User's Guide

Motor Control SDK Universal Project and Lab

ABSTRACT

This document explains the steps needed to run the motor drive evaluation kits with the Universal Motor Control Lab project in MotorControlSDK, how to migrate the lab project to a custom board, and how to port the lab project to a new C2000™ device.

Table of Contents

List of Figures

List of Tables

Trademarks

C2000™, FAST™, InstaSPIN™, InstaSPIN-FOC™, Code Composer Studio™, LaunchPad™, NexFET™, and BoosterPack™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

The Universal Motor Control Lab project described in this guide is intended for you to not only experiment with various motor control algorithms but also to use as a reference for your own design. The universal motor control solution, as well as the lab project, is located within the MotorControl SDK.

The Universal Motor Control Lab project provides an example using the F28002x, F28003x, and F280013x series C2000 MCU. This is a single project with build examples for different Sensorless (FAST™, eSMO, InstaSPIN™-BLDC) and Sensored (Incremental Encoder, Hall) motor control techniques (FOC, Trapezoidal), with included system features and debug interfaces that can be used across a variety of three-phase inverter motor evaluation kits.

The FAST library (which is used to estimate the motor Flux, Angle, Speed, and Torque) is implemented with InstaSPIN-FOC™ in this Universal Motor Control Lab project. This library enables the use of the FAST observer for InstaSPIN-FOC with FPU enabled and C2000Ware-MotorControl-SDK supported C2000 devices. The user no longer needs to use a C2000 device with special ROM content in order to use FAST or InstaSPIN-FOC.

In this user's guide you will learn how to modify the *user_mtr1.h* file, which is the header file that stores all of the user parameters. Some of these parameters can be manipulated through CCS during run-time, but the parameters must be updated in the *user_mtr1.h* file to be saved permanently in your project. You will learn how to migrate the lab to your own hardware board, and port the lab project to the other C2000 MCU controllers by modifying the *hal.h* and *hal.c* files.

The lab project provides several interface functions to start/stop the motor and set the reference speed by using push a button, potentiometer, or CAN interface.

The Motor Control Universal Lab project is built within the MotorControl SDK folder and additionally uses files from C2000Ware. The MotorControl SDK software includes firmware that runs on C2000 motor control evaluation modules (EVMs) and TI designs (TIDs). A copy of C2000Ware is provided as part of the MotorControl SDK and offers various projects, ranging from device-specific drivers and support software to complete example system applications.

The Universal MotorControl Lab requires:

- Code Composer Studio™ v12.0.0 or newer
- C2000 Compiler v22.6.0 LTS or newer
- C2000Ware MotorControl SDK V4.01.00 or newer

2 Motor Control Theory

Permanent Magnet Synchronous motor (PMSM) has a wound stator, a permanent magnet rotor assembly and internal or external devices to sense rotor position. The sensing devices provide position feedback for adjusting frequency and amplitude of stator voltage reference properly to maintain rotation of the magnet assembly. The combination of an inner permanent magnet rotor and outer windings offers the advantages of low rotor inertia, efficient heat dissipation, and reduction of the motor size.

