} at each Hall transition event. In other words, the estimated Hall angle θ_{est} is reset to the actual Hall angle θ_{est} at each Hall transition event.

With Hall PLL disabled, when the motor is accelerating or decelerating, the estimated Hall angle θ_{est} would jump abruptly at each Hall transition event.

2.8.4 Hall Zero-Speed Check

When the motor control state machine is in 'MOTORRUN' state, if the time interval between the two sequential Hall transition events is longer than a threshold T_{zf}, then it is considered as a Hall zero frequency fault. The threshold T_{zf} is calculated following this equation $T_{zf} = 4096 \times T_{PWM}$. Once the time interval between the two sequential Hall transition events is shorter than the threshold T_{zf}, this fault is automatically cleared.

The equivalent motor speed that would trigger Hall zero frequency fault consistently with 2 or 3 digital Hall sensor configurations can be calculated as follows:

$$\omega_{zf_3Hall}\left(rpm\right) = \frac{1}{4096 \times T_{PWM}} \times \frac{1}{6} \times \frac{60}{pole_pair}$$

If this Hall zero frequency fault lasts as long as $T_{HallTimeOut}$, then a 'Hall Timeout' Fault is confirmed. The threshold $T_{HallTimeOut}$ can be configured using the parameter 'HallTimeoutPeriod' following this equation: $T_{HallTimeOut} = HallTimeOutPeriod \times 16ms$.

This fault is to detect rotor lock condition when Hall sensors are being used.

2.8.5 Hall Pattern Validation

Hall pattern is formed as a binary number ($[H3, H2, H1]_b$) by using the 3 digital inputs, H1, H2 and H3 of Hall Event Capture block, and assumes that H3 is bit 2, H2 is bit 1, and H1 is bit 0 as shown in the following Figure 27. For example, if H3 is logic high, H2 is logic low, and H3 is logic high, then the Hall pattern is recognized as $[101]_b = 5$.

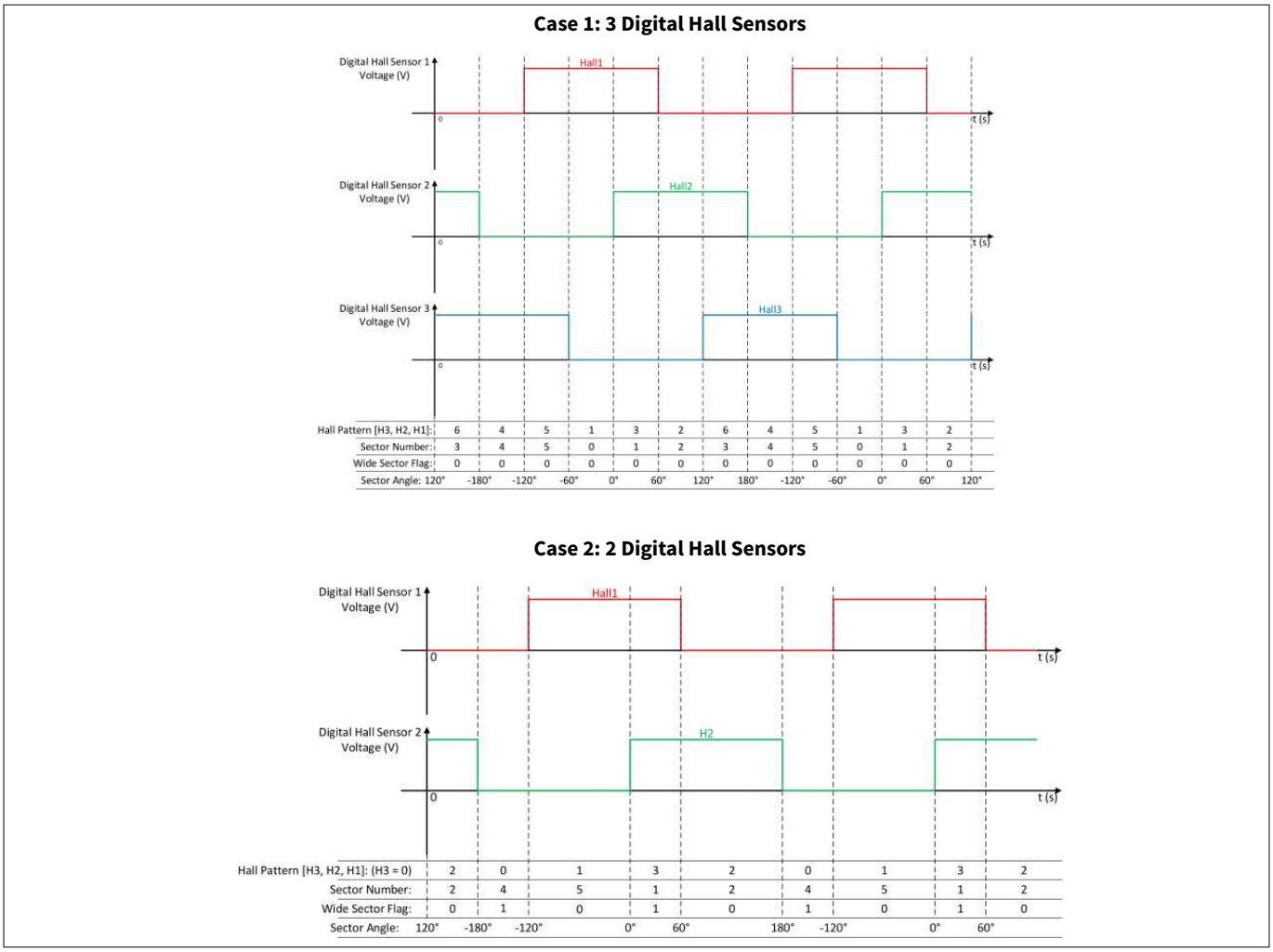


Figure 27 **Calculation of Hall Pattern, Sector Number, Wide Sector Flag, and Sector Angle from Hall Inputs**

Hall pattern validation starts by comparing the newly sampled Hall pattern with an expected Hall pattern from a pre-determined Hall pattern sequence based on motor rotating direction.

If the newly sampled Hall pattern is [111] or [000], then it is considered as an invalid pattern fault. If two consecutive occurrences of the invalid pattern fault are detected, then 'Hall Invalid' fault is confirmed and the 15th bit of variable 'FaultFlags' is set. Notice this invalid pattern check is only applicable to 3 digital Hall configuration.

If the newly sampled Hall pattern is valid but doesn't match either the expected Hall pattern from the CW rotating Hall pattern sequence or from the CCW rotating Hall pattern sequence, then it is considered as an unexpected pattern fault. If three consecutive occurrences of the unexpected pattern fault are detected, then 'Hall Invalid' fault is confirmed and the 15th bit of variable 'FaultFlags' is set.

If the newly sampled Hall pattern is validated successfully, then a new sector number (0~5) is extracted based on a mapping relationship between Hall patterns and sector numbers as shown in Figure 27.

2.8.6 Hall Angle Offset

For 2 analog Hall sensor configuration, assume that the angle difference between the zero-crossing of UV line to line back-EMF voltage waveform and the zero-crossing of analog Hall 1 differential waveform is defined as θ_{offset} as shown in the following Figure 28.

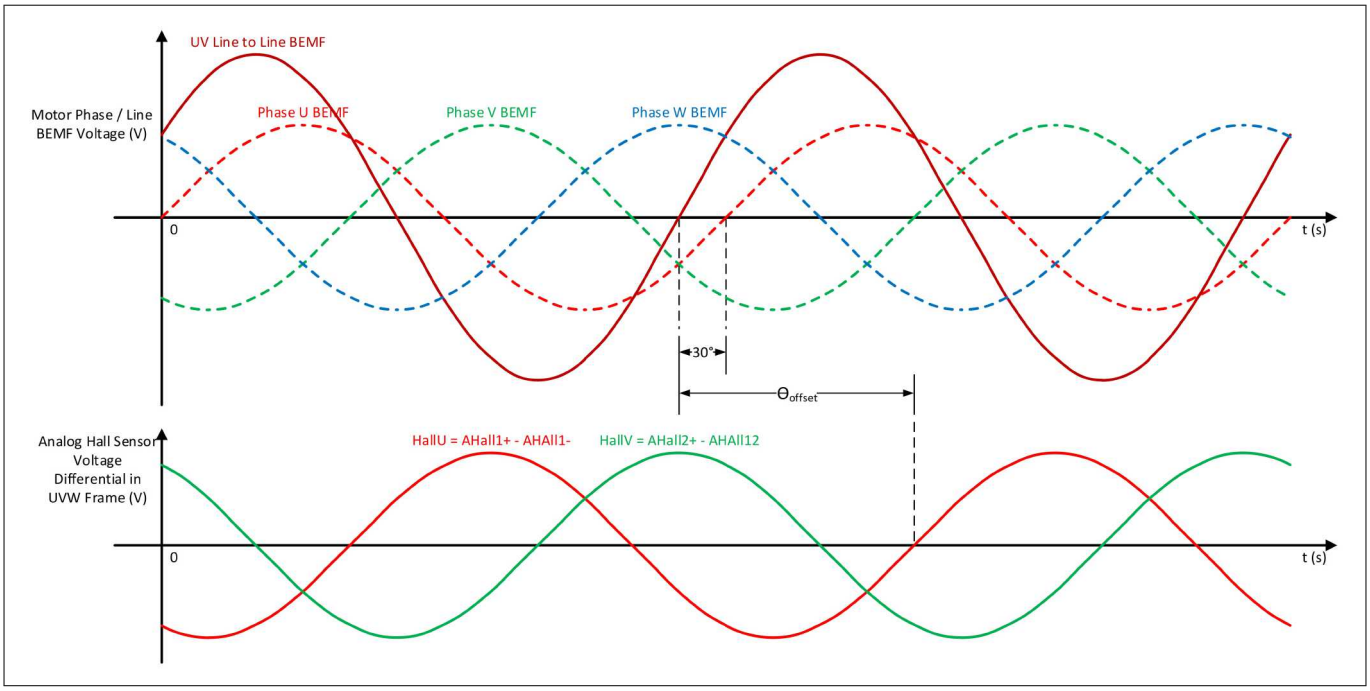


Figure 28 Angle Offset Definition Diagram for 2 Analog Hall Sensor Configuration

For 2 or 3 digital Hall sensor configuration, assumes that the angle difference between the zero-crossing of UV line to line back-EMF voltage waveform and the zero-crossing of analog Hall 1 differential waveform is defined as θ_{offset} as shown in the following Figure 29.

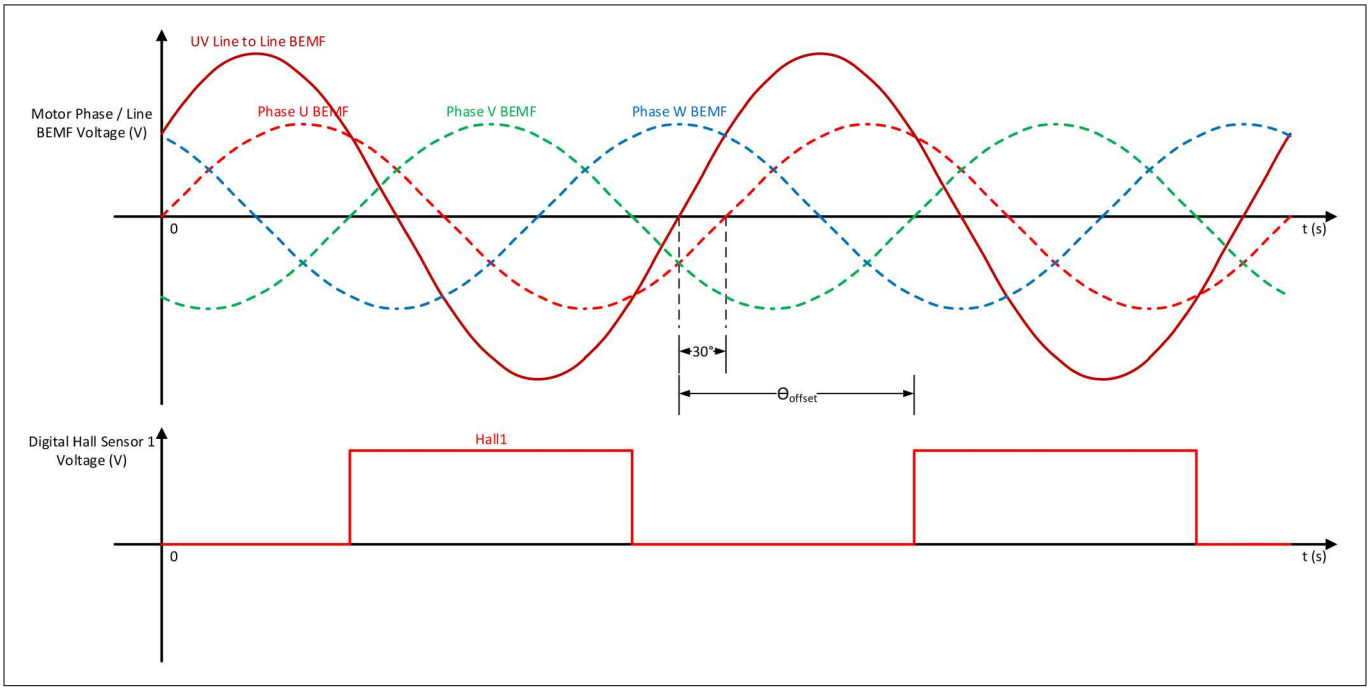


Figure 29 Angle Offset Definition Diagram for 2 or 3 Digital Hall Sensor Configuration

The parameter 'HallAngleOffset' shall be calculated following this equation:

$$HallAngleOffset = \left(\theta_{offset} - 90°\right) \times \frac{16384}{90°}$$

This parameter is used in the final calculation of variable 'HallAngle' during each motor PWM cycle to compensate for the angle difference between the rotor position and the Hall sensor (DHALL1 or AHALL1) mounting position.

2.8.7 Atan Angle Calculation

The above-mentioned Hall angle calculation method based on Hall zero-crossing events using comparators renders a variable estimated angle error correction rate. The lower the motor speed is, the longer it takes for a Hall input transition event to occur, and the lower the estimated angle error correction rate becomes. As a result, when the motor is starting up using this Hall angle calculation method, it is inevitable that the estimated Hall angle would not accumulate smoothly during the first a few sectors due to the nature of lower Hall input transition event occurrence rate. This would sometimes cause undesirable acoustic noise and unsmooth motor start-up performance.

The MCE provides an Atan angle calculation method to complement the estimated Hall angle calculation during start-up for 2 analog Hall sensor configuration. The Atan angle calculation method can be enabled or disabled by using the 5th bit of the parameter 'HallConfig'. When Atan angle calculation method is enabled and Hall angle or hybrid angle is selected by the parameter 'AngleSelect'. Parameter HallATanPeriod specifies the number of sectors for which Hall Atan angle, represented by the parameter 'Atan_Angle', is being used as rotor angle during start-up.

The Atan angle calculation process is shown in the following Figure 30. The analog Hall sensor input (AHALL1+, AHALL1-, AHALL2+, and AHALL2-) voltage levels are sampled during each motor PWM cycle, and the voltage differential of each analog Hall sensor is calculated as HallU = AHALL1+ - AHALL1-, and HallV = AHALL2+ - AHALL2-. The MCE performs Clarke transformation to convert HallU and HallV components in UVW reference frame to Hallα and Hallβ components in a stationary αβ reference frame. Then $Atan\left(\frac{Hall_\beta}{Hall_\alpha}\right)$ calculation is performed to generate Atan angle represented by the variable 'Atan_Angle' with the addition of Hall angle offset specified by the parameter 'HallAngleOffset'.

Using the complementary Atan angle calculation method, the rotor angle using Atan angle is expected to accumulate more smoothly with minimal acoustic noise during motor start-up compared to using the above-mentioned Hall angle estimation method. The analog Hall sensor signals bear higher order harmonics in some cases. As a result, the Atan angle calculation would yield undesired fluctuation that is not a true reflection of the rotor speed variation. Consequently, it is recommended to limit the usage of Hall Atan angle calculation method to a short duration during start-up for just several number of sectors as needed by configuring the parameter 'HallATanPeriod.

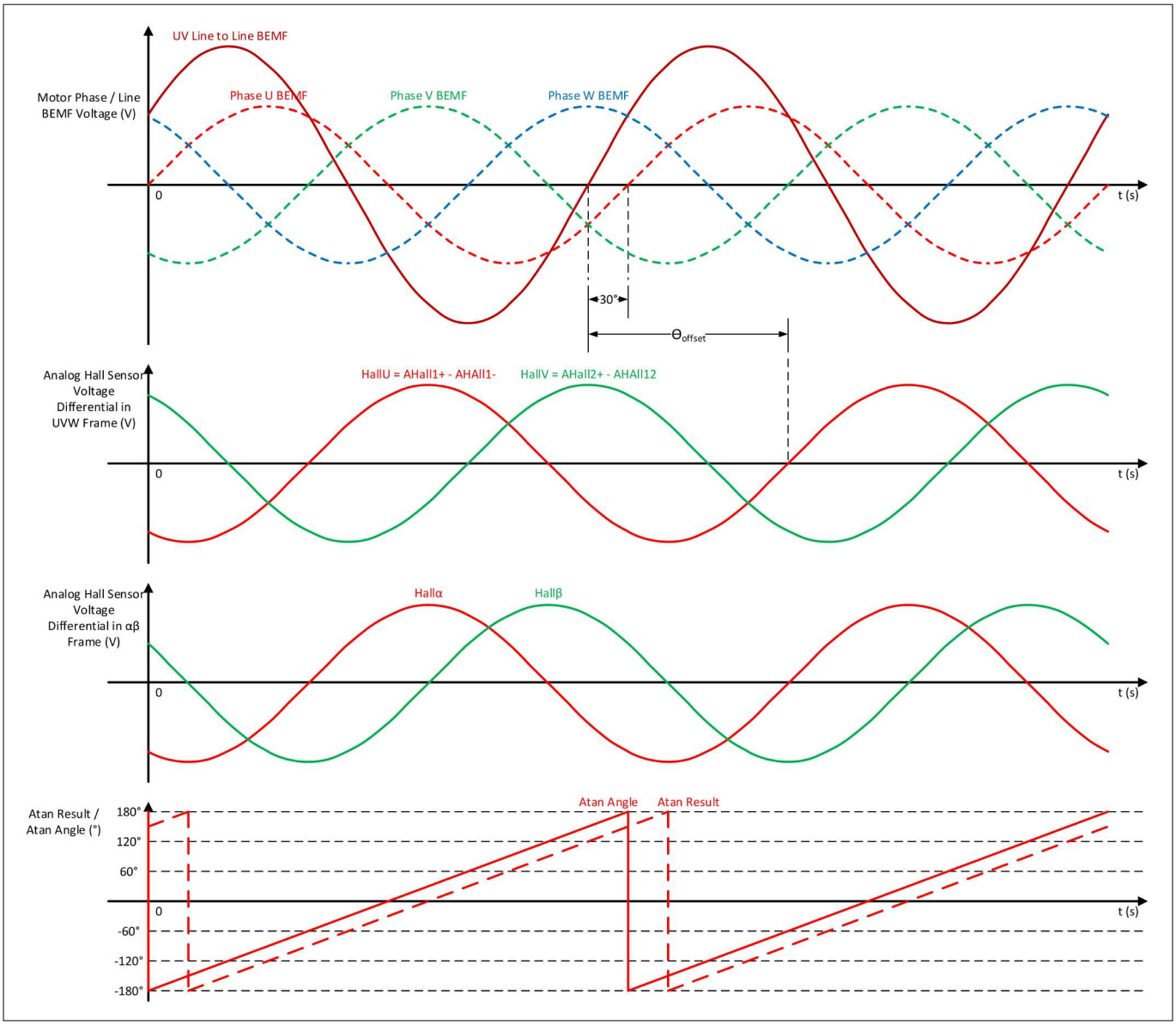


Figure 30 **Atan Angle Calculation Based on 2 Analog Hall Sensor Inputs (AHall1+, AHall1-, AHall2+, AHall2-)**

2.8.8 Hall Initial Position Estimation

For digital 2 or 3 Hall configurations, as well as analog 2 Hall configuration without using Atan angle calculation method, the initial rotor position at the start-up is estimated by the MCE based on the initial Hall inputs. The MCE assumes that the rotor starts in the middle of the angle range which is interpreted from the initial Hall pattern. The following Table 4 and Table 5 show the initial angle estimation details for 3 Hall and 2 Hall configurations.

Table 4 **Hall Initial Position Estimation (3 Hall, HallAngleOffset = 0)**

Hall pattern [H3, H2, H1]	1 (001_b)	3 (011_b)	2 (010_b)	6 (110_b)	4 (100_b)	5 (101_b)
Angle range	-60° to 0°	0° to 60°	60° to 120°	120° to 180°	180° to 240°	240° to 300°
Initial angle	-30°	30°	90°	150°	210°	270°

Table 5 Hall Initial Position Estimation (2 Hall, HallAngleOffset = 0)

Hall pattern [H3, H2, H1] (H3 = 0)	1 (001_b)	3 (011_b)	2 (010_b)	0 (000_b)
Angle range	-120° to 0°	0° to 60°	60° to 180°	180° to 240°
Initial angle	-60°	30°	120°	210°

2.8.9 Hall Sensor/Sensorless Hybrid Operation

The MCE supports a hybrid mode where both the Hall sensor interface driver and the flux estimator and flux PLL are active. As shown in the following Figure 31, the rotor angle uses estimated Hall angle from the Hall sensor interface driver during the start-up. As the motor speed increases to more than the Hall-to-Flux speed threshold configured by the parameter 'Hall2FluxThr', the rotor angle switches over to using estimated flux angle from the flux estimator and flux PLL. While the rotor angle is fed from flux angle, if the motor speed decreases to below the Flux-to-Hall speed threshold configured by the parameter 'Flux2HallThr', the rotor angle switched back to using estimated Hall angle from the Hall sensor interface driver. In hybrid mode, both the Hall sensor interface driver and the flux estimator and flux PLL are running concurrently although only one out of the two outputs is being used as rotor angle to close the angle loop.

While the MCE offers an advanced sensorless algorithm with excellent performance, some applications require better performance at start-up and/or very low speed operations. In this case, using Hall sensors can complement the sensorless option in providing superior start-up and low speed performance. Thus, it is recommended to select hybrid mode to take advantage of both the sensorless mode and the Hall sensor mode to ensure a consistent high performance of a drive system across a wide speed range including start-up.

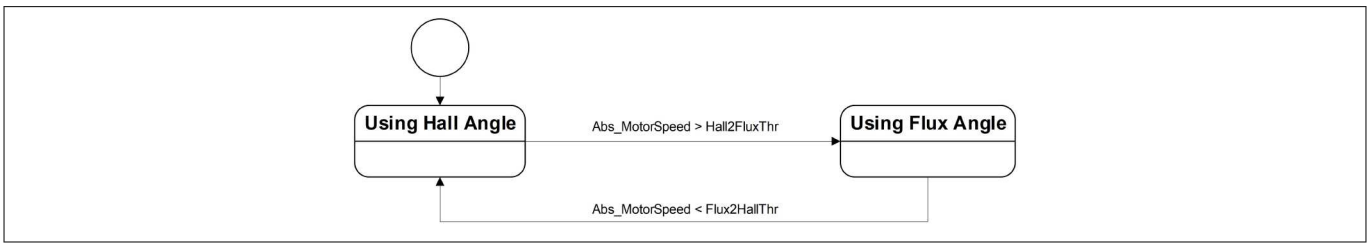


Figure 31 Hall Sensor/Sensorless Hybrid Mode Diagram

2.9 DC Bus Compensation

DC bus voltage typically has high frequency ripple as well as 2 times AC input line frequency (F_{line}) ripple. The low frequency ripple is dominant due to limited size of DC bus capacitors. The instantaneous DC bus voltage is part of the motor current control loop gain. Thus, if the current loop bandwidth is not high enough, then there is not enough loop gain at 2 x F_{line} frequency. As a result, the current loop won't be able to adjust the Modulation Index (MI) accordingly to ensure stable inverter output voltage. The resulting motor phase current would inevitably be modulated by 2 x F_{line} frequency DC bus voltage ripple.

The MCE provides a DC bus voltage feedforwarding function to compensate for the effect of the DC bus voltage variation on the current control loop gain, so that the actual MI is not affected by the DC bus voltage ripple.

As shown in the following Figure 6, if the DC bus compensation is enabled, Valpha and Vbeta, that are the 2 orthogonal components of the desired inverter output voltage, are adjusted by a factor of the ratio between 50% of the DC bus full range voltage to the instantaneous DC bus voltage. The adjusted results, Malpha and Mbeta, are the 2 orthogonal components of the desired output voltage vector, based on which the SVPWM block generates the three phase PWM switching signals. Additionally, the vector voltage limit (parameter 'VdqLim') is also adjusted inversely by the DC bus compensation factor to make full inverter voltage available. If DC bus compensation is disabled, Valpha and Vbeta are directly coupled with Malpha and Mbeta without any

additional adjustment. Flux estimator parameters are scaled based on 50% of the DC bus full range voltage. So, If the DC bus compensation is disabled, it is required to compensate Voltage alpha (ValphaComp) and Voltage beta (VbetaComp) components used in flux estimator based on the DC bus voltage. If the DC bus compensation is enabled, no compensation required in voltage alpha and voltage beta components used in flux estimator. Motor voltage (Vdq) is calcuated based on Vd and Vq values. Calculation is done every 1 ms.

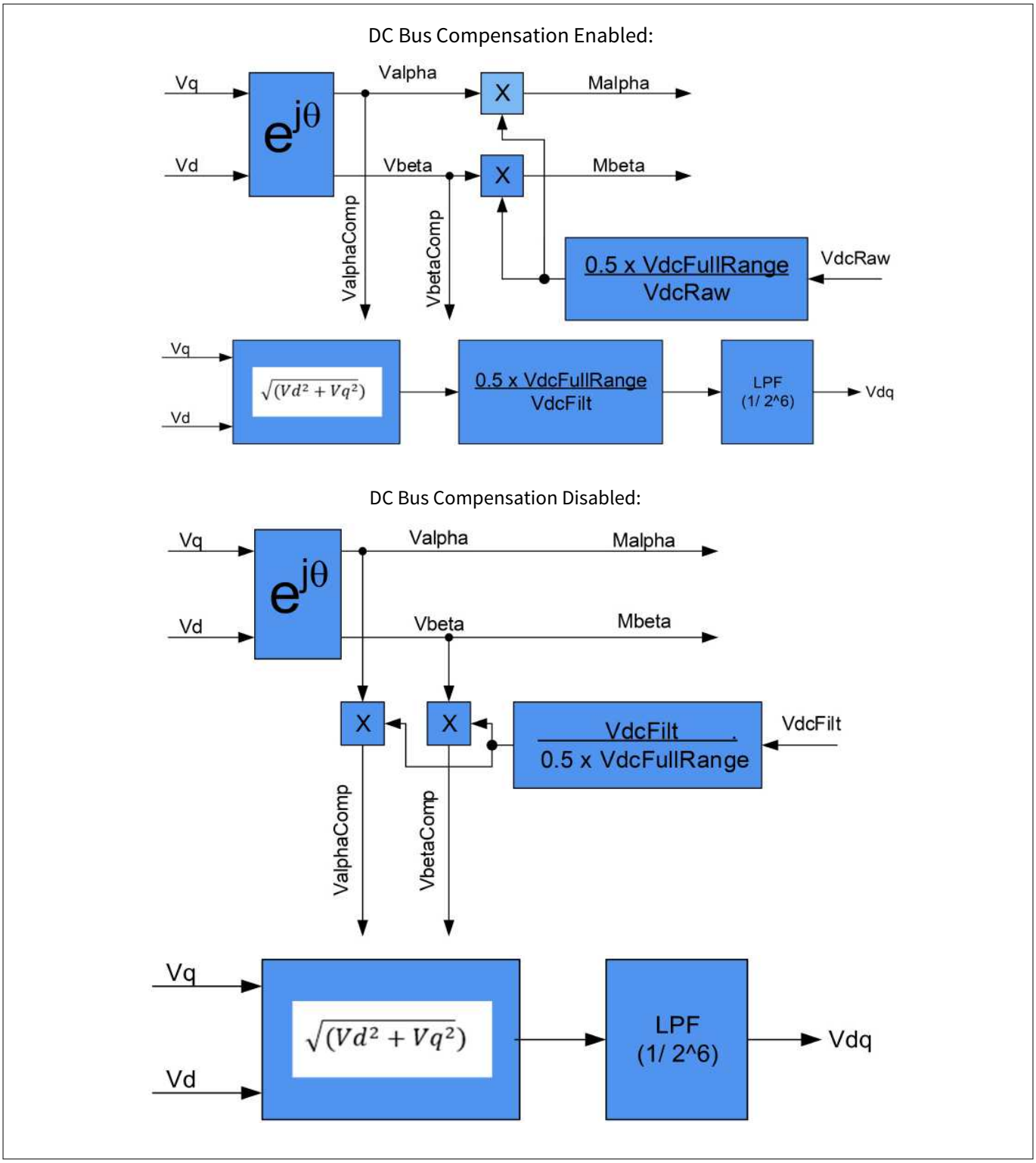


Figure 32 **DC Bus Compensation Functional Diagram**

The DC bus full range voltage is the maximum DC bus voltage that the ADC can sample up to given a specified voltage divider for DC bus voltage sensing. Referring to Figure 5, it can be calculated using the following equation.

$$V_{DCFullRange} = V_{ADC_REF} \times \frac{R1 + R2}{R2}$$

DC bus compensation function can be enabled by setting bit [0] of 'SysConfig' parameter.

With the DC bus compensation function disabled, the actual MI can be estimated using the variable 'MotorVoltage' following this equation:

$$MI = \frac{MotorVoltage}{4974}$$

With the DC bus compensation function enabled, the actual MI can be estimated using the variables 'MotorVoltage' and 'VdcRaw' following this equation:

$$MI = \frac{MotorVoltage \times \frac{2048}{VdcRaw}}{4974} \times 100\,\%$$

2.10 Motor Current Limit Profile

Some applications (such as a fan) don't require a high current at low speed. In other words, full torque is only required above a certain speed. The MCE provides a configurable dynamic motor current limit feature which reduces the current limit in the low speed region for a smooth startup. This feature provides smooth and quiet start up, and it also can reduce the rotor lock current.

Figure 33 depicts that the motor current limit changes dynamically as a function of motor speed. The MCE enables the motor load to work in both motoring mode (1st and 3rd quadrants in Figure 33) and regenerating mode (2nd and 4th quadrants Figure 33).

In motoring mode, when the absolute value of the motor speed is below the minimum speed specified by the parameter 'MinSpd' ($|MotorSpeed| \leq MinSpd$), the maximum motor current is limited to a threshold configured by parameter 'LowSpeedLim'. When the absolute value of the motor speed is between the minimum speed and the low speed threshold, the motor current limit increases linearly as the speed increases following the relationship as below.

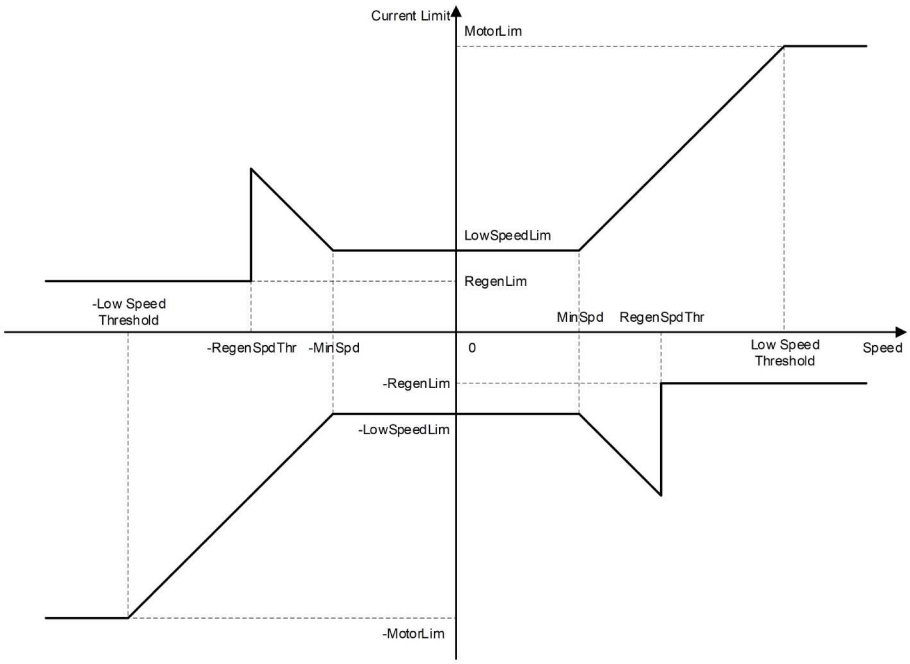
$$MotorCurrentLimit = LowSpeedLim + (|MotorSpeed| - MinSpd) \times LowSpeedGain$$

When the motor speed goes beyond the low speed threshold, the maximum motor current is limited to the upper boundary specified by the parameter 'MotorLim'.

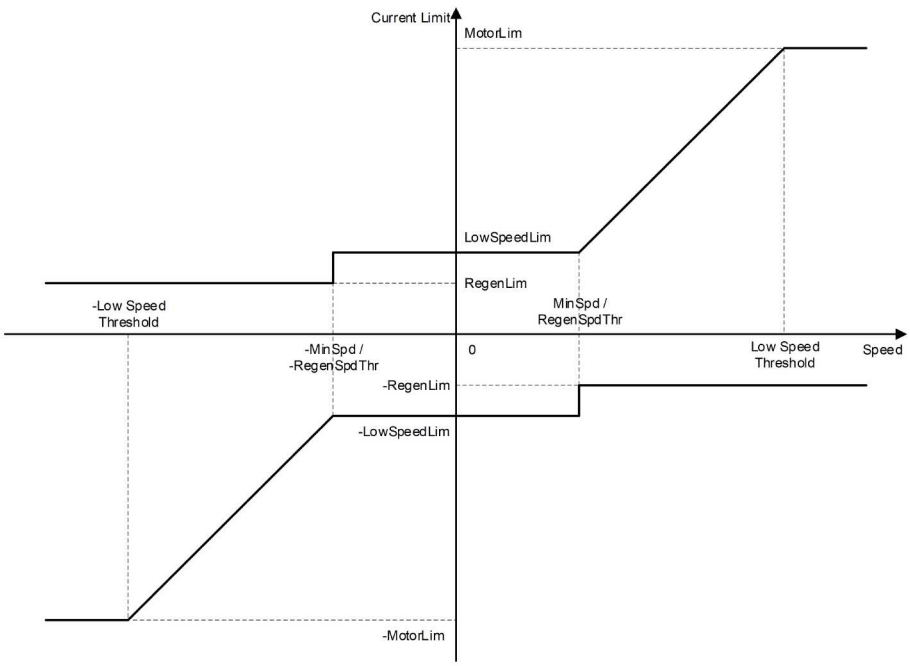
In regenerating mode, when the absolute value of motor speed is below a threshold specified by the parameter 'RegenSpdThr', the motor current limit follows the above-mentioned linear relationship. When the absolute value of motor speed goes beyond the threshold specified by the parameter 'RegenSpdThr', the maximum motor current is limited to a threshold specified by the parameter 'RegenLim'.

Having the freedom to adjust the motor current limit in motoring mode and regenerating mode independently allows users to tailor the acceleration torque as well as the regenerative braking torque separately to achieve optimal drive performance. If further customization of motor current limit is required, users can take advantage of script code to program the motor current limit ('MotorLim' parameter) to any arbitrary profile.

Case 1: MinSpd < RegenSpdThr



Case 2: MinSpd = RegenSpdThr



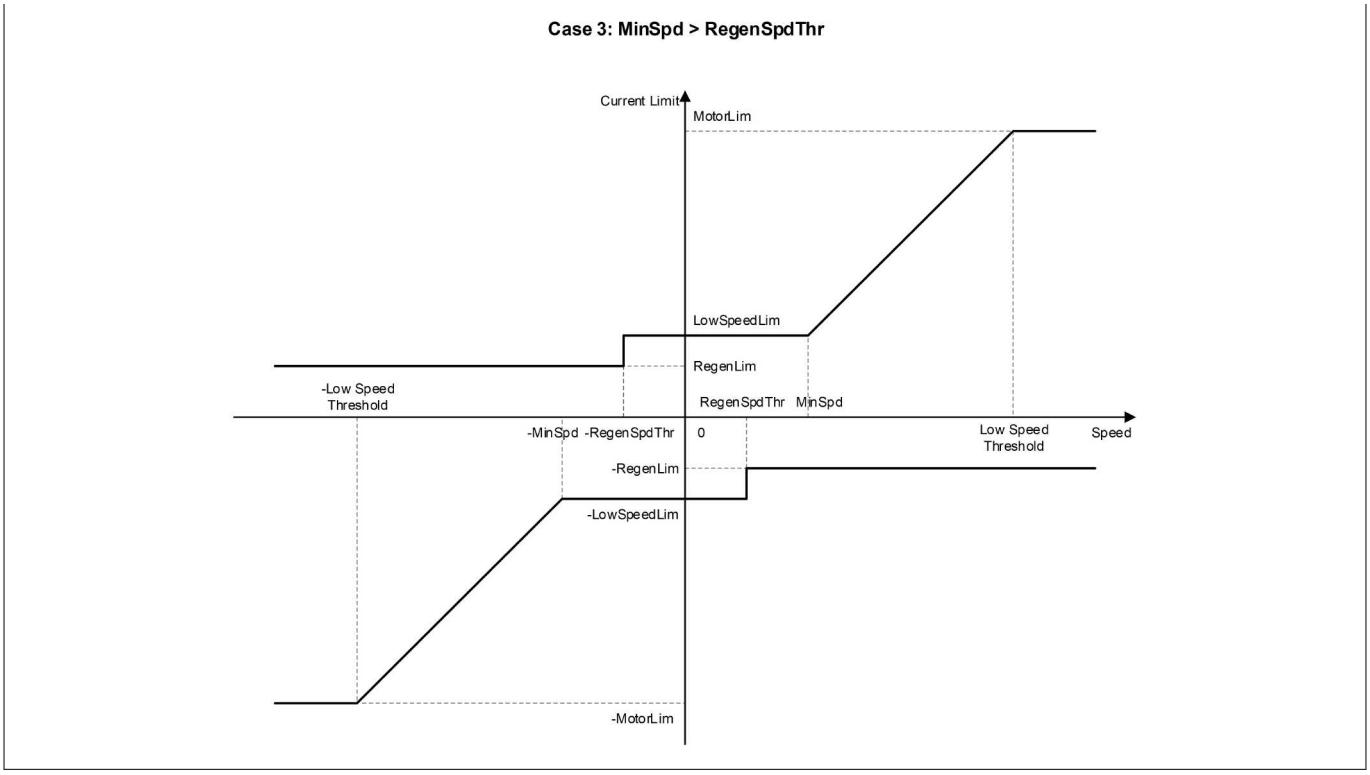


Figure 33 **Motor Current Limit Profile**

2.11 Over-Modulation

As shown in the following Figure 34, the linear modulation range is defined by the disk that fits within the hexagonal active voltage vector (a, b) timing limit boundary. The modulation index can be up to $\frac{\sqrt{3}}{2} = 0.866$ if the modulation stays within linear range. If maximizing output power is the priority and non-linear modulation is acceptable, then the modulation index can go up to 1 so that the active voltage vector goes outside the disk into the grey area to make full use of the DC bus voltage.