- Synchronous motor construction: Permanent magnets are rigidly fixed to the rotating axis to create a constant rotor flux. This rotor flux usually has a constant magnitude. The stator windings when energized create a rotating electromagnetic field. To control the rotating magnetic field, it is necessary to control the stator currents.
- The actual structure of the rotor varies depending on the power range and rated speed of the machine. Permanent magnets are suitable for synchronous machines ranging up-to a few Kilowatts. For higher power ratings the rotor usually consists of windings in which a DC current circulates. The mechanical structure of the rotor is designed for number of poles desired, and the desired flux gradients desired.
- The interaction between the stator and rotor fluxes produces a torque. Since the stator is firmly mounted to the frame, and the rotor is free to rotate, the rotor will rotate, producing a useful mechanical output as shown in Figure 2-1.
- The angle between the rotor magnetic field and stator field must be carefully controlled to produce maximum torque and achieve high electromechanical conversion efficiency. For this purpose a fine tuning is needed after closing the speed loop using sensorless algorithm to draw minimum amount of current under the same speed and torque conditions.
- The rotating stator field must rotate at the same frequency as the rotor permanent magnetic field; otherwise the rotor will experience rapidly alternating positive and negative torque. This will result in less than optimal torque production, and excessive mechanical vibration, noise, and mechanical stresses on the machine parts. In addition, if the rotor inertia prevents the rotor from being able to respond to these oscillations, the rotor will stop rotating at the synchronous frequency, and respond to the average torque as seen by the stationary rotor: Zero. This means that the machine experiences a phenomenon known as *pull-out*. This is also the reason why the synchronous machine is not self starting.
- The angle between the rotor field and the stator field must be equal to 90ºC to obtain the highest mutual torque production. This synchronization requires knowing the rotor position to generate the right stator field.
- The stator magnetic field can be made to have any direction and magnitude by combining the contribution of different stator phases to produce the resulting stator flux.

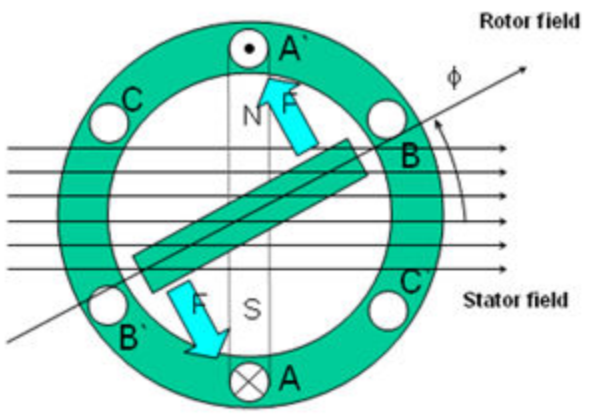


Figure 2-1. The Interaction Between the Rotating Stator Flux, and the Rotor Flux Produces a Torque

2.1 Mathematical Model and FOC Structure of PMSM

The FOC structure for a PMSM is illustrated in Figure 2-2. In this system, the eSMO is used for achieving the sensorless control an IPMSM system, and the eSMO model is designed by utilizing the back EMF model together with a PLL model for estimating the rotor position and speed.

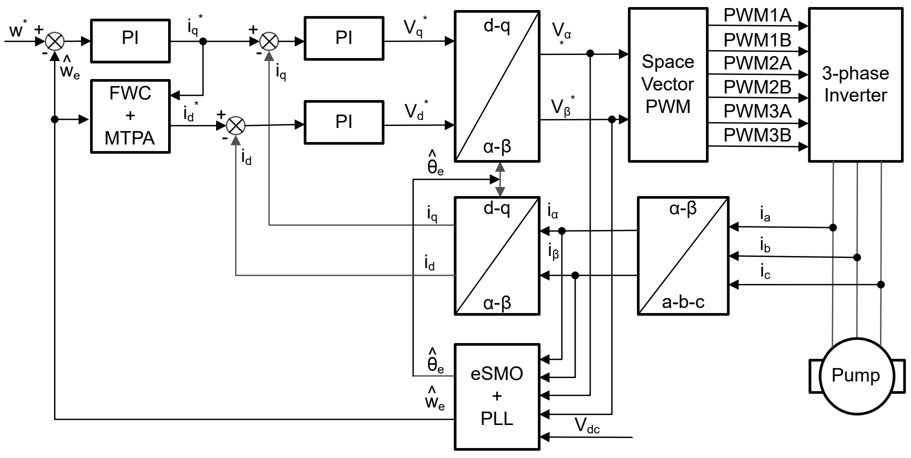


Figure 2-2. Sensorless FOC Structure of an PMSM System

An IPMSM consists of a three-phase stator winding (a, b, c axes), and permanent magnets (PM) rotor for excitation. The motor is controlled by a standard three-phase inverter. An IPMSM can be modeled by using phase a-b-c quantities. Through proper coordinate transformations, the dynamic PMSM models in the d-q rotor reference frame and the α-β stationary reference frame can be obtained. The relationship among these reference frames are illustrated in Equation 1. The dynamic model of a generic PMSM can be written in the d-q rotor reference frame as:

$$\begin{bmatrix} v_d \\ v_q \end{bmatrix} = \begin{bmatrix} R_s + pL_d & -\omega_e L_q \\ \omega_e L_d & R_s + pL_q \end{bmatrix} \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_e \lambda_{pm} \end{bmatrix} \quad (1)$$

Where v_d and v_q are the q-axis and d-axis stator terminal voltages, respectively; i_d and i_q are the d-axis and q-axis stator currents, respectively; L_d and L_q are the q-axis and d-axis inductances, respectively, *p* is the derivative operator, a short notation of $\frac{d}{dt}$; λ_{pm} is the flux linkage generated by the permanent magnets, R_s is the resistance of the stator windings; and ω_e is the electrical angular velocity of the rotor.

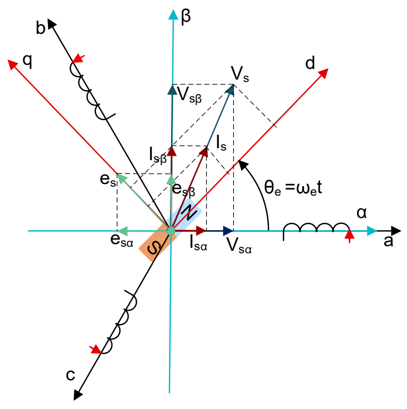


Figure 2-3. Definitions of Coordinate Reference Frames for PMSM Modeling

By using the inverse Park transformation as shown in Figure 2-3, the dynamics of the PMSM can be modeled in the α-β stationary reference frame as:

$$\begin{bmatrix} v_\alpha \\ v_\beta \end{bmatrix} = \begin{bmatrix} R_s + pL_d & \omega_e(L_d - L_q) \\ -\omega_e(L_d - L_q) & R_s + pL_q \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} + \begin{bmatrix} e_\alpha \\ e_\beta \end{bmatrix} \tag{2}$$

Where the e_a and e_β are components of extended electromotive force (EEMF) in the α-β axis and can be defined as:

$$\begin{bmatrix} e_\alpha \\ e_\beta \end{bmatrix} = (\lambda_{pm} + (L_d - L_q)i_d)\omega_e \begin{bmatrix} -\sin(\theta_e) \\ \cos(\theta_e) \end{bmatrix} \tag{3}$$

According to Equation 2 and Equation 3, the rotor position information can be decoupled from the inductance matrix by means of the equivalent transformation and the introduction of the EEMF concept, so that the EEMF is the only term that contains the rotor pole position information. And then the EEMF phase information can be directly used to realize the rotor position observation. Rewrite the IPMSM voltage equation Equation 4 as a state equation using the stator current as a state variable:

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \frac{1}{L_d} \begin{bmatrix} -R_s & -\omega_e(L_d - L_q) \\ \omega_e(L_d - L_q) & -R_s \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} + \frac{1}{L_d} \begin{bmatrix} V_\alpha - e_\alpha \\ V_\beta - e_\beta \end{bmatrix} \tag{4}$$

Since the stator current is the only physical quantity that can be directly measured, the sliding surface is selected on the stator current path:

$$S(x) = \begin{bmatrix} \hat{i}_\alpha - i_\alpha \\ \hat{i}_\beta - i_\beta \end{bmatrix} = \begin{bmatrix} \tilde{i}_\alpha \\ \tilde{i}_\beta \end{bmatrix} \tag{5}$$

where $\hat{i}_\alpha$ and $\hat{i}_\beta$ are the estimated currents, the superscript **^** indicates the estimated value, the superscript "~" indicates the variable error which refers to the difference between the observed value and the actual measurement value.