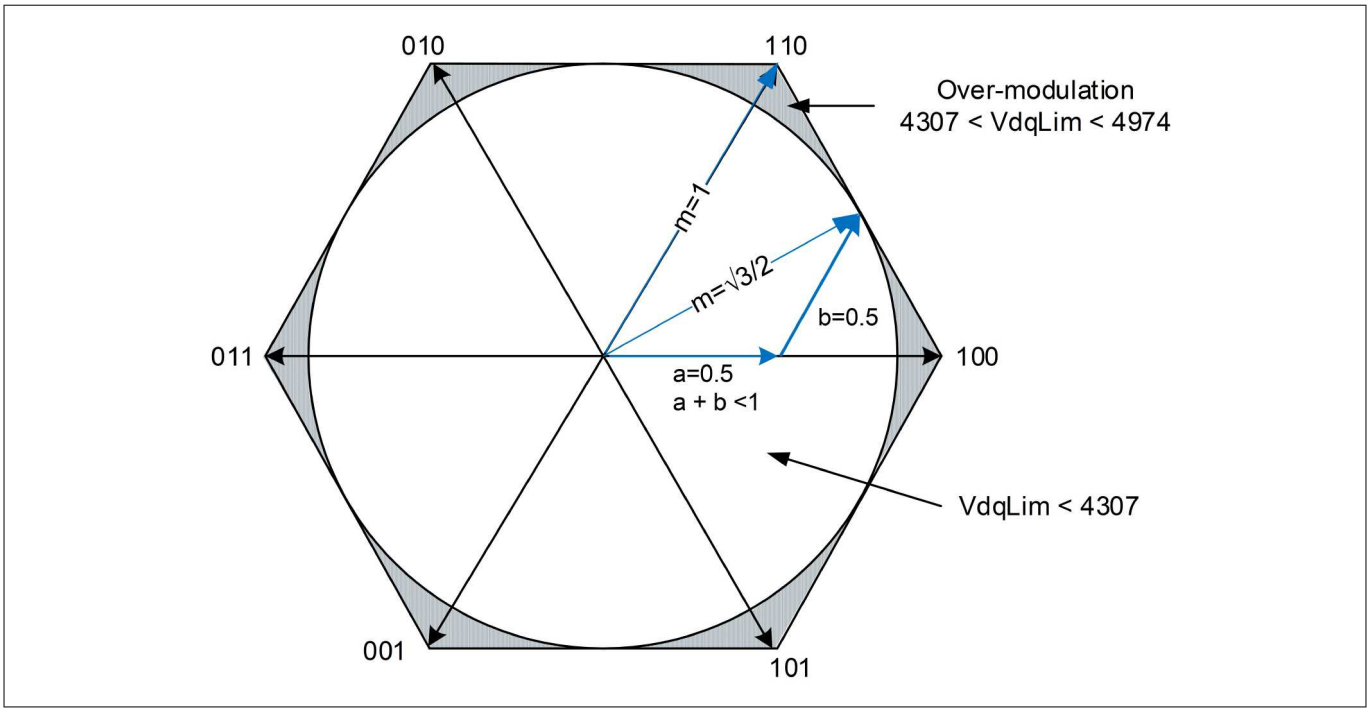


Figure 34 SVPWM Vector Timing Limit Diagram

The MCE offers the parameter 'VdqLim' to configure the desired modulation index limit. 100% modulation corresponds to 4974 counts for parameter 'VdqLim'. If users need to limit the modulation to only linear range, the parameter 'VdqLim' shall be set up to 4974 x 0.866 = 4307. If users need to take advantage of over-modulation, then the parameter 'VdqLim' shall be set up to 4974.

Although utilizing over-modulation helps maximize DC bus voltage utilization, it would introduce acoustic noise associated with the additional harmonics, and compromise the flux PLL operation and result in errors in RMS current and voltage based power or torque calculations.

2.12 2-Phase Modulation

MCE supports 2-phase type 3 (low-side clamping) space vector PWM modulation with a configurable switch-over threshold. As shown in the following Figure 35, 2-phase type 3 modulation clamps one motor winding to the negative inverter rail. Thus, it eliminates switching of one of the 3 inverter legs in each sector to reduce switching loss while keeping the output line voltage the same as compared to the case of 3-phase modulation. This is done by not using zero vector [111] and allocating all the zero vector time to the zero vector [000].

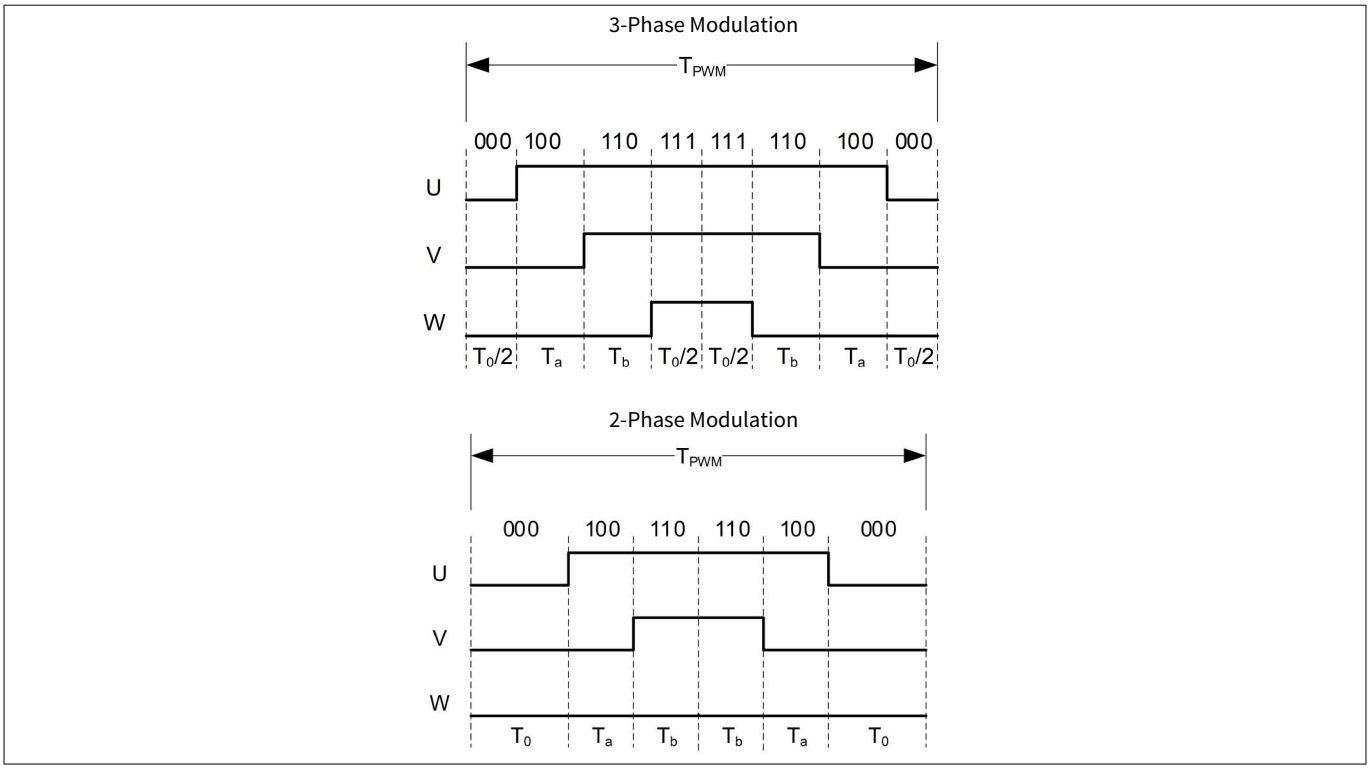


Figure 35 **3-Phase/2-Phase Type 3 SV PWM Modulation Diagram**

2-phase type 3 PWM modulation cannot be used at low speeds when the high side gate driver uses a bootstrap diode to charge up the voltage rail. The bootstrap capacitor must be sized sufficiently to hold enough charge to drive the high side gate for the full duration of a SV PWM Modulation sector.

Bit field [4:3] of the parameter ‘HwConfig’ is used to enable 2-phase type 3 PWM modulation. As shown in the following Figure 36, if 2-phase type 3 SVM is enabled, at start-up 3-phase PWM modulation is used. When the motor absolute speed (variable ‘Abs_MotorSpeed’) goes above a configurable threshold (parameter ‘Pwm2PhThr’), MCE would switch to using 2-phase type 3 PWM modulation. When the absolute motor speed goes below the configurable threshold (parameter ‘Pwm2PhThr’) with a hysteresis of 256 counts (1.6% of motor maximum RPM), MCE would switch back to 3-phase PWM modulation.

If the value of the parameter ‘Pwm2PhThr’ is 256 or lower (≤ 1.6% of motor maximum RPM), and 2-phase PWM modulation is enabled, after MCE has switched to 2-phase PWM modulation, it would not switch back to 3-phase PWM modulation automatically until motor is stopped and then is restarted.

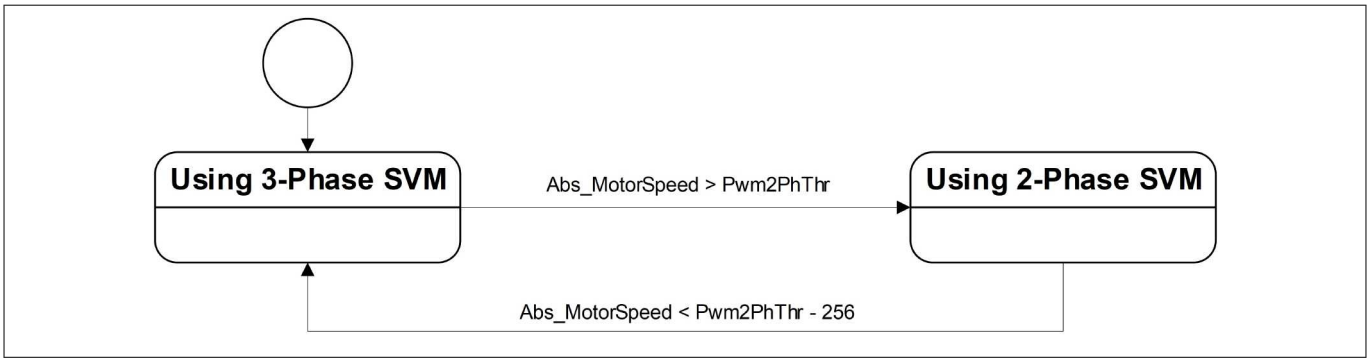


Figure 36 **3-Phase SVM and 2-Phase SVM State Transition Diagram**

2.13 Torque Compensation

For single rotary compressor based air-conditioners or refrigerator applications, big variation in the torque demand exists within a mechanical cycle from absorption stage and compression stage. Because of the limited speed loop bandwidth, the motor speed would vary due to the varying torque demand within one mechanical cycle, causing noticeable mechanical vibration and undesirable noise. To solve this problem, the MCE provides a torque compensation function that is able to detect and synchronize with the mechanical cycle and uses a feed forwarding loop to modulate torque reference following a sinusoidal compensation curve per mechanical cycle to minimize speed variation and so reduce vibration. This function uses the torque reference and flux angle (rotor electrical angle in sensorless mode) as inputs. It has two primary operating modes over a configurable speed range. In one mode it synchronizes with the peak load torque within a mechanical cycle, while in the other mode it calculates the feedforward compensation torque. The MCE's torque compensation function supports 4-pole or 6-pole compressor motor types.

The torque compensation function can be enabled or disabled by using bit [1] of the parameter 'SysConfig'.

Figure 37 depicts the state transitions for the MCE's torque compensation function.

When torque compensation is disabled by resetting bit [1] of the parameter 'SysConfig', it stays in TC_Disabled state and goes through an initialization process (TorqueComp_init()) where relevant variables including 'TrqCompBaseAngle', 'TrqCompStatus' and 'TrqCompOutput' are reset.

When torque compensation is enabled, by setting bit [1] of the parameter 'SysConfig', it shifts to TC_Enabled state.

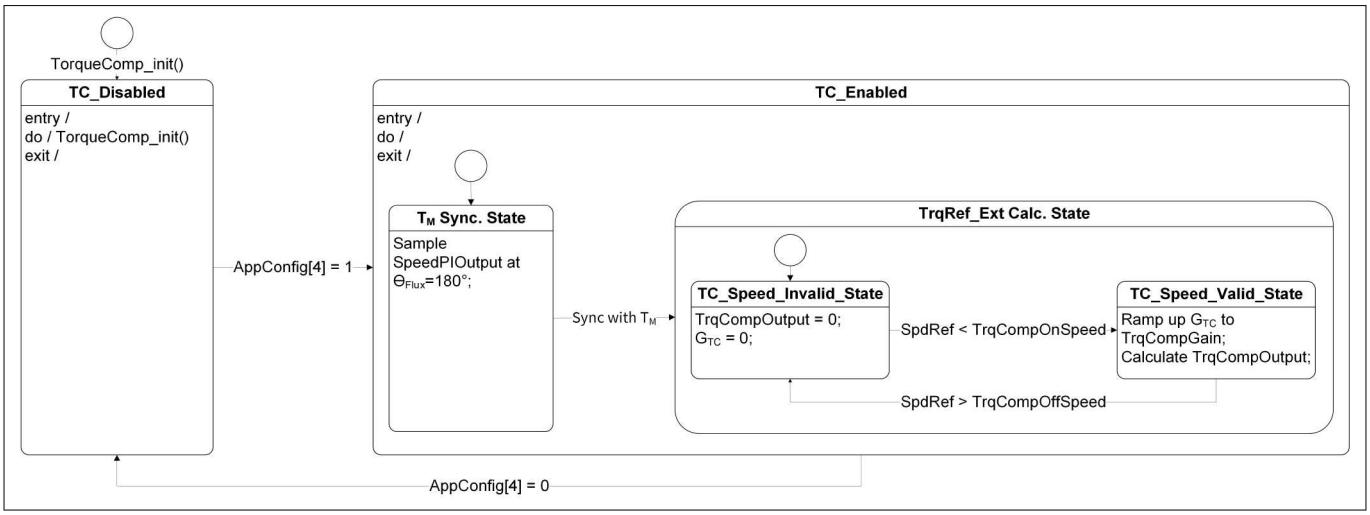


Figure 37 Torque Compensation State Transition Diagram

As shown in Figure 37 and Figure 38, there are 2 sub-states within TC_Speed_Valid state. When entering TC_Speed_Valid state, it starts from T_M Synchronization sub-state where it samples the torque reference (variable 'SpeedPIOutput') once every electrical cycle when flux angle θ_{Flux} = 180°. If the torque reference samples at the k_{th} sample time match the following criteria: SpeedPIOutput [k] > SpeedPIOutput [k-1], SpeedPIOutput [k] > SpeedPIOutput [k-2], SpeedPIOutput [k-3] > SpeedPIOutput [k-1], SpeedPIOutput [k-3] > SpeedPIOutput [k-2] for 10 consecutive mechanical cycles T_M, then it is considered as having synchronized with peak load torque within a mechanical cycle T_M, and that moment marks the zero point of torque compensation base angle θ_{base} (variable 'TrqCompBaseAngle'). Then it shifts to TrqCompOutput Calculation sub-state.

There are two sub-states inside TrqCompOutput Calculation sub-state. If the motor speed reference (variable 'SpeedRef) is lower than the turn-on threshold configured by the parameter 'TrqCompOnSpeed', then it shifts to TC_Speed_Valid sub-state where torque compensation function becomes active. While it is in TC_Speed_Valid sub-state, if the motor speed reference becomes higher than the turn-off threshold configured by the parameter 'TrqCompOffSpeed', then it shifts back to TC_Speed_Invalid sub-state where torque compensation fuction becomes inactive with $TrqCompOutPut$ and G_{TC} being reset to zero.

It shall be pointed out that once the torque compensation function achieves synchronization with mechanical cycle T_M, it does not lose synchronization with mechanical cycle T_M whether the motor speed reference is within the valid speed range (active) or not (inactive).

The active status of torque compensation function is reflected in bit [0] of variable 'TrqCompStatus'.

When torque compensation fuction is active, the desired sinusoidal compensation torque reference (variable 'TrqCompOutput') is synthesized following this equation:

$$TrqCompOutPut = G_{TC} \times Trq\mathrm{Re}\,f_{Filt} \times \cos\left(\frac{\theta_{Flux} - 180°}{pole_pair} + \theta_{base} + \theta_{os}\right) \times k_{CORDIC}$$

G_{TC} represents the gain factor for the desired compensation torque reference 'TrqCompOutput'.

$TrqRef_{Filt}$ represents the averaged value of the desired torque reference from speed PI regulator output. It is the low-pass filtered result from variable 'SpeedPIOutput' with upper limit. As shown in Figure 38, if $TrqRef_{Filt}$ is greater than the value of the parameter 'TrqCompLim', then it is limited to the value of 'TrqCompLim'.

k_{COR} is an internal fixed gain factor ($k_{COR} = 1.647$).

The amplitude of the desired sinusoidal compensation torque reference is $G_{TC} \times Trq\mathrm{Re}f_{Filt} \times k_{CORDIC}$. G_{TC} starts from zero and ramps up at a rate of 8 counts per electrical cycle till it reaches the value of parameter 'TrqCompGain'.

The torque compensation base angle θ_{base} increments by 120° (6-pole) or 180° (4-pole) every electrical cycle.

The torque compensation angle offset θ_{TCos} (parameter 'TrqCompAngOfst') specifies the angle difference between the peak load torque within a mechanical cycle and the peak of the synthesized sinusoidal compensation torque reference.

The status of synchronization with mechanical cycle T_M is reflected in bit [1] of variable 'TrqCompStatus'.

As shown in Figure 38, the synthesized compensation torque reference 'TrqCompOutput' is summed up with 'SpeedPIOutput' to form total torque reference (variable 'TrqRef'), which is used in the following IPM (Interior Permanent Magnet) control block to generate current references for d and q axis current loops.

If it is needed to restart the synchronization with the mechanical cycle T_M, the torque compensation function shall be disabled and enabled again by toggling bit [1] of the parameter 'SysConfig'.

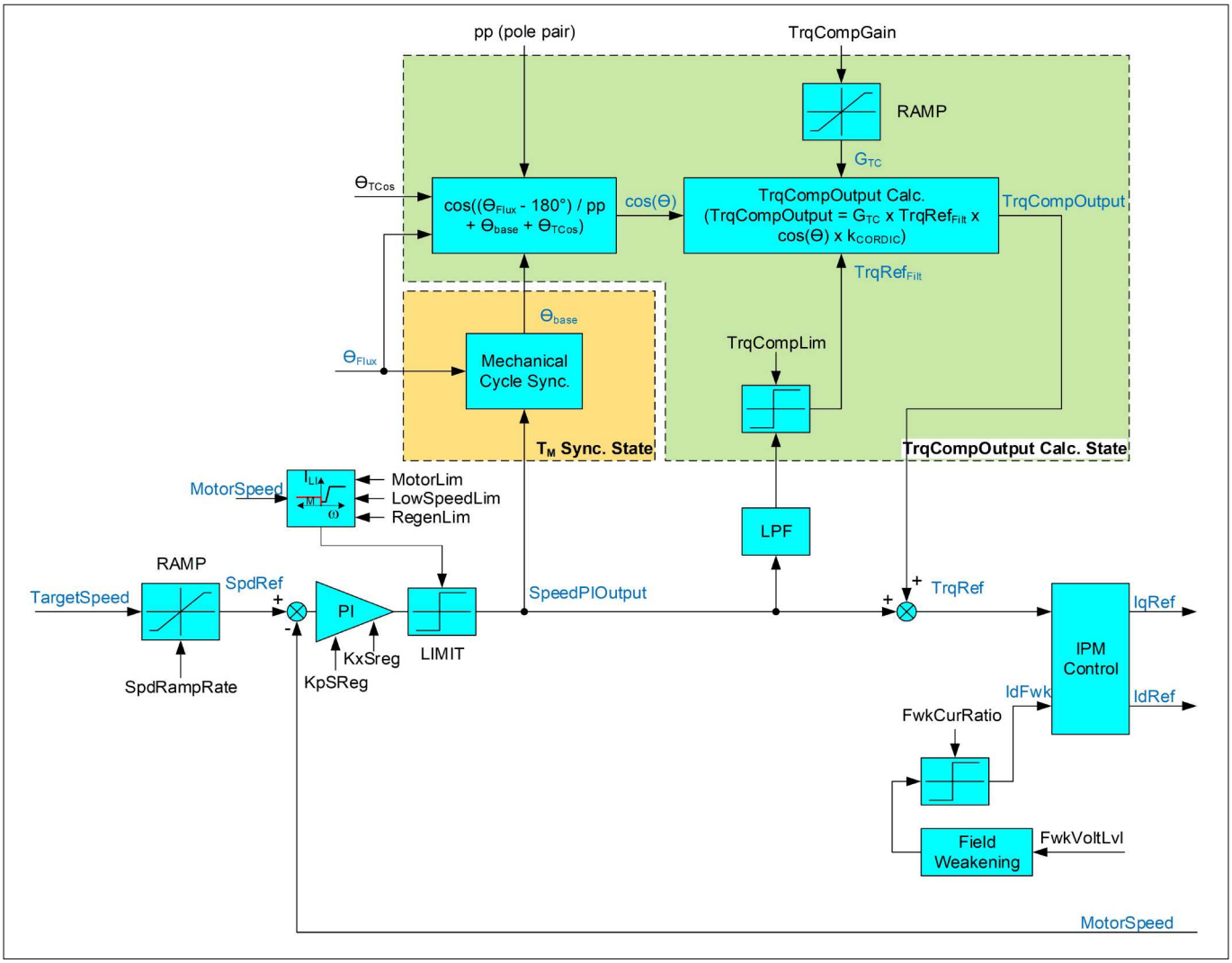


Figure 38 **Torque Compensation Top-Level Algorithm Diagram**

The following steps are recommended to tune the torque compensation related parameters.

1. Set initial parameter values: G_{TC} (TrqCompGain = 128) and 50% compensation torque limit (TrqCompLim = 2048)
2. Set the speed rising threshold (TrqCompOffSpeed) above which torque compensation function shall be inactive. Set the speed falling threshold (TrqCompOnSpeed) below which torque compensation function shall be active. These two parameters shall have about 2% hysteresis to avoid oscillation
3. Set bit [1] of the parameter 'SysConfig' to enable torque conpensation function
4. Use tracing function in Solution Designer to plot variable 'SpeedError'
5. Adjust 'TrqCompAngOfst' value to a value with which the amplitude of 'SpeedError' as well as compressor vibration is minimized
6. Increase 'TrqCompGain' value to further reduce the compressor vibration if needed

2.14 Zero-Vector Braking

In some application there is a need to slow down the motor quickly and bring it to a full stop. The MCE offers zero-vector braking on request which can be initiated by setting an MCE parameter. When zero-vector braking is applied, the MCE clamps all the low-side switches of the inverter. This shorts the motor phases and the BEMF of the motor drives phase currents through the windings as shown in Figure 39. The kinetic energy stored in the motor/load is then dissipated as an electric loss in the motor windings.

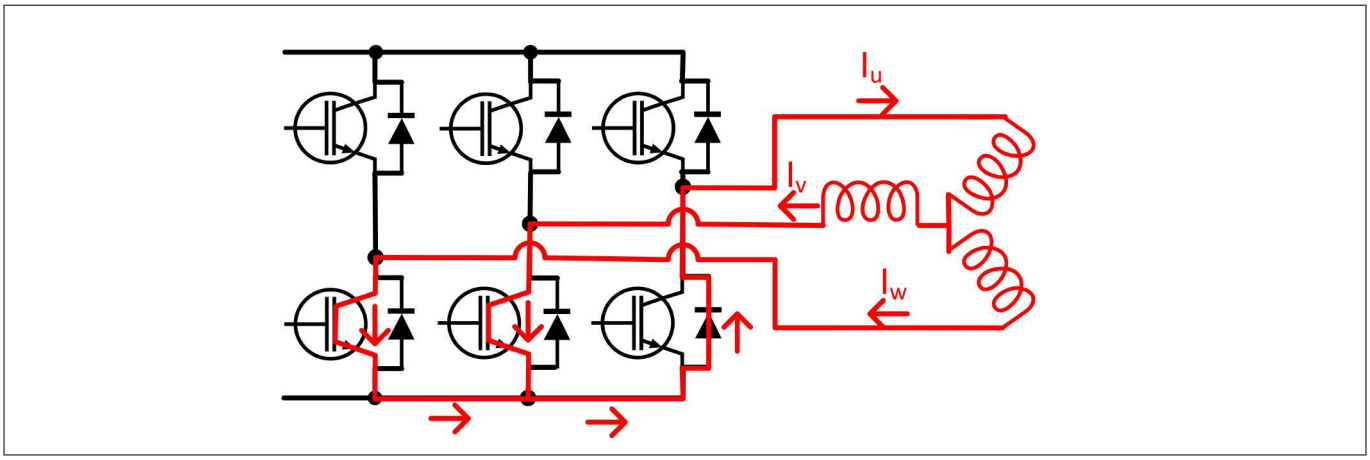


Figure 39 **Zero-Vector Braking by Clamping of Low-Side Switches (IGBT Case)**

During zero-vector braking the MCE firmware handles all required state transitions and continues to service all applicable protection functions. Note that the MCE also utilizes zero-vector clamp in case of the fault Critical Over Voltage.

2.14.1 Parameters

Zero-vector braking is applied when the parameter APP_MOTOR0.ZeroVectorBrake is set to '1'. Similar, zero-vector braking is canceled by setting the parameter APP_MOTOR0.ZeroVectorBrake to '0'.

Current state of the zero-vector can be checked be reading the bitfield ZeroVecBrake of parameter APP_MOTOR0.MotorStatus where '1' indicates that zero-vector is applied and '0' indicates that zero-vector is not applied. The parameter APP_MOTOR0.MotorStatus reflects the state of zero-vector which is not necessarily the same as APP_MOTOR0.ZeroVectorBrake = 1.

2.14.2 State Dependencies

When the parameter APP_MOTOR0.ZeroVectorBrake is set to '1' the system automatically switches to STOP state. Zero-vector-braking is applied until one of the flowing events happens:

1. Zero-vector braking is canceled by setting APP_MOTOR0.ZeroVectorBrake = 0
2. A motor start command is received from Solution Designer, scripting or a control interface. A start command automatically clears APP_MOTOR0.ZeroVectorBrake to '0'
3. A gate-kill fault. If APP_MOTOR0.ZeroVectorBrake = 1 prior to the gate-kill the parameter remains set and zero-vector braking is re-applied when the gate-kill fault is cleared

Zero-vector braking can only be applied when the system is in one the following states: RUN (4), RUN_HALL (10), RUN_HYBID (11), RUN_OPENLOOP (12), FAULT (5) and STOP (1).

Zero-vector braking cannot be applied when the system is in the flowing states: CATCHSPIN (6), ANGLESENSE (9), PARKING (7), OPENLOOP (8), IDLE (0), OFFSETCAL (2) and STANDBY (13).

The number in parenthesis refers to the value of MCEOS.Motor_SequencerState in the respective states. Refer to Chapter 2.1 for information on the state machine.

If zero-vector braking is requested while the system is in a state where zero-vector braking is not supported, the request will be temporarily ignored but remembered. When the system reaches a state where zero-vector braking is supported, the zero-vector will be applied. For example, if zero-vector braking is requested during the state PARKING, it will be ignored and the system will continue through OPENLOOP and into RUN where the zero-vector braking then takes effect.

While zero-vector braking is applied, the motor current measurements are updated. However, the currents are only observable with leg-shut current sensors. With single-shunt current sensor, the phase currents are

updated but, since the motor currents are not flowing through the single shunt current sensor, they are not observable.

2.14.3 Zero-Vector Braking During Faults

Should a critical overvoltage fault or gate-kill fault occur during zero-vector-braking, the fault takes priority. In the case of critical overvoltage, the system applies zero-vector to protect the inverter. If zero-vector braking is canceled during the critical overvoltage, the zero-vector continues to be applied due to the fault.

In case of a gate-kill fault when zero-vector braking is applied, all gate signals are put in the inactive state which effectively cancels zero-vector braking. Once the gate-kill fault is cleared, the system resumes zero-vector braking.

The bitfield ZeroVecBrake of parameter APP_MOTOR0.MotorStatus will always indicate whether zero-vector is applied or not.

All other faults have no effect on zero-vector braking.

2.15 Catch Spin

Before turning on the inverter, due to some external force, for example wind air flow in fan applications, the motor may be already spinning. The MCE offers ‘Catch Spin’ feature which is designed to synchronize the flux estimator and flux PLL with the actual motor speed before providing the torque to drive the motor. Catch spin cannot be done if the motor back EMF voltage is higher than the DC bus voltage, which usually occurs when the motor is running above rated speed. Hence, the catch spin is generally effective up to the rated speed of the motor. The catch spin starting process is part of the motor state machine and is executed at start-up if catch spin function is enabled.

In catch spin, the controller tracks the back EMF in order to determine if the motor is turning, and if so, in which direction. Catch spin sequence begins after the bootstrap capacitor charging stage is completed. During catch spin, both IqRef and IdRef are set to 0 (Speed regulator is disabled), meanwhile flux PLL attempts to lock to the actual motor speed (variable ‘MotorSpeed’) and rotor angle (variable ‘RotorAngle’). Catch spin time, defined by TCatchSpin parameter. Once catch spin time is elapsed, calculated motor speed check with “DirectStartThr” parameter value. If motor speed is more than or equal to “DirectStartThr” parameter value, normal speed control starts, current motor speed will become the initial speed reference and also set as the speed ramp starting point. Depending on the set target speed, motor will decelerate (via regenerative braking) or accelerate to reach the desired speed. If motor speed is less than “DirectStartThr” parameter value, motor state changes to “ANGLESENSING” state.

Depending upon the direction of rotation, there are 3 types of catch spin scenarios

- Zero Speed Catch Spin
- Forward Catch Spin
- Reverse Catch Spin

2.15.1 Zero Speed Catch Spin

If the motor is stationary, then the catch spin sequence is termed as ‘Zero Speed Catch Spin’. Figure 40 (A) shows an example for ‘Zero Speed Catch Spin’. In this example, at the start command, the motor is stationary. After the start command, ‘Zero Speed Catch Spin’ sequence begins. During the catch spin sequence, no motoring current is injected. After the catch spin time has elapsed, the motor speed at that instance (which is 0 RPM) becomes initial speed reference and starting point for speed ramp reference. The motor continues to accelerate, following the speed ramp reference to reach the set target speed.

If catch spin is disabled, normal speed control starts immediately after the start command, without waiting for PLL to be locked. As shown in Figure 41 (B), after the start command, motoring current is injected directly as

there is no catch spin sequence. The motor starts accelerating, following the speed ramp reference to reach the set target speed.

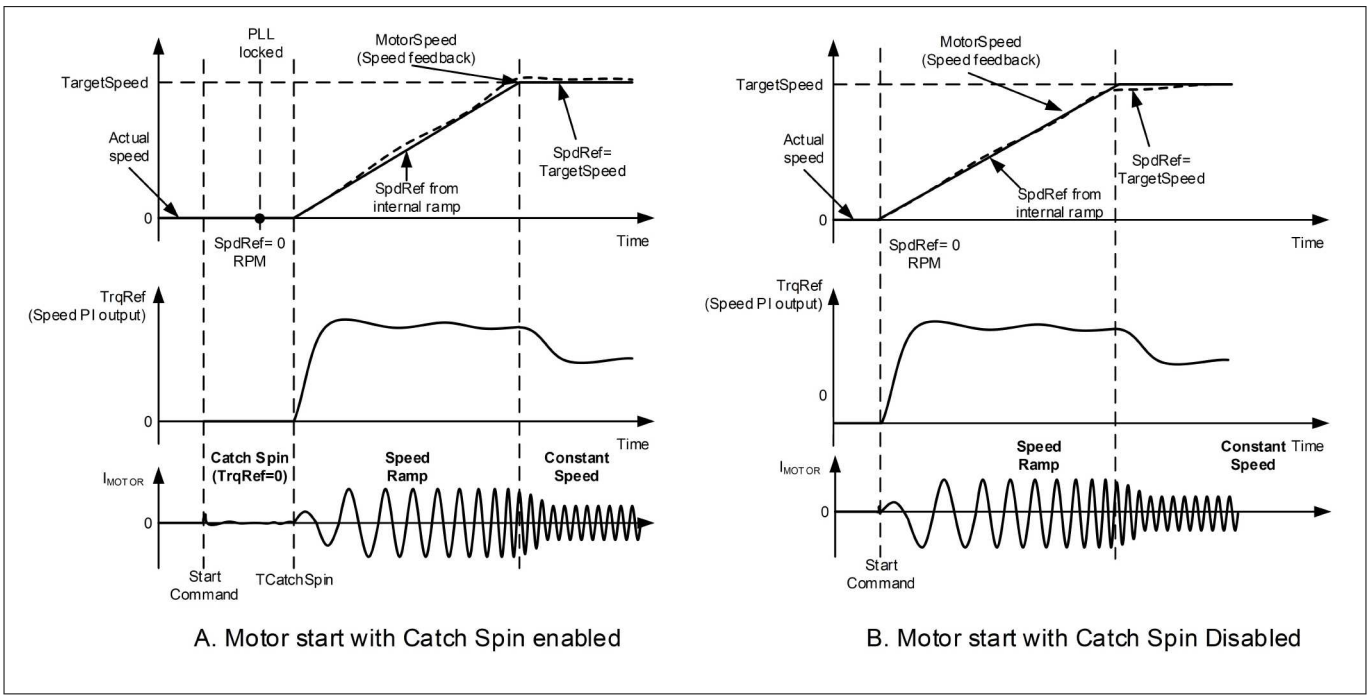


Figure 40 **Zero Speed Catch Spin - Motor start with/without catch spin**

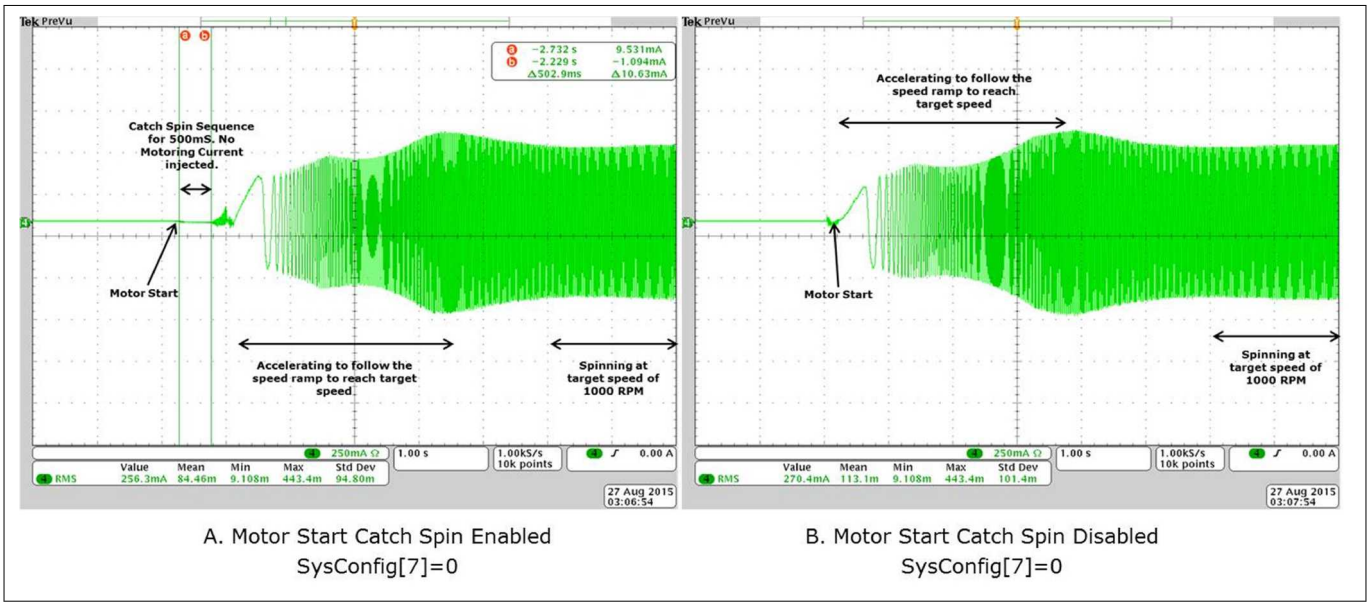


Figure 41 **Motor Phase Current - Zero Speed Catch Spin - Motor start with/without catch spin**

2.15.2 Forward Catch Spin

If the motor is spinning in the same direction as desired, then the catch spin sequence is termed as 'Forward Catch Spin'. Figure 42 (A) shows an example for 'Forward Catch Spin'. In this example, at the start command the motor is already spinning (in the desired direction). During the catch spin sequence, no motoring current is injected. After the catch spin time has elapsed, assuming the flux PLL locks to the actual motor speed, the motor speed at that instance becomes initial speed reference and starting point for speed ramp reference. The motor continues to accelerate or decelerate, following the speed ramp reference to reach the set target speed.

If catch spin is disabled, normal speed control starts immediately after the start command, without waiting for PLL to be locked. Usually the control would still be able to start a spinning motor, but motor speed may not increase/decrease seamlessly. As shown in Figure 42 (B), after the start command, the actual motor speed is higher than speed reference (variable 'SpeedRef). Hence, the motor is decelerated (using regenerative braking) to force the motor to follow the speed reference (variable 'SpeedRef). As the speed of the motor is higher than Regen Speed Threshold (variable 'RegenSpdThr'), the negative torque injected in the motor to achieve deceleration is limited by the value in RegenLim parameter. Once the motor speed matches the speed reference, the motor starts accelerating, following the speed ramp reference to reach the set target speed.

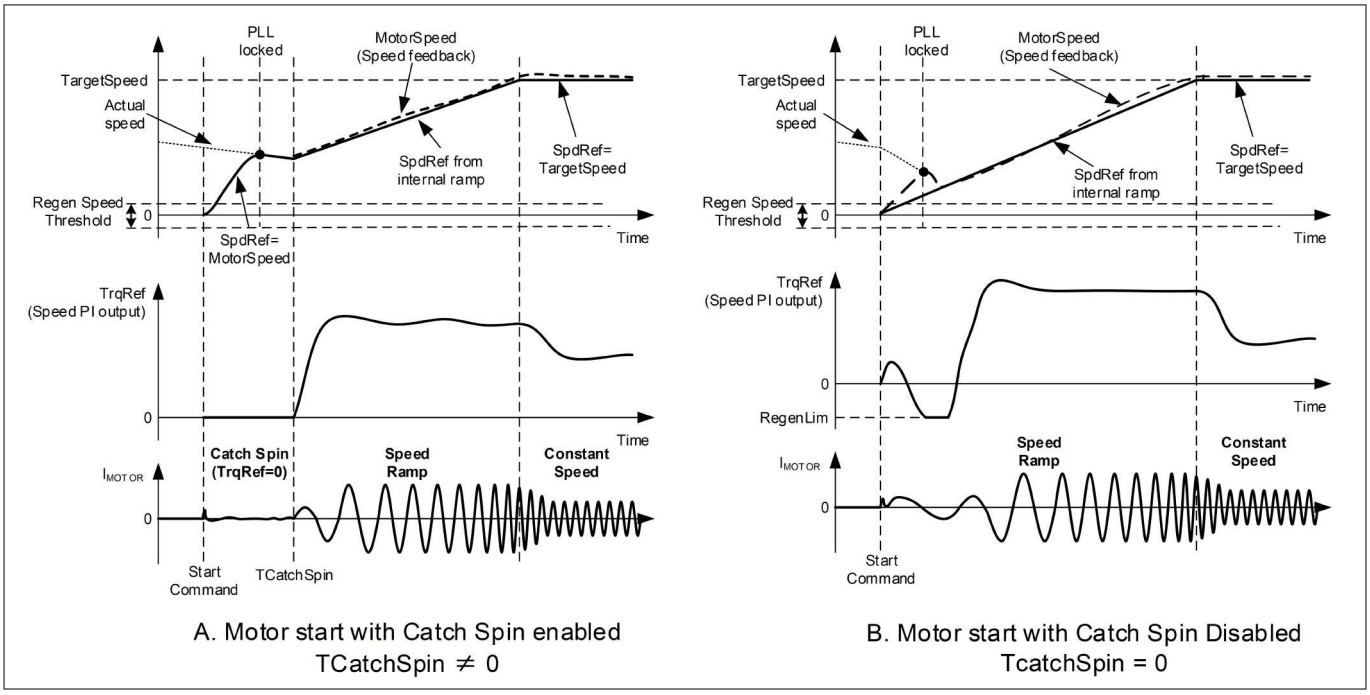


Figure 42 **Forward Catch Spin - Motor start with/without catch spin**

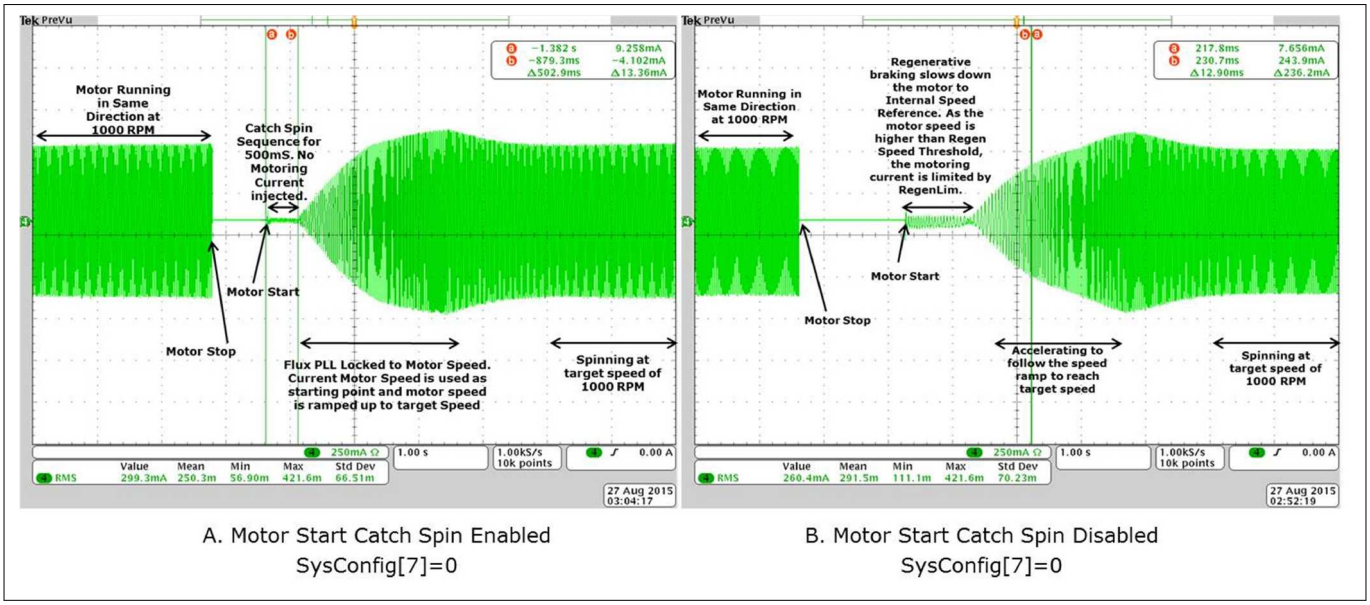


Figure 43 **Motor Phase Current Waveform - Forward Catch Spin - Motor start with/without catch**

2.15.3 Reverse Catch Spin

If the motor is spinning in the opposite direction as desired, then the catch spin sequence is termed as ‘Reverse Catch Spin’. Figure 44 (A) shows an example of ‘Reverse Catch Spin’. In this example, at the start command, the motor is already spinning (in the opposite direction). During the catch spin sequence, no motoring current is injected. After the TCatchSpin time has elapsed, the motor is still spinning in opposite direction at a speed higher than Regen Speed Threshold (RegenSpdThr), so an injected torque, limited by the value defined in RegenLim parameter, forces the motor to decelerate via regenerative braking. Once the speed of the reverse spinning motor falls below Regen Speed Threshold (RegenSpdThr), the injected torque is limited by MotorLim (RegenLim < = MotorLim). The injected torque forces the motor to come to a stop and start accelerating in the desired spin direction, following the speed ramp reference to reach the set target speed.