2.2 Field Oriented Control of PM Synchronous Motor

To achieve better dynamic performance, a more complex control scheme needs to be applied, to control the PM motor. With the mathematical processing power offered by the microcontrollers, we can implement advanced control strategies, which use mathematical transformations to decouple the torque generation and the magnetization functions in PM motors. Such de-coupled torque and magnetization control is commonly called rotor flux oriented control, or simply Field Oriented Control (FOC).

In a direct current (DC) Motor, the excitation for the stator and rotor is independently controlled, the produced torque and the flux can be independently tuned as shown in Figure 2-4. The strength of the field excitation (for example, the magnitude of the field excitation current) sets the value of the flux. The current through the rotor windings determines how much torque is produced. The commutator on the rotor plays an interesting part in the torque production. The commutator is in contact with the brushes, and the mechanical construction is designed to switch into the circuit the windings that are mechanically aligned to produce the maximum torque. This arrangement then means that the torque production of the machine is fairly near optimal all the time. The key point here is that the windings are managed to keep the flux produced by the rotor windings orthogonal to the stator field.

To achieve better dynamic performance, a more complex control scheme needs to be applied, to control the PM motor. With the mathematical processing power offered by the microcontrollers, we can implement advanced control strategies, which use mathematical transformations to decouple the torque generation and the magnetization functions in PM motors. Such de-coupled torque and magnetization control is commonly called rotor flux oriented control, or simply Field Oriented Control (FOC).

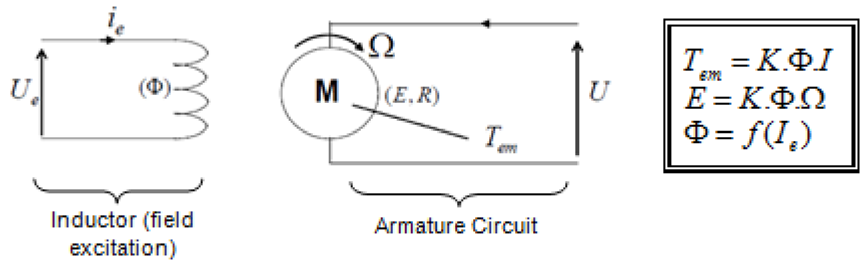


Figure 2-4. Flux and Torque are Independently Controlled in DC Motor Model

The goal of the FOC (also called vector control) on synchronous and asynchronous machine is to be able to separately control the torque producing and magnetizing flux components. FOC control will allow us to decouple the torque and the magnetizing flux components of stator current. With decoupled control of the magnetization, the torque producing component of the stator flux can now be thought of as independent torque control. To decouple the torque and flux, it is necessary to engage several mathematical transforms, and this is where the microcontrollers add the most value. The processing capability provided by the microcontrollers enables these mathematical transformations to be carried out very quickly. This in turn implies that the entire algorithm controlling the motor can be executed at a fast rate, enabling higher dynamic performance. In addition to the decoupling, a dynamic model of the motor is now used for the computation of many quantities such as rotor flux angle and rotor speed. This means that their effect is accounted for, and the overall quality of control is better.

According to the electromagnetic laws, the torque produced in the synchronous machine is equal to vector cross product of the two existing magnetic fields as Equation 6.

$$\tau_{em} = \vec{B}_{stator} \times \vec{B}_{rotor} \tag{6}$$

This expression shows that the torque is maximum if stator and rotor magnetic fields are orthogonal meaning if we are to maintain the load at 90°. If we are able to ensure this condition all the time, if we are able to orient the flux correctly, we reduce the torque ripple and we ensure a better dynamic response. However, the constraint is to know the rotor position: this can be achieved with a position sensor such as incremental encoder. For low-cost application where the rotor is not accessible, different rotor position observer strategies are applied to get rid of position sensor.

In brief, the goal is to maintain the rotor and stator flux in quadrature: the goal is to align the stator flux with the q axis of the rotor flux, for example, orthogonal to the rotor flux. To do this, the stator current component in quadrature with the rotor flux is controlled to generate the commanded torque, and the direct component is set to zero. The direct component of the stator current can be used in some cases for field weakening, which has the effect of opposing the rotor flux, and reducing the back-emf, which allows for operation at higher speeds.

The Field Orientated Control consists of controlling the stator currents represented by a vector. This control is based on projections which transform a three phase time and speed dependent system into a two co-ordinate (d and q co-ordinates) time invariant system. These projections lead to a structure similar to that of a DC machine control. Field orientated controlled machines need two constants as input references: the torque component (aligned with the q co-ordinate) and the flux component (aligned with d co-ordinate). As Field Orientated Control is simply based on projections, the control structure handles instantaneous electrical quantities. This makes the control accurate in every working operation (steady state and transient) and independent of the limited bandwidth mathematical model. The FOC thus solves the classic scheme problems, in the following ways:

- The ease of reaching constant reference (torque component and flux component of the stator current)
- The ease of applying direct torque control because in the (d, q) reference frame the expression of the torque is defined in Equation 7.

$$\tau_{em} \propto \psi_R \times i_{sq} \tag{7}$$

By maintaining the amplitude of the rotor flux (ψ_R) at a fixed value we have a linear relationship between torque and torque component (i_{Sq}). We can then control the torque by controlling the torque component of stator current vector.

Space Vector Definition and Projection

The 3-phase voltages, currents and fluxes of AC-motors can be analyzed in terms of complex space vectors. With regard to the currents, the space vector can be defined as follows. Assuming that i_a, i_b, i_c are the instantaneous currents in the stator phases, then the complex stator current vector is defined in Equation 8.

$$\bar{i}_s = i_a + \alpha i_b + \alpha^2 i_c \tag{8}$$

where $\alpha = e^{j\frac{2}{3}\pi}$ and $\alpha^2 = e^{j\frac{4}{3}\pi}$ represent the spatial operators.

Figure 2-5 shows the stator current complex space vector.

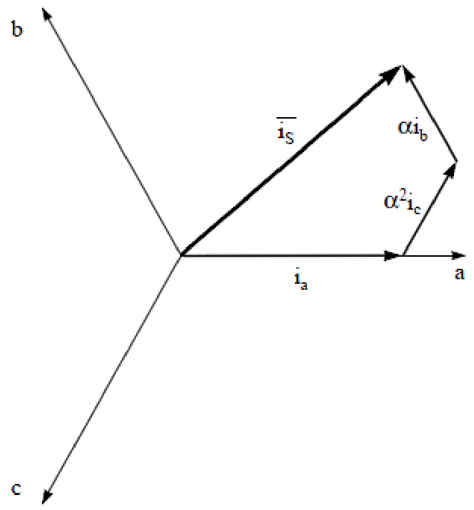


Figure 2-5. Stator Current Space Vector and its Component in (a,b,c) Frame

Where (a,b,c) are the three phase system axes. This current space vector depicts the three phase sinusoidal system. It still needs to be transformed into a two time invariant co-ordinate system. This transformation can be split into two steps:

- $(a,\ b) \Rightarrow (\alpha, \beta)$ (Clarke transformation) which outputs a 2-coordinate time-variant system
- $(\alpha, \beta) \Rightarrow (d,\ q)$ (Park transformation) which outputs a 2-coordinate time-invariant system

The $(a,\ b) \Rightarrow (\alpha, \beta)$ Clarke Transformation

The space vector can be reported in another reference frame with only two orthogonal axis called (α, β). Assuming that the axis a and the axis α*lpha* are in the same direction we have the following vector diagram as shown in Figure 2-6.