If catch spin is disabled, normal speed control starts immediately after the start command, without waiting for PLL to be locked. Usually, the control would still be able to start a spinning motor, but motor speed may not increase/decrease seamlessly. As shown in Figure 44 (B), after the start command, the motor is still spinning at a speed higher than Regen Speed Threshold (RegenSpdThr), hence the injected torque limited by the value defined in RegenLim parameter, forces the reverse spinning motor to decelerate via regenerative braking. Once the speed of the reverse spinning motor falls below Regen Speed Threshold (RegenSpdThr), the injected torque is limited by MotorLim (RegenLim < = MotorLim). The injected torque forces the motor to come to a stop and start accelerating in the desired spin direction, following the speed ramp reference to reach the set target speed.

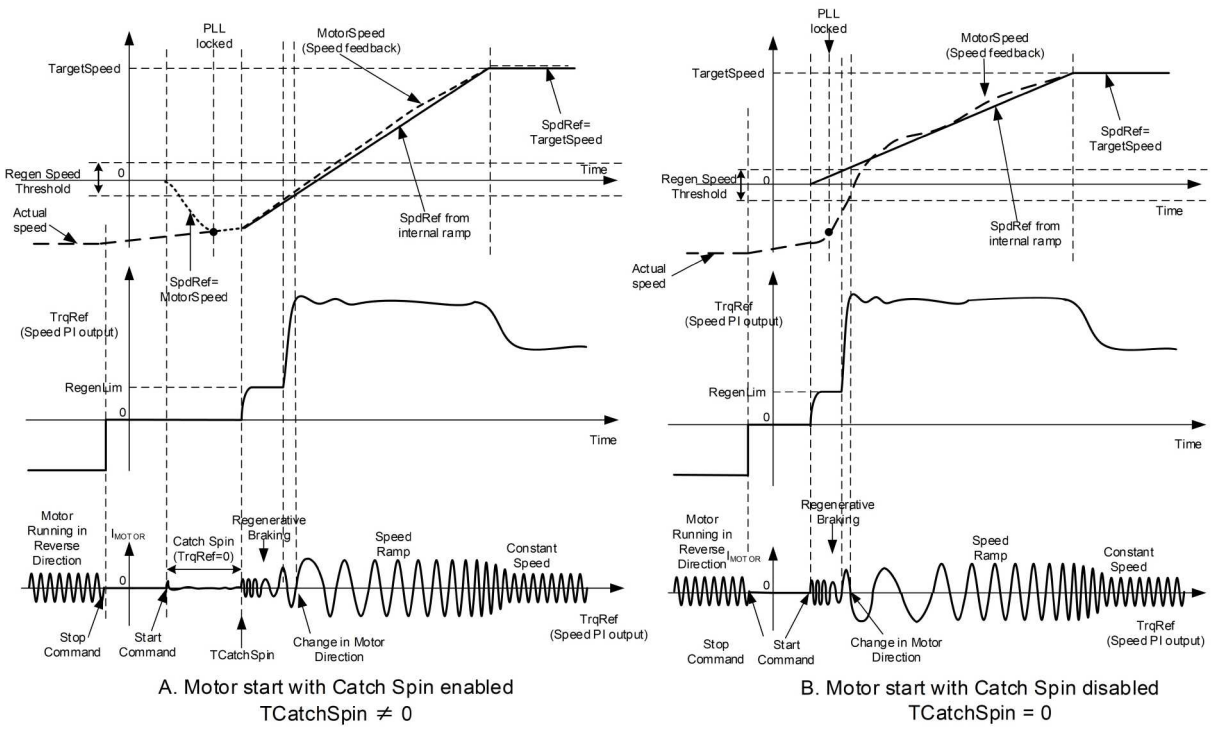


Figure 44 **Reverse Catch Spin - Motor start with/without catch spin**

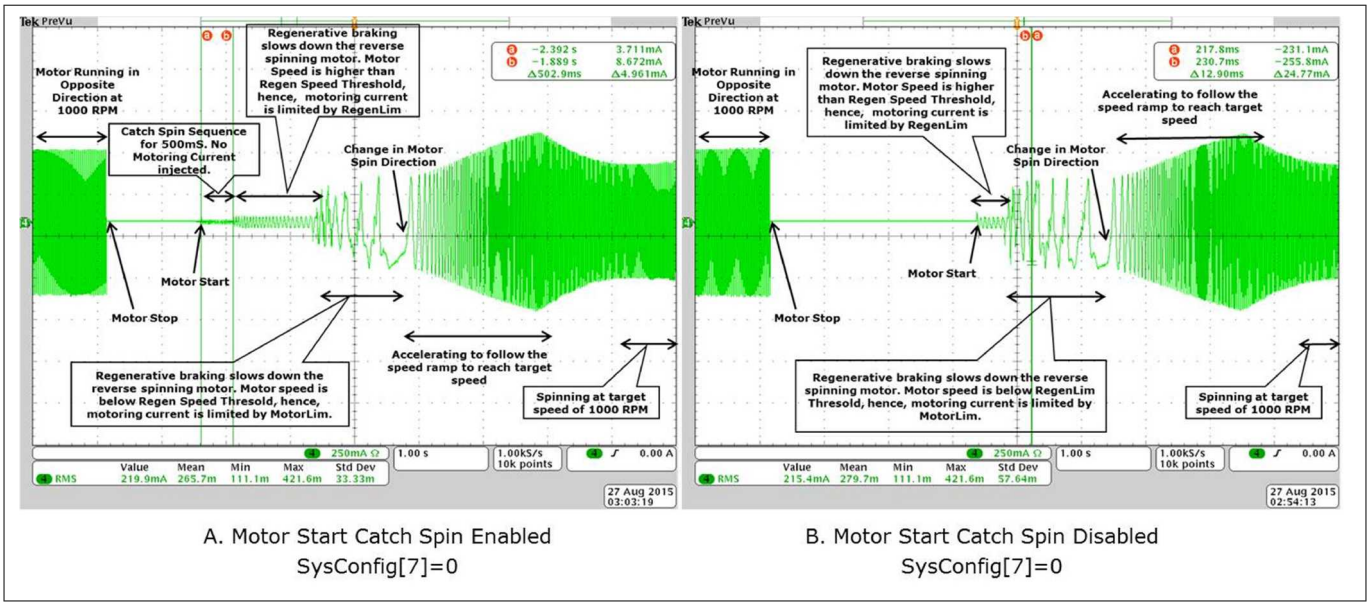


Figure 45 **Motor Phase Current Waveform - Reverse Catch Spin - Motor start with/without catch spin**

2.16 Control Input

MCE is able to control the motor from 4 types of inputs. Type of control input can be configured using Solution Designer.

- UART control
- Vsp analog input
- Frequency input
- Duty cycle input

2.16.1 UART control

In UART control mode, motor start, stop and speed change are controlled by UART commands. Target speed can be positive or negative; motor will spin in reverse direction if Target Speed is negative. If any fault condition happens, motor will stop and stay in fault status. It is up to master controller when to clear the fault and restart the motor.

2.16.2 Vsp Analog Input

In Vsp Analog Input control mode, the motor operations like motor start, motor stop and speed change are controlled by applying an analog voltage signal. Direction of the motor is controlled by a separate pin. If the direction pin is LOW, target speed will be set as positive and if the direction pin is HIGH, target speed will be set as negative value; motor will spin in reverse direction if target speed is negative. MCE uses “VSP” pin as the Vsp Analog input and uses “DIR” pin as motor direction input. The relationship between Vsp voltage and motor target speed is shown in Figure 46.

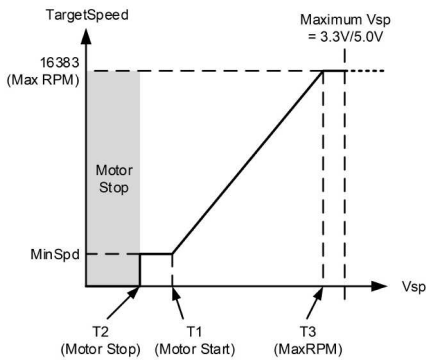


Figure 46 Vsp Analog Input

There are three input thresholds used to define the relationship between input voltage and target Speed.

- T1 (Input threshold for motor start): if the Vsp analog voltage is above this threshold, motor will start
- T2 (Input threshold for motor stop): if the Vsp analog voltage is below this threshold, motor will stop
- T3 (Input threshold for maximum RPM): if the Vsp analog voltage is higher or equal to this threshold, “TargetSpeed” variable will be 16383 which is maximum speed

Solution Designer uses these three input thresholds to calculate the value of three parameters: “CmdStart”, “CmdStop” and “CmdGain”

$$CmdStop = Integer\left\{\left(\frac{T2 \times 2}{V_{adcref}} \times 2048\right) + 0.5\right\}$$

Where T2 = Analog Vsp Motor Stop Voltage in V.

$$CmdStop = Integer\left\{\left(\frac{T1 \times 2}{V_{adcref}} \times 2048\right) + 0.5\right\}$$

Where T1 = Analog Vsp Motor Start Voltage in V.

$$CmdGain = Integer\left\{\left(\frac{Speed_{Max} - Speed_{Min}}{Speed_{Max}} \times 2^{12}\right) \times \left(\frac{2^{14}}{\left(\left(4096 \times 32 \times \frac{T3}{V_{adcref}}\right) - \left(CmdStart \times 32\right)\right)}\right) + 0.5\right\}$$

Where:

- T3 = Analog Vsp Motor Max RPM Voltage in V
- $Speed_{Max}$ = Maximum motor speed in RPM
- $Speed_{Min}$ = Minimum motor speed in RPM

Table 6 Specification for Analog Input Voltage

Recommended input range	Vsp Analog input (0.1 V to V_{adcref})
T1	< 50% of V_{adcref}
T2*	< 50% of V_{adcref}
T3**	< V_{adcref}

Note: ** T2 must be < T1 and **T3 must be > T2*

Refer data sheet for input range for specific devices and pin details. This feature is not available in UART control mode.

2.16.3 Frequency input

In Frequency Input control mode, the motor operations like motor start, motor stop and speed change are controlled by applying a square wave frequency signal on digital IO pin. Direction of the motor is controlled by a separate pin. If the direction pin is LOW, target speed will be set as positive and if the direction pin is HIGH, target speed will be set as negative value; motor will spin in reverse direction if target speed is negative. MCE uses “DUTYFREQ” pin as the frequency input and uses “DIR” pin as motor direction input. The relationship between Frequency and motor target speed is shown in Figure 47

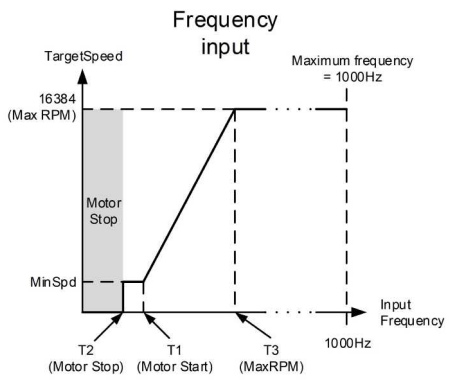


Figure 47 Frequency Input

There are three input thresholds used to define the relationship between frequency input and target Speed.

- T1 (Input threshold for motor start): if the frequency input is above this threshold, motor will start
- T2 (Input threshold for motor stop): if the frequency input is below this threshold, motor will stop
- T3 (Input threshold for maximum RPM): if the frequency input is higher or equal to this threshold, target Speed will be 16383 which is maximum speed

Solution Designer uses these three input thresholds to calculate the value of three parameters: “CmdStart”, “CmdStop” and “CmdGain”

$$CmdStop = Integer\{T2 \times 10 + 0.5\}$$

Where T2 = Motor Stop Speed Frequency in Hz.

$$CmdStop = Integer\{T1 \times 10 + 0.5\}$$

Where T1 = Motor Start Speed Frequency in Hz.

$$CmdGain = Integer\left\{\left(2^{12} \times \frac{\left(16384 - \left(\frac{Speed_{Min}}{Speed_{Max}} \times 16384\right)\right)}{(T3 - T1) \times 32 \times 10}\right) + 0.5\right\}$$

Where:

- T1 = Motor Start Speed Frequency in Hz
- T3 = Motor Maximum Speed Frequency in Hz
- $Speed_{Max}$ = Maximum motor speed in RPM
- $Speed_{Min}$ = Minimum motor speed in RPM

Table 7 Specification of Frequency Input

Recommended input range	Frequency input (5 Hz – 1000 Hz , 10% – 90% duty cycle)
T1	≤ 255 Hz
T2*	≤ 255 Hz
T3**	≤ 1000 Hz

***Note**: T2 must be < T1 and **T3 must be >T2*

Refer data sheet for input range for specific devices and pin details. This feature is not available in UART control mode.

2.16.4 Duty Cycle Input Control

In Duty Cycle Input control mode, the motor operations like motor start, motor stop and speed change are controlled by varying the duty cycle of a rectangular wave signal on digital IO pin. Direction of the motor is controlled by a separate pin. If the direction pin is LOW, target speed will be set as positive and if the direction pin is HIGH, target speed will be set as negative value; motor will spin in reverse direction if target speed is negative. MCE uses “DUTYFREQ” pin as the duty input and uses “DIR” pin as motor direction input. The relationship between duty cycle and motor target speed is shown in Figure 48.

In duty cycle control mode, the pre-scaler of capture timer has much wider range than frequency control mode. This allows higher input frequency in duty cycle control mode; the recommended input frequency range is 5 Hz to 20 kHz. Please note that any external R/C low pass filter on the input pin may affect the duty cycle measurement especially when the input frequency is above 1 kHz.

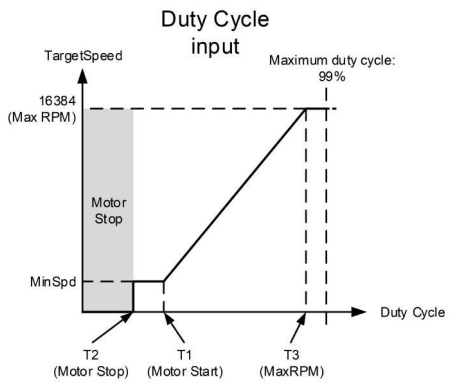


Figure 48 Duty Cycle Input

There are three input thresholds used to define the relationship between duty cycle input and target Speed.

- T1 (Input threshold for motor start): if the duty cycle input is above this threshold, motor will start
- T2 (Input threshold for motor stop): if the duty cycle input is below this threshold, motor will stop
- T3 (Input threshold for maximum RPM): if the input reaches or above this threshold, “TargetSpeed” variable will be 16383 which is maximum speed

Solution Designer uses these three input thresholds to calculate the value of three parameters: “CmdStart”, “CmdStop” and “CmdGain”

$$CmdStop = Integer\{T2 \times 10 + 0.5\}$$

Where T2 = Motor Stop Speed Duty Cycle in %.

$$CmdStop = Integer\{T1 \times 10 + 0.5\}$$

Where T1 = Motor Start Speed Duty Cycle in %.

$$CmdGain = Integer\left\{\left(\frac{Speed_{Max} - Speed_{Min}}{Speed_{Max}} \times 2^{12}\right) \times \left(\frac{2^{14}}{((T3 \times 10) - (CmdStart)) \times 32}\right) + 0.5\right\}$$

Where:

- T1 = Motor Start Speed Duty Cycle in %
- T3 = Motor Maximum Speed Duty Cycle in %
- SpeedMax = Maximum motor speed in RPM
- SpeedMin = Minimum motor speed in RPM

Solution Designer uses these three input thresholds to calculate the value of three parameters: “CmdStart”, “CmdStop” and “CmdGain”

Table 8 Specification of Duty Cycle Input

Recommended input range	Duty cycle input (5 Hz – 20 kHz, 1% – 99% duty cycle)
T1	< 50%
T2*	< 50%
T3**	≤ 99%

Note: ** T2 must be < T1 and **T3 must be > T2*

Refer data sheet for input range for specific devices and pin details. This feature is not available in UART control mode.

2.16.5 Automatic Restart

In Vsp, frequency or duty cycle control input mode, users have an option to specify retry times and intervals to restart the motor after any fault occurs and stops the motor. ‘FaultRetryNumber’ parameter configures the number of retry times after fault. A non-zero value of ‘FaultRetryNumber’ enables retry after fault. ‘FaultRetryPeriod’ parameter configures the retry interval.

This feature is not available in UART control mode.

2.16.6 Forced control input change

If required by some debug purpose, it is possible to change the control inputs by sending UART command from master controller (or PC), and then a new mode will be effective immediately. If the control input is switched to UART control from the other three inputs, motor status (run/stop and “TargetSpeed” variable) will be unchanged until it receives a new motor control command.

2.16.7 PG output

The MCE can output a pulse train (PG output) that represents the rotor postion. In case of Hall sensor/Hybrid mode, PG output will be enabled always irrespective of the motor state. In case of Sensorless mode, PG output will be enabled only in RUN state by default.

The 'PGDeltaAngle' parameter configures the PG output according to PGDeltaAngle = 256*(Motor poles)/PPR, where PPR is Pulses Per Revolution. For example, 4 PPR for an 8 poles motor (1 pulse per electrical cycle), then: PGDeltaAngle = 256 * 8/4 = 512. Writing 0 to PGDeltaAngle will disable the PG output.

PG output is updated every PWM cycle, so the maximum PG output frequency is ½ Fpwm. The maximum value for PGDeltaAngle is 16383, which means 1 PG pulse take 32 electrical cycle (16384/512 = 32), on an 8 poles motor, the PG output will be 0.125PPR.

If PGDeltaAngle is 2^n (2,4,8,16….8192,16384), PG pulse will be synchronized with rotor angle. For example, if PGDeltaAngle=512 for an 8 poles motor (4PPR). There are 4 PG pulses every 4 electrical cycles and the PG transition (high to low or low to high) will happen at 0 and 180 electrical degree.

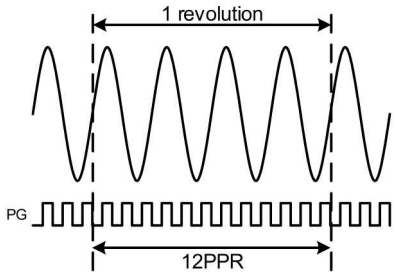


Figure 49 **PG Output**

2.16.8 Control Input Customization

By default, the relationship between the control input (VSP analog input/frequency input/duty cycle input) and the motor target speed is linear as shown in Figure 46, Figure 47, and Figure 48. If an application requires implementation of an arbitrary mapping relationship between the control input and the motor target speed, then one can choose to disable the default linear control input method and use script language to realize control input customization.

For VSP analog input control method, the analog input voltage can be read from' 'adc_result0' variable using script.

To enable frequency or duty input control customization, one needs to set the 6th bit of 'AppConfig' variable, so that the 'FrequencyInput' and 'DutyInput' variables get updated with the relevant frequency and duty cycle measurement results every 10 ms. Supported input frequency range: 5 Hz – 5000 Hz. Supported input duty cycle range: 1% - 99%.

For frequency input control method, the measured input frequency can be read from 'FrequencyInput' variable using script.

For duty cycle input control method, the measured input duty cycle can be read from 'DutyInput' variable using script.

2.17 Protection

Table 9 **List of Motor Control and Common Protection**

Type of Protection	Description	UL60730-1 Certification
Over Current (Gate kill)	This fault is set when there is over current and shutdown the PWM. This fault cannot be masked	Yes

(table continues...)

Table 9 (continued) List of Motor Control and Common Protection

Type of Protection	Description	UL60730-1 Certification
Critical Over Voltage	This fault is set when the DC voltage is above a threshold; all low side switches are clamped (zero-vector-braking) to protect the drive and brake the motor. The zero-vector is held until fault is cleared. This fault cannot be masked	No
DC Over Voltage	This fault is set when the DC Bus voltage is above a threshold	No
DC Under Voltage	This fault is set when the DC Bus voltage is below a threshold	No
Flux PLL Out of Control	This fault is set when motor flux PLL is not locked which could be due to wrong parameter configuration	Yes
Over Temperature	This fault is set when the temperature is above a threshold	No
Rotor Lock	This fault is set when the rotor is locked	Yes
Execution	This fault occurs if the CPU load is more than 95% or if background tasks are not executed at least once every 60s	Yes
Phase Loss	This fault is set if one or more motor phases are not connected	Yes
Parameter Load	This fault occurs when parameter block in flash is faulty	Yes
Link Break	This fault is set when there is no UART communication for a defined time limit	Yes
Hall Invalid	This fault is set when hall interface receives invalid Hall pattern	Yes
Hall Timeout	This fault is set when no Hall input transition is detected for a defined period of time. This fault is to detect rotor lock condition in Hall sensor/hybrid mode	Yes
Current Offset Calibration	This fault is checked in current OFFSET calibration state after offset measurement is completed. When this protection happens, system enters into fault state	No

2.17.1 Flux PLL Out-of-Control Protection

When the Flux PLL is locked to the correct rotor angle, Pll_M, which represent the flux of the permanent magnet of the motor, should be a DC value normalized at 2048 counts. Instead, if the PLL is not locked to correct rotor angle, Pll_M becomes either unstable or its value is far off from 2048 counts. Flux PLL out-of-control protection is the mechanism designed to detect this fault condition.

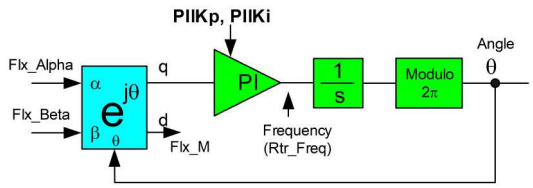


Figure 50 Simplified block diagram of a Flux PLL

The MCE keeps monitoring Pll_M, within certain time slot (configured by 'FluxFaultTime' parameter), if Pll_M value is below 512 or above 8192, and if this happens in 8 continuous time slots (each slot time is equal to FluxFaultTime/8), flux PLL is considered "out-of-control". See Figure 51 for details.

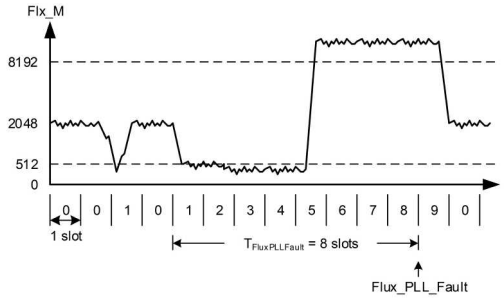


Figure 51 Flux PLL Out-of-Control Protection

If the Flux PLL out-of-control fault is confirmed, then it will be reported by setting the bit 4 in FaultFlags motor variable, and the motor speed loop gets reset. If the bit 4 in 'FaultEnable' motor dynamic parameter is set, then this fault will be reflected in 'SwFaults' motor variable and the motor state machine will shift to FAULT state causing the motor to stop running. If this bit is not set, then the corresponding bit in 'SwFaults' variable will be masked by 'FaultEnable' parameter, so that this fault will not be reflected in 'SwFaults' variable, and the motor state machine will not shift to FAULT state. This protection is also able to detect phase loss condition.

The PLL out-of-control fault response time can be configured by setting motor parameter 'FluxFaultTime'. The valid range of its value is from 0 to 65535. The value of 1 corresponds to 0.016 seconds. The default value is set to 500, which corresponds to a response time of 8 seconds.

2.17.2 Rotor Lock Protection

As shown in the following Figure 52, rotor lock fault is detected if the speed PI regulator output (variable 'TrqRef') is being saturated for a defined amount of time $T_{Rotor\ Lock}$. The rock lock detection time $T_{Rotor\ Lock}$ can be configured by using parameter 'RotorLocktime' following this equation $T_{Rotor\ Lock} = RotorLockTime \times 16ms$. Rotor lock protection is active when the motor speed ranges from min motor speed to 25% of maximum speed. Rotor lock protection becomes inactive when the motor speed goes beyond 25% of maximum speed to avoid erroneous fault reporting.

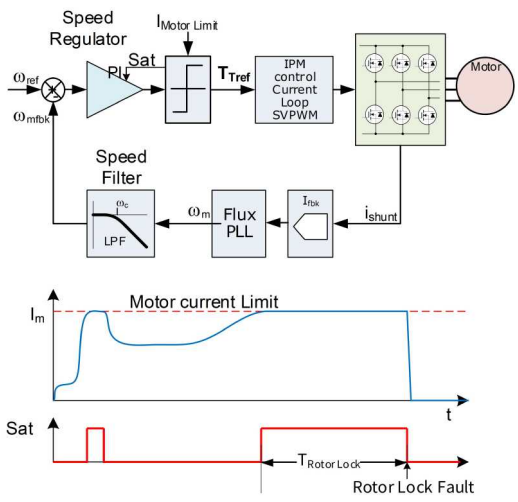


Figure 52 Rotor Rock Protection Mechanism Diagram

If the rotor lock fault is confirmed, then it will be reported by setting the bit 7 in FaultFlags motor variable. If the bit 7 in FaultEnable motor dynamic parameter is set, then this fault will be reflected in SwFaults motor variable, and the motor state machine will shift to FAULT state causing the motor to stop running. If this bit is not set, then the corresponding bit in SwFaults variable will be masked by FaultEnable parameter, so that this fault will not be reflected in SwFaults variable, and the motor state machine will not shift to FAULT state and the motor will keep running.

Please note if rotor lock detect time $T_{Rotor\ Lock}$ is set too short, it might trigger the fault during acceleration or momentary high load conditions.

Rotor lock detection is not 100% guaranteed to report the fault especially when the motor is running at low speed. The reason is, in rotor lock condition, the PLL might be locked at a false speed which may not cause speed PI output to be saturated.

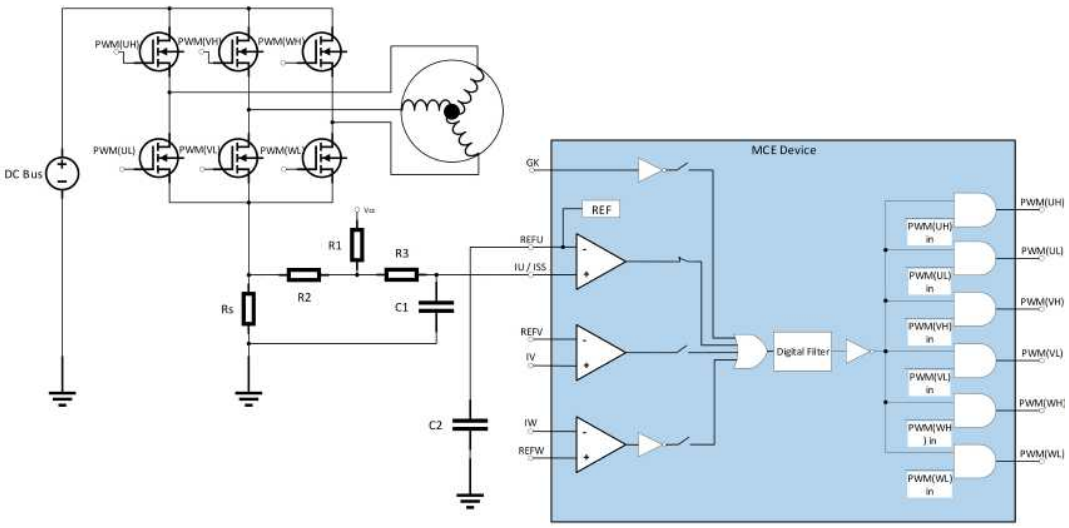
2.17.3 Motor Over Current Protection (OCP)

Motor gatekill fault is set during over current condition. This over current condition can be detected by the following two input sources.

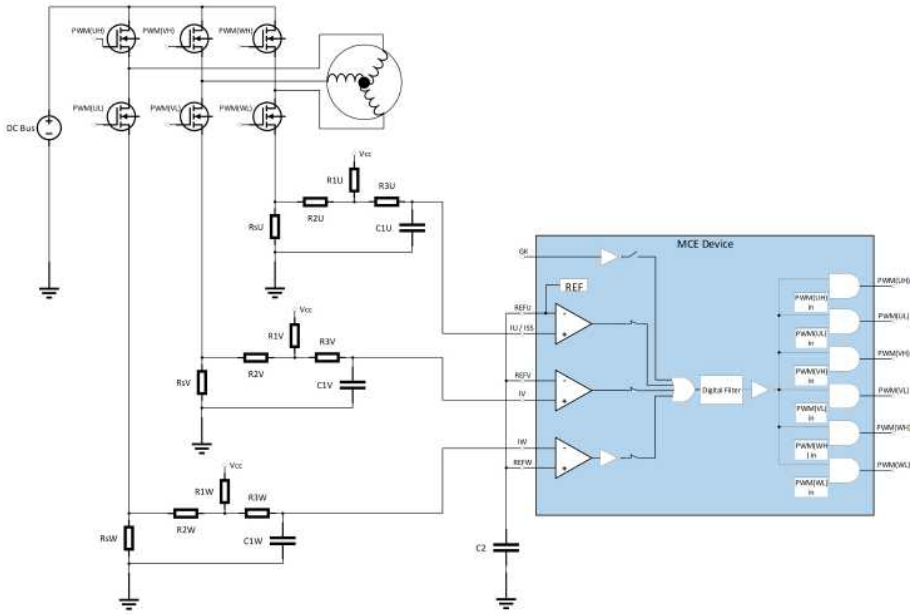
1. Direct GK pin: Gatekill fault is set if input is LOW
2. Internal comparators

It is possible to select either both or any one of the two sources for over current detection logic. Over current detection source can be selected by Solution Designer. Bit 0 in FaultFlag will be set in case of over current condition detected via any of the two sources. In case over current condition is detected via direct gate kill pin, bit 5 of FaultFlags will also set apart from bit 0 of FaultFlags.

If direct GK pin is included in the source of over current detection logic, the internal pull-up of IMC products is configurable by Solution Designer, it can be disabled or enabled, the default value is enabled. The internal pull-up can be configured to disabled when external pull-up is used. The internal pull-up must be enabled when there is no external pull-up. The internal pull-up of IMD, IMM and IMI is always enabled.



(1) Single Shunt Configuration



(2) Leg Shunt Configuration

Figure 53 Typical Motor OCP Implementation Using Internal Comparators

User can select using either the dedicated GK pin or the internal comparators to realize the over-current protection function. In the case of using the GK pin, it is configured to be active LOW. In the case of using the internal comparators, the exact tripping voltage level can be specified by setting the ‘CompRef’ motor parameter. The current tripping level for the internal comparator can be configured using Solution Designer, the ‘CompRef’ parameter holds the current trip level value. As shown in Figure 53, for single shunt current measurement configuration, only one internal comparator is used. For leg shunt current measurement configuration, three internal comparators are used to detect over current condition as shown in Figure 53.

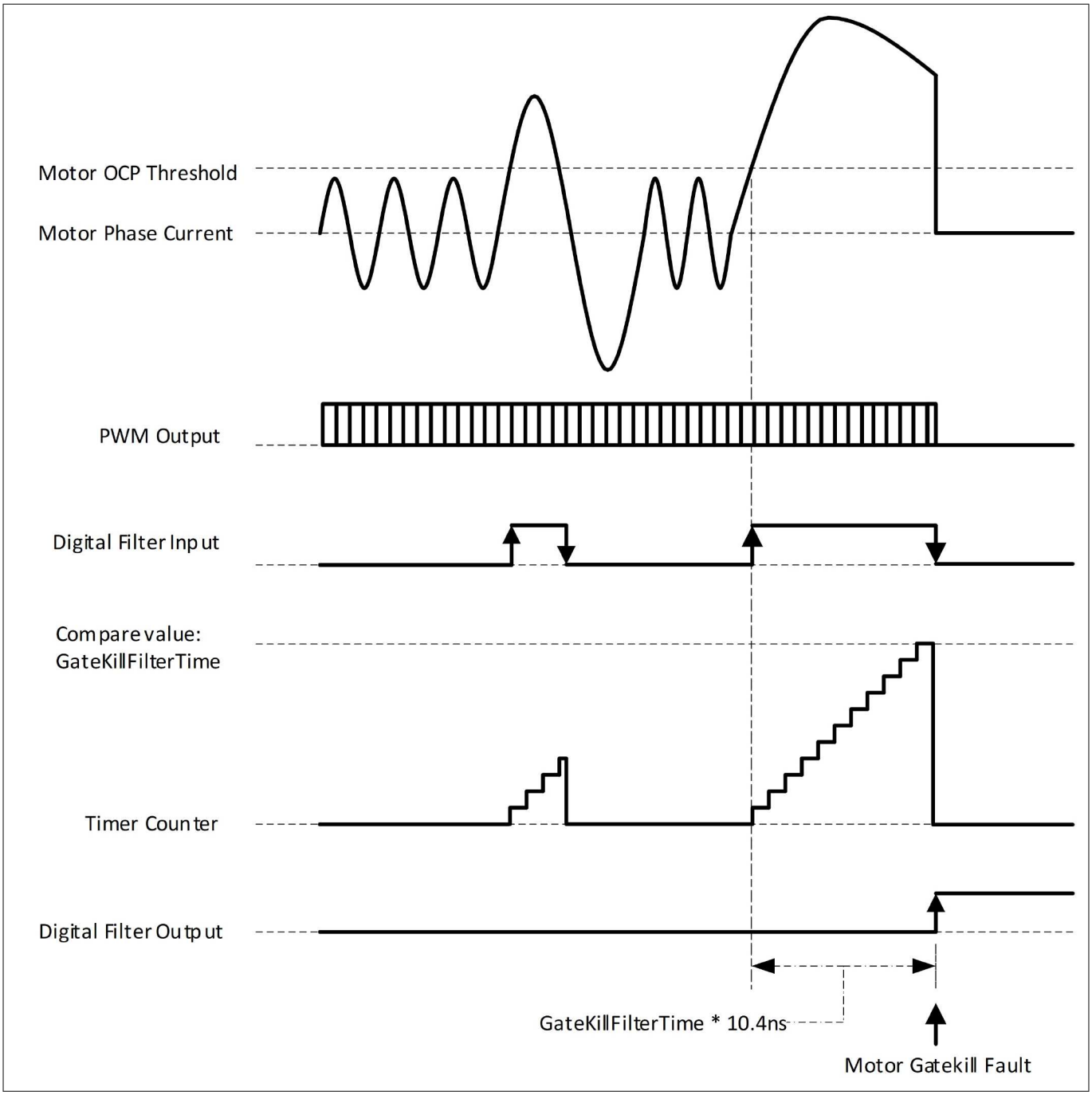


Figure 54 **Digital Filter Timing Diagram for Motor Gatekill Fault**

An internal configurable digital filter is used to de-bounce the input signal to prevent high frequency noise from mis-triggering a gate kill fault. “GatekillfilterTime” parameter holds the gate kill filter time value in clock cycles. Input signal needs to remain stable for the duration of the specified gate kill filter time to trigger the fault condition.

Gatekill filter timer is configured to be level triggered by the external GK pin or the internal comparator output. As shown in Figure 54, if the phase current goes beyond the specified OCP threshold, a timer in the digital filter starts counting up. If the digital filter input goes to logic LOW (external GK pin goes logic HIGH or the internal comparator output voltage level changes to logic LOW), then the timer gets reset. If the over-current condition is persistent when the timer counts up to ‘GateKillFilterTime’ value, then the digital filter output immediately goes to logic HIGH which forces entering Trap State upon which the PWM outputs all go to the programmed passive levels. The motor gatekill fault can only be cleared by writing 1 to the ‘FaultClear’ motor variable. This

fault cannot be masked, so that it will be reflected in SwFaults motor variable, and the motor state machine will shift to FAULT state, causing the motor to stop running.

GateKillFilterTime is a type of static motor parameter that specifies the gatekill response time for over-current fault detection. The valid range of its value is from 4 to 960 in clock cycles. The value of 1 corresponds to 1/96 MHz = 10.4167ns. The default value is 96, which is 1µs.

2.17.4 Over Temperature Protection

As shown in the following Figure 55, MCE provides an over-temperature protection (OTP) function with the help of an external NTC thermistor. Typically, the NTC thermistor and a pull-up resistor form a voltage divider. The MCE senses the output of the voltage divider and compares with a configurable OTP shutdown threshold $V_{Shutdown}$ that corresponds to the desired temperature $T_{Shutdown}$ where the system shall be shut down. If the output of the thermistor voltage divider is below $V_{Shutdown}$, then an OTP fault would be reported. The OTP shutdown threshold $V_{Shutdown}$ can be configured using the parameter 'Tshutdown'.

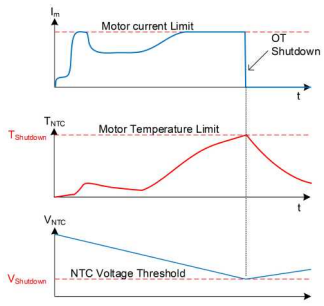


Figure 55 Over-Temperature Protection Mechanism Diagram

The action corresponding to the occurrence of over-temperature fault can be configured by use of the bit 6 in FaultEnable dynamic motor parameter. If this bit is set, then the motor state machine will go to FAULT state and the motor will stop running. If this bit is not set, then the motor state machine will not go to FAULT state and the motor will keep running.

2.17.5 DC Bus Over/Under Voltage Protection

Over/under voltage fault is detected when DC bus voltage goes above or below the relevant protection voltage threshold values.

DC bus voltage is being sampled every motor PWM cycle. The sampled DC bus voltage goes through a Low-Pass Filter to attenuate high-frequency noise, which can be read from the variable 'VdcFilt'. The time constant of the LPF depends on the motor control PWM frequency, and it can be calculated using the following equation:

$$T_{decay} = \frac{Fast_Control_Rate}{F_{PWM} \times Ln\left(\frac{2^{16}}{2^{16} - 2^{11}}\right)}$$

For example, if the motor control PWM frequency is 15 kHz, then the DC bus voltage sampling rate is 15 kHz. In that case, the time constant T_{decay} is about 2.1ms, and the cut-off frequency is about 76 Hz.

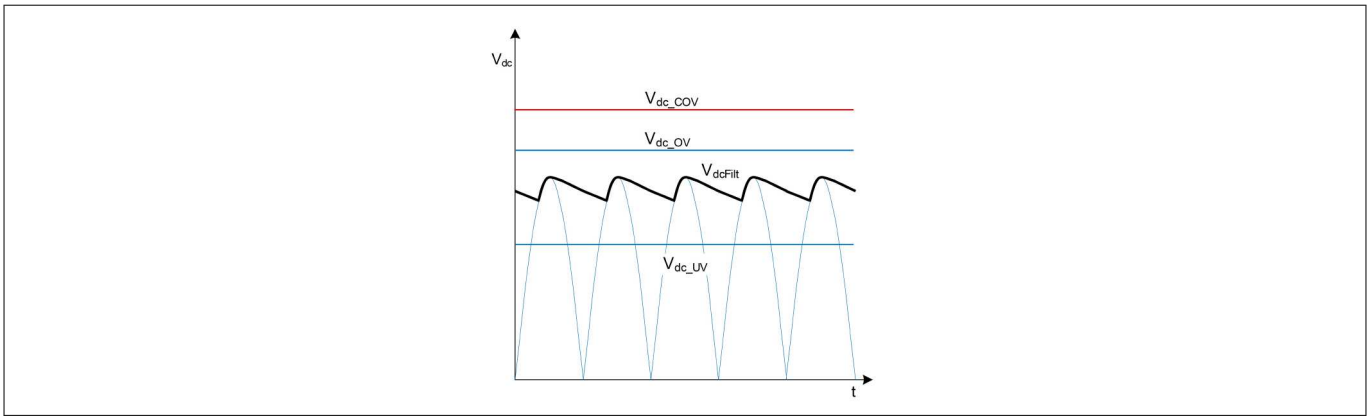


Figure 56 DC Bus Over/Under Voltage Protection Threshold Diagram

As shown in Figure 56, if the 'VdcFilt' value is greater than V_{dc_OV} (configured by the variable 'DcBusOvLevel'), then a corresponding bit 2 in FaultFlags motor variable is set. If the bit 2 in FaultEnable motor dynamic parameter is set, then this fault will be reflected in SwFaults motor variable, and the motor state machine will shift to FAULT state causing the motor to stop running. If this bit is not set, then the corresponding bit in SwFaults variable will be masked by FaultEnable parameter, so that this fault will not be reflected in SwFaults variable, and the motor state machine will not shift to FAULT state and the motor will keep running.

If the 'VdcFilt' value is lower than V_{dc_UV} (configured by the variable 'DcBusLvLevel'), then a corresponding bit 3 in FaultFlags motor variable is set. If the bit 3 in FaultEnable motor dynamic parameter is set, then this fault will be reflected in SwFaults motor variable, and the motor state machine will shift to FAULT state causing the motor to stop running. If this bit is not set, then the corresponding bit in SwFaults variable will be masked by FaultEnable parameter, so that this fault will not be reflected in SwFaults variable, and the motor state machine will not shift to FAULT state and the motor will keep running.

If the 'VdcFilt' value is above V_{dc_COV} (configured by the variable 'CriticalVdcOvLevel'), motor will be stopped immediately and zero vector [000] is applied until the fault is cleared, during which time 'critical over voltage' fault would be reported. This 'critical over voltage' fault cannot be disabled.

2.17.6 Phase Loss Protection

The MCE is capable of detecting motor phase loss fault. If one of the motor phases is disconnected, or the motor windings are shorted together, the parking currents will not have the correct value. If any of the phase current value is less than I_{phase_loss} at the end of PARKING state, then phase loss fault is confirmed.

The I_{phase_loss} can be configured by using the parameter 'PhaseLossLevel'. The default value of 'PhaseLossLevel' is automatically calculated by Solution Designer following this equation:

$$PhaseLossLevel = 25\,\% \times \frac{LowSpeedLim}{4096} \times I_{rated_rms} \times \sqrt{2} \times R_s \times G_{ext} \times G_{\text{int}} \times \frac{4096}{V_{ref_ADC}}$$

When phase loss fault is confirmed, if bit [8] of the parameter 'FaultEnable' is set, then this fault will be reflected in the variable 'SwFaults', and the motor state machine will shift to FAULT state causing the motor to stop running. If this bit is not set, then the corresponding bit in SwFaults variable will be masked by 'FaultEnable' parameter, so that this fault will not be reflected in 'SwFaults' variable, and the motor state machine will not shift to FAULT state and the motor will keep running.

2.17.7 Current Offset Calibration Protection

This protection function is executed in the OFFSET calibration state after the offset measurement is completed.

If any of measured current input offset values are not within specified limits, the currentoffset fault will be triggered and the system enters into fault state. The fault will be reported by setting bit 9 of FaultFlags parameter. The status flag is cleared when FaultClear is requested.

If bit [9] of the FaultEnable parameter is set, the fault will be reflected in SwFaults parameter and the motor state machine will change to FAULT state. If bit [9] of FaultEnable is not set, the corresponding bit of SwFaults will be masked and the fault will not be reflected in SwFaults. With the fault masked, the motor state machine will not change to FAULT state and move to STOP state regardless of the measured current offset.

The configurable limits CurrentOffsetMax and CurrentOffsetMin define the maximum and minimum levels of the measurmed current offset. The level is defined in ADC counts with 4095 representing the full ADC voltage reference value.

2.17.8 Staus LED

The MCE support a visual indication of motor- and PFC fault status through the LED pin. The fault status is encoded as shown in the table below. It is assumed the LED is lit when the LED pin is logic ‘low’.

Table 10 Status LED

Fault Status	LED pin encoding	Description
No Motor Fault and no PFC Fault	Logic ‘low’	LED is lit
Motor Fault	Toggle between logic ‘low’ and logic ‘high’ every 100 ms	LED is blinking fast
PFC Fault	Toggle between logic ‘low’ and logic ‘high’ every 1000 ms	LED is blinking slow
Motor Fault and PFC Fault	1 sec of toggle between logic ‘low’ and logic ‘high’ every 100 ms followed by 1 sec logic ‘low’.	LED is blinking fast for 1 sec followed by 1 sec lit

2.17.9 Execution Fault

Two conditions lead the MCE to detecting an Execution Fault. Firstly, the MCE monitors CPU execution load and if it exceeds 95%, Execution Fault becomes active. Secondly, if any of the background tasks (UART interface, Control input and script Task1) are not executed at least once every 60s, Execution Fault becomes active.

The MCE always reports Execution Fault by setting bit[10] of the parameter ‘FaultFlags’. Furthermore, if bit[10] of the parameter ‘FaultEnable’ is set, Execution Fault will be reflected in the parameter ‘SwFaults’ and the motor state machine will shift to FAULT state causing the motor to stop running when the fault is detected. If bit[10] of ‘FaultEnable’ is not set, the corresponding bit of ‘SwFaults’ will be masked and and the motor state machine will not shift to FAULT state.

The user can configure the controller to reset when any of the background tasks are not executed for 60s. To enable reset, set bit[8] (enable background task monitor reset) of the “SysTaskConfig” parameter.

The MCE supports Scripting based controller reset which makes it possible to reset the controller with timing different from 60s. Reset from scripting is performed by setting “Script_command = 0xAE51”. The reset command can be issued from both Task0 and Task1. However, since Task1 background function itself, it cannot be relied on to monitor execution background task. The reset command must be issued from Task0.

If Class B is enabled (MCEOS.SafetyFunctions = 52020), the evaluation period of background task execution interval is reduced to 1 second and if any of background tasks are not executed for 1s, the MCE enters to failsafe mode. If Class B is disabled (MCEOS.SafetyFunctions = 255) and BG monitoring is enabled (MCEOS. SysTaskConfig.BGtask_Monitoring_EN = 1, i.e bit 6 of “SysTaskConfig”), software reset is performed in case any of the background task is not executed at least once 60s period.

3 Power Factor Correction

Power Factor Correction (PFC) is a technique used to match the input current waveform to the input voltage, as required by government regulation in certain applications. The power factor, which varies from 0 to 1, is the ratio between the real power and apparent power in a load. A high power factor can reduce transmission losses and improve voltage regulation. The MCE supports full digital control and protection of a Continuous Conduction Mode (CCM) boost PFC using average current control scheme.

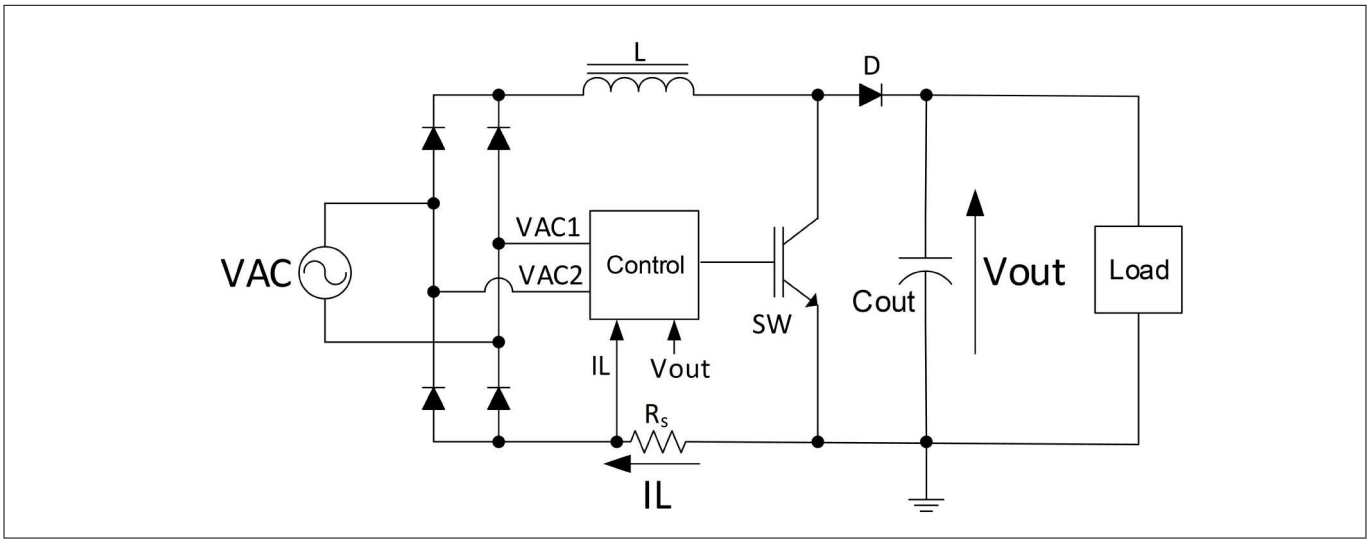


Figure 57 Boost PFC topology

Starting from the left-hand side of Figure 57, an AC-voltage (VAC) is rectified by a full bridge rectifier. The boost converter itself consists of an inductor (L), a diode (D), and a switch (SW). The output is filtered by a DC-capacitor (Cout) that smooths the output voltage (Vout). To measure the inductor current (IL), a resistive shunt (R_s) is placed typically in the return path of the input current. There are several other options for measuring current but, by putting the sensor in the return path, both the sensing circuit and the gate of the switch can be referenced to the same potential as the output voltage. The boost converter requires a controller to regulate the inductor current as well and the output voltage. As feedback the controller relies on the measurements of the inductor current, the output voltage, and the AC voltage.

3.1 PFC Algorithm

Closed-loop control ensures that the output voltage is kept at its desired value and that the AC current is sinusoidal and in phase with the AC voltage. The PFC control algorithm of the MCE is a multiplier-based average current control scheme, which means there are two control loops: an inner current loop and an outer voltage loop. In addition, there are feedforward terms which enhances dynamic response. The output of the voltage controller is multiplied by the instantaneous rectified AC voltage value and then divided by the square of AC voltage Root-Mean-Square (RMS) value to produce a reference for the current controller which in turn generates the duty-cycle command. This PFC control scheme requires sensing of the inductor current, AC line voltage and DC-bus voltage. With Figure 58 as reference, each of the main element of the MCE control algorithm will be described.

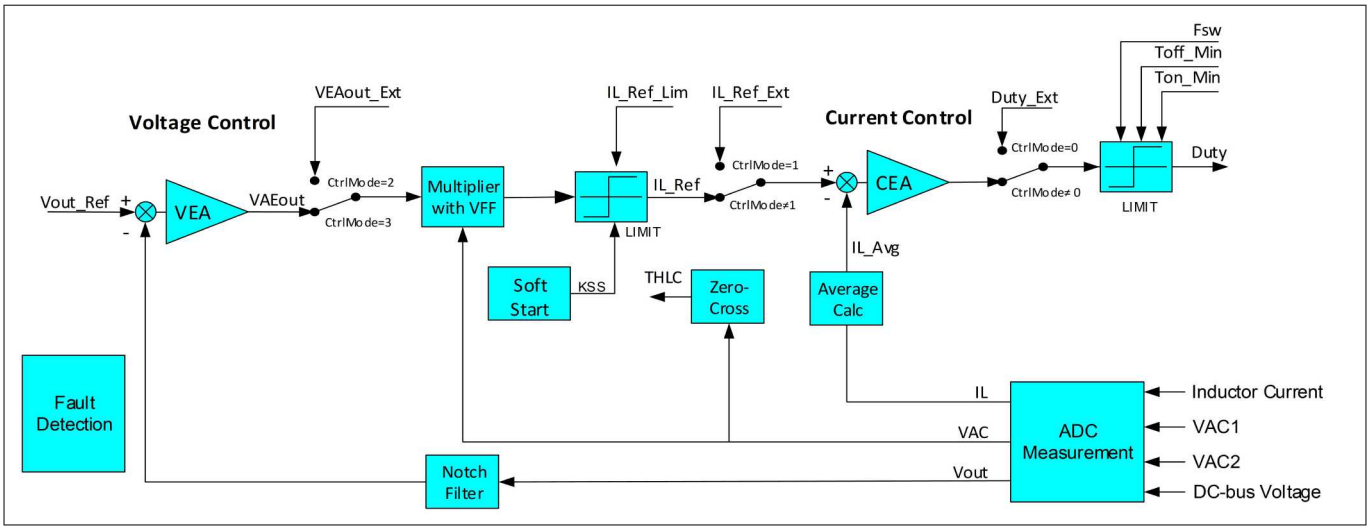


Figure 58 **Control system for the Boost PFC**

3.1.1 ADC Measurement

As feedback, the system in Figure 58 requires three measurements for close loop control:

1. The DC-bus voltage, Vout, to ensure that it is maintained at the reference level Vout_Ref
2. The AC voltage, VAC, to provide a sinusoidal shaped reference for the input current
3. The inductor current, IL, to ensure that it tracks the reference IL_Ref

These three signals are also used for over- and under voltage protection and for over current protection.

Vout is measured across the DC-link and with reference to power ground. VAC is measured in front of the rectifier and therefore not referenced to power ground. However, by measuring both the phase voltage, VAC1, and neutral voltage, VAC2, (see Figure 57) the actual AC voltage, VAC, is reconstructed as:

$$VAC = VAC1 - VAC2$$

All three signals are measured at the update rate of the current control loop (base rate). Scheduling and sample rates will be discussed in more detail in Chapter 3.3.

3.1.2 Current Control

The inner control loop ensures that the inductor current tracks the current reference, IL_Ref. The central element of the loop is a Current Error Amplifier (CEA) which calculates the duty-cycle command for the boost converter switch. As feedback, the loop relies on the inductor current averaged over a PWM switching period, IL_avg. The output of the CEA is fed to limiter that ensures minimum on/off times are observed.

The bandwidth of the current controller is determined by tuning but typically falls in the range of 3-9 kHz. Solution Designer calculates parameters for the current controller for optimized performance.

3.1.3 Average Current Calculation

Continuous Conduction Mode (CCM) is preferred for higher power boost PFC converters due to EMC and power component utilization. However, at low load the inductor current can become discontinuous during a portion of each half-line cycle and the boost converter enters what referred to as Discontinuous Conduction Mode (DCM). From a control point of view the two conduction modes are quite different and require separate handling. The algorithm in the MCE is optimized for CCM but it has additional features that enhances input current waveform during DCM (low load).

Figure 59 (A) shows the gate signal and corresponding inductor current during CCM. During the ON-time of the gate, T_{ON}, the inductor is charged by the input voltage and during the OFF-time, T_{OFF}, the charge is released to the DC-link. The PWM period, T_{PWM}, equals $T_{PWM} = T_{ON} + T_{OFF}$. At no point during the PWM period does the inductor current drop to 0A and a continuous current flows through the inductor.

Figure 59 (B) shows the gate signal and corresponding inductor current during DCM. As with CCM, the ON-time of the gate, T_{ON}, charges the inductor and during the OFF-time, T_{OFF}, the charge is released to the DC-link. The difference is that the inductor current drops all the way to 0 during the OFF-time, meaning there is period, T_{DIS}, where no current flows through the inductor. In other words, the current flow is discontinuous. T_{DIS} depends on a number of factors, such as input voltage, output voltage, duty-cycle, inductor size and switching frequency.

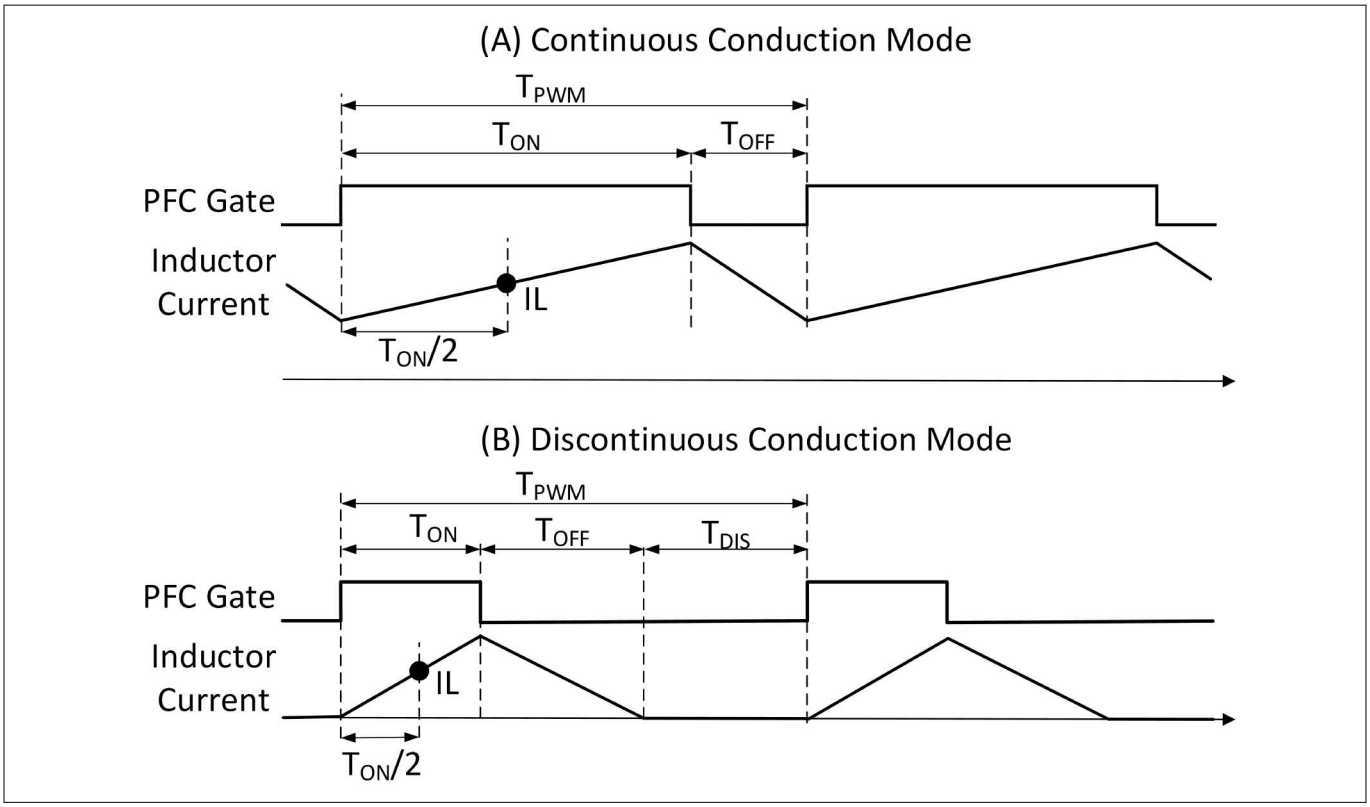


Figure 59 (A) Continuous Conduction Mode. (B) Discontinuous Conduction Mode

The digital PFC control algorithm requires a measurement of average inductor current once per control period. With CCM, the average current is easily obtained by sampling at the center of the ON-time, that is at $T_{ON}/2$. In Figure 59 this measurement is shown as the dot labeled IL. With DCM, extraction of the average current is more complicated as the duration of the non-conducting interval, T_{DIS}, has to be considered. The average inductor current can be expressed as a function of T_{ON}, T_{OFF} and T_{DIS}.

$$IL_Avg = \frac{IL \times (T_{ON} + T_{OFF})}{T_{ON} + T_{OFF} + T_{DIS}}$$

Where IL is the inductor current measured at the center of the ON-time. With Continuous Conduction Mode, $T_{DIS} = 0$ and IL_Avg = IL. With Discontinuous Conduction Mode $T_{DIS} > 0$, and IL_Avg ≠ IL.

Using IL as feedback in DCM will result in phase current distortion and reduced control performance. To properly control the current regardless of conduction mode, the average must be used as feedback for the loop. Measuring the average current is difficult in a digital system but estimation through calculation is possible.

The PFC algorithm in the MCE is capable determining the average current regardless of conduction mode. An estimator takes IL as input and, given operating conditions, calculates T_{DIS}. Based on T_{DIS} the average inductor current is estimated and then used a feedback for the current control loop.

It should be noted that the PFC algorithm in the MCE is optimized for CCM and should be used together with a converter designed for CCM. The purpose of the average estimator is to enhance performance during low load conditions where the converter enters DCM.

3.1.4 Voltage Control

The Voltage Error Amplifier (VEA) calculates the input current amplitude required to maintain the DC-bus voltage at the reference, Vout_Ref, under varying load and input voltage conditions. Feedback for the loop is a notch filtered version of the actual DC-bus voltage. Output from the compensator, VEAout, expresses the desired magnitude of the input current and is passed on to the multiplier where it is shaped according to the input voltage waveform, Chapter 3.1.5.

The outer loop is much slower than the inner loop with a typically the bandwidth of a few 10s-of-Hertz. Update rate of the loop is configurable but the loop runs at a sub-rate (Primary Rate) of the inner current loop (Base Rate)

3.1.5 Multiplier with Voltage Feed-Forward

The Multiplier has two purposes. First, it has to shape the current reference so it resembles the waveform of the input voltage. Second, it has to ensure constant voltage control loop gain during all operating conditions, commonly referred to as voltage feed-forward or VFF. The multiplier used in the MCE algorithm is shown in Figure 60.

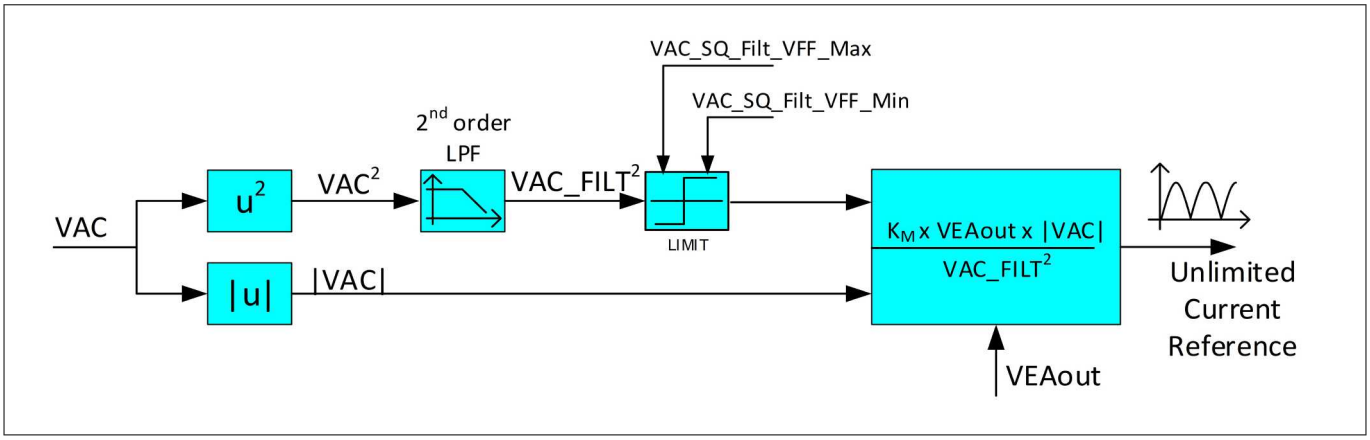


Figure 60 Reference Multiplier with Voltage Feed-Forward

The inputs to the multiplier are the absolute value of instantaneous input voltage, |VAC|, and a 2nd ordered low-pass filtered version of the squared input voltage, VAC_FILT^2. |VAC| shapes the current reference to be proportional to the input voltage after the rectifier, hence ensuring unity power factor. VAC_FILT^2 is representing the squared RMS of the input voltage.

The overall gain of the voltage loop is proportional to the square of the input RMS voltage. That means the loop has a higher gain at high AC input voltages than it does at low AC input voltages. Since a boost PFC typically operates over a wide AC input voltage range, it is impossible to design one controller that operates well under all conditions. If the controller is optimized for high voltage operation it becomes sluggish at low voltage. Even worse, if the controller is optimized for operation at low voltage, it could become unstable at high voltage.

The MCE algorithm ensures constant voltage loop gain by normalizing the output of the voltage controller, VEAout, with the inverse of the RMS input voltage squared, see Figure 60. This make the control loop independent of the input voltage throughout the universal input range. The factor K_M is calculated by Solution Designer to ensure the current reference can reach the peak current needed to deliver the specified maximum power at minimum AC input voltage when the VEAout reaches maximum limit.

The multiplier uses a lowpass filtered version of the squared AC line voltage to represent the squared RMS value of the AC line voltage. The filter is designed to attenuate the component at twice the AC line frequency. However, with a finite attenuation some AC content will remain at twice the AC line frequency and this ripple couples through multiplier and ends up modulating onto the current reference as a 2nd order harmonic distortion. The current controller can easily track a 2nd order component so the distortion ends up in the actual input current as a 3rd order harmonic. Solution Designer helps the user designing the filter based on acceptable 3rd harmonic input current requirement.

VAC_FILT^2 gets limited if it falls outside the configurable parameters VAC_SQ_Filt_VFF_Min and VAC_SQ_Filt_VFF_Max. This effectively disables voltage feedforward when operating beyond these limits.

The output of the Multiplier represents the unlimited inductor current reference. The unlimited reference is passed on to a limiter that ensures the current reference never exceeds IL_Ref_Lim.

3.1.6 Notch Filter

With both the input voltage and current being sinusoidal, the power drawn from the grid has a squared sinusoidal waveform pulsating at twice the grid frequency. For example, at 50 Hz supply the power will pulsate at 100 Hz. The job of the DC-link capacitor is to filter out this pulsating component so the load sees an output voltage close to an ideal DC. However, due to cost and physical size constraints it is not possible to fully eliminate the DC-bus voltage ripple. The result is that any one-phase boost PFC will have a DC-bus voltage ripple alternating at twice the grid frequency.

Typically, the DC-bus ripple does not have major negative effect on the load. However, voltage ripple couples through the outer voltage control loop and ends up modulating the current reference amplitude as a second order harmonic distortion. The current controller can easily track a second order component, so the distortion ends up in the actual input current.

The two feedback loops of the PFC boost have somewhat conflicting objectives. A fast outer loop gives good performance in terms of disturbance rejection and stabilizes the output voltage under all operating conditions. However, a strongly tuned voltage loop will deteriorate the power factor by commanding an input current that ensures a fixed output voltage rather than the desired sinusoidal-shaped current. To limit the distortion of the input current reference, the traditional approach is to reduce the control loop gain at the second harmonic frequency. This approach attenuates the voltage ripple coupled through the loop, but it is undesirable in terms of dynamic control performance.

The PFC algorithm in the MCE, solves the problem of second harmonic distortion caused by the voltage control loop by introducing a second order notch-filter in the feedback path of the voltage loop, see Figure 58. The filter is tuned to have high attenuation (notch) at twice the grid frequency, and therefore removes the voltage ripple from the feedback signal, while leaving all other frequencies unaltered. With the voltage ripple removed from the feedback, detuning of the voltage loop gain/bandwidth to avoid current distortion, is no longer needed.

Figure 61 shows the magnitude plot of the notch filter implemented in the MCE when tuned for a 100 Hz center frequency (notch) which is suitable for a 50 Hz input supply. The bandwidth and the attenuation of the notch filter are configurable. In this example, the filter was designed for a 20 Hz width of notch (-3 dB to -3 dB) and the attenuation at the notch is designed for -100 dB. The filter is updated at the primary rate, which in this example is 2500 Hz. The notch filter is fully tuned and parametrized by Solution Designer.

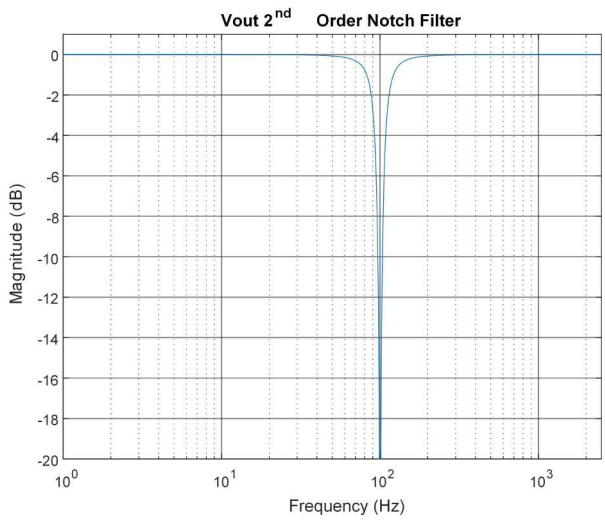


Figure 61 **2nd order Notch Filter tuned to remove 100 Hz ripple**

3.1.7 Zero-Cross Detection

The PFC algorithm relies on information about the line frequency and half line cycle period, THLC. Both of these values are determined by measuring the time between zero-crossings of the line voltage as illustrated in Figure 62. The top part of the figure shows the line voltage, VAC, along with an AC Polarity signal. The AC Polarity signal indicates whether VAC is in a positive- or negative half cycle of the voltage and it changes state at every zero-crossing. The period of the half line cycle, THLC, is time between zero-crossings of the line voltage.

In Figure 62, a positive-to-negative and a negative-to-positive going zero-crossing are highlighted by dashed eclipses and close-up views are shown in the bottom half of the figure. Stating with the positive-to-negative zero-cross detection (bottom left), the first step is to determine when VAC is less than the threshold ZCD_Step1_Thr but greater than 0V. If the voltage stays within this range for a deglitch time of ZCD_Step1_Deglitch_Time, the detection proceeds to the second step. If the voltage fails to stay within the thresholds throughout the deglitch window, the detection starts over from the beginning.

Second step of the detection is to validate the zero-crossing from first step. For a successful completion of the second step, VAC must stay below the threshold -ZCD_Step2_Thr for a deglitch window with a duration of ZCD_Step2_Deglitch_Time. When that happens, a new-zero crossing has been detected and the half line cycle time gets updated based on the elapsed time since last zero-crossing. If the voltage fails to stay below the threshold throughout the deglitch window, the detection goes back to the beginning of the first step.

Similarly, with the negative-to-positive zero-cross detection, the first step is to determine when VAC is greater than the threshold -ZCD_Step1_Thr but less than 0V. If the voltage stays within this range for a deglitch time of ZCD_Step1_Deglitch_Time, the detection proceeds to the second step. If the voltage fails to stay within the thresholds throughout the deglitch window, the detection starts over from the beginning.

For a successful completion of the second step, VAC must stay above the threshold ZCD_Step2_Thr for a deglitch window with a duration of ZCD_Step2_Deglitch_Time. If the voltage fails to stay above the threshold throughout the deglitch window, the detection goes back to the beginning of the first step.

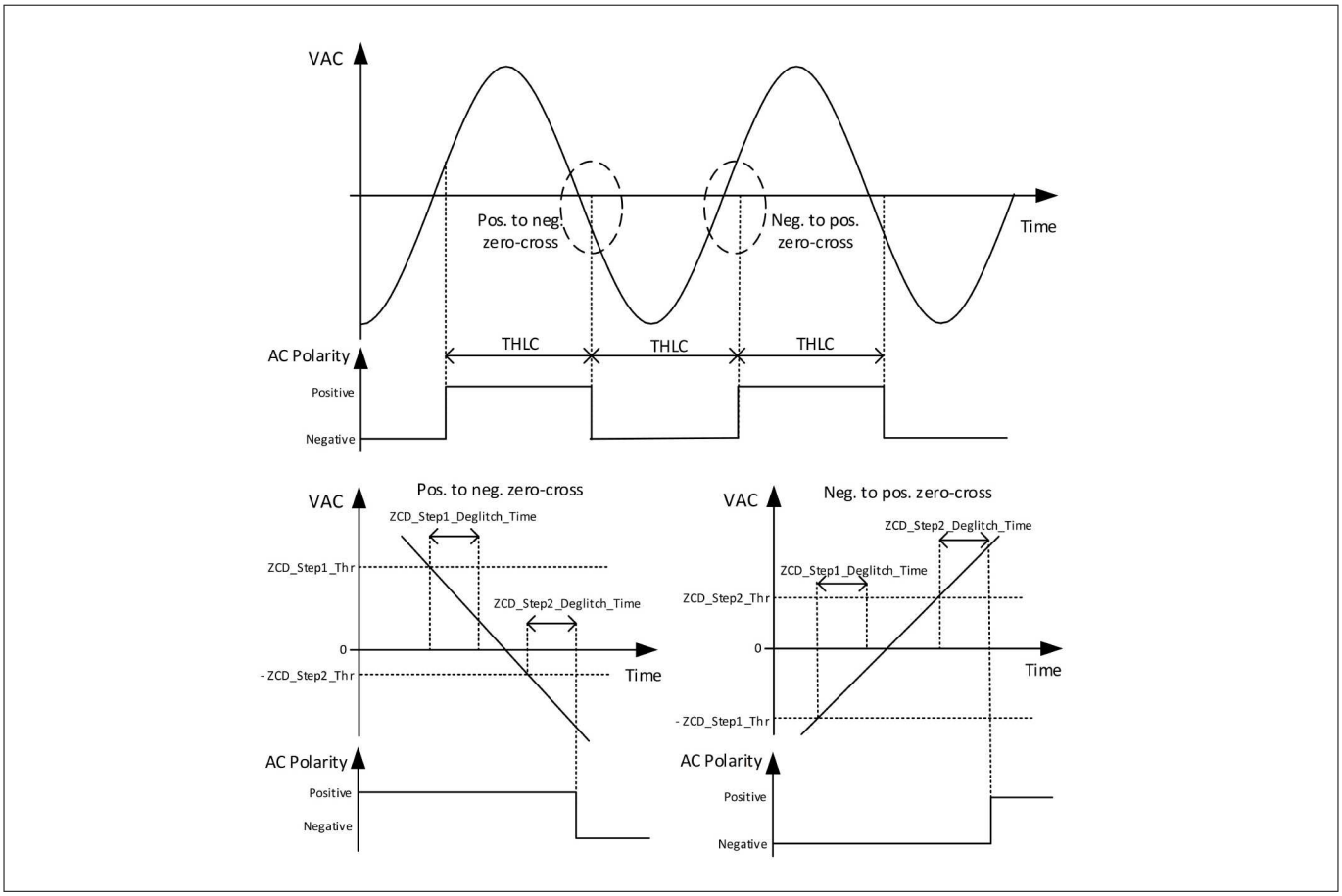


Figure 62 Zero-cross detection of the line voltage

In addition to the described thresholds and deglitch windows, there is a timeout on step 2. After completion of step 1, the voltage must drop below -ZCD_Step2_Thr within a time of ZCD_Step2_Check_Time to avoid time-out. In case of a time-out, the detection starts over from the beginningZCD_Step2_Check_Time.

If the detection algorithm fails to find a valid zero-cross within the configurable time ZCDTimeout_Thr, the parameter ZCDTimeoutFlag is set to 1 to indicate the system is supplied by a DC source. If a zero-cross is detected within ZCDTimeout_Thr, the parameter ZCDTimeoutFlag is set to 0 to indicate the system is supplied by an AC source.

3.1.8 Soft Start

At startup, there is typically a big difference between the actual DC-bus voltage and the requested voltage, Vout_Ref. That results in a large control error which can lead to DC-bus overshoot when starting up with no load or light load thanks to low bandwidth of the voltage control loop. To avoid this the MCE has a Soft Start feature that gently charges the DC-bus capacitor by gradually increasing a scaling factor, KSS, for the inductor current limit, IL_Ref_Lim. Soft start is complete when the current reference reaches 100%.

Soft Start ramps the DC-bus voltage up at every start of the PFC, including when PFC operation has been interrupted by a fault. The actually state of the soft start sequence can be read from bitfield SSStatus in parameter PFCStatus.

3.1.9 Vout Ready Monitor

The MCE has Vout Ready monitor function that checks the DC-bus voltage against a configurable threshold. One possible use case for this check is during sequencing of PFC- and motor startup where the monitor function can be used to determine a safe time to start the motor.

The working principle of the Vout Ready monitor function is illustrated in Figure 63 . When the DC-bus voltage exceeds Vout_Ready_Thr, and remains higher than the threshold during a deglitch window of length Vout_Ready_Deglitch_Time, bit 8 of PFCStatus is set. If the voltage drops below the threshold during the deglitch window, the status bit is not set. The status bit is cleared if voltage drops below Vout_Ready_Thr minus a hysteresis, Vout_Ready_Hyst, and stays below the threshold during a deglitch window with a length of Vout_Norm_Deglitch_Time.

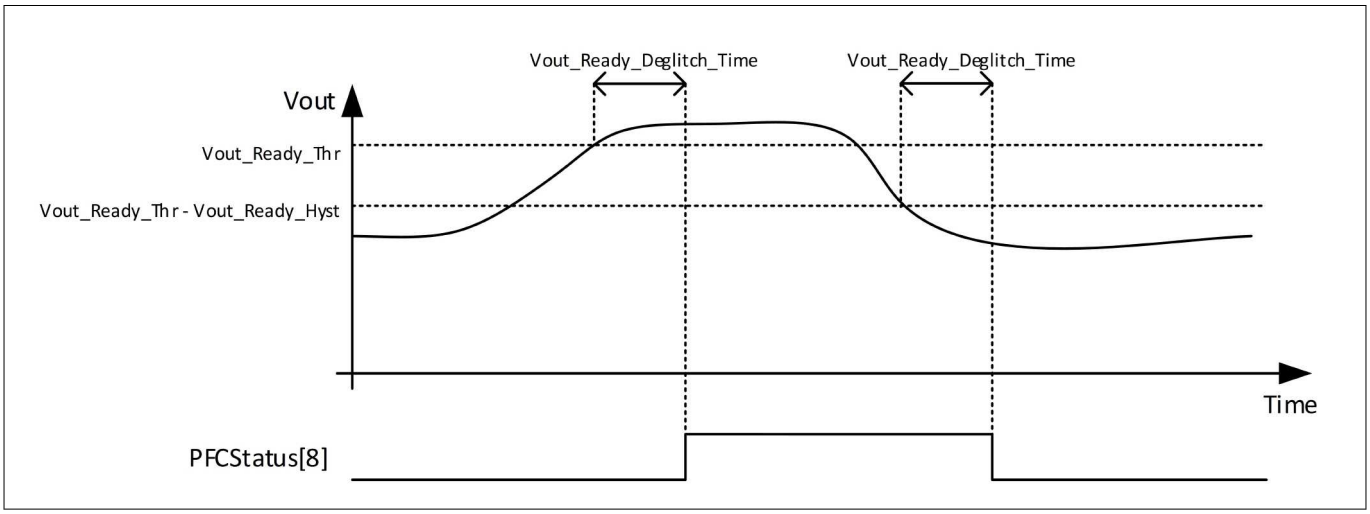


Figure 63 Vout Ready Check

3.1.10 Control Modes

The PFC system offers manual control modes that let the user overwrite parts of the closed-loop control system. In Figure 58 the control modes are symbolized by switches whose positions are determined by setting of the bitfield CtrlMode of parameter SysConfig. The supported modes are:

Table 11 Control Modes

Control Mode	Function
0	Open loop current control and open loop voltage control. External input, Duty_Ext, sets the PFC duty cycle
1	Closed loop current control and open loop voltage control. External input, IL_Ref_Ext, sets the PFC current reference
2	Closed loop current control and open loop voltage control but with multiplier enabled. External input, VEAout_Ext, sets the PFC voltage error amplifier output
3	Closed loop current control and closed loop voltage control with multiplier enabled

Normal PFC operation happens with ControlMode = 3. When ControlMode = 0-2 is selected, it is the user's responsibility to set appropriate external references.

3.2 State Handling

The Motion Control Engine includes a built-in state machine which manages sequencing of the PFC. The state machine is updated at the Sequencer Rate (1kHz). Current state of sequencer is stored in PFC_SequencerState parameter. The states and transitions are listed inTable 12 and illustrated in Figure 64.