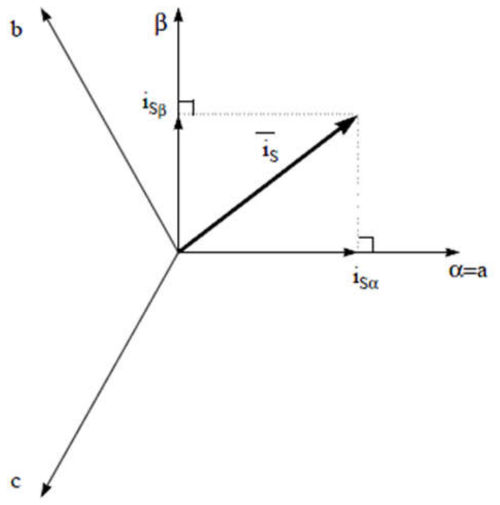


Figure 2-6. Stator Current Space Vector in the Stationary Reference Frame

The projection that modifies the 3-phase system into the (α, β) 2-dimension orthogonal system is presented in Equation 9.

$$i_{s\alpha} = i_a$$
$$i_{s\beta} = \frac{1}{\sqrt{3}} i_a + \frac{2}{\sqrt{3}} i_b \tag{9}$$

The two phase (α, β) currents are still depends on time and speed.

The $(\alpha, \beta) \Rightarrow (d,\ q)$ **Park Transformation**

This is the most important transformation in the FOC. In fact, this projection modifies a 2-phase orthogonal system (α, β) in the (d, q) rotating reference frame. If we consider the d axis aligned with the rotor flux, Figure 2-7 shows the relationship for the current vector from the two reference frame.

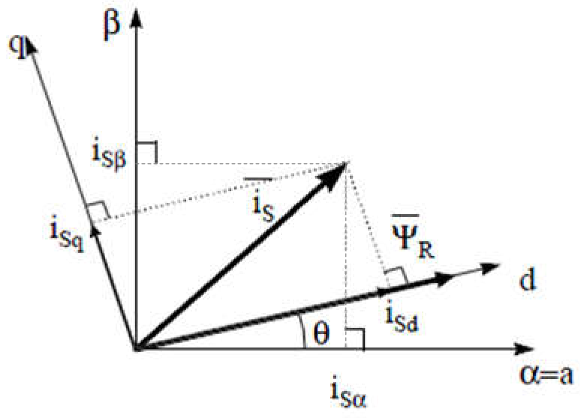


Figure 2-7. Stator Current Space Vector in The d,q Rotating Reference Frame

The flux and torque components of the current vector are determined by Equation 10.

$$i_{sd} = i_{s\alpha}\cos(\theta) + i_{s\beta}\sin(\theta)$$
$$i_{sq} = -\,i_{s\alpha}\sin(\theta) + i_{s\beta}\cos(\theta) \tag{10}$$

where θ is the rotor flux position

These components depend on the current vector (α, β) components and on the rotor flux position; if we know the right rotor flux position then, by this projection, the d,q component becomes a constant. Two phase currents now turn into dc quantity (time-invariant). At this point the torque control becomes easier where constant i_{sd} (flux component) and i_{sq} (torque component) current components controlled independently.

The Basic Scheme of FOC for AC Motor

Figure 2-8 summarizes the basic scheme of torque control with FOC:

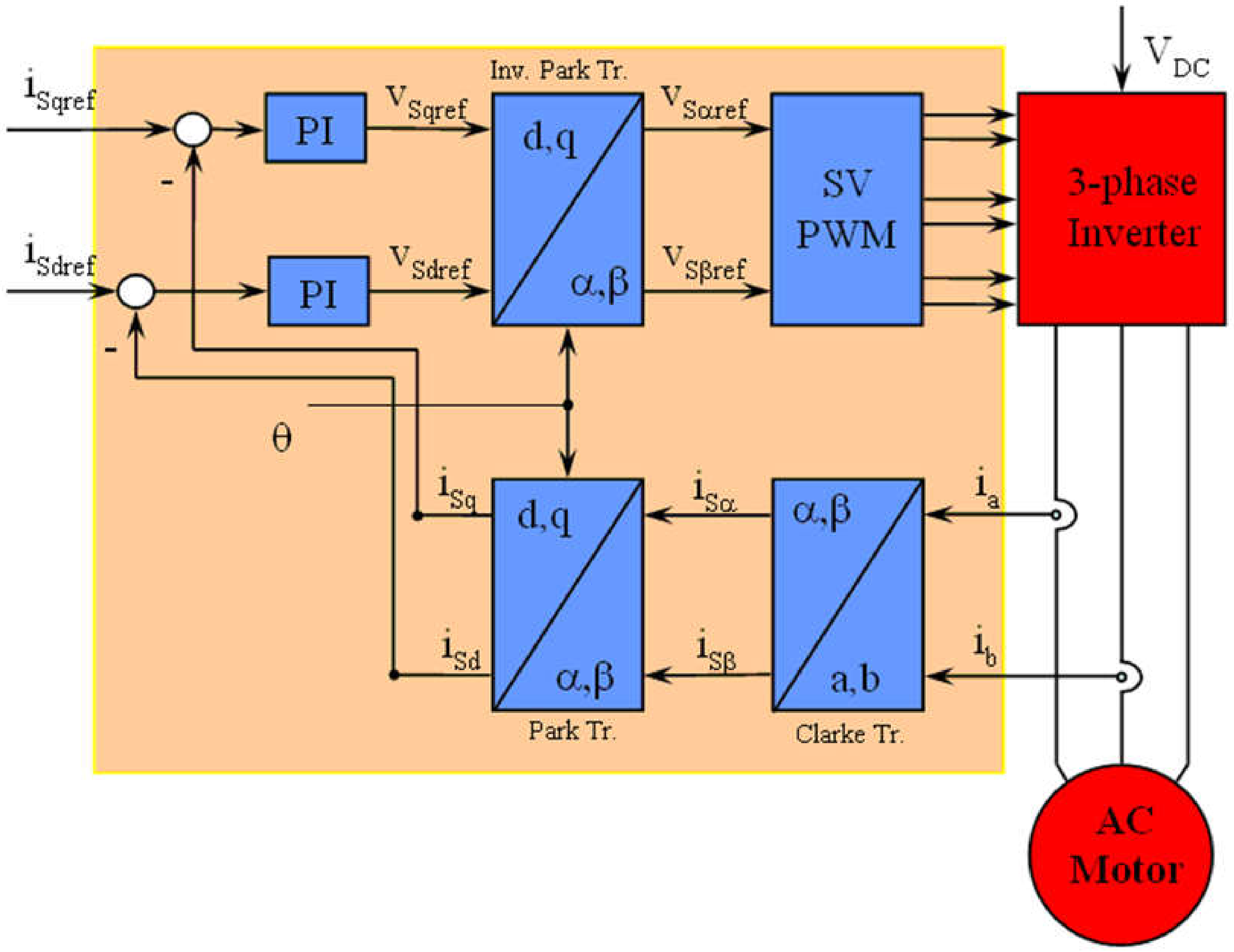


Figure 2-8. Basic Scheme of FOC for AC Motor

Two motor phase currents are measured. These measurements feed the Clarke transformation module. The outputs of this projection are designated $i_{s\alpha}$ and $i_{s\beta}$. These two components of the current are the inputs of the Park transformation that gives the current in the d,q rotating reference frame. The i_{sd} and i_{sq} components are compared to the references i_{sdref} (the flux reference component) and i_{sqref} (the torque reference component). At this point, this control structure shows an interesting advantage: it can be used to control either synchronous or induction machines by simply changing the flux reference and obtaining rotor flux position. As in synchronous permanent magnet a motor, the rotor flux is fixed determined by the magnets; there is no need to create one. Hence, when controlling a PMSM, i_{sdref} should be set to zero. As an AC induction motor needs a rotor flux creation to operate, the flux reference must not be zero. This conveniently solves one of the major drawbacks of the *classic* control structures: the portability from asynchronous to synchronous drives. The torque command i_{sqref} could be the output of the speed regulator when we use a speed FOC. The outputs of the current regulators are Vsdref and V_{sqref}; they are applied to the inverse Park transformation.