Table 12 State Description and Transition

Sequence State	PFC_SeqeuncerState	State Functionality	Condition for Next State
PFC_IDLE	0	After power-up, a PFC reset (Command=2) or end of PFC Standby, the control enters this state. Setup and configuration of the PFC	Valid parameter set
PFC_OFFSETCAL	1	Offset calculation for the PFC current measurement channel. This state takes $2^{OffsetCalTotalTime}$ PWM cycles to complete	If the offset falls within min/max levels, proceed to a RUN_CTRLMODEx state. If not, proceed to PFC_FAULT
PFC_FAULT	5	Fault state if current measurement offset check failed or if execution fault is detected	Once entered, the PFC stays in this state until PFC is reset either through power cycling or by setting Command = 2
RUN_CTRLMODE0	2	Run mode with external duty-cycle reference. Control is either waiting for an enable command (Command = 1), applying PWM or shut down by a fault	Once entered, the PFC can only leave this state in case of an execution fault or a PFC reset (Command = 2)
RUN_CTRLMODE1	3	Run mode with external current reference. Control is either waiting for an enable command (Command=1), applying PWM or shut down by a fault	Once entered, the PFC can only leave this state in case of an execution fault or a PFC reset (Command = 2)
RUN_CTRLMODE2	4	Run mode with external multiplier reference. Control is either waiting for an enable command (Command = 1), applying PWM or shut down by a fault	Once entered, the PFC can only leave this state in case of an execution fault or a PFC reset (Command = 2)
RUN_CTRLMODE3	6	Normal PFC run mode with full close loop control. Control is either waiting for an enable command (Command = 1), applying PWM or shut-down by a fault	Once entered, the PFC can only leave this state in case of an execution fault or a PFC reset (Command = 2)

(table continues...)

Table 12 (continued) State Description and Transition

Sequence State	PFC_SeqeuncerState	State Functionality	Condition for Next State
PFC_STANDBY	7	The MCE lowers standby power consumption by reducing the CPU clock and by switching off some of the controller's peripherals. To enter STANDBY the PFC must be in RUN_CTRLMODE3, no PFC fault is present, and PFC PWM disabled. Motor is in STOP state with Zero-Vector-Braking disabled. In addition, a configured delay time must expire before entering STANDBY	A motor- or PFC start command or a fault

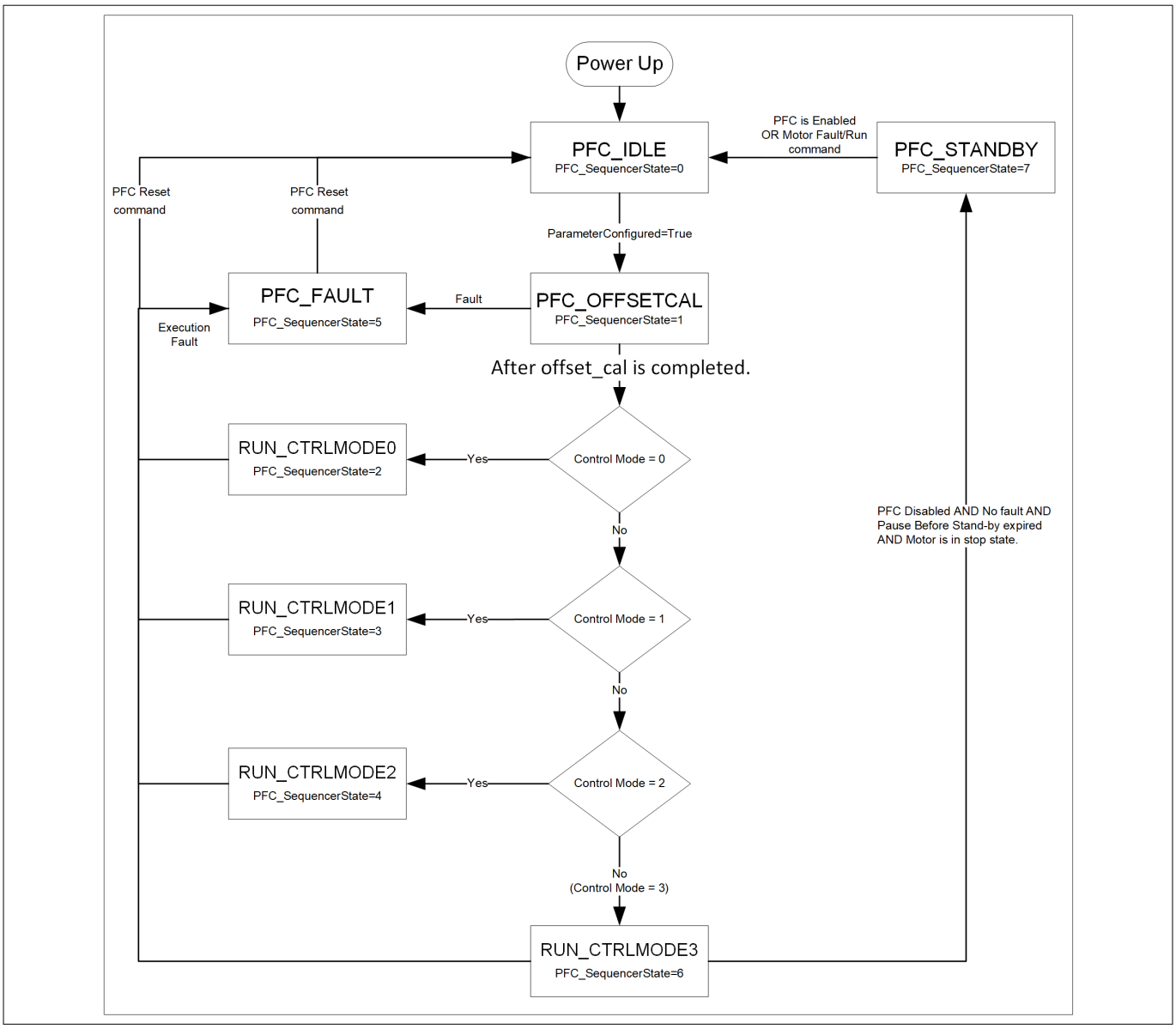


Figure 64 State handling of the PFC

Once a RUN_CTRLMODEx (x = 0, 1, 2 or 3) state has been entered, the system stays in that state unless one of 3 exceptions occur. (1) If an execution fault is detected, and fault is enabled, the system enters PFC_FAULT. (2) If in RUN_CTRLMODE3 and the conditions for low-power standby are met, the system enters PFC_STANDBY. (3) If the system is reset (Command=2) the system transitions to PFC_IDLE. PFC operation can be enabled/disabled by setting of the Command parameter but the system stays in the RUN_CTRLMODEx state regardless of the command. In case of a fault (excluding current offset fault and execution fault), the PFC gate operation is shut down until the fault clears but the system remains in the RUN_CTRLMODEx state throughout the fault condition. In case of current offset fault or execution fault, the PFC enters PFC_FAULT state and stays in that state until the power cycles or the PFC is reset (Command = 2).

In state RUN_CTRLMODE3 the system can transition to PFC_STANDBY if the PFC is disabled and a configured delay time has expired. In standby the MCE lowers power consumption by reducing the CPU clock and by switching off some of the controller's peripherals. PFC_STANDBY is terminated when a motor start (Command =1) or PFC reset (Command=2) is received, or a motor fault occurs. PFC_STANDBY is left through the PFC_IDLE state and followed by an offset calibration before normal PFC operation is resumed.

3.3 Scheduling and Timing

The time constants involved in the control of a Boost PFC varies greatly. For best control performance, and to minimize execution load, it is beneficial to split the algorithm into sub systems based on time constants and execute those subsystems at different rates. The PFC algorithm in the MCE is updated at 4 different rates as listed in the table below:

Table 13 Execution Rates

Rate Name	Execution Events	Update Rate
PWM	Switching frequency of PFC gate	Configurable. Typical 20-100 kHz
Base	Measurement System Reference Multiplier Current Controller PWM duty-cycle update VAC zero-cross detection	Configurable. Sub-rate of PWM Rate. Possible ratios are 1:1, 1:2 and 1:3 Typical 20-60 kHz
Primary	Voltage Controller Current Reference Limiter Notch filter Soft Start Feed-Forward voltage calculation Vout UV/OV update PFC Status update	Configurable. Sub-rate of Base Rate. Typical 1-10 kHz
Sequencer	Preparation of Power calculation Preparation of VAC RMS calculation Preparation of IAC RMS calculation VAC Drop-out update VAC UV/OV update AC Frequency validation Vout Ready update Overcurrent trip status update	Fixed at 1 kHz

The current control loop is executed at the Base Rate and it is the most time critical part of the system. As mentioned, the Base Rate is derived as a sub rate of the PWM rate. When the processor load allows, it is preferred to use a 1:1 ratio between the PWM- and Base Rate, meaning the PWM duty-cycle is updated every PWM cycle. In systems where a high PWM rate is required, say 100 kHz, a 1:1 ratio is not allowed due to the execution load of the MCE. For these high PWM rate systems, the MCE supports a 1:2 as well as a 1:3 ratio between the PWM rate and Base Rate.

Base- and primary rate are set by parameters FastControlRate and PrimaryControlLoop.

3.3.1 1:1 Base Rate

If the PWM rate allows the current controller to be updated every PWM cycle, a 1:1 ratio between the PWM rate and Base rate is preferred. In this case $T_{BaseRate} = T_{PWM}$. A timing diagram of this operating mode is shown in Figure 65.

From the top down, the figure shows the PFC Gate signal which is defined by its on-time, T_{ON}, its off-time, T_{OFF}, and its PWM switching period, $T_{PWM} = T_{ON} + T_{OFF}$. During the on-time the Inductor Current, IL, increases and during the off-time it decreases as the stored energy is release to the DC-link. Assuming Continuous Conduction

Mode, the Inductor Current assumes its average value at the center of the on-time. Under ideal conditions, the center of the on-time is the correct instant to sample the current but the system has delays, such as gate driver propagation delay and measurement channel delay. To compensate for the delays, the ADC Trigger is offset by $T_{SHDelay}$. In addition to the inductor current IL, the AC voltages, VAC1/VAC2, and the DC-bus voltage, Vout are also measured by the ADC. The inductor current is sampled during the gate on-time and the 3 voltages are sequentially sampled during the gate off-time. Each measurement consists of a sample-and-hold stage and a conversion state. The sample-and-hold stage takes T_{SH}= 333 ns to complete and the conversion takes T_{conv}= 767 ns to complete.

When the ADC is finished converting IL, the current loop is updated based on the newly acquired feedback. The result is a new duty-cycle which is applied at beginning of the next PWM cycle.

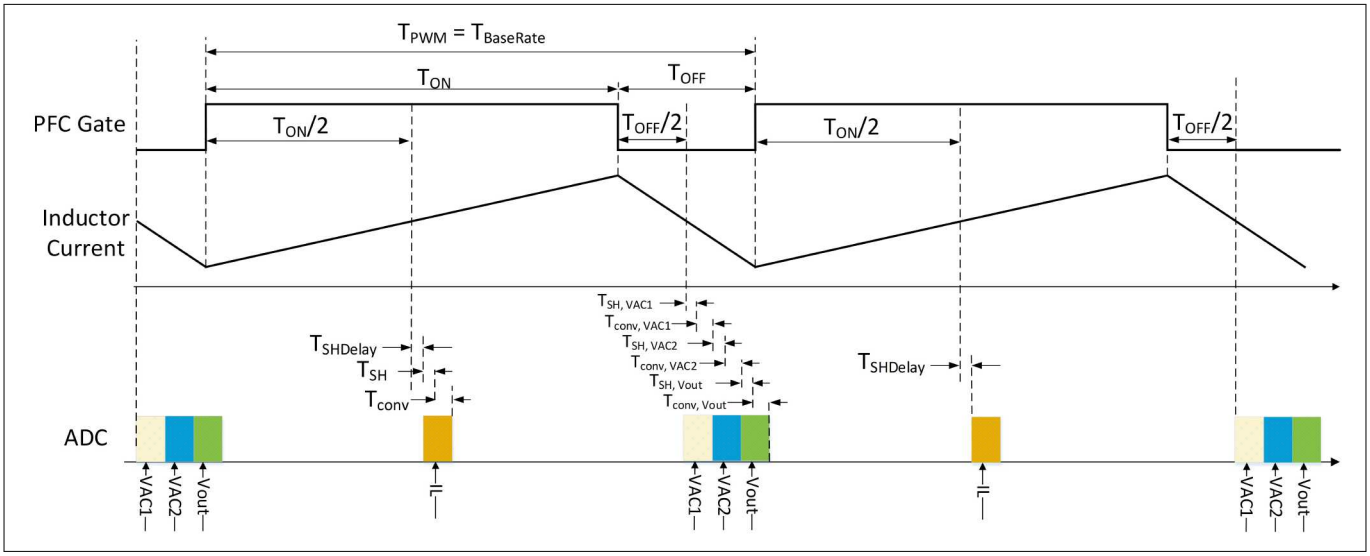


Figure 65 Timing diagram when the Base rate to PWM rate is 1:1

3.3.2 1:2 Base Rate

At higher PWM rates the execution load of the PFC algorithm does not leave enough room for the motor control algorithm. For these cases, the MCE offers a 1:2 base rate, meaning the current controller is only updated every second PWM cycle. In this case $T_{BaseRate}$ = 2 x T_{PWM}. The timing diagram in Figure 66, illustrates operation with a 1:2 Base Rate. Note how the Inductor Current sampling and execution of the control algorithm are skipped every second PWM cycle.

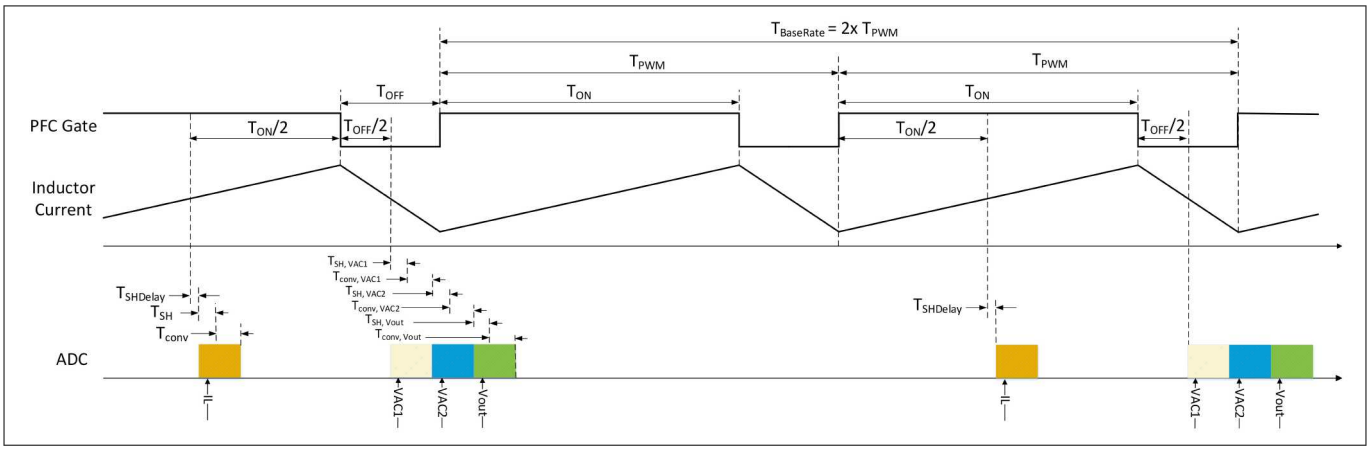


Figure 66 Timing diagram when the Base rate to PWM rate is 1:2

3.3.3 1:3 Base Rate

For highest possible PWM rate the MCE supports a 1:3 base rate, meaning the current controller is only updated every third PWM cycle. In this case $T_{BaseRate} = 3 \times T_{PWM}$. The timing diagram in Figure 67, illustrates operation with a 1:3 Base Rate. Note how the Inductor Current sampling and execution of the control algorithm are skipped every third PWM cycle.

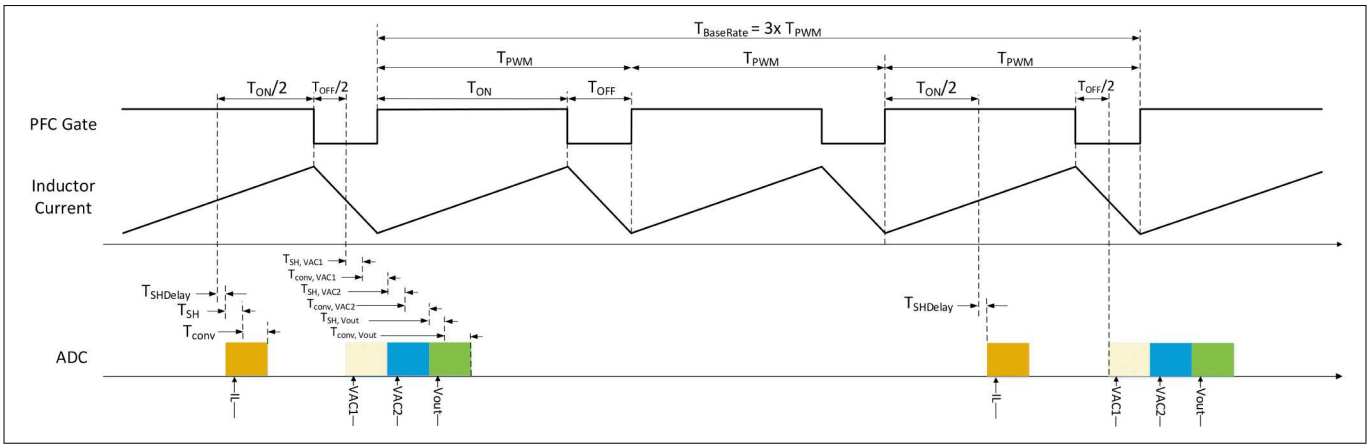


Figure 67 **Timing diagram when the Base rate to PWM rate is 1:3**

3.3.4 Co-existence of PFC and Motor Control Algorithm

Both the PFC and the Motor Control are real-time control system that must be executed in a time consistent manner. This requires special attention when both algorithms are running on the same single core device. The PFC algorithm typically executes at a higher rate than the motor control algorithm but the PFC execution time is only a fraction of motor execution time. On the MCE, execution of the PFC algorithm is given highest possible priority and it can preempt execution of the motor algorithm. This ensures both PFC- and motor algorithms can coexist on the same device. Neither phase nor rate of PFC PWM are synchronized to Motor PWM.

3.4 Protection

The MCE has a total of 9 PFC protection functions as summarized in the table below. The functions have been designed to protect the PFC from operating under potentially damaging conditions while at the same time ensure maximum robustness of the PFC operation.

The parameter FaultEnable can be used to enable/disable maskable protection function. The PFC has 3 non-maskable protections, Over Current Fault (OCP), Vout Overvoltage Fault and Current Measurement Offset Fault. These non-maskable protection functions are always enabled regardless of the setting of FaultEnable and the system will always take protective action for these functions. All other protection functions can be disabled by setting the corresponding bit in the parameter FaultEnable.

Fault status is reported in the parameters FaultFlags and SwFaults. Non-maskable faults are always reported in both FaultFlags and SwFaults. Maskable faults are always reported in FaultFlags but reporting in SwFaults depends on the setting of the parameter FaultEnable.

If a maskable protection function's corresponding bit in parameter FaultEnable is set, the fault will be reported in parameter SwFaults and the system will take protective action for that function. If the protection function's corresponding bit in parameter FaultEnable is not set, the fault will not be reflected in parameter SwFaults and the system will not take protective action for that function.

Table 14 Protection Functions

Protection Function	Description	Fault Actions		UL60730-1 certification
		With Fault Enabled	With Fault Disabled	
Over Current (OCP)	Fast, HW-based cycle-by-cycle overcurrent protection	PFC gate to inactive level. Automatic recover at the beginning of the following PWM cycle when fault clears. FaultFlags and SwFaults status update	PFC gate to inactive level. Automatic recover at the beginning of the following PWM cycle when fault clears. FaultFlags and SwFaults status update. Non-maskable fault	Yes
VAC Drop-out	Low instantaneous input voltage	PFC operation uninterrupted. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	No
VAC Overvoltage	High RMS input voltage protection	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	Yes
VAC Brown-out	Low RMS input voltage (undervoltage) protection	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	Yes
VAC Frequency	Out of range AC line frequency	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	Yes
Vout Overvoltage	DC-bus overvoltage protection	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update. Non-maskable fault	No

(table continues...)

Table 14 (continued) Protection Functions

Protection Function	Description	Fault Actions		UL60730-1 certification
		With Fault Enabled	With Fault Disabled	
Vout Open-Loop	DC-bus voltage open-loop (undervoltage) protection	Limit PFC gate duty-cycle to 0. Automatic recover when fault clears. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	No
Current Measurement Offset	Out of range current measurement offset before start of PFC	Abort start-up of PFC. Enter PFC_FAULT state and latch until power is cycled or set Command = 2. FaultFlags and SwFaults status update	Abort start-up of PFC. Enter PFC_FAULT state and latch until power is cycled or set Command = 2. FaultFlags and SwFaults status update. Non-Maskable fault	No
Execution	CPU execution load exceeding 95% or background tasks are not executed at least once every 60s	Enter PFC_FAULT state and latch until power is cycled or set Command = 2. FaultFlags and SwFaults status update	PFC operation uninterrupted. FaultFlags status update	Yes

Except for VAC Drop-out protection, the protection system automatically brings the system into a safe mode when a fault is detected and enabled. Most protection functions force the PFC duty-cycle to 0 in case of a fault and, when the fault clears, restores normal operating conditions. The exception to this handling approach is OCP fault, Current Measurement Offset fault and Execution fault. In case of OCP fault, the PFC gate is switched to the inactive state for the remainder of the PWM cycle but by the start of the following PWM cycle the gate is automatically reenabled if the OCP condition is cleared. A Current Measurement Offset fault and Execution fault forces the system into an inactive state (PFC_FAULT) form which there is no recovery until power is cycled or the PFC is reset.

Bitfield SWStatus of parameter PFCStatus represents the status of the PFC power stage switch. SWStatus will be set to Disable when either the Command parameter is set to Disable, or the duty-cycle is forced to zero by any fault conditions. Note that the faults capable of forcing the duty-cycle to zero are VAC Overvoltage-, VAC Brown-out-, VAC Frequency-, Vout Overvoltage- and Vout Open-Loop fault.

It should be noted that a PFC fault does not shut down operation of the motor. Likewise, a motor fault does not shut down operation of the PFC. If such an application specific fault synchronization is required, the application script must take care of the required handling. Fault status is accessible to the script through the parameters FaultFlags and SwFaults.

3.4.1 Over Current Protection

PFC over-current fault is detected by comparing the instantaneous inductor current against a pre-configured over current protection (OCP) threshold value. The PWM output is disabled when the inductor current exceeds the OCP threshold.

The OCP function is fully implemented by hardware and operates independently of the software. The over-current tripping mechanism makes use of an internal comparator that can be configured to both non-inverting

and inverting topology for inductor current sensing. Below diagram depicts the OCP mechanism for non-Inverting current sense topology.

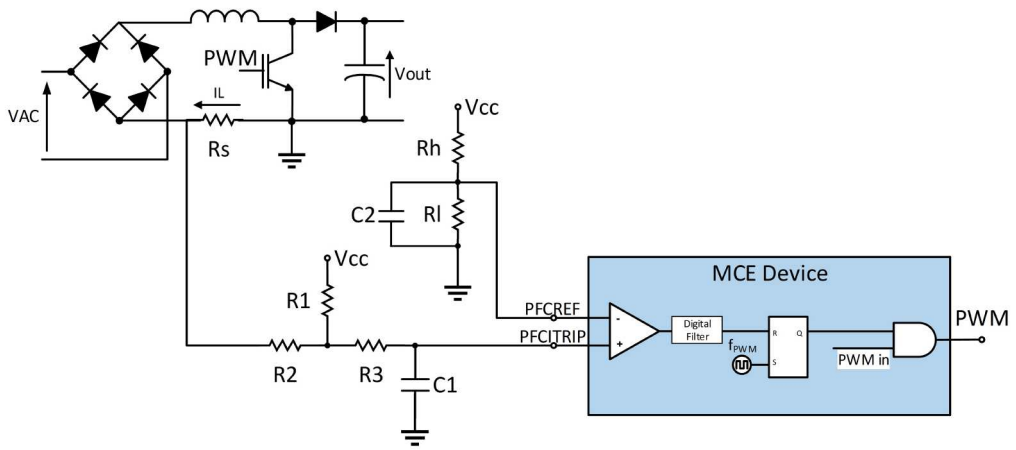


Figure 68 Cycle-by-cycle OCP Application Diagram for Non-Inverting current sense

The tripping level is set using external resistors R_h and R_l which set the tripping level at the PFCREF pin. (inverting input of the comparator). The voltage across the resistive shunt R_s, is scaled and offset by R_1, R_2 and R_3 and then fed to the PFCITRIP pin (Non-inverting input of the comparator). If the voltage at the PFCITRIP pin is below the voltage at the PFCREF pin, the comparator output goes low. Comparator output is connected to an internal digital filter that is used to de-bounce the input signal to prevent high frequency noise from mis-triggering of over current fault. User can configure the digital filter value using OCP_BlankingTime parameter. The comparator signal needs to remain stable (low) for the specified digital filter period to trigger the over-current fault. As a result, the PWM output immediately goes to configured passive level, and stays until the end of this PWM cycle, even if the inductor current drops below the PFC OCP threshold. At the beginning of the following PWM cycle, if the inductor current is below the PFC OCP threshold, then PWM output resumes. If the inductor current is still higher than the PFC OCP threshold, then the PWM output remains logic LOW. This type of OCP is commonly referred to as cycle-by-cycle protection.

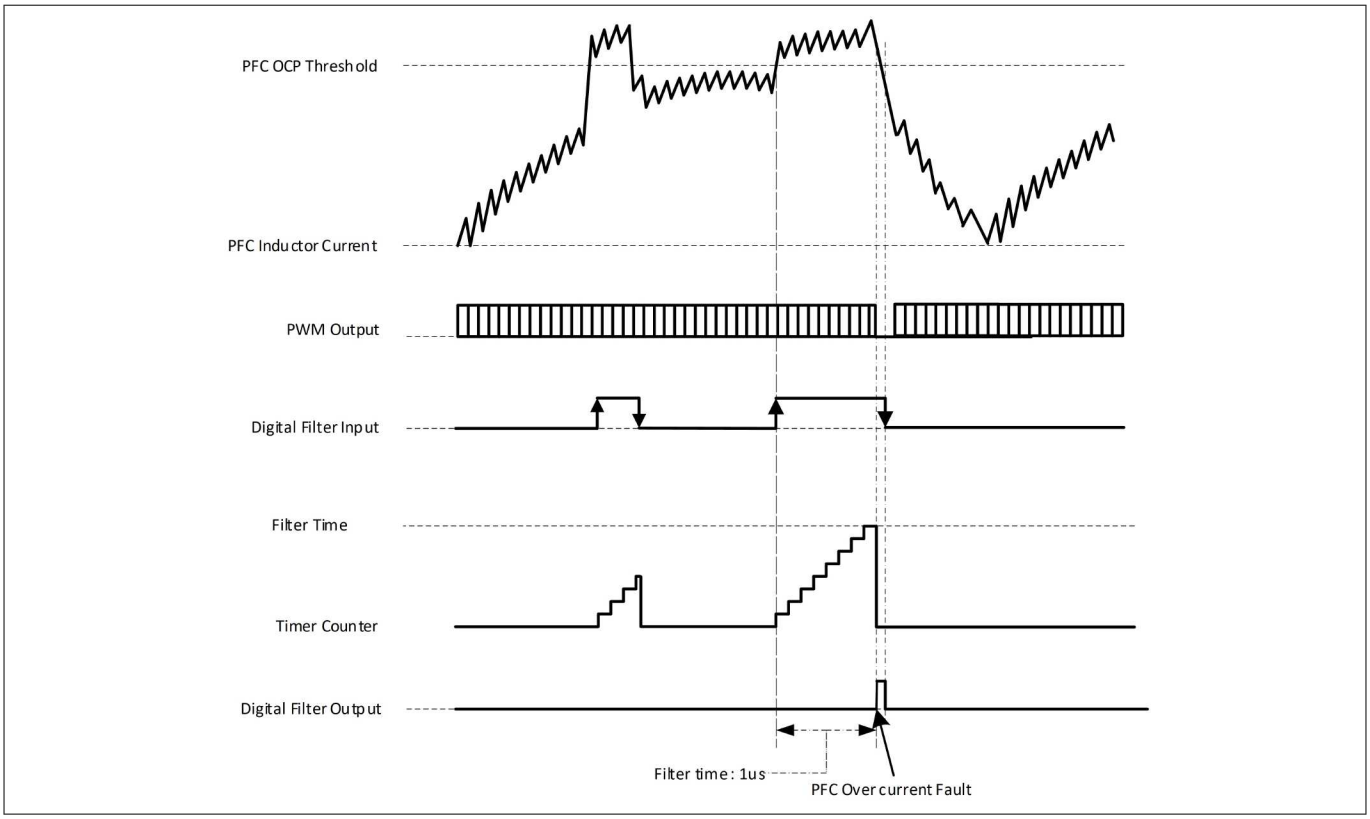


Figure 69 OCP Filter Timing Diagram

A timing diagram of cycle-by-cycle OCP is shown below. PWM operates with a switching period of T_{PWM}. During the on-time the Inductor current, IL, increases and during the off-time the inductor current decreases. When the inductor current exceeds the Trip Level, PWM is forced to the inactive state to prevent damage to the converter. At the beginning of the next PWM cycle, the fault is cleared and normal PWM resumes. In below diagram a second OCP fault is detected the following PWM cycle and the sequence repeats.

Note that PWM reenable is synchronized to the beginning of a new PWM cycle, guaranteeing the PWM switching frequency remains constant and as configured even during OCP conditions.

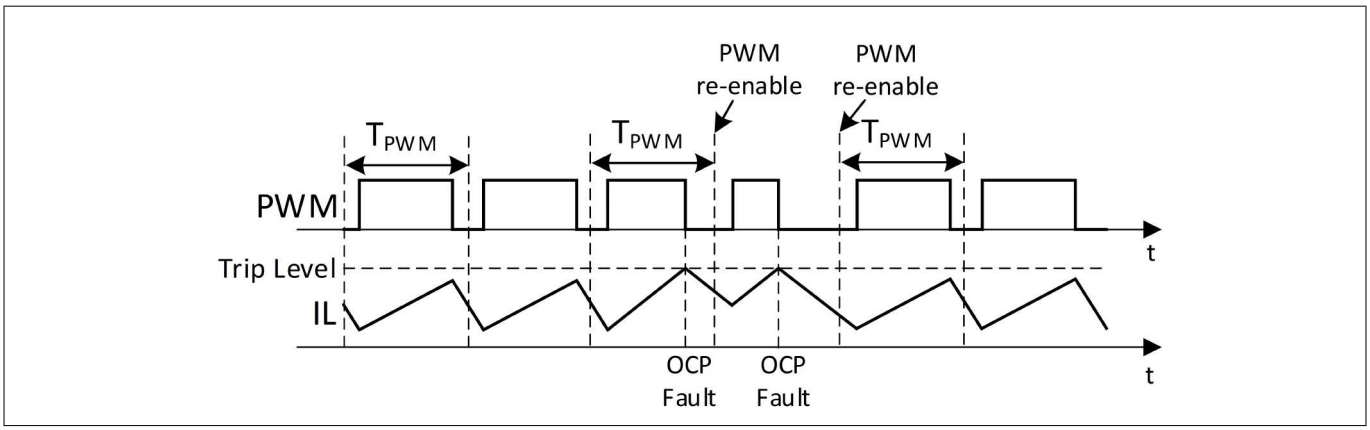


Figure 70 Cycle-by-cycle OCP timing diagram

OCP fault notification is illustrated in below diagram. When a fault occurs, an Internal Trap Flag is set. A configurable update time, OCP_Status_Update_Time, determines how often the trap flag gets read and copied to bit 0 of the parameters FaultFlags and SWFaults. Upon latch of the trap flag, and if the OCP fault condition is no longer present, the flag is cleared. If the fault persists, then the trap flag remains set. Bit 0 of FaultFlags and SWFaults will be set to 1 for a duration of OCP_Status_Update_Time and then automatically cleared. If the

application requires system level handling of the OCP fault, it is up to the application script to capture FaultFlags or SWFaults in a timely manner and take the appropriate action.

In cased of OCP fault, the PFC state machine remains in the 'Run State' and PWM will be chopped by the OCP comparator on a cycle-by-cycle basis until user stops the PFC by setting the parameter Command to disable.

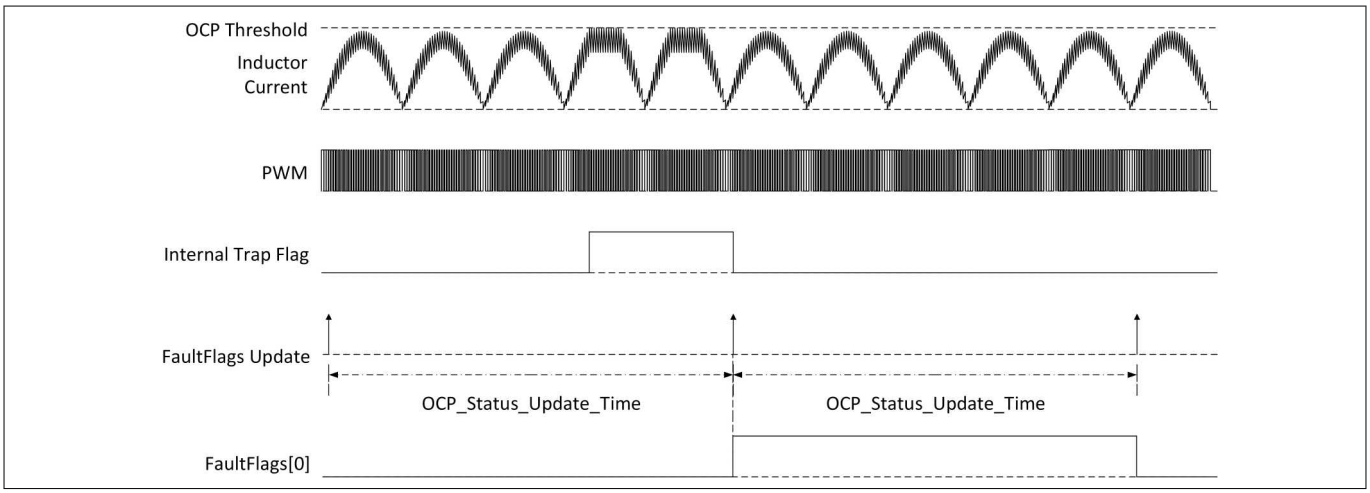


Figure 71 OCP fault signaling

OCP mechanism in MCE for inverting and non- Inverting current sense polarity topology is shown below. The MCE, inverts the output of the internal comparator in case of inverting current sense.

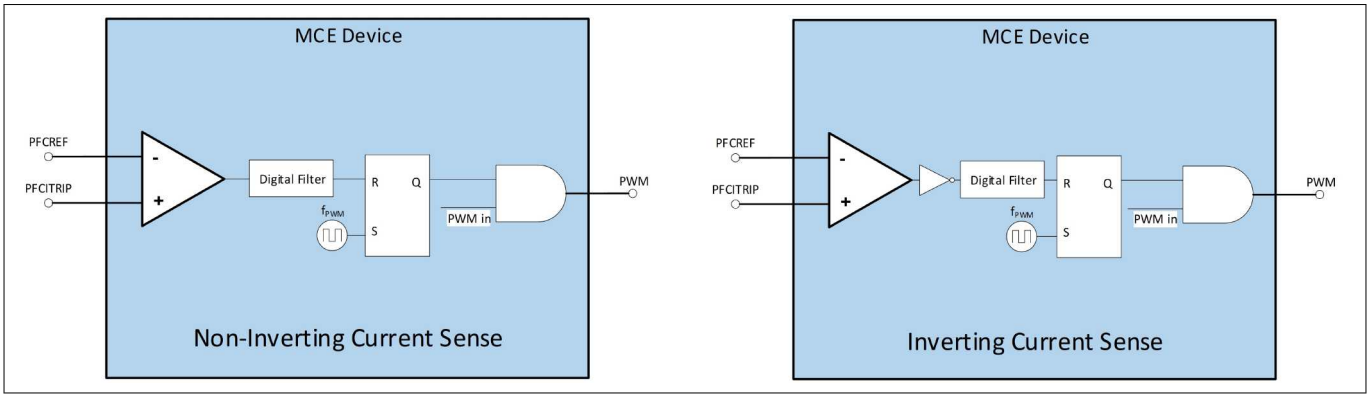


Figure 72 OCP Mechanism for inverting and non-inverting polarity topology

3.4.2 VAC Drop-out Protection

VAC Drop-out fault becomes active when the instantaneous, absolute value of the AC input voltage drops below a configurable threshold. A hysteresis and a deglitch window are added to prevent rapid toggling between normal- and fault conditions. VAC Drop-out Fault does not force the PFC duty-cycle to 0 meaning PFC operation will continue in the event of a fault. It is up to the application script to take the appropriate action in case of VAC drop-out fault.

The AC input voltage is sampled every PFC base-rate cycle and drop-out detection relies on the absolute value of this measurement. The protection function is executed at Sequencer Period (1 ms).

VAC Drop-out detection and clear is illustrated in Figure 73. If the instantaneous absolute value of the input voltage drops below VAC_DO_Thr, and remains lower than the threshold during a deglitch window of length VAC_DO_Deglitch_Time, bit 6 of FaultFlags is set. If the voltage exceeds the threshold during the deglitch window, the fault is not set. To clear the drop-out fault, the voltage must exceed VAC_DO_Thr plus a hysteresis, VAC_DO_Hyst, and stay above this threshold for the duration of a deglitch window with a length of

VAC_Recov_Deglitch_Time. If the voltage fails to stay above the threshold throughout the deglitch window, FaultFlags [6] remains set.

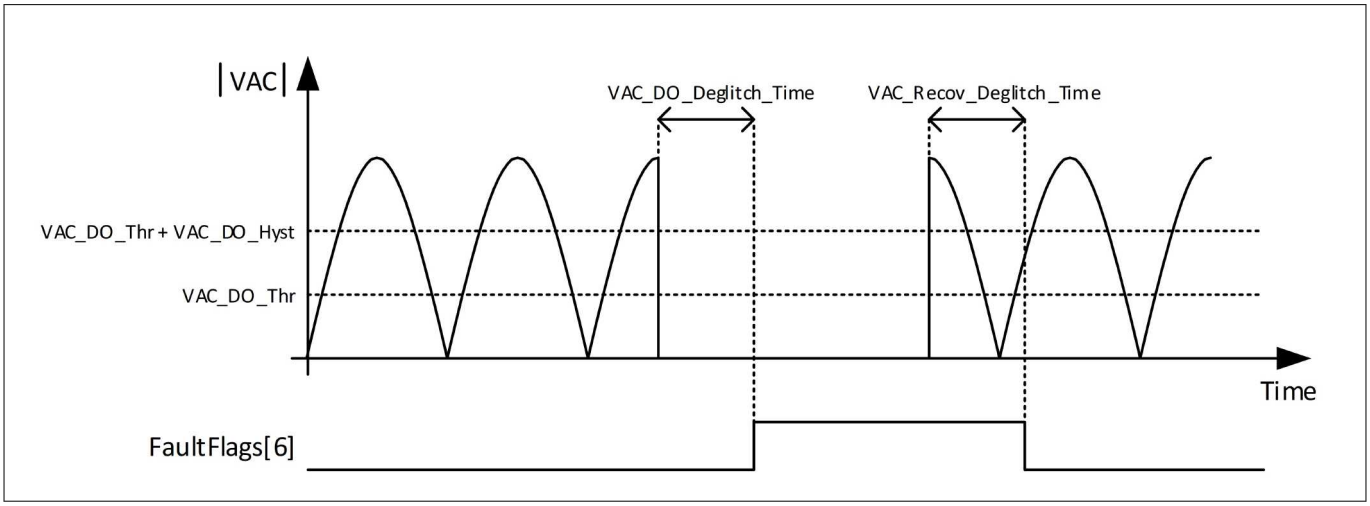


Figure 73 **VAC drop-out voltage detection can clear**

3.4.3 VAC Over Voltage and Brown-out Protection

AC Over Voltage fault becomes active when the AC input voltage RMS value is above a configurable threshold and AC brown-out (undervoltage) fault becomes active when the AC input voltage RMS value is below a configurable threshold. A hysteresis and a deglitch window are added to prevent rapid toggling between normal- and fault conditions.

Both over voltage- and brown-out protection operates on the RMS of the input voltage. The instantaneous AC input voltage is sampled every PFC base-rate cycle and the RMS value of the AC input voltage is updated every half-line cycle when AC input voltage zero-crossing is detected.

The VAC over-voltage detection and clear is illustrated in Figure 74 with the absolute value of the instantaneous AC voltage, VAC, shown on the top, RMS value of the AC voltage, VAC RMS, in the middle and fault reporting parameter FaultFlags, at the bottom. Note how the VAC RMS voltage is updated at every zero-crossing of VAC. If the VAC RMS value exceeds VAC_OVP_Thr, and remains higher than the threshold during a deglitch window of length VAC_OVP_Deglitch_Time, bit 5 of FaultFlags is set. If the voltage drops below the threshold during the deglitch window, the fault is not set. To clear the overvoltage fault, the voltage must drop below VAC_OVP_Thr minus a hysteresis, VAC_OVP_Hyst and stay below this threshold for the duration of a deglitch window with a length of VAC_Norm_Deglitch_Time. If the voltage fails to stay below the threshold throughout the deglitch window, FaultFlags[5] remains set. The length of the deglitch window is an integer number of half-line-cycle periods, THLC.

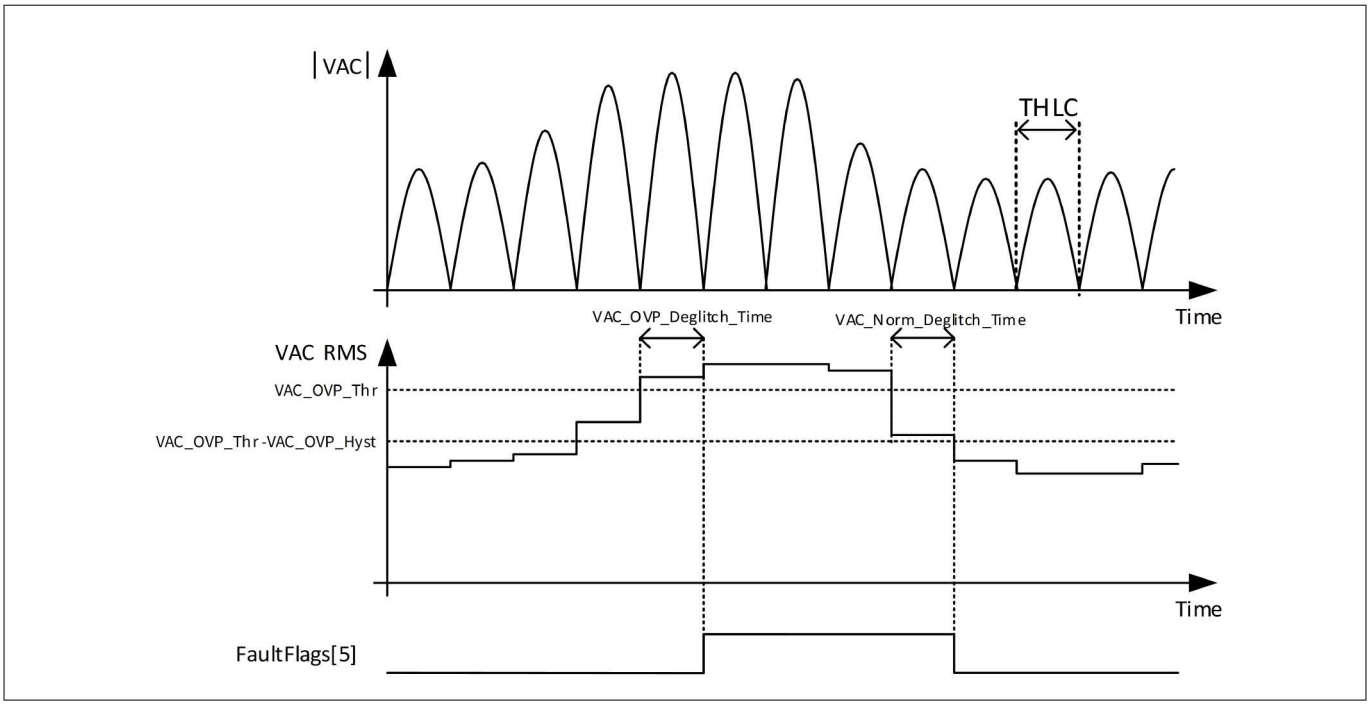


Figure 74 VAC over voltage detection and clear

The VAC Brown-out (undervoltage) detection and clear is illustrated in Figure 75, with the absolute value of the instantaneous AC voltage, VAC, shown on the top, RMS value of the AC voltage, VAC RMS, in the middle and fault reporting parameter FaultFlags, at the bottom. If the calculated VAC RMS drops VAC_BO_Thr, and remains lower than the threshold during a deglitch window of length VAC_BO_Deglitch_Time, bit 4 of FaultFlags is set. If the voltage exceeds the threshold during the deglitch window, the fault is not set. To clear the brown-out fault, the voltage must exceed VAC_BO_Thr plus a hysteresis, VAC_BO_Hyst, and stay above this threshold for the duration of a deglitch window with a length of VAC_Norm_Deglitch_Time. If the voltage fails to stay above the threshold throughout the deglitch window, FaultFlags[4] remains set.

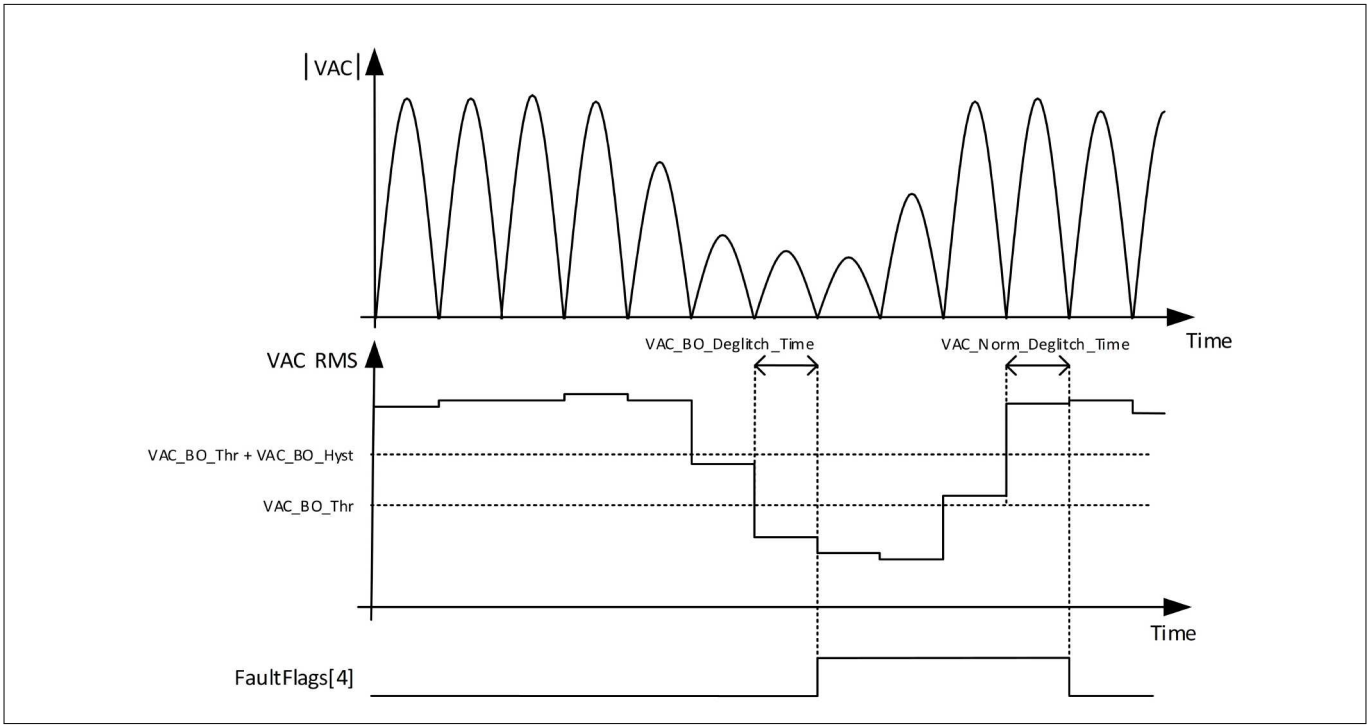


Figure 75 VAC brown-out detection and clear

3.4.4 Input Frequency Protection

The MCE monitors the actual AC line frequency and asserts a fault if it falls outside a configured window. The user can choose between 50 Hz or 60 Hz as nominal line frequency. Valid range of the actual AC input frequency is also configurable. For example, with a nominal AC frequency of 50 Hz, a typical valid range of actual AC input frequency is from 47 to 53 Hz.

The AC input frequency is determined by measuring the time between zero-crossings of the input voltage. During each Base Rate cycle the MCE checks for zero-crossing and increments a counter when a zero-crossing is *not* detected. When a zero-crossing is detected the counter value is latched and stored in parameter THLC which then holds the number of Base Rate cycle per half line cycle period. Note, that a long half line cycle period corresponds to a low frequency. AC input frequency is checked against min/max limits at the Sequencer Update Rate (1 kHz).

The principle behind the Input Frequency Protection function is shown in Figure 76. If the measured positive- or negative half line cycle period, THCL, is greater than the maximum limit, THLC_Validation_Max_Thr, and remains higher than the threshold during a deglitch window of length THLC_Validation_Deglitch_Time, bit 3 of FaultFlags is set to indicate a low line frequency. If the half line cycle period drops below the threshold during the deglitch window, the fault is not set. The frequency fault is cleared when the half line cycle period drops below THLC_Validation_Max_Thr minus a hysteresis, THLC_Validation_Max_Hyst and stays below this threshold for the duration of a deglitch window with a length of THLC_Validation_Deglitch_Time. If the half line cycle period fails to stay below the threshold throughout the deglitch window, FaultFlags[3] remains set.

Similarly, If the measured half line cycle period, THCL, is less than the max limit, THLC_Validation_Min_Thr, and remains lower than the threshold during a deglitch window of length THLC_Validation_Deglitch_Time, bit 3 of FaultFlags is set to indicate a high line frequency. If the half line cycle period exceeds the threshold during the deglitch window, the fault is not set. The frequency fault is cleared when the half line cycle period exceeds THLC_Validation_Min_Thr plus a hysteresis, THLC_Validation_Min_Hyst and stays above this threshold for the duration of a deglitch window with a length of THLC_Validation_Deglitch_Time. If the half line cycle period fails to stay above the threshold throughout the deglitch window, FaultFlags[3] remains set.

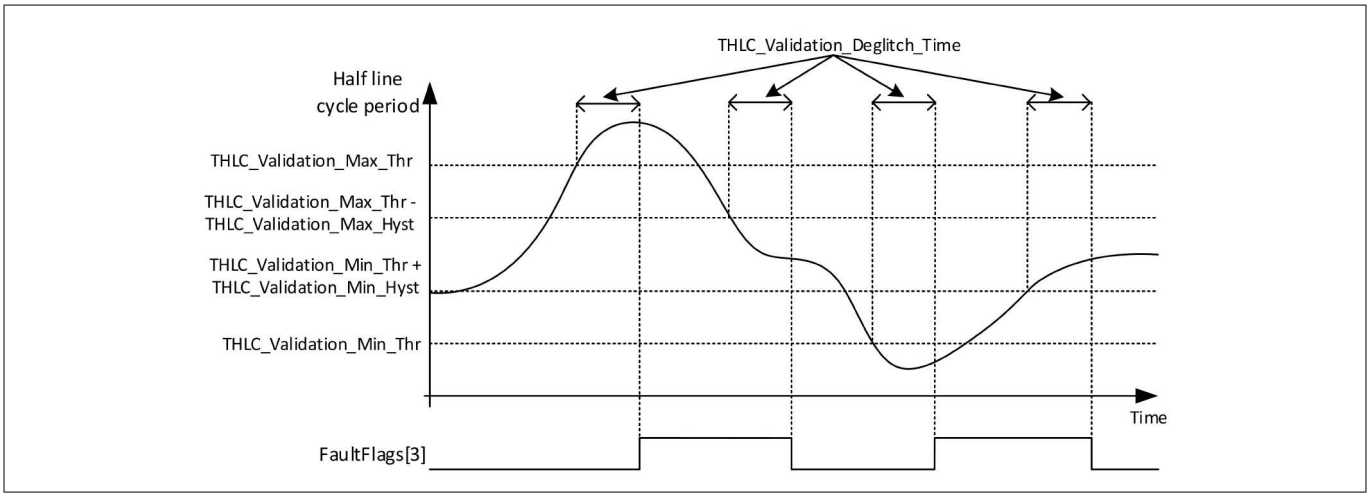


Figure 76 **Min/max input frequency detection and clear**

3.4.5 Vout Over Voltage and Open Loop Protection

Vout over voltage fault is active when the DC-bus voltage exceeds a configurable threshold and Vout Open Loop (undervoltage) fault is active when the DC-bus voltage drops below a configurable threshold. A hysteresis and a deglitch window are added to prevent rapid toggling between normal- and fault conditions. The DC-bus voltage is sampled every PFC switching cycle and can be read from the parameter Vout.

The DC-bus over-voltage detection and clearing is illustrated in Figure 77. If the DC-bus voltage exceeds Vout_OVP_Thr, and remains higher than the threshold during a deglitch window of length

Vout_OVP_Deglitch_Time, bit 2 of FaultFlags is set. If the voltage drops below the threshold during the deglitch window, the fault is not set. To clear the overvoltage fault, the voltage must drop below Vout_OVP_Thr minus a hysteresis, Vout_OVP_Hyst and stay below this threshold for the duration of a deglitch window with a length of Vout_Norm_Deglitch_Time. If the voltage fails to stay below the threshold throughout the deglitch window, FaultFlags[2] remains set.

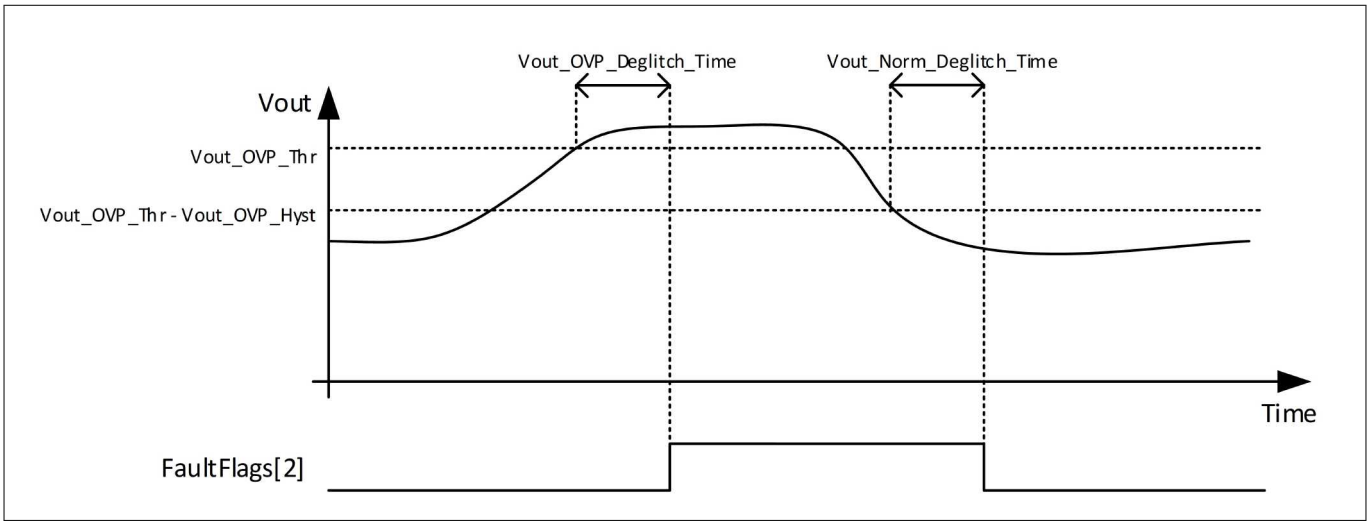


Figure 77 Vout over voltage detection and clear

The Vout Open-Loop (undervoltage) detection and clear is illustrated in Figure 78. If the DC-bus voltage drops below Vout_OLP_Thr, and remains lower than the threshold during a deglitch window of length Vout_OLP_Deglitch_Time, bit 1 of FaultFlags is set. If the voltage rises above the threshold during the deglitch window, the fault is not set. To clear the brown-out fault, the voltage must exceed Vout_OLP_Thr plus a hysteresis, Vout_OLP_Hyst, and stay above this threshold for the duration of a deglitch window with a length of Vout_Norm_Deglitch_Time. If voltage fails to stay above the threshold throughout the deglitch window, FaultFlags[1] remains set.

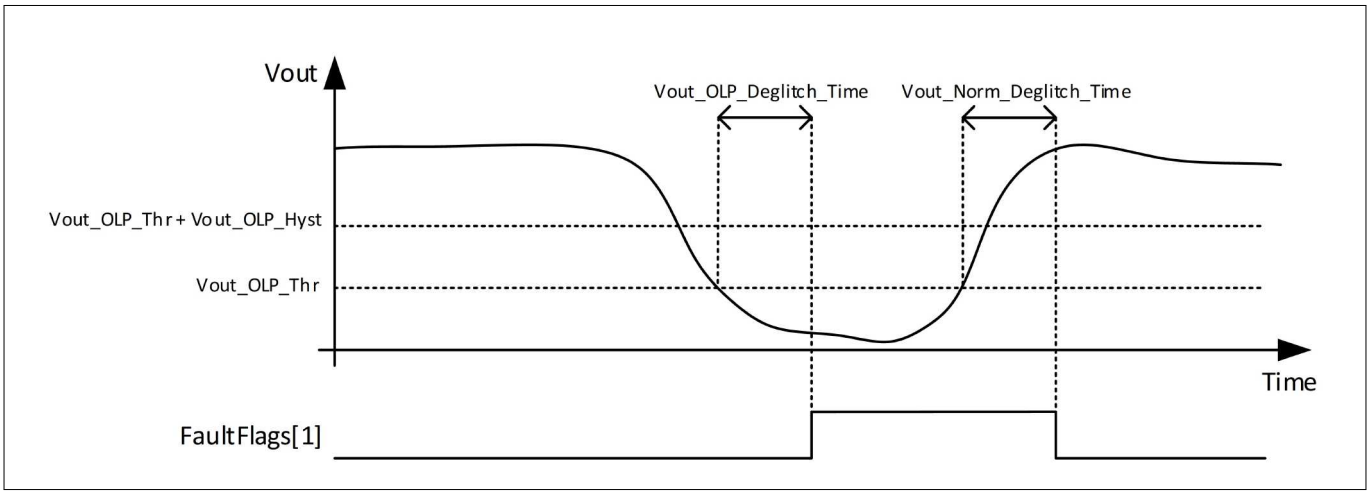


Figure 78 Vout open loop detection and clear

3.4.6 Current Measurement Offset

As part of the PFC startup, the offset of the current measurement channel, ILOffset, is determined and used for calibration of the inductor current samples. If this measured offset falls outside configurable limits, ILOffset_Min and ILOffset_Max, a protection function prevents the PFC from starting and raises a FaultFlags[7]. Out of range Current Measurement Offset brings the PFC into a dedicated fault state, PFC_FAULT.

An out of range offset indicates a faulting measurement circuit that cannot be relied on for closed loop current control. Only an MCE reset or a PFC reset (Command = 2) will initiate a new start attempt.

3.4.7 Execution

CPU execution load is monitored at a system level and takes all running tasks into account. If the execution load exceeds 95% or if background tasks are not executed at least once every 60s, FaultFlags[9] is set to indicate a fault. If the fault is enabled, the PFC enters a dedicated fault state, PFC_FAULT. Only an MCE reset or a PFC reset (Command = 2) will initiate a new start attempt.

If Class B is enabled, the evaluation period of background task execution interval is reduced to 1 second. If Class B is disabled and BG monitoring is enabled (MCEOS. SysTaskConfig.BGtask_Monitoring_EN == 1, i.e bit 6 of “SysTaskConfig”), software reset is performed in case any of the background task is not executed at least once 60s period.

4 System

4.1 Internal Oscillator Calibration

4.1.1 Overview

The internal oscillator frequency of MCE varies as the temperature changes. The accuracy of the internal oscillator can be improved by a calibration process with respect to temperature changes. The MCE implements a run-time calibration routine that measures the die temperature using its on-chip temperature sensor, and applies an offset value to adjust the internal oscillator accordingly to achieve higher accuracy. This calibration routine is executed every 20 ms automatically.

This internal oscillator calibration function can be enabled by setting the 3rd bit of ‘SysTaskConfig’ parameter. See respective product datasheets for detailed specification of achievable accuracy.

4.2 Multiple Parameter Programming

4.2.1 Parameter Page Layout

In iMOTION ™ product, 4k bytes of flash memory are used to store control parameter data. There are totally 16 parameter blocks, each parameter block is 256 bytes in size. Multiple parameter blocks up to a maximum of 16 can be used to support different motor types or hardware.

Active parameter set is specified by a parameter set number, which can be configured using iMOTION ™ Solution Designer. iMOTION ™ Solution Designer output (out.ldf) that contains the parameter values, can be programmed into the parameter block using iMOTION ™ Solution Designer. iMOTION ™ Solution Designer output file contains the specified parameter set number. Solution Designer loads the parameter values into the corresponding parameter block. Each parameter block can be updated individually multiple times.

Each parameter set will take one parameter block. The valid parameter set IDs can range from 0(0x00) to 15(0x0F).

4.2.2 Parameter Block Selection

MCE supports to select the parameter block in 3 different methods; UART, Analog input and GPIO pins.

Parameter block selection input configuration is available in iMOTION ™ Solution Designer and iMOTION ™ Solution Designer updated "ParSetConf" parameter. Table 15 shows relationship between ParSetConf[3:0] and configuration of parameter selection.

Table 15 Configuration of Parameter Selection

MCEOS.ParSetConf[3:0]	Parameter Selection
0	UART control
1	Multiple parameter handling is not enabled. Parameter sets specified in MCEOS.ParPageConf[9:4] is loaded as a default parameter set
2	Analog input
3	GPIO pins (PAR0, 1, 2 and 3)

***Note**: Not all of the 4 methods to select parameter block are available in all iMOTION ™ devices, due to pin availabilities. Refer specific device datasheet for available methods to select parameter block.*

4.2.2.1 UART Control

Specific UART messages are defined to load the parameter block from flash to RAM and save the parameter set from RAM to flash. Refer Chapter 5.1.7.10 for message format.

4.2.2.2 Analog Input

Parameter block is selected based on the analog input value. MCE uses “PARAM” pin as the Analog input for parameter set selection. Mapping between parameter page selections based on Analog input mentioned below

$$ParameterBlock = Integer\left\{\left(\frac{AnalogInput}{V_{adcref}} \times 15\right)\right\}$$

Example if AnalogInput = 1.2 V and V_{adcref} = 3.3 V, then ParameterBlock = 5

***Note**: Maximum value of parameter block is 15 (0x00 to 0x0F).*

4.2.2.3 GPIO Pins

Parameter block is selected based on the four GPIO pins. GPIO pins used for parameter set selection are named as “PAR0”, “PAR1”, “PAR2” and “PAR3”. Mapping between parameter page selections based on GPIO pins are listed in the Table 16.

Table 16

GPIO Input				Parameter Block
PAR3	PAR2	PAR1	PAR0	
0	0	0	0	0 (0x00)
0	0	0	1	1 (0x01)
0	0	1	0	2 (0x02)
0	0	1	1	3 (0x03)
0	1	0	0	4 (0x04)

(table continues...)

Table 16 (continued)

GPIO Input				Parameter Block
PAR3	PAR2	PAR1	PAR0	
0	1	0	1	5 (0x05)
0	1	1	0	6 (0x06)
0	1	1	1	7 (0x07)
1	0	0	0	8 (0x08)
1	0	0	1	9 (0x09)
1	0	1	0	10 (0x0A)
1	0	1	1	11 (0x0B)
1	1	0	0	12 (0x0C)
1	1	0	1	13 (0x0D)
1	1	1	0	14 (0x0E)
1	1	1	1	15 (0x0F)

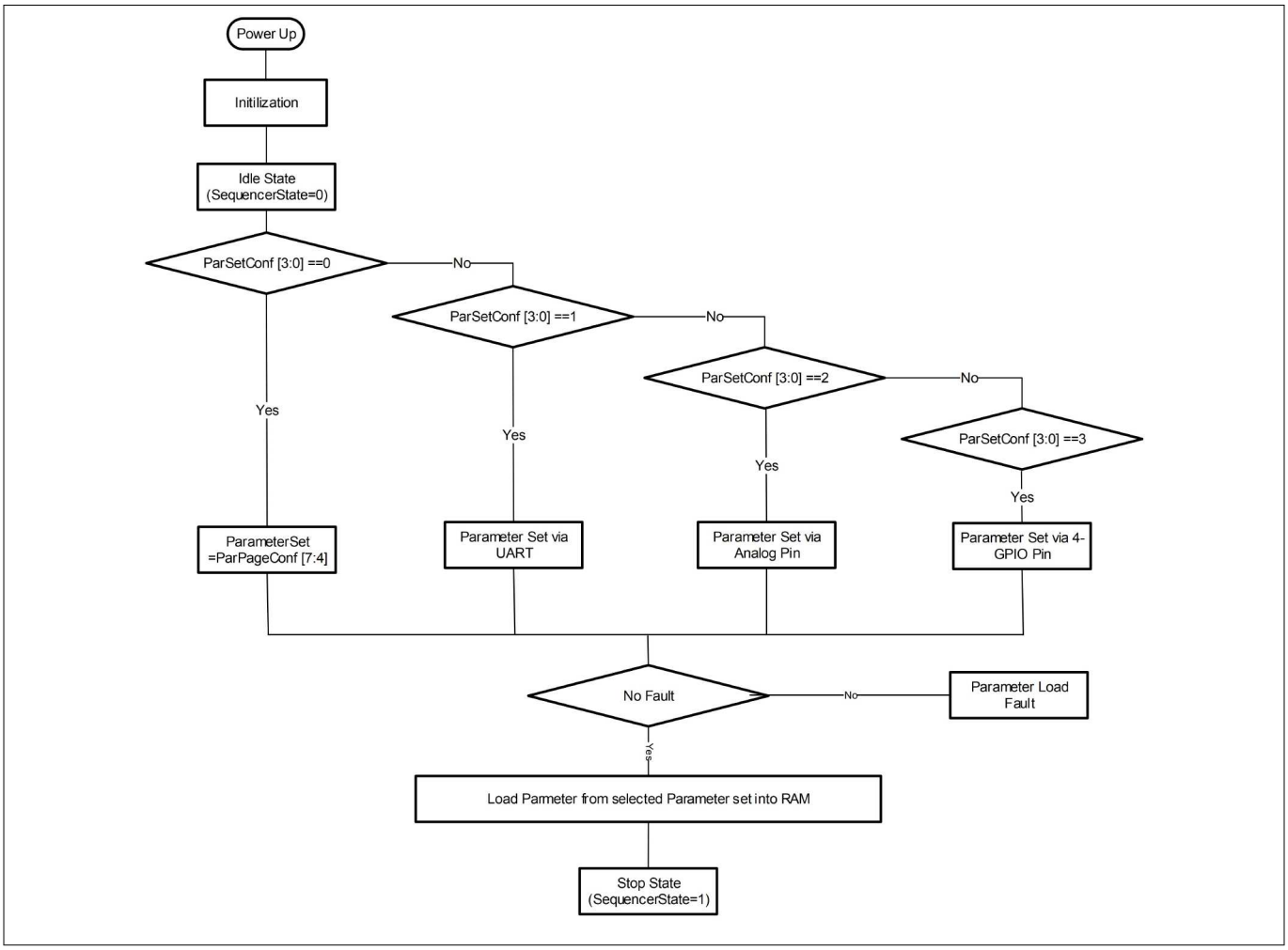


Figure 79 Parameter Load Procedure

4.2.3 Parameter load fault

If there is no parameter data available in the selected parameter block, MCE stays in IDLE state. It is not possible to start the motor from IDLE state. If there is no valid parameter data is available in the selected parameter block, MCE report parameter load fault and stays in IDLE state. In this condition, it is required to load the right parameter data or select right parameter block.

If there is no other fault, the MCE load parameter values into RAM then go to STOP state and is ready to run the motor.

4.3 Low Power Standby

The MCE provides two types of standby modes, CPU Idle Sleep and Low Power, that enabled reduced power consumption of the MCE when the motor/PFC are not operating. These two modes can be enabled together or independently. CPU Idle Sleep puts the CPU into sleep mode when there are no tasks to execute. Power reduction depends on the MCE idle time (CPU load) and power reduction is limited. Low Power Mode achieves more power reduction by reducing the CPU clock and switching off some of the internal controller peripherals. In Low Power mode, the Motor PWM unit and the PFC PWM unit (if supported) are disabled. In Low Power mode, VDC is measured every 1 ms. To detect a critical over voltage, over voltage, or under voltage fault two consecutive VDC measurements must be above the over voltage threshold for a fault to be triggered, giving a total response time of 2 ms. The measured VDC is not filtered and the raw value is used for the voltage protection functions.

The highest power saving is achieved when enabling both CPU Idle Sleep and Low Power Mode simultaneously. The Bitfields of the parameter ‘StandbyConf’ enables/disables the standby modes.

Available features depend on the standby mode, where CPU Idle Sleep generally supports all available features and Low Power Mode only supports a subset. The table below gives a general overview of the features supported in each mode.

Group	Protection Fault	CPU Idle Sleep	Low Power Mode
Interface	iSD DashBoard	x	x
	iSD Oscilloscope	x	x
	JCOM	x	x
	USER UART	x	x
	Digital/Analog Hall	x	
	PFC	x	
Control Input	VSP	x	x
	DUTY	x	x
	FREQ	x	x
Scripting	Scripting	x	x
	IR Interface	x	RC-5 only
	Data Storage	x	x
	TRIAC Control	x	x
	I2C	x	x

Available protection functions also depend on the standby mode. With CPU Idle Sleep all protection functions are supported and with Low Power Mode only a subset of protection functions is supported. The table below gives a general overview of the functions are supported in each mode.

Group	Protection Fault	CPU Idle Sleep	Low Power Mode
Motor	Critical Over Voltage	x	x
	Over Voltage	x	x
	Under Voltage	x	x
	Gate Kill	x	
	Gate Kill Pin	x	
	Flux	x	
	Over Temperature	x	x
	Rotor Lock	x	
	Phase Loss	x	
	Current Offset	x	
PFC	Over Current (OCP)	x	
	VAC Drop-out	x	
	VAC Overvoltage	x	
	VAC Brown-out	x	
	VAC Frequency	x	
	Vout Overvoltage	x	
	Vout Open-Loop	x	
	Current Meas. Offset	x	
System	UART link break	x	x
	CPU Execution	x	x
	Parameter Load	x	x

4.3.1 Entry and Exit Conditions

If enabled, CPU Idle Sleep is entered as soon as the MCE is in an execution idle state. No other conditions need to be met. CPU Idle Sleep is exited when there are tasks to execute.

Before entering Low Power mode, a configurable delay time must expire and specific conditions must be met. During this time the system remains fully active and can cancel entry to Low Power mode by running the motor or triggering a fault. The delay time is configured by the 'StandbyPauseTimeout' parameter with a maximum of 65535 milliseconds. This parameter cannot be modified at runtime from script.

Following are the conditions for entering and exiting to/from Low Power mode for MOTOR and PFC (if supported):

Entry Conditions:

1. Motor is in STOP state
2. PFC is disabled (if supported)
3. No motor faults are present
4. Zero-Vector-Braking is not active
5. Pause before entering low power mode has expired

Exit Conditions:

1. Motor start command has been received

2. PFC is enabled (if supported)
3. Fault has occurred

A motor start command or a fault is needed to wakeup the MCE from Low Power Mode. The start command can come from any of the active sources which include Control Input (VSP/FREQ/DUTY), scripting, JCOM, User UART or Solution Designer.

4.3.2 Scripting

Script functionality is fully supported with instructions limitations per Execution Step and a minimum a task period of 1 millisecond (Task 0).

- Up to 40 instructions when using Motor and no PFC
- Up to 20 instruction when Motor and PFC

Both Motor_SequencerState = 13 and PFC_SequencerState = 7 (if supported) will indicate that MOTOR and PFC are in Low Power Mode. For more information on state handling during standby, refer to Chapter 2.1 and Chapter 3.2

4.3.2.1 Timing

Entering Low Power Mode takes 'Overhead Time' + 'Pause Before Entering to Low Power' + '1 millisecond state transition' in the worst case. Average Overhead Time is 106 microseconds, corresponding to ramp-down process and stand-by request.

Exiting from STAND-BY mode takes 'Overhead Time' + '1 millisecond state transition' in the worst case. Average Overhead Time is 34.24 microseconds where the system reconfigures what was disabled in the ramp down process.

5 Communication Interface

5.1 User Mode UART

The user mode UART communication is designed to provide a simple, reliable and scalable communication protocol for motor control application. The protocol is simple so that it can be easily implemented even in low-end microcontrollers which work as master to control the motor. It supports networking (up to 15 nodes on same network) which is required in some industrial fan/pump applications. Each UART commands are processed every 1 ms.

If users intend to implement a customized UART communication protocol, it can be realized by using those configurable UART driver methods described in Chapter 6.12.1.

5.1.1 Baud Rate

The MCE supports the following Baud rate configuration for user mode UART: 2400 bps, 9600 bps, 19200 bps, 57600 bps, 115200 bps, and 230400 bps.

5.1.2 Data Frame

The format of the data frame is shown in Figure 80 . Notice that it follows little endian format.

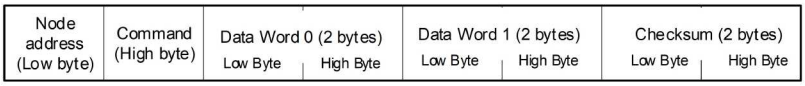


Figure 80 **UART Data Frame**

5.1.3 Node Address

Node address is the first byte in a data frame. It is designed to allow one master controlling multiple slaves in the same network. Each slave node has its unique node ID. The slave only acknowledges and responds to the message with same ID. There are two broadcast addresses (0x00 and 0xFF) defined for different usage. If a message is received with address = 0x00, all the slaves execute the command but will not send a reply to the master. This is useful in a multiple slave network and the master needs to control all the slaves at the same time, for example, turn on all the motor by sending only one message. If received a frame with address = 0xFF, the slave will execute the command and send a reply to the master. This is useful in 1-to-1 configuration when the master does not know or does not need to know the slave node address.

Table 17 **Node Address Definition**

Node Address	Command
0x00	All nodes receive and execute command, no response
0x01 to 0x0 F	Only the node that has same address executes the command and replies to the master
0x10 to 0xFE	Reserved
0xFF	All nodes receive and execute the command and reply to the master. Only used in 1-to-1 configuration. It will cause conflict if multiple nodes connected to the same network

5.1.4 Link Break Protection

Link break protection is to stop the motor if there is no UART communication for certain period of time. In some application, the main controller maintains communication with the motor controller. In case of a loss of communication or line break, it is desired to stop the motor for protection. This protection feature is enabled or disable and Link break timeout is configured in Solution Designer.

5.1.5 Command

UART command is the second byte in a data frame. Bit [5:0] specifies the command code. Bit [7]indicates the direction of the data frame. All data frames sent by master must have bit 7 cleared (= 0), all reply data frames sent by slave will have bit 7 set (= 1). Bit [6] in the reply data frame indicates the success (=0) or error (=1) of the command.

Table 18 **UART Command Definition**

Command (Bit[5:0])	Description
0	Read Status
1	Request to clear fault flag
2	Select Control input mode

(table continues...)

Table 18 (continued) UART Command Definition

Command (Bit[5:0])	Description
3	Set motor control target speed
4	Not used, slave will not reply to master
5	Read Register
6	Write Register
7	Not used, slave will not reply to master
8	Register Write Low-Word and buffer
9	Register Write High-Word to buffer
10	Register Read High-Word from buffer
11 - 31	Not used, slave will not reply to master
32	Load parameter set
33-63	Not used, slave will not reply to master

5.1.6 Verification

There are two methods for user mode UART communication verification. One is checksum, the other is CRC(Cyclic Redundancy Check). The two methods can be configured in iSD Configure Wizard page.

Checksum is 16-bit format and it shall be calculated as below:

[Command: Node address] + Data Word 0 + Data Word 1 + Checksum = 0x0000

Notice that when sending the checksum word to the user UART interface, little endian format shall be followed as shown in Figure 80.

Checksum calculation example:

Input: Node address = 1, command = 2, Data Word 0 = 0x1122 and Data Word 1 = 0x3344

[Command: Node address] = 0x0201

Checksum = -1 x (0x0201 + 0x1122 + 0x3344) = 0xB999

Data frame:

Node address (Low byte)	Command (High byte)	Data Word0 (2 bytes)		Data Word1 (2 bytes)		Checksum (2 bytes)	
		Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte
0x01	0x02	0x22	0x11	0x44	0x33	0x99	0xB9

Figure 81 Data Frame to be transmitted

CRC(Cyclic Redundancy Check) is CRC16-CCITT polynomial 0x1021 (x16 + x12 + x5 + 1), this allows a hamming distance of 4 as the number of transmitted bits is below 32751.

[Command: Node address], Data Word 0, Data Word 1 all should be included in CRC calculation, the calculation method is as below.

CRC model :	CRC16-CCITT x16+x12+x5+1
Width:	16

Ploy(Hex):	0x1021
INIT(Hex):	0xFFFF
XOROUT:	0
REFIN:	No
REFOUT:	No
Switch:	Switch high 8-bit and low 8-bit of CRC 16-bit output

CRC calculation example:

Node address	Command	Data Word0 (2 bytes)		Data Word1 (2 bytes)		CRC (2 bytes)	
Low byte	High byte	Low byte	High byte	Low byte	High byte	Low byte	High byte
0x01	0x06	0x01	0x40	0x01	0x00	0x1D	0xDE

5.1.7 UART message

5.1.7.1 Read Status: Command = 0x00

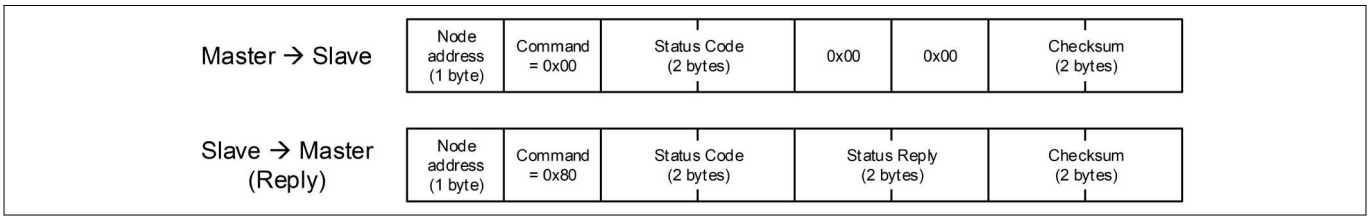


Figure 82 **Read Status command**

Table 19 **Status code and status reply**

Status code	Status reply
0x0000	Fault Flags
0x0001	Motor Speed
0x0002	Motor State
0x0003	Node ID
0x0004 – 0xFFFF	0x0000

5.1.7.2 Clear Fault: Command = 0x01

Master → Slave	Node address (1 byte)	Command = 0x01	0x00	0x00	0x00	0x00	Checksum (2 bytes)
Slave → Master (Reply)	Node address (1 byte)	Command = 0x81	0x00	0x00	0x00	0x00	Checksum (2 bytes)

Figure 83 **Clear fault command**

5.1.7.3 Change Control Input Mode: Command = 0x02

Master → Slave	Node address (1 byte)	Command = 0x02	0x00	0x00	0x00	00: UART 01: Analog 02: Freq 03: Duty	Checksum (2 bytes)
Slave → Master (Reply)	Node address (1 byte)	Command = 0x82	0x00	0x00	0x00	00: UART 01: Analog 02: Freq 03: Duty	Checksum (2 bytes)

Figure 84 **Control input mode command**

5.1.7.4 Motor Control: Command = 0x03

Master → Slave	Node address (1 byte)	Command = 0x03	0x00	0x00	TargetSpeed (2 bytes)	Checksum (2 bytes)
Slave → Master (Reply)	Node address (1 byte)	Command = 0x83	SequencerState (2 bytes)		MotorSpeed (2 bytes)	Checksum (2 bytes)

Figure 85 **Motor control Command**

Note: *Target Speed = 0: motor stop, TargetSpeed ≠ 0: motor start*

5.1.7.5 Register Read: Command = 0x05

Master → Slave	Node address (1 byte)	Com-mand = 0x05	FB ID (1 byte)	Register ID (1 byte)	0x00	0x00	Checksum (2 bytes)

Slave → Master (reply)	Node address (1 byte)	Com-mand = 0x85	FB ID (1 byte)	Register ID (1 byte)	Register Value (lower 16 bit) (2 bytes)	Checksum (2 bytes)

***Note**:* *Reading 32 bit registers provides the lower 16 bit of the register value immediately. The higher 16 bit are copied into a register buffer and can be read by command 0x0A.*

5.1.7.6 Register Read High-Word: Command = 0x0A

Master → Slave	Node address (1 byte)	Com-mand = 0x0A	0x00	0x00	0x00	0x00	Checksum (2 bytes)

Slave → Master (reply)	Node address (1 byte)	Com-mand = 0x8A	0x00	0x00	Register Buffer (higher 16 bit) (2 bytes)	Checksum (2 bytes)

***Note**:* *Reading the higher 16 bit of a 32 bit register requires to update the register buffer by calling register read command 0x05 in advance.*

Attention**:* ***The register buffer is shared between read and write command

5.1.7.7 Register Write: Command = 0x06

Master → Slave	Node address (1 byte)	Com-mand = 0x06	FB ID (1 byte)	Register ID (1 byte)	Register Value (2 bytes)	Checksum (2 bytes)

Slave → Master (reply)	Node address (1 byte)	Com-mand = 0x86	FB ID (1 byte)	Register ID (1 byte)	Register Value (2 bytes)	Checksum (2 bytes)

5.1.7.8 Register Write Low-Word: Command = 0x08

Master → Slave	Node address (1 byte)	Com-mand = 0x08	FB ID (1 byte)	Register ID (1 byte)	Register Value (2 bytes)	Checksum (2 bytes)
Slave → Master (reply)	Node address (1 byte)	Com-mand = 0x88	FB ID (1 byte)	Register ID (1 byte)	Register Value (2 bytes)	Checksum (2 bytes)

***Note**:* *Writing a 32 bit register requires to update the register buffer by calling register write high-word command 0x09 in advance. This command writes the content of the register buffer as high-word combined with the register value as low-word into the register at once.*

5.1.7.9 Register Write High-Word: Command = 0x09

Master → Slave	Node address (1 byte)	Com-mand = 0x09	0x00	0x00	Register Buffer (higher 16 bit) (2 bytes)	Checksum (2 bytes)
Slave → Master (reply)	Node address (1 byte)	Com-mand = 0x89	0x00	0x00	Register Buffer (higher 16 bit) (2 bytes)	Checksum (2 bytes)

***Note**:* *This command updates the register buffer as high-word only. Writing a 32 bit register requires calling register write low-word command 0x08 afterward.*

Attention**:* ***The register buffer is shared between read and write command

5.1.7.10 Load Parameter Set: Command = 0x20

'Load parameter set' command will trigger device reset before to load the parameters from the specified parameter set stored in FLASH into the RAM. The valid range of the parameter set number is: 0 – 15. In the reply frame, data 0 word contains the value of 'Status' (0: success; 1: fail; 2: parameter set number not supported.)

Master →Slave	Node address (1 byte)	Command =0x20	0x0022	Param Set No	0x00	Checksum (2 bytes)
Slave →Master (Reply)	Node address (1 byte)	Command =0xA0	0x0022	Status	0x00	Checksum (2 bytes)

Figure 86 Load Parameter with Reset Command

5.1.8 Connecting multiple nodes to same network

It is possible to connect multiple MCE to same UART network, see Figure 87 detail.

For the TXD pin of each MCE node, it needs to connect a Schottky diode before connect to the same wire, and on the master controller side, a 4.7 kOhm pull up resister is required.

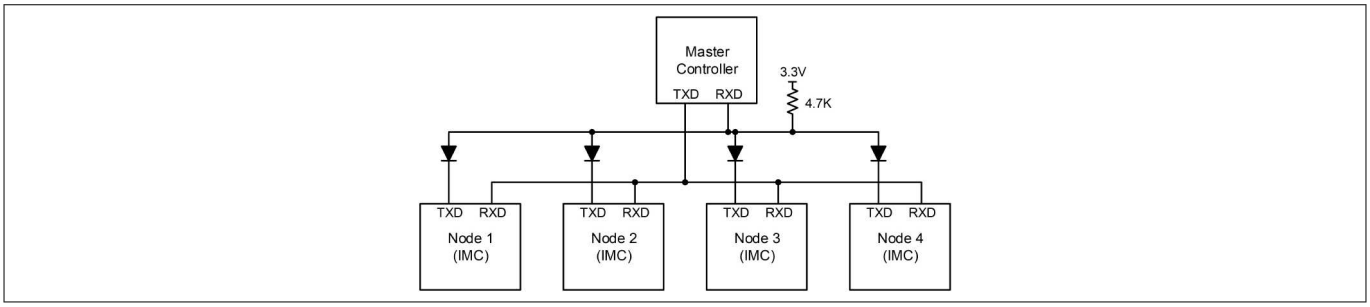


Figure 87 UART network connection

5.1.9 UART Transmission Delay

A configurable delay (bit [14:7] of parameter 'UARTConf') can be inserted between the reception of a message from the host and the transmission of a response message.

5.2 JCOM Inter-Chip Communication

The JCOM interface is designed to provide a means of point-to-point bi-directional communication for dual-core products between the motor control core running the MCE (named T core hereafter) and the integrated MCU (named A core hereafter). JCOM interface utilizes an internal serial port. JCOM protocol assumes one master and one slave during communication. JCOM interface can be enabled by using bit field [5:3] of the parameter 'TargetInterfaceConf' and parameter 'JCOMConf'.

5.2.1 Operation Mode

JCOM interface supports asynchronous mode between the master and the slave.

5.2.1.1 Asynchronous Mode

In asynchronous mode, the A core (MCU) serves as the master, while the T core (MCE based motor control) serves as the slave. All communication activities are initiated by the master.

From the slave side, JCOM interface driver is interrupt driven to ensure that the response from T core is handled with minimum delay. As soon as enough data is accumulated in the reception FIFO, the JCOM interrupt handler is triggered where the received frame is parsed to extract the message payload. Based on the Message Object

(MO) number, relevant action is executed per the Command and Response Protocol. Then, the response frame is constructed and sent to the transmission FIFO.

5.2.2 Baud Rate

The Baud rate of JCOM interface can be configured at the start-up or during run-time. The valid range is from 6.1 Kbps to 6 Mbps. The default Baud rate is 1 Mbps.

If the T core JCOM interface experiences some frame error more than 3 times due to mismatch of Baud rate configuration between the A core and the T core, then the Baud rate of JCOM interface of the T core would be reset to the default value (1 Mbps) automatically.

5.2.3 Message Frame Structure

Each JCOM message frame consists of the following fields assuming transmission sequence is from left to right. The following Table 20 shows the details of the JCOM message frame structure.

Table 20 JCOM Message Frame Structure

Flag	Seq	Res	MO	Data [0]	Data [1]	Data [2]	Data [3]	Data [4]	Data [5]	CRC	Flag
			Message								
1 byte (0x7E)	2 bit	2 bit	4 bit	6 bytes						1 byte	1 byte (0x7E)

Flag: Indication of the start and end of a frame.

Seq: This sequence number is used to detect a wrong sequence fault. During normal operation, Seq number is incremented per frame and checked at the receiver side. If the Seq number doesn't match, then the entire frame is ignored and no response is sent.

Res: Reserved for future use.

MO: This Message Object number defines how the data is interpreted.

Data [x]: These data fields contain the payload of the message.

CRC: The CRC byte is calculated over the **"Message"** fields including the MO number. If CRC check fails, then the entire frame is ignored and no response is sent.

5.2.4 Command and Response Protocol

The command and response protocol is used when JCOM interface works in asynchronous mode. The message contains a Message Object number and 4 data bytes. Under the 'direction' column found in the following Message Structure figures, 'DS' refers to communication from master (A core) to slave (T core), and 'US' refers to communication from slave (T core) to master (A core). If a command frame sent from the master is successfully received by the slave and passes CRC check, then a corresponding response frame would be sent from the slave. If the command frame sent from the master is out of synchronization due to Seq number mismatch, or fails the CRC check, then the entire command frame is ignored by the slave with no response. Some time-out recovery mechanism is recommended from the master side to deal with those faults. The following Table 21 summarizes the functions corresponding to different MO numbers.

Table 21 Message Object Function Table

Message Object	Functions
0	State machine inquiry; Execution time and CPU load inquiry
1	System configuration protection; Reset T core; Access static parameter; Set boot mode; Set JCOM Baud rate
6	Get parameter
7	Set parameter
8	Get parameter request
Others	Reserved for future use

5.2.4.1 Message Object: 0

The following Table 22 shows the details of the message structure with MO set to 0. With MO = 0, data [0] contains a status byte that represents the type of objects whose status is requested.

Table 22 Message Structure (MO = 0)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[3]	Data[4]	Comments
DS	STATUS	x	x	x	x	x	Status = 0: returns the state number of the SM0 (motor) and SM1 (PFC) state machines.
US	SM0 state	SM1 state	0xFF	0xFF	0xFF	0xFF	Status = 0
US	reserved		reserved		reserved		Status = 1

5.2.4.1.1 State Machine Inquiry

If the status byte = 0 in the command frame, then the relevant state numbers of the motor and PFC state machines are requested by the master. The response frame is supposed to contain the state number (‘Motor_SequencerState’) of the motor state machine in data[0] and the state number (‘PFC_SequencerState’) of the PFC state machine in data[1].

5.2.4.2 Message Object: 1

The following Table 23 shows the details of the message structure with MO set to 1. With MO = 1, the command frame contains a Command word in data[0] and data[1] and a Value word when applicable in data[2] and

data[3]. The response frame is supposed to contain the same Command word in data[0] and data[1] and the same Value word in data[2] and data[3] to acknowledge successful reception.

Table 23 Message Structure (MO = 1)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Comments
	Command		Value		reserved		
System Configuration							
DS	0x0000		p		x		Configuration protection: p = 0: protected 0 < p < 3: unprotected for the next p commands
DS	0x0001		0		x		Reset (immediately)
DS	0x0002		a		x		Static parameter access: a = 0: disable a = 1: enable
DS	0x00BD		(~bmd<<8)+bmd		x		Set boot mode
JCOM Configuration							
DS	0x0100		Baud rate		x		Set JCOM Baud rate
Parameter Handler Commands							
DS	0x0200		x		x		Disable coherent parameter handling
DS	0x0201		x		x		Enable coherent parameter handling
DS	0x0202		x		x		Set parameter coherently and disable coherent parameter handling.
DS	0x0203		x		x		Flush coherent parameter FIFO.
DS	0x0204		x		x		Get parameters coherently.
File Handler Commands							
DS	0x0303		file#		x		Load parameter file with Reset
Response							
US	Command		Value		x		Response, Acknowledge from slave

5.2.4.2.1 System Configuration Protection

Changing system configuration requires going through a 2-step unlock process for safety concerns. Those operations include resetting T core, accessing static parameters, as well as setting boot mode.

The 1st step is to have the master send a command frame (MO = 1) with Command = 0x0000 and Value = p to unprotect the next p commands. p can be set to 1 or 2.

The 2nd step is to have the master send a command frame (MO = 1) with one of those system configuration related commands to change system configuration.

5.2.4.2.2 Reset T Core

A core can perform a reset request for T core by the following steps.

The 1st step is to have the master send a command frame (MO = 1) with Command = 0x0000 and Value = 1 to unprotect the next 1 command.

The 2nd step is to have the master send a command frame (MO = 1) with Command = 0x0001 and Value = 0. Upon receiving this frame, the T core will immediately reset itself with no response US frame.

5.2.4.2.3 Access Static Parameter

Writing to those static type of parameters is not allowed by default. A 2-step unlock process is needed to obtain write access to the static type of parameters. Without going through this process, attempting to write to those static type of parameters would have no effect.

The 1st step is to have the master send a command frame (MO = 1) with Command = 0x0000 and Value = 1 to unprotect the next 1 command.

The 2nd step is to have the master send a command frame (MO = 1) with Command = 0x0002 and Value = 1 to grant write access to those static type of parameters.

Then the master has the right to write to those static type of parameters using a command frame with MO = 7. After the write operation is completed, it is recommended to disable the write access to those static type of parameters by the same 2-step lock process.

The 1st step is to have the master send a command frame (MO = 1) with Command = 0x0000 and Value = 1 to unprotect the next 1 command.

The 2nd step is to have the master send a command frame (MO = 1) with Command = 0x0002 and Value = 0 to disable write access to those static type of parameters.

5.2.4.2.4 Set Boot Mode

By default T core (MCE) operates in Application Mode. A core can request changing the MCE to Configuration Mode (BMD = 0xCD) or Boot-Loader Mode (BMD = 0x5D) by the following steps.

The 1st step is to have the master send a command frame (MO = 1) with Command = 0x0000 and Value = 1 to unprotect the next 1 command.

The 2nd step is to have the master send a command frame (MO = 1) with Command = 0x00BD and Value = 0x32CD to set the boot mode to Configuration Mode, or Value = 0xA25D to set the boot mode to Boot-Loader Mode.

5.2.4.2.5 Set JCOM Baud Rate

The master can request changing the Baud rate of the JCOM interface of the slave by sending a command frame (MO = 1) with Command = 0x0100 and Value = desired Baud rate (bps)/100.

5.2.4.3 Message Object: 6

5.2.4.3.1 Get Parameter

The following Table 24 shows the details of the message structure with MO set to 6. Each parameter can be addressed by using the FB ID and Register ID number which are described in Parameter Reference Manual. With MO = 6, the command frame contains the FB ID byte in data[0] and the Register ID byte in data[1] of the specified parameter or variable. The response frame is supposed to contain the same FB ID byte in data[0], the same Register ID byte in data[1], and the Value word of the requested parameter or variable from data[2] to data[5] to confirm a successful operation. In case of error, the response frame is supposed to contain a value of 0xFF in both data[0] and data[1], and the status value in data[2] and data[3], and the remaining value's data bytes contain invalid data (0xFF).

Table 24 Message Structure (MO = 6)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Comments
DS	FB ID	Register ID	0x00000000				Get parameter
Response							
US	FB ID	Register ID	Value				Send requested parameter
US	0xFF	0xFF	Status		0xFFFF		Status Error: 1: Index Unknown 2: Access not allowed 3: Value out of range 4: Error

5.2.4.4 Message Object: 7

5.2.4.4.1 Set Parameter

The following Table 25 shows the details of the message structure with MO set to 7. With MO = 7, the command frame contains the FB ID byte in data[0], the Register ID byte in data[1], and the Value word in data[2] and data 3] of the specified parameter or variable. The response frame is supposed to contain the same FB ID byte in data[0], the same Register ID byte in data[1], and the same Value word of the requested parameter or variable from data[2] to data[5] to confirm a successful operation. In case of error, the response frame is supposed to contain a value of 0xFF in both data[0] and data[1], and the status value in data[2] and data[3] and the remaining value's data bytes contain invalid data (0xFF).

Table 25 Message Structure (MO = 7)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Comments
DS	FB ID	Register ID	Value				Set parameter
Response							
US	FB ID	Register ID	Value				Send back parameter for confirmation

(table continues...)

Table 25 (continued) Message Structure (MO = 7)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Comments
US	0xFF	0xFF	Status		0xFFFF		Status Error: 1: Index Unknown 2: Access not allowed 3: Value out of range 4: Error

5.2.4.5 Message Object: 8

5.2.4.5.1 Get Parameter Request

The following Table 26 shows the details of the message structure with MO set to 8. Each parameter can be addressed by using the FB ID and Register ID number which are described in Parameter Reference Manual. With MO = 8, the command frame contains the FB ID byte in data[0] and the Register ID byte in data[1] and the Value word of 0 from data[2] to data[5] to confirm a successfully operation. In case of error, the response frame is supposed to contain a value of 0xFF in both data[0] and data[1], and the status value in data[2] and data[3] and the remaining value's data bytes contain invalid data (0xFF).

Table 26 Message Structure (MO = 8)

Direction	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Comments
DS	FB ID	Register ID	Value				Get parameter request
Response							
US	FB ID	Register ID	0x00000000				Operation Status.
US	0xFF	0xFF	Status		0xFFFF		Status Error: 1: Index Unknown 2: Access not allowed 3: Value out of range 4: Error

6 Script Engine

Script engine is a light-weight “C” language like virtual machine running on the MCE. The script engine enables users to implement system level functionalities beyond motor control and PFC function. Key advantages of script engine are:

- Extend capabilities of manipulating additional digital and analog pins that are not used by motor control and/or PFC
- Scalable for any future functional extension beyond motor control and PFC
- Read and write all MCE parameters and variables
- Multi-tasking capability

6.1 Overview

Script code follows 'C'-like syntax. The script engine executes the script code from two different tasks (Task 0 and Task 1) with different priority. The script engine supports arithmetic, binary logical operators, decision statement (if...else statement) and loop statement (FOR statement). In the MCE, 8448Bytes of flash memory is reserved for script byte code and constant data. Consequently, the maximum allowed script byte code size is 8448 Bytes (Approximately 0.8 k lines of code).If the multi parameter management function is used, this space will be further reduced. 256 bytes of data memory is allocated for script global variables and 128 bytes of data memory is allocated for local variables in each task separately.

6.2 Script Program Structure

The script program consists of the following parts:

- Set Commands: Define script user version and script task execution period
- Functions: Script code should be written inside four predefined functions- Script_Task0_init (), Script_Task0 (), Script_Task1_init () and Script_Task1 ()
- Variables, parameters and script methods
- Statement and Expressions: Each individual statement must be ended with a semicolon
- Comments: Starts with a slash asterisk/* and ends with an asterisk slash */for multiple line comments or prefix double slash//to comment single lines

6.3 Script Program Execution

The script engine executes script code from two independent tasks, named Task0 and Task1. Both the tasks are executed periodically.

Global priority of Script language tasks is lower than that of the MCE embedded tasks such as the FOC, PFC tasks, and others. In the other word CPU computation resource is allocated to the MCE first and then to Script language tasks by utilizing the remaining CPU resource of MCE. If the embedded MCE function of FOC and PFC utilizes a full amount of CPU loading (that is high PWM carrier update for the FOC and/or PFC) in a specific application environment, then Script language tasks have no room for their computation. Therefore, the CPU resource availability is highly dependent on a specific application condition.

Task execution period can be configured using "SCRIPT_TASK0_EXECUTION_PERIOD" and "SCRIPT_TASK1_EXECUTION_PERIOD", for each task. Each task has separate initialization functions (Script_Taskx_init ( )) to initialize script variables and MCE parameters. Also, it is possible to write script code inside the initialization function. These init functions are called only once during start-up. Task0/Task1 script functions (Script_Taskx()) are called periodically based on task execution period value.

Among script tasks, Task0 has higher priority than Task1.

For Task0, by default, the execution step is 1 and the execution period is 50 (50 x 1 ms = 50 ms). So, Task0 executes one line of script code or script instruction every 1 ms by default, and starts over the execution of the entire script loop every 50 ms. For Task1, by default, the execution step is 10, the execution period is 10 (10 x 10 ms = 100 ms). So Task1 executes 10 lines of script code or script instruction every 10 ms by default, and starts over the execution of the entire loop every 100 ms.

Total script execution time for Task0 or Task1 can be calculated based on number of script instructions in the script code. For example, if the number of script instructions in Task0 is 20, then by default, Task0 takes 20 ms to finish executing the entire script code. No script code is executed in the remaining 30 ms. After 50 ms, Task0 starts to execute the first script instruction again.

Execution step and execution period of each task is configurable. For example, if Task0 execution period is set to 100 ms (SCRIPT_TASK0_EXECUTION_PERIOD =100), then Task0 execution is repeated every 100 ms.

If Task0 execution period is set to 100 ms (SCRIPT_TASK0_EXECUTION_PERIOD = 100), and number of lines in Task0 is 150, task0 script function takes 150 ms to execute the complete script code once. After finishing execution, it immediately starts over again.

6.3.1 Execution Time Adjustment

As mentioned, Task0 executes one line of script code or script instruction every 1 ms and Task1 executes 10 lines of script code or script instruction for every 10 ms by default. It is possible to increase the number of lines executed by Task0 or Task1 per step, to accelerate the script execution.

Number of lines to be executed every 1 ms in Task0 can be configured in “SCRIPT_TASK0_EXECUTION_STEP”. If Task0 execution period is set to 100 ms (SCRIPT_TASK0_EXECUTION_PERIOD = 100), Task0 number of lines to be executed every 1 ms is set to 2 (SCRIPT_TASK0_EXECUTION_STEP = 2) and number of lines in Task0 is 100. Task0 script function takes 50 ms to execute the complete script code once.

Similarly, in Task1, number of lines to be executed every 10 ms can be configured in “SCRIPT_TASK1_EXECUTION_STEP”.

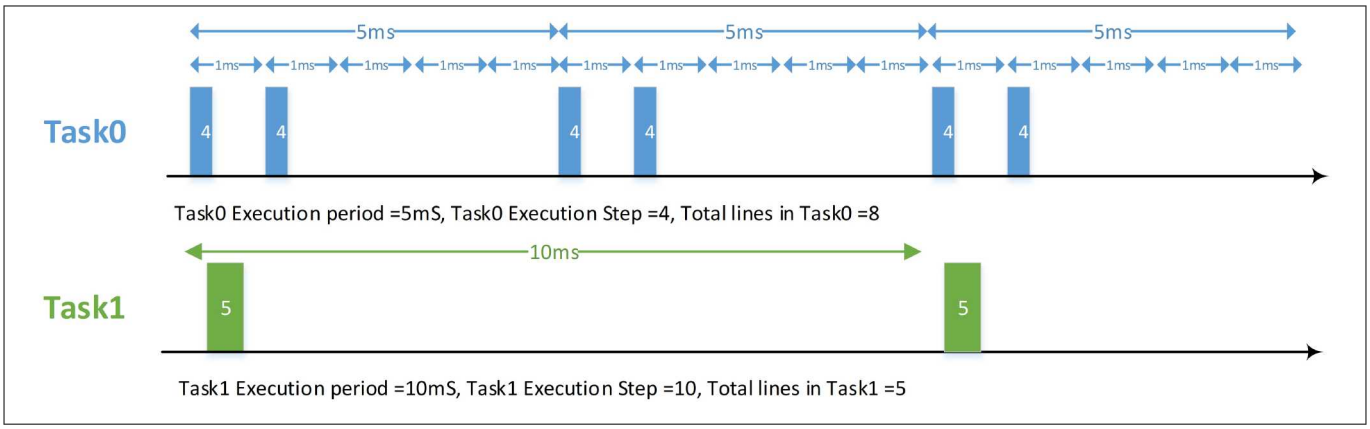


Figure 88 **Script Task Execution**

6.4 Constants

Script supports only integer literals in decimal and hexadecimal representation. Hexadecimal value should be prefixed with 0x. Constant value should not have any suffix, for example U or L.

If any variable is assigned with float literals, digits after the decimal place are ignored by the script translator.

Script translator supports up to 100 constant definitions. To define a constant, use descriptor CONST or const in front of the variable type keyword.

6.5 Variable types and scope

The script engine supports global and local variables. The global variables can be accessed from both tasks and local variables can only be accessed within the respective task.

The script engine supports the following variable types:

Table 27 **Script Variable Types**

Type	Storage Size	Value range	Description
uint8_t	1 byte	0 to 255	Byte length unsigned integer
int8_t	1 byte	-128 to 127	Byte length integer

(table continues...)

Table 27 (continued) Script Variable Types

Type	Storage Size	Value range	Description
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

In the MCE, 256 bytes of data memory is allocated for script global variables and 128 bytes of data memory is allocated for local variables in each task.

Script variable name should only consist of alphanumerical characters and underscore symbol ('_'). Variable name is case-sensitive. All the variable names including global and local should be unique.

Variables declared outside the Task0 or Task1 functions are treated as global variables. Variables declared inside Task0 or Task1 functions are local to Task0 or Task1 respectively.

***Note**: Variable cannot be initialized during declaration unless is a constant variable.*

6.5.1 Script Array define

The script engine also supports the array define and the types of the array should be consistent with previous description

```
uint8_t arr[5];
arr[0] = 10;
arr[1] = 11;
arr[2] = 12;
arr[3] = 13;
arr[4] = 14;
```

As shown above, an array containing 5 elements is defined, and the array type is unit8_t. Of course, user can define corresponding data according to actual application needs

6.5.2 Mapping of Variables to Parameters

The parameter handler provides access to global and local variables of the script engine. The access to different data-types (8 bit, 16 bit, 32 bit) is done by parameter mapping.

The variables memory space is visible in the range 0xF000 to 0xF7FF:

- 0xF000 to 0xF07F local task 0 variables with 8 bit access
- 0xF080 to 0xF0FF local task 1 variables with 8 bit access
- 0xF100 to 0xF17F local task 0 variables with 16 bit access
- 0xF180 to 0xF1FF local task 1 variables with 16 bit access
- 0xF300 to 0xF37F local task 0 variables with 32 bit access
- 0xF380 to 0xF3FF local task 1 variables with 32 bit access
- 0xF400 to 0xF4FF global variables with 8 bit access
- 0xF500 to 0xF5FF global variables with 16 bit access
- 0xF700 to 0xF7FF global variables with 32 bit access

There is no variable range check or data-type check implemented. As a result, the full variables memory space can be accessed in any data-type mapping regardless the declaration in the script source code.

***Note**:* *Access to the gaps (0xF200 to 0xF2FF and 0xF600 to 0xF6FF) is detected and results in an error response of the parameter handler.*

6.6 MCE Parameter Access

All MCE parameters and variables can be accessed from script. Parameters and variables can be used directly in the script code without declaration. Only DYNAMIC type parameters and READWRITE type variables are writable from the script code.

A set of parameters and variables can be updated simultaneously using the coherent update method. Two methods (EnableCoherentUpdate () and DoCoherentUpdate ()) are defined in script to do simultaneous update of parameter and variables.

If Coherent update is enabled (by called EnableCoherentUpdate () method), write operation will not update parameters and variables values immediately. Instead, all the values are stored into a buffer and all parameters and variables are updated simultaneously after calling DoCoherentUpdate (). Script supports simultaneous update of up to 32 parameters and variables (Chapter 6.11.2).

6.7 User defined function

1.3.3 version of the iSD tool or higher supports user defined function functionality in the script. This can fully utilize the limited script space and improve code utilization. Below are examples to illustrate how to define.

```
uint16_t function_uint16_t(uint16_t val)
{
  return val*2;
}
```

This function implements a variable that returns a double value

```
void function_void_t(void)
{
  TestVal = TestVal + 1;
}
```

Alternatively, user can define a function without a return value. For example, in this function, implement the function of increasing variables.Of course, user can define corresponding functions according to the actual application needs.

The order in which scripts compile code is Global.mcs first, then Script_Task0.mcs, and finally Script_Task1.mc. Therefore, when calling a function, please ensure that the function definition is before the call.

6.8 Operators

An operator is a symbol that informs the script to perform a specific mathematical or logical function. A list of operators supported in script function are listed below:

Table 28 Arithmetic Operators

Operator	Description
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides numerator by de-numerator
%	Modulus Operator, remainder after an integer division

Table 29 Binary Operators

Operator	Description
|	Binary OR Operator copies a bit to the result if it exists in either operand
&	Binary AND Operator copies a bit to the result if it exists in both operands
^	Binary XOR Operator copies a bit to the result if it is set in one operand but not both
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand

Table 30 Assignment Operators

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand

Table 31 Relational Operators

Operator	Description
==	Checks if the values of two operands are equal. If yes, then the condition becomes true
!=	Checks if the values of two operands are not equal. If yes, then the condition becomes true
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true

Table 32 Logical Operators

Operator	Description
&&	Logical AND operator used to combine two or more conditions. Operator returns true when both the conditions in consideration are satisfied. Otherwise it returns false

(table continues...)

Table 32 (continued) Logical Operators

Operator	Description
||	Logical OR Operator used to combine two or more conditions. Operator returns true when any one of the conditions in consideration are satisfied. Otherwise it returns false

The precedence and associativity of all the operators in script languages are summarized in the table below.

Table 33 Script Operator Precedence

Precedence	Operator	Description	Associativity
8 Highest	~ -	Bitwise NOT (One's Complement) Unary minus	Right to left
7	* / %	Multiplication Division Modulo (remainder)	Left to right
6	+ -	Addition Subtraction	Left to right
5	<< >>	Bitwise left shift Bitwise right shift	Left to right
4	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right
3	== !=	Equal to Not equal to	Left to right
2	& | ^	Bitwise AND Bitwise OR Bitwise XOR (exclusive or)	Left to right
1 Lowest	&& ||	Logical AND Logical OR	Left to right

The order of precedence can be overridden by using parentheses. Simply enclose within a set of parentheses the part of the equation that needs to be executed first.

6.9 Decision Structures

Decision structures are used for branching. The script engine provides if-statement for decision making. If statements can be followed by an optional else statement, which executes when the Boolean expression is false. Boolean expression can consist of relational operator and logical operators. Syntax of if…else statement in script language is shown below:

If...else statement syntax

```
if(boolean_expression)
{
  /*Statement(s) will execute if the expression is true*/
}
else
{
  /*Statement(s) will execute if the expression is false*/
}
If and else statement should be followed by curly braces
```

Script programming assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value. Depth of nested if condition is limited to 15.

6.10 Loop Structures

The MCE supports FOR-statements for repeat processes. Syntax of FOR statement in script language is shown below:

For statement syntax

```
for(<ScriptVariable> = <Startvalue> : <Endvalue>)
 {
   /*Statement(s) will execute for defined loop time*/
 }
```

Statements inside the for loop are executed for Endvalue - Startvalue+1 time. The FOR statement does not support count down mode (decreasing index). The start value must always be less than end value.

6.11 Methods

Predefined methods are available for specific operations. Methods supported in script functions are described in the following sections.

6.11.1 Bit access Methods

Three methods are defined in the script to read or write particular bit of script variables or motor control/PFC related variables or parameters.

Table 34 Bit Access Methods

Methods	Description
void SET_BIT(<Var>, <bitposition>)	Set the particular bit of variable
void CLEAR_BIT(<Var>, <bitposition>)	Clear the particular bit of variable
uint8_t GET_BIT(<Var>, <bitposition>)	Read the particular bit of variable

***Note**: Bit position value must be 0 to 15.*

6.11.2 Coherent update methods

These methods are used for updating motor control and/or PFC parameters and variables simultaneously.

Table 35 Coherent Methods

Methods	Description
EnableCoherentUpdate()	Enable simultaneous update of parameter/variables
DoCoherentUpdate()	Trigger simultaneous update of parameter/variables

***Note**: Maximum 32 parameters/variables can be updated simultaneously.*

When a coherent update is enabled, values are not updated into parameter/variables immediately. Instead values are stored into buffer and update the actual variable/parameter after trigger the coherent update.

6.11.3 User GPIOs

The Script enables access to digital pins and analog inputs not used by motor control and PFC. Read and write of digital pins is supported and read of analog inputs are supported.

6.11.3.1 Digital Input and Output Pins

Digital pins available to users can be configured as input or output pins. All configured digital input/output pins values are read/write by the script every 1 ms.

Four dedicated variables are defined in the MCE to read or write digital input/output pins.

Variable Name	Type	Description
FB_GPIO.GPIO_Status	READONLY	Holds digital input/output (GPIO0 to GPIO29) pin values
FB_GPIO.GPIO_Set	READWRITE	Sets or resets digital output pin (GPIO0 to GPIO29)

The logic level of a GPIO pin can be read via the read–only registers “FB_GPIO.GPIO_Status”. Read “FB_GPIO.GPIO_Status” register always returns the current logical value of the GPIO pin, regardless of the pin direction (input or output). It is possible to read the complete variable or binary data.

“FB_GPIO.GPIO_Set” register determines the value of a digital pin when it is configured as output. Writing a 0 to a bit position delivers a low level at the corresponding output pin. Likewise, writing a 1 to a bit position delivers a high level at the corresponding output pin. It is possible to read the complete variable or binary data.

6.11.3.2 Analog pins

Analog pins available to the user are read by MCE every 1 ms. The result value is accessible to the script code.

12 dedicated variables are defined in the MCE to read analog input pins value.

Variable Name	Type	Description
FB_ADC.adc_result[0]	READONLY	Holds AIN0 analog input value (12 bit value)
FB_ADC.adc_result[1]	READONLY	Holds AIN1 analog input value (12 bit value)
FB_ADC.adc_result[2]	READONLY	Holds AIN2 analog input value (12 bit value)

Variable Name	Type	Description
FB_ADC.adc_result[3]	READONLY	Holds AIN3 analog input value (12 bit value)
FB_ADC.adc_result[4]	READONLY	Holds AIN4 analog input value (12 bit value)
FB_ADC.adc_result[5]	READONLY	Holds AIN5 analog input value (12 bit value)
FB_ADC.adc_result[6]	READONLY	Holds AIN6 analog input value (12 bit value)
FB_ADC.adc_result[7]	READONLY	Holds AIN7 analog input value (12 bit value)
FB_ADC.adc_result[8]	READONLY	Holds AIN8 analog input value (12 bit value)
FB_ADC.adc_result[9]	READONLY	Holds AIN9 analog input value (12 bit value)
FB_ADC.adc_result[10]	READONLY	Holds AIN10 analog input value (12 bit value)
FB_ADC.adc_result[11]	READONLY	Holds AIN11 analog input value (12 bit value)

6.12 Plugins

6.12.1 Configurable UART

The MCE firmware includes an UART interface that consists of a plug-in of the scripting engine and script APIs. Based on UART API provided an UART communication protocol can be implemented.

To find out what pins the device's pins are, refer to your device's datasheet.

Table 36 Configurable UART API

API name	Brief description
UART_DriverInit()	Initializes the UART hardware driver
UART_DriverDeinit()	De-initializes the UART hardware driver
UART_FifoInit()	Initialize UART hardware FIFO
UART_BufferInit()	Initialize UART software buffer
UART_GetStatus()	Get the status word for the UART communication status
UART_GetRxDelay()	Returns the delay time between receive frames
UART_Control()	Writes to the Control Word that defines UART control commands
UART_RxFifo()	Returns one byte from the receive FIFO
UART_TxFifo()	Puts one byte to the transmit FIFO
UART_RxBuffer()	Returns one byte from the receive buffer from a specified location
UART_TxBuffer()	Puts one byte in the transmit buffer at a specified location

6.12.1.1 UART_DriverInit()

Declaration:

```
void UART_DriverInit(channel, rxInvert, txInvert, baudrate, dataBits, stopBits)
```

Input Parameters	Min	Max	Description
channel	0	1	Selects which UART channel to be used. 0: UART 0 1: UART 1
rxInvert	0	1	Configures the data interpretation logic for the received data. 0: non-inverting 1: inverting
txInvert	0	1	Configures the data interpretation logic for the transmitted data. 0: non-inverting 1: inverting
baudrate	600 bps	115,200 bps (230,400 bps in FIFO mode)	Configures the baudrate for the UART in bits-per-second
dataBits	5 bits	8 bits	Configures the length of the data bits in a UART byte
parity	0	3	Select parity: 0: no parity 2: even parity 3: odd parity
stopBits	1 bit	2 bits	Configures the number of stop bits in a UART byte. 1: 1 stop bit 2: 2 stop bits

Description:
This API initializes the UART driver.

6.12.1.2 UART_DriverDeinit()

Declaration:

```
void UART_DriverDeinit(void)
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:
This API deinitializes the UART driver.

6.12.1.3 UART_FifoInit()

Declaration:

```
void UART_FifoInit(rxFifoSize, txFifoSize)
```

Input Parameters	Min	Max	Description
rxFifoSize	1 byte	31 bytes	Size of the FIFO buffer allotted for receive in bytes
txFifoSize	1 byte	31 bytes	Size of the FIFO buffer allotted for transmit in bytes

Description:

This API initializes the UART FIFO.

6.12.1.4 UART_BufferInit()

Declaration:

```
void UART_BufferInit(halfDuplex, rxTimeout, txDelay, txByteDelay, rxFlag, txFlag, rxDataLength,
txDataLength)
```

Input Parameters	Min	Max	Description
halfDuplex	0	1	Configure the UART buffer for half or full duplex communication. 0: Full duplex 1: Half duplex
rxTimeout	0	65535	Configure the longest expected time to receive a frame. If a frame is not received within this time an RxTimeout will occur
txDelay	0	65535	Configure the delay time from having received a frame and starting to transmit a frame in ms
txByteDelay	0	65535	Configure the delay time between each byte in a transmit frame
rxFlag	0	65535	RxFlag is a byte that signifies the beginning of a receive frame. 0-255: valid flag byte 256-65535: invalid flag/no flag byte is used

Input Parameters	Min	Max	Description
txFlag	0	65535	TxFlag is a byte that signifies the beginning of a transmit frame. 0-255: valid flag byte 256-65535: invalid flag/no flag byte is used
rxDataLength	1 byte	8 bytes	Configure the length of the receive frame, in bytes, not including the start flag byte
txDataLength	1 byte	8 bytes	Configure the length of the transmit frame, in bytes, not including the start flag byte

Description:
This API configures the UART software buffer.

6.12.1.5 UART_GetStatus()

Declaration:

```
int32_t UART_GetStatus(void)
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Returns the status word whose bitfield representation is described below **FIFO status:** Bit 0 – **IsRxFIFOEmpty:** is receive FIFO empty bit • 0: receive FIFO is not empty • 1: receive FIFO is empty Bit 1 – **IsRxFIFOFull:** is receive FIFO full bit • 0: receive FIFO is not full • 1: receive FIFO is full Bit 2 – **IsTxFIFOEmpty:** is transmit FIFO empty bit • 0: transmit FIFO is not empty • 1: transmit FIFO is empty Bit 3 – **IsTxFIFOFull:** is transmit FIFO full bit • 0: transmit FIFO is not full • 1: transmit FIFO is full Bit 4:7- **reserved:** read as ‘0’ **Buffer status:** Bit 8 – **IsRxBufferFull:** is receive buffer full bit • 0: receive buffer is not full • 1: receive buffer is full Bit 9 – **reserved:** read as ‘0’ Bit 10 – **IsTxBufferEmpty**: is transmit buffer empty bit • 0: transmit buffer is not empty • 1: transmit buffer is empty Bit 11:14 – **reserved:** read as ‘0’ Bit 10 – **IsBufferMode:** is Buffer Mode Initialized bit • 0: the frame buffer and the driver handler is not initialized • 1: the frame buffer and the driver handler is initialized **Handler status:** Bit 16 – **IsRxTimeout:** is receive frame timeout bit • 0: receive frame is not timed out • 1: receive frame is timed out Bit 17 – **IsCollision:** is collision detected bit • 0: collision is not detected • 1: collision is detected Bit 18:19 – **HandlerState:** Handler state bitfield • 00: FRAME_START • 01: FRAME_RECEIVE • 10: FRAME_DELAY • 11: FRAME_TRANSMIT Bit 20:22- **reserved:** read as ‘0’ Bit 23 – **IsHalfDuplex:** is half duplex bit

Return Type	Description
	• 0: the driver handler is not initialized in half-duplex mode • 1: the driver handler is initialized in half-duplex mode **Driver status:** Bit 24 – **IsRxNoiseDetected:** is receive noise detected bit • 0: noise on the receive line has not been detected • 1: noise on the receive line has been detected Bit 25 – **IsParityError:** is parity error bit • 0: a parity error has not occurred • 1: a parity error has occurred Bit 26 – **IsStopBitError:** is stop bit error • 0: a stop bit error has not occurred • 1: a stop bit error has occurred Bit 27:30 – **reserved:** read as ‘0’ Bit 31 – **IsInitialized:** is initialized bit • 0: the driver is not initialized • 1: the driver handler is initialized

Description:
This API returns the status word.

6.12.1.6 UART_GetRxDelay()

Declaration:

```
int32_t UART_GetRxDelay(void)
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return type	Description
int32_t	Returns the time, in ms, between receive frames. Timing begins from the last byte of the current receive frame and ends at the first byte of the next receive frame

Description:
This API returns the delay, in ms, between receive frames.

6.12.1.7 UART_Control()

Declaration:

```
void UART_Control(command)
```

Input Parameters	Description
command	Writes the Control Word whose bit field representation is described below. **FIFO control:** Bit 0 – **reserved:** Bit 1 – **ClrRxFIFO:** Clear RX FIFO bit • 0: N/A • 1: clear the receive FIFO Bit 2 – **reserved:** Bit 3 – **ClrTxFIFO:** Clear TX FIFO bit • 0: N/A • 1: clear transmit FIFO Bit 4:7 – **reserved:** **Buffer control:** Bit 8 – **ClrRxBufferFlag**: Clear RX Buffer flag bit • 0: N/A • 1: clear receive buffer flag Bit 9 – **reserved**: Bit 10 –**SendTxBuffer**: Send TX Buffer flag • 0: N/A • 1: Initiate the sending of bytes from the transmit buffer through the specified UART channel. Bit 11:15 – **reserved**: **Handler control:** Bit 16 – **ClrRxTimeoutFlag**: Clear RX time-out Flag bit • 0: N/A • 1: clear receive time-out flag Bit 17 – **ClrCollisionFlag**: Clear Collision detected Flag bit • 0: N/A • 1: clear collision detection flag Bit 18 – **RstBufferControl**: Reset Buffer Control bit • 0: N/A • 1: reset buffer control state machine Bit 19:23 – **reserved**: **Driver Control:** Bit 24 – **ClrRxNoiseFlag**: Clear RX Noise Flag bit • 0: N/A • 1: clear the receive noise flag Bit 25 – **ClrParityErrorFlag**: Clear Parity Error Flag bit • 0: N/A • 1: clear the parity error flag Bit 26 – **ClrStopbitErrorFlag**: Cleare Stop bit Error Flag bit

Input Parameters	Description
	• 0: N/A • 1: clear the stop bit error flag Bit 27:31 – **reserved**:

Description:
This API controls the UART's buffer mode, FIFO mode, driver control, and handler control.

6.12.1.8 UART_RxFifo()

Declaration:

```
int32_t UART_RxFifo(void)
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Returns one byte from the receive FIFO

Description:
This API returns one byte of data from the receive FIFO in First In First Out order

6.12.1.9 UART_TxFifo()

Declaration:

```
void UART_TxFifo(data)
```

Input Parameters	Min	Max	Description
data	0	255	One byte of data placed in the transmit FIFO

Description:
This API pushes data into the transmit FIFO in First In First Out order.

6.12.1.10 UART_RxBuffer()

Declaration:

```
int32_t UART_RxBuffer(int32_t idx)
```

Input Parameters	Min	Max	Description
idx	0	7	Specifies which byte of data in the receive buffer to return

Return Type	Description
int32_t	Returns one byte from the receive buffer specified by idx

Description:

This API returns one byte of data from the receive buffer. In buffer mode one can select which byte of data to be returned by specifying the byte using idx.

6.12.1.11 UART_TxBuffer()

Declaration:

```
void UART_TxBuffer(idx, data)
```

Input Parameters	Min	Max	Description
idx	0	7	Specifies at which index to place one byte of data in the transmit buffer
data	0	255	One byte of data to be placed at index idx in the transmit buffer

Description:

This places one byte of data into the transmit buffer. In Buffer mode one can place the byte of data anywhere in the buffer specified by idx.

6.12.2 Flash Data Storage

The MCE firmware includes an interface that allows the user access to the embedded flash of the device. The user is allowed access to 160 bytes of storage in which script variables can be persistent stored. This interface consists of a plug-in to the script engine, providing the user a set of APIs for easy use. Therefore, this interface is compatible with any iMOTION ™ device that supports the scripting feature.

6.12.2.1 Flash Data Type

Script variables declared with the keyword ‘flash’ can be stored in flash upon request from the user. The script variables will be initialized by stored flash value at initialization. All variable types, supported by scripting, can be declared as flash variables and both local and global types are supported. To declare a flash variable, use the syntax ‘`flash varType varName`’, for example:

```
flash uint8_t FlashVar1;
flash uint16_t FlashVar2;
flash int32_t FlashVar3;
```

This specifies that this variable will be stored in flash after the Flash_Write() API is called. If a variable with the same name has already been stored, the stored value will be assigned to this variable.

Up to 160 bytes of flash variables are supported in any combination of script variable types. For example, 160 8-bit size variables, 80 16-bit size variables, 40 32-bit size variables, or any size mix of variables.

When flash is empty, a variable of type ‘flash’ will be initialized to 0 after reset. When flash data is invalid, no content will be loaded from flash. It is up to the user to handle the correct initialization of variables in these situations.

During execution of the script, the user can write to flash variables at any point. However, the content of that variable will not be committed to flash until the user calls the API Flash_Write(). Only then, the variable is stored in flash. It is up to the user to decide when an appropriate time to commit to flash is. At initialization, the variable is assigned the value last committed to flash.

6.12.2.2 Flash Data Storage APIs

The APIs of the Flash Data Storage plug-in are summarized in the table below.

API name	Brief description
Flash_Write()	Writes all “flash” type variables to flash
Flash_Erase()	Erases all data in allocated storage
Flash_GetWriteCount()	Returns amount of times flash has been written over lifetime
Flash_GetStatus()	Returns status from flash driver

6.12.2.3 Flash_Write()

Declaration:

```
int32_t Flash_Write()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0 – Write status 0: Write success 1: Write failure

Description:

Writes all “flash” type variables to flash and returns number based on success or failure to write.

***Note**: Motor/PFC must be stopped before calling.*

6.12.2.4 Flash_Erase()

Declaration:

```
int32_t Flash_Erase()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0 – Erase status 0: Erase success 1: Erase failure

Description:

Erases all data in storage allocated for data storage interface. Returns status based on success of erase cycle. Note Flash_Erase() erases content of the flash data storage only. Variables of type flash are initialized after reset and the erase value will not be assigned to the flash variables until next initialization.

6.12.2.5 Flash_WriteCount()

Declaration:

```
int32_t Flash_WriteCount()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0-31 – Number of writes

Description:

Returns the number of times flash has been written. Write Counter is cleared when flash is erased.

6.12.2.6 Flash_GetStatus()

Declaration:

```
int32_t Flash_GetStatus(void)
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0 – **FlashErased:** Indicates if flash has been erased • 0: data in flash • 1: flash has been erased Bit 1 – **FlashInvalid:** Indicates if flash is corrupt • 0: flash content is valid • 1: flash is corrupt Bit 2 – **isFlashWriteError:** Indicates if flash failed to write • 0: no error exists • 1: flash failed to write Bit 3:31 – **reserved**

Description:

Returns status message from Flash Data Storage driver. Status of Flash that has never been used or flash that has been erased will show that 'Flash has been erased' (bit 0). If checksum of stored data does not match content, status will show 'Flash content is not valid' (bit 1). It is up to the user to initialize flash variable appropriately when flash is empty or not valid.

6.12.2.7 Timing Considerations

The APIs Flash_Write and Flash_Erase() are blocking meaning all other tasks and functions will not be serviced while performing flash write/erase. Calling these APIs must be done at non-critical times and tightly synchronized to the system state. Before calling Flash_Write and Flash_Erase(), it is up to the user to make sure the motor and PFC (if supported) are in a stopped and passive state. Therefore, the user can only use the data storage interface to write and erase while the motor is stopped-, fault or standby state.

Flash write time is determined by the carateristics of the embedded flash and of how many variables are written. Maximum write time of flash wariables is 8.6 ms.

The erase process takes 7.0 ms and leaves the content of flash initialized to 0xF's. When flash is empty, variable of type 'flash' will be initialized to 0 after reset.

6.12.2.8 Endurance Considerations

The Flash Data Storage utilizes the MCE's embedded flash which has less endurance than traditional EEPROM. Maximum numbers of writes supported is 50000 and it is up to be user to keep track of number of write cycles using the API Flash_WriteCount() and make sure the endurance of the flash is not exceeded.

6.12.3 Infrared Interface

The MCE firmware includes an IR interface that consists of a plug-in of the scripting engine and script APIs. This allows IR signals to be interpreted directly from an IR sensor, as long as the transmitter's protocol is supported. This can, for example, be used for setting the motor speed based on the press of a remote's button, or customized for setting of MCE parameters. This can be done by creating a simple script, then connecting an IR sensor to the chosen device input pin.

6.12.3.1 Infrared Protocols

The IR Interface supports the following protocols: NEC, NEC Extended, RC-5 Phillips. The protocols are characterized by:

Protocol	Number of bits	Order of transmitted parts	Length of each transmission	Carrier Frequency
NEC	32	Address, inverted address, command, inverted command	67.5 ms	38.222 kHz
NEC extended	32	Address low 8 bits, address high 8 bits, command, inverted command	67.5 ms	38.222 kHz
Phillips RC-5	12	1 toggle bit + 5 address bits + 6 command bits	24.892 ms	36.0 kHz

NEC, NEC Extended, and RC-5 operate with 'regular data frames' for transmission of commands and with 'repeat frames' for transmission of a repeated command. Both types of frames are supported by the IR script plug-in.

6.12.3.2 IR Pins

The MCE supports IR data on 3 different pins: IR0, IR1 and IR2 . Not all options are available on all devices. The user must select one of these when utilizing the IR interface. The pins are enabled and assigned IR-function through the API IR_DriverInit().

When using the IR interface make sure other pin functions colliding with the IR pin are disabled.

Refer to the latest version the datasheet of the device for pin assignment.

6.12.3.3 Infrared Interface APIs

The APIs of the Infrared Interface plug-in are summarized in the table below.

API name	Brief description
IR_DriverInit()	Initializes IR Driver based on key parameters
IR_DriverDeinit()	De-initializes IR Driver
IR_RxBuffer()	Returns most recent transmission
IR_GetStatus()	Returns status
IR_RxCommand()	Returns “Command” section of transmission
IR_RxAddress()	Returns “Address” section of transmission
IR_RxRepeats()	Returns numbers of transmissions repeated
IR_RxReceived()	Returns true if transmission has been received
IR_RxRepeating()	Returns true if transmission has not been fully received

6.12.3.4 IR_DriverInit()

Declaration:

```
int32_t IR_DriverInit(channel, rxInvert, protocol, address)
```

Input Parameters	Min	Max	Description
channel	0	2	Specifies the channel (pin) to received IR data. Availability of pins depends on the device. 0: IR0 1: IR1 2: IR2
rxInvert	0	1	Indicates if the IR receiver sensor is sending the signal inverted or non-inverter. 0: Disable invert 1: Enable Invert
protocol	0	2	Selects the IR protocol 0: RC-5 1: NEC 2: NEC Extended
address	0	65535	Configure the address to use. Must match transmitter (remote) address. Range varies based on protocol

Return Type	Description
int32_t	Initialization status. 0 – Driver successfully initialized 1 – IR driver not available 2 – Protocol recognized 3 - Address out of range

Description:
Initializes driver and related peripherals.

6.12.3.5 IR_DriverDeinit()

Declaration:

```
int32_t IR_DriverDeinit()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Returns success/failure of API execution 0 – Driver successfully de-initialized 1 - IR driver was not available

Description:
De-initializes driver and peripherals.

6.12.3.6 IR_RxBuffer()

Declaration:

```
int32_t IR_RxBuffer()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Returns raw data buffer. Data are organized according to protocol. **Phillips RC-5** • Bit 0 – toggle bit • Bit 1:5 – address • Bit 6:11 – command **NEC** • Bit 0:7 – address • Bit 8:15 – address • Bit 16:23 – command • Bit 24:31 – command **NEC extended** • Bit 0:15 – address • Bit 16:23 – command • Bit 24:31 – command

Description:

This API returns the raw content of the data buffer without any interpretation. Calling the API clears bit 2 and 3 of IR_GetStatus() return value.

6.12.3.7 IR_GetStatus()

Declaration:

```
int32_t IR_GetStatus()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	IR interpreted signal status Bit 0:1 – **ProtocolSelected**: • 0: Phillips RC-5 • 1: NEC • 2: NEC Extended Bit 2 – **IsRawDataAvailable**: • 0: no data received • 1: data received Bit 3 – **IsRawDataValid**: • 0: invalid IR signal received or received address does not match configured address • 1: valid IR signal received and received address matches configured address Bit 4 – **IsDataAvailable**: • 0: invalid IR signal received or received address does not match address • 1: valid IR signal received and received address matches configured address Bit 5 – **IsReceiving**: • 0: waiting for IR transmission • 1: target is currently receiving data Bit 6 – **IsAddressIncorrect**: • 0: no mismatch • 1: received address doesn't match address defined in IR_DriverInit() Bit 7:30 – **reserved** Driver Status Bit 31 – **IsInitialized**: • 0: driver not initialized • 1: driver Initialized

Description:

Returns status of IR driver. Note that status bits IsRawDataAvailable and IsRawDataValid are cleared when calling API IR_RxBuffer(). Staus bit IsDataAvailable is cleared when calling API IR_RxCommand().

6.12.3.8 IR_RxCommand()

Declaration:

```
int32_t IR_RxCommand()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	The API returns the command of a full frame when address and protocol match configuration. On repeat codes, the last valid command is kept as return value

Description:

Returns command section of transmission as long as the protocol and address match configuration. The API clears bit 4 of IR_GetStatus() return value.

6.12.3.9 IR_RxAddress()

Declaration:

```
int32_t IR_RxAddress()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	The API returns the address of a full frame when address and protocol match configuration. On repeat codes, the last valid address is kept as return value

Description:

Returns address section of IR transmission.

6.12.3.10 IR_RxRepeats()

Declaration:

```
int32_t IR_RxRepeats()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Contains the number of transmissions repeated after IR remote's button has been released. If target is currently receiving, 0 is returned. Calling this API clears the return value to 0

Description:

Returns the number of transmissions repeated after IR remote's button has been released. This API allows for script to respond according to how long a button is pressed, such as holding an "increase" button.

6.12.3.11 IR_RxReceived()

Declaration:

```
int32_t IR_RxReceived()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0 – Receive status 0: IR message frame has not been received 1: IR message frame has been received

Description:

Returns 1 after the first frame of a new transmission is received. Calling IR_RxCommand automatically clears the return value of this API.

6.12.3.12 IR_RxRepeating()

Declaration:

```
int32_t IR_RxRepeating()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit 0 – Repeat status 0: Transmission is not repeating (has stopped) 1: Transmission is repeating (button is held down)

Description:

Indicates a command is currently repeating via a button being held down. Returns 0 when the transmission stops.

6.12.3.13 Timing Considerations

The IR buffer update period is 50 ms and it is recommended the script checks for new data at a rate similar or faster than this. This is to ensure no data are lost in cases where commands are transmitted at a high rate. Practical limitations, such as the implausibility of pushing a remote button every 50 ms, may enable lower update rates.

6.12.3.14 Configuration

In order to configure the IR interface, scripting shall be utilized. The process of programming a script to a device is described in the app note “How to use iMOTION™ Script Language”. The first step in configuring the IR interface is to use the IR_Driverinit(), an API that initializes the IR driver. This API let's the user choose which device input pin the IR receiver sensor will be connected to, which IR protocol will be read, and what address will be sent by the transmitter. After executing this API intrepered commands sent by the transmitter are read with IR_RxCommand() and by using conditional statements.

6.12.4 I2C Interface

The MCE firmware provides a way to communicate directly with I2C enabled devices. Devices can communicate at data rates of 100 kHz or 400 kHz and is compatible with devices that have 10-bit or standard 7-bit addresses. This interface is made up of a plug-in to the script engine where users may utilize APIs that deal with sending signals and accessing the FIFO where the firmware collects data sent back to the device.

Because of the low-level nature of this interface it is necessary to understand how your specific device utilizes I2C.

Refer to the latest version of device’s datasheet for pins assignment.

6.12.4.1 I2C Interface API

The APIs of the I2C Interface plug-in are summarized in the table below:

Table 37 I2C API

API name	Brief description
I2C_DriverInit()	Initializes the I2C driver and specifies data rate and address size
I2C_DriverDeInit()	De-initializes the driver and peripheral
I2C_MasterStart()	Sends a start condition and target device address
I2C_MasterRepeatedStart()	Sends a start condition after communication has already started
I2C_MasterStop()	Sends stop condition
I2C_Transmit()	Transmits one byte
I2C_GetDataACK()	Returns received data and sends back an acknowledge bit
I2C_GetDataNACK()	Returns received data without sending back an acknowledge bit
I2C_GetRxFifo()	Returns byte most recently received
I2C_GetStatus()	Provides the status for FIFO and I2C driver
I2C_Control()	Clears FIFO and I2C driver errors

6.12.4.2 I2C_DriverInit()

Declaration:

```
int32_t I2C_DriverInit(dataRate, addressMode);
```

Input Parameters	Min	Max	Description
`dataRate`	0	1	Select the data rate 0: standard mode 100k Hz 1: fast mode 400 kHz
`addressMode`	0	1	Select the device address size 0: standard mode 7-bit 1: 10-bit

Return Type	Description
`int32_t`	Initialization status 0 – Driver was initialized successfully 1 – Device does not support I2C 4 – Speed mode/address mode is invalid

Description:

Initializes I2C driver with given data rate and address mode. Internally, this API first de-initializes the I2C driver and then initializes the driver and peripheral.

6.12.4.3 I2C_DriverDeInit

Declaration:

```
int32_t I2C_DriverDeInit();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	De-initialization status 0 – Driver was de-initialized successfully 1 – Device does not support I2C

Description:

De-initializes the I2C driver and peripheral.

6.12.4.4 I2C_MasterStart()

Declaration:

```
int32_t I2C_MasterStart(address);
```

Input Parameters	Min	Max	Description
`address`	0	N/A	I2C target device address

Return Type	Description
`int32_t`	Bit 0 – Returns transmission status 0: Successful transmission 1: Driver not available

Description:

Sends the start condition and address. Use this API only when the iMOTION ™ device is serving as the master.

6.12.4.5 I2C_MasterRepeatedStart()

Declaration:

```
int32_t I2C_MasterRepeatedStart(address);
```

Input Parameters	Min	Max	Description
`address`	0	N/A	I2C target device address

Return Type	Description
`int32_t`	Bit 0 – Returns transmission status 0: Successful transmission 1: Driver not available

Definition:

Use this API after a start condition has already been sent and before a stop condition has been sent.

6.12.4.6 I2C_MasterStop()

Declaration:

```
int32_t I2C_MasterStop();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	Bit 0 – Stop condition status 0: Successful transmission 1: Driver not available

Definition:

Sends stop condition to I2C target device.

6.12.4.7 I2C_Transmit()

Declaration:

```
int32_t I2C_Transmit(data);
```

Input Parameters	Min	Max	Description
`data`	0	0xFF	Data to be interpreted by target device

Return Type	Description
`int32_t`	Bit 0 – Transmission status 0: Successfully transmitted 1: Driver not available

Definition:

Sends one byte to I2C target device.

6.12.4.8 I2C_GetDataACK()

Declaration:

```
int32_t I2C_GetDataACK();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	Reads received data or an error code if an error occurred -1 – Driver is not available -2 – No response; waiting time timeout

Definition:

Returns data received from other I2C devices and sends back an acknowledge bit. This API is a blocking function; the script will not move on until data is received or the timeout time is reached. Data will successfully return if data is received prior to this API as it will be stored in the interface's FIFO. If there is no data in FIFO the device will wait 250 µs before returning an error code.

6.12.4.9 I2C_GetDataNACK()

Declaration:

```
int32_t I2C_GetDataNACK();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	Returns data received or error code if an error occurred. -1 – Driver is not initialized. -2 – No response; waiting time timeout

Definition:

Returns data received from other I2C devices and does not send back an acknowledge bit. This API is a blocking function; the script will not move on until data is received or the timeout time is reached. Data will successfully return if data is received prior to this API as it will be stored in the interface's FIFO. If there is no data in FIFO the device will wait 250 µs before returning an error code.

6.12.4.10 I2C_GetRxFifo()

Declaration:

```
int32_t I2C_GetRxFifo();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	Returns data received or error code if an error occurred -1 – Driver is not initialized -2 – No response; waiting time timeout

Definition:

This should be called when I2C_GetStatus() is either indicating the reception buffer full or not empty.

6.12.4.11 I2C_GetStatus()

Declaration:

```
int32_t I2C_GetStatus();
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
`int32_t`	**FIFO status:** Bit 0 – **IsReceiveBufferEmpty** • 0: The receive buffer is not empty. Use I2C_GetRxFifo API to read the data • 1: The receive buffer is empty Bit 1 - **IsReceiveBufferFull** • 0: The receive buffer is not full. It could be empty or contain data, but not full. Use I2C_GetRxFifo API to read the data • 1: The receive buffer is full Bit 2 - **IsTransmitBufferEmpty** • 0: Transmit buffer contains data to be transmitted. The transmit buffer is not empty • 1: The transmit buffer is empty Bit 3 - **IsTransmitBufferFull** • 0: The transmit buffer is not full. It could be empty or contain data, but not full • 1: The transmit buffer is full Bit 1:7 – **reserved:** **Driver status:** Bit 24 – **Arbitration lost:** • 0: No arbitration lost error in the transmission • 1: Arbitration lost error in the transmission Bit 25 – **I2C error:** • 0: No error in the transmission • 1: Error in the transmission Bit 26 – **No response:** • 0: Target device responded properly • 1: Target device did not respond Bit 27:30 – **reserved:** Bit 31 – **Initialized:** • 0: Driver is not initialized • 1: Driver is initialized

Definition:

Provides the status for FIFO and driver error. Use the I2C_Control() API to clear arbitration lost error, I2C error, and no response errors. FIFO status are clear automatically.

6.12.4.12 I2c_Control()

Declaration:

```
int32_t I2C_Control(status);
```

Input Parameters	Min	Max	Description
`status`	0	0x7000000	Control Bits: • Bit 0-23 - Reserved • Bit 24 - Arbitration Lost - 0 - There is no I2C arbitration lost error in the transaction - 1 - There is an I2C arbitration lost error in the transaction • Bit 25 - I2C Error - 0 - There is no I2C error (frame format) in the transaction - 1 - There is an I2C error (frame format) in the transaction • Bit 26 - Clear both RX/TX FIFO buffers - 0 - Do not clear both Rx/Tx FIFO buffers - 1 - Do clear both Rx/Tx FIFO buffers • Bit 27-31 - Reserved

Return Type	Description
`int32_t`	Returns success/failure of API execution 0 – Successfully cleared driver status error 1 – Driver is not available 4 – Mode is invalid

Definition:

Clears driver status errors. There are three driver status errors: arbitration lost error, I2C error, and no response error. Clearing the no response error also serves as a way to clear the FIFO.

6.12.5 TRIAC Control

The MCE supports TRIAC based voltage control through a script plug-in. With TRIAC control, the turn-on instance is delayed with respect to the zero-crossing of the AC-line voltage and thereby reduces the RMS voltage across the load. Typically, TRIAC control is handled by a separate microcontroller, but the MCE integrates the

feature and only a few external components are required. The TRIAC plug-in offers a set a APIs to implement custom control function in scripting. TRIAC control is supported in Stand-by mode.

The MCE based TRIAC control system is shown in Figure 89. An AC voltage, V_{AC}, is feeding a load through a TRIAC. The TRIAC's gate pulse is supplied by a Gate Drive circuit which is driven by a gate pulse from the MCE. To position the gate pulse correctly with respect to the AC-line, the MCE requires information about the AC Polarity. That signal is generated by a Zero-Cross Detection circuit.

The TRIAC gate and the zero-cross detection circuits are reference to the AC-line potential which is typically not the reference potential for the MCE. For that reason, an isolation barrier is needed, for example based on opto-couplers.

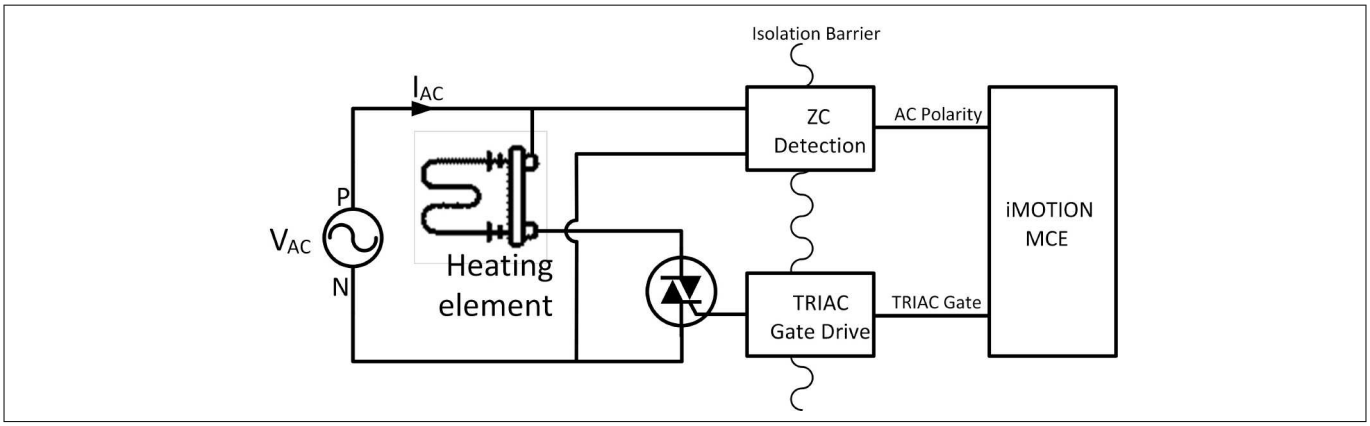


Figure 89 TRIAC Control of a Heating Element

A timing diagram of the TRIAC control is shown in Figure 90. The AC-line, V_{AC}, is a 50/60 Hz AC voltage. A Zero-Cross detection circuit monitors this AC voltage and generates the signal AClineState. Low-high and high-low transitions of the AClineState signal indicates zero-cross of the AC-line.

The MCE generates a TRIAC Gate pulse which is delayed offTime with respect to the zero-crossing of the AC-line and with a pulse width of onTime. The gate pulse turns the TRIAC on and current, I_{AC}, starts to flow through the load. The TRIAC turns off by itself when the current has decayed to zero and the sequence starts over.

At offTime = 0 the load sees the full AC-line voltage but as offTime is increased, the voltage across the load is reduced.

The required pulse width of the gate pulse is device specific. With too short a time, the TRIAC may fail to tun fully ON.

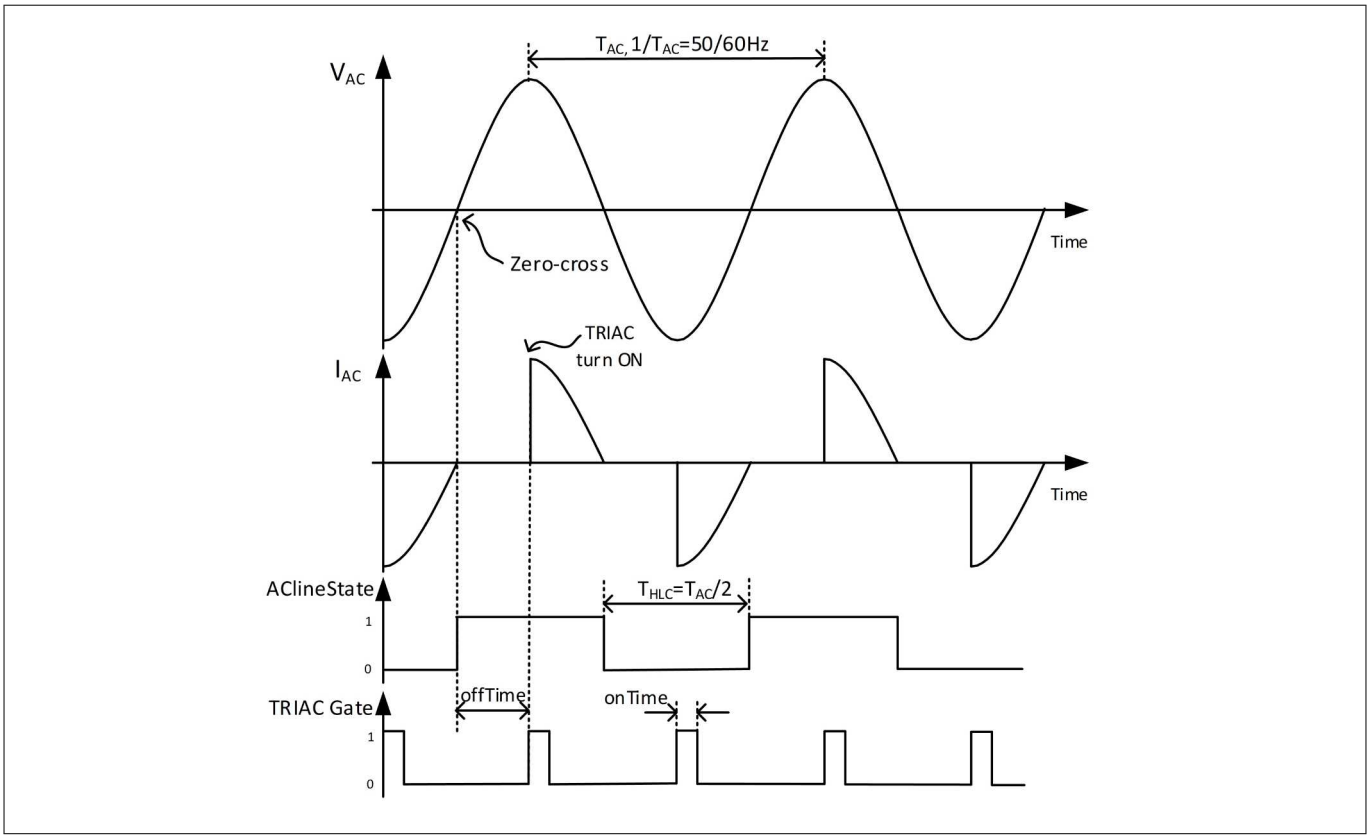


Figure 90 Phase Control Of a TRIAC

The user can set the offTime and onTime with the API TRIAC_SetONOFFTime() at any point during a half-line cycle. The requested times take effect at the following zero-cross of the AC-line as shown in Figure 91.

By latching the onTime and offTime at AC-line zero-crossing the update rate is kept constant at half the AC-line frequency. The MCE handles all the time critical tasks and ensure the correct synchronization to the AC-line. The onTime and offTime can be set with a resolution of 1 micro-second. The maximum allowed onTime and offTime is 21.84 milli-seconds.

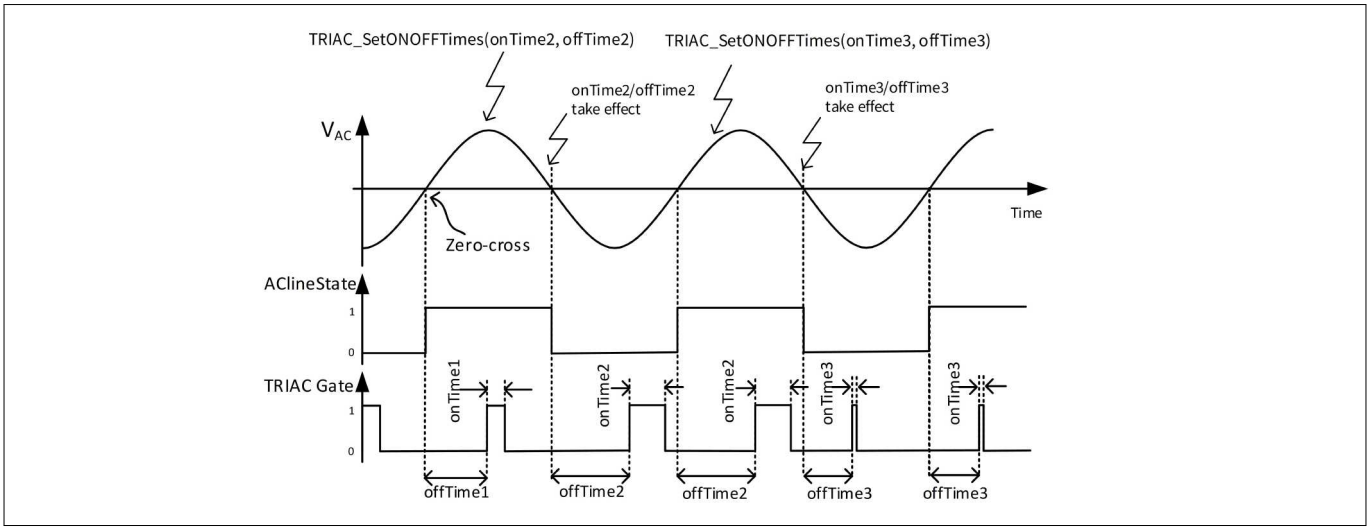


Figure 91 Setting and Latching of Gate Pulse Width and Turn-on Delay

With TRIAC control, a new cycle starts at every zero-cross of the AC-line voltage. If the requested offTime and onTime do not fit within a half line cycle (T_{HLC}), the gate pulse gets truncated. This process is illustrated in Figure 92.

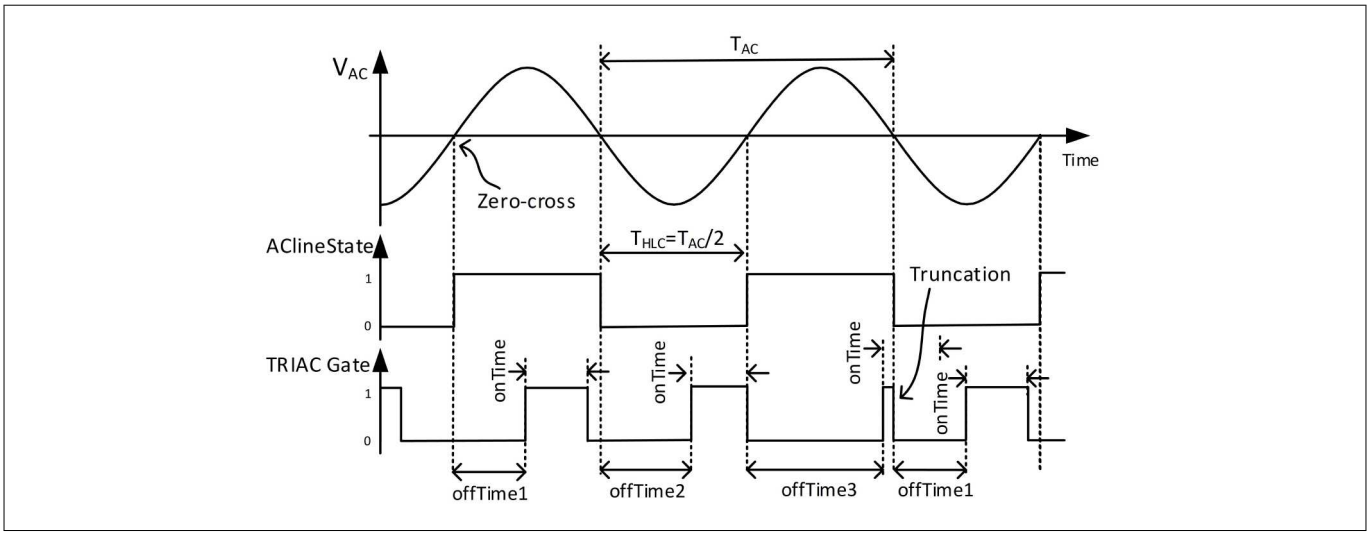


Figure 92 Truncation of Gate Pulse Near Zero-crossing

To avoid truncation, offTime + onTime < T_{HCL}. In Figure 92, offTime1 + onTime < T_{HCL} and the gate pulse is completed in time before the next zero-crossing. The borderline case is offTime2 + onTime = T_{HCL} and no truncation happens in this case. In case of offTime3 + onTime > T_{HCL} the gate pulse is truncated to fit within the half-line cycle.

6.12.5.1 TRIAC Pins

The TRIAC control has one input pin (AC-line zero-cross) and one output pin (TRIAC Gate). For flexibility, the MCE offers the option to select between two different input pins (TRIN0 and TRIN1) and two different output pins (TROUT0 and TROUT1). Availability of the TRIAC pins depends on the device and what other functions are supported in the application. Not all input/output options are available on all devices.

Refer to the latest version the datasheet of the device for pin assignment.

TRIAC input/output pins are selected during initialization of the TRIAC plug-in. The pins are enabled and assigned through the API TRIAC_DriverInit(). When using the TRIAC interface, make sure the TRIAC pins are not colliding with other functions and that the pins are not enabled in in the User Pin Configuration in Solution Designer.

The TRIAC input pin has an internal pull-down resistor. The TRIAC output pin is a push-pull stage. Active level of the output pin can be configured by the API TRIAC_DriverInit API().

6.12.5.2 TRIAC Interface APIs

The APIs of the TRIAC Interface plug-in are summarized in the table below.

API name	Brief description
TRIAC_DriverInit()	Initializes TRIAC interface and initial state of the control
TRIAC_DriverDeInit()	De-initializes TRIAC interface
TRIAC_SetONOFFTimes()	Sets the TRIAC gate on-time and delay-time
TRIAC_GetStatus()	Returns status bits
TRIAC_ClearStatus()	Clears sticky status bits
TRIAC_GetOnTime()	Returns the actual TRIAC gate on-time
TRIAC_GetOffTime()	Returns the actual TRIAC gate delay-time

6.12.5.3 TRIAC_DriverInit()

Declaration:

```
int32_t TRIAC_DriverInit(onTime, offTime, inputChannel, outputChannel, outputInvert)
```

Input Parameters	Min	Max	Description
`onTime`	0	21845	Specifies the TRIAC gate pulse width in micro-seconds. To keep the gate constantly OFF (inactive level), set onTime = 0 and offTime = 0. To keep the gate constantly ON (active level), set onTime = 0xFFFF and offTime = 0
`offTime`	0	21845	Specifies the TRIAC gate delay time in micro-seconds. To keep the gate constantly OFF (inactive level), set onTime = 0 and offTime = 0. To keep the gate constantly ON (active level), set onTime = 0xFFFF and offTime = 0
`inputChannel`	0	1	Specifies the pin for the AC-line status signal. Availability of pins depends on the device 0: TRIN0 1: TRIN1
`outputChannel`	0	1	Specifies the pin for the TRIAC gate signal. Availability of pins depends on the device 0: TROUT0 1: TROUT1
`outputInvert`	0	1	Specifies the active level of the TRIAC gate signal 0: Do not invert, active level is 'high' 1: Invert, active level is 'low'

Return Type	Description
int32_t	Returns initialization status. 0 – Driver successfully initialized. 1 – TRIAC driver not available or the selected channel(s) is not supported by product.

Description:
Initializes driver, sets initial gate condition and assigns I/O pins.

6.12.5.4 TRIAC_DriverDeInit()

Declaration:

```
int32_t TRIAC_DriverDeInit()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Returns de-initialization status. 0 – TRIAC driver was de-initialized successfully. 1 – Driver not available or TRIAC driver was not initialized before to de-initialized

Description:
De-initializes driver and I/O pins.

6.12.5.5 TRIAC_SetONOFFTimes()

Declaration:

```
int32_t TRIAC_SetONOFFTimes(onTime, offTime)
```

Input Parameters	Min	Max	Description
`onTime`	0	21845	Specifies the TRIAC gate pulse width in micro-seconds. To keep the gate constantly OFF (inactive level), set onTime = 0 and offTime = 0. To keep the gate constantly ON (active level), set onTime= 0xFFFF and offTime = 0
`offTime`	0	21845	Specifies the TRIAC gate delay time in micro-seconds. To keep the gate constantly OFF (inactive level), set onTime = 0 and offTime = 0. To keep the gate constantly ON (active level), set onTime = 0xFFFF and offTime = 0

Return Type	Description
int32_t	Result of setting timing variables. 0 – Update was successful. 1 – Driver not available or TRIAC driver is not initialized.

Description:
Sets the TRIAC gate pulse width and the TRIAC turn-on delay time.

6.12.5.6 TRIAC_GetStatus()

Declaration:

```
int32_t TRIAC_GetStatus()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Bit mapped variable indicating the status of AC-line signal. Bit 0 – **ZeroCross** (sticky): • 0: Zero-crossing of AC-line not detected • 1: Zero-crossing of AC-line detected Bit 1 – **AClineState**: • 0: AC line signal is low • 1: AC-line signal is high Bit 2:31 – **Reserved**

Description:
Reports status of the AC-line signal. ZeroCross (Bit 0) is sticky. Once set, it remains set until cleared manually by calling API TRIAC_ClearStatus(). AClineState is not sticky and always reports the current level of the AC-line signal.

6.12.5.7 TRIAC_ClearStatus()

Declaration:

```
int32_t TRIAC_ClearStatus(clearMask)
```

Input Parameters	Min	Max	Description
`clearMask`	0	255	Mask to clear sticky bits returned by TRIAC_GetStatus()

Return Type	Description
int32_t	Result of status clear. 0 – Clear was successful 1 – Clear was unsuccessful

Description:
The bits of the status returned by TRIAC_GetStatus() are sticky and must be cleared manually by calling this API with the appropriate mask. To clear bit 0, set mask to 1.

6.12.5.8 TRIAC_GetOnTime()

Declaration:

```
int32_t TRIAC_GetOnTime()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Current TRIAC gate on-time in micro-seconds

Description:

This API returns the last on-time set by either TRIAC_DriverInit or TRIAC_SetONOFFTimes.

6.12.5.9 TRIAC_GetOffTime()

Declaration:

```
int32_t TRIAC_GetOffTime()
```

Input Parameters	Min	Max	Description
N/A	N/A	N/A	N/A

Return Type	Description
int32_t	Current TRIAC delay-time in micro-seconds

Description:

This API returns the last delay-time set by either TRIAC_DriverInit or TRIAC_SetONOFFTimes.

Revision history

Document version	Date of release	Description of changes
1.0	2021-11-09	Initial release
1.1	2022-12-21	Document updated to reflect MCE software revision MCE FW_V5.1.0
1.2	2023-08-28	Update to include new features introduced with 5.2
1.3	2024-01-10	Include UL Class-B certification column for Motor and PFC protection table. Update Execution Fault with Class-B protection. Load and Save command section updated.
1.4	2024-10-28	Document updated to reflect MCE software revision MCE V5.4.0 • User Mode UART: Read and Write commands for 32 bit registers • User Mode UART: Remove description of 'Save' command; not implemented • User Mode UART: Correct number of parameter sets for 'Load' command (0 .. 15) • Script Engine: Mapping of Variables to Parameters
1.5	2025-06-04	• Script Engine: Script Array define • Script Engine: User defined function • Script Engine: Correct the script room size
1.6	2026-05-21	• User Mode UART: Add CRC check description for • Motor Control - Protection: Add internal pull-up disable or enable description for IMC products • Script Engine: Correct the script room size according to MCE V5.5.0

Trademarks

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

Edition 2026-05-21

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-eml1689935802070

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, 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.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

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.

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.