The outputs of this projection are $V_{s\alpha ref}$ and $V_{s\beta ref}$ which are the components of the stator vector voltage in the (α, β) stationary orthogonal reference frame. These are the inputs of the Space Vector PWM. The outputs of this block are the signals that drive the inverter. Note that both Park and inverse Park transformations need the rotor flux position. Obtaining this rotor flux position depends on the AC machine type (synchronous or asynchronous machine).

Rotor Flux Position

Knowledge of the rotor flux position is the core of the FOC. In fact if there is an error in this variable the rotor flux is not aligned with d-axis and i_{sd} and i_{sq} are incorrect flux and torque components of the stator current. Figure 2-9 shows the (a, b, c), (α, β) and (*d, q*) reference frames, and the correct position of the rotor flux, the stator current and stator voltage space vector that rotates with d,q reference at synchronous speed.

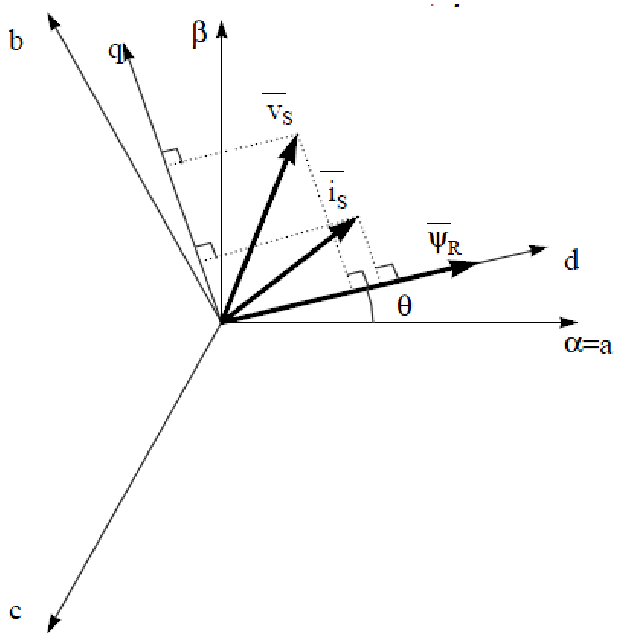


Figure 2-9. Current, Voltage and Rotor Flux Space Vectors in the (d, q) Rotating Reference Frame

The measure of the rotor flux position is different if we consider synchronous or asynchronous motor:

- In the synchronous machine the rotor speed is equal to the rotor flux speed. Then θ (rotor flux position) is directly measured by position sensor or by integration of rotor speed.
- In the asynchronous machine the rotor speed is not equal to the rotor flux speed (there is a slip speed), then it needs a particular method to calculate θ. The basic method is the use of the current model which needs two equations of the motor model in *d, q* reference frame.

Theoretically, the field oriented control for the PMSM drive allows the motor torque be controlled independently with the flux like DC motor operation. In other words, the torque and flux are decoupled from each other. The rotor position is required for variable transformation from stationary reference frame to synchronously rotating reference frame. As a result of this transformation (so called Park transformation), q-axis current will be controlling torque while d-axis current is forced to zero. Therefore, the key module of this system is the estimation of rotor position using enhance Sliding-Mode Observer (eSMO) or FAST estimator.

Figure 2-10 shows the overall block diagram of sensorless FOC of a PMSM motor using eSMO with flying start in this document.

Figure 2-11 shows the overall block diagram of sensorless FOC of a PMSM motor using eSMO with field weakening control (FWC) and maximum torque per ampere (MTPA) in this document.

Figure 2-13 shows the overall block diagram of sensorless FOC of a PMSM motor using FAST with flying start in this document.

Figure 2-13 shows the overall block diagram of sensorless FOC of a PMSM motor using FAST with field weakening control (FWC) and maximum torque per ampere (MTPA) in this document.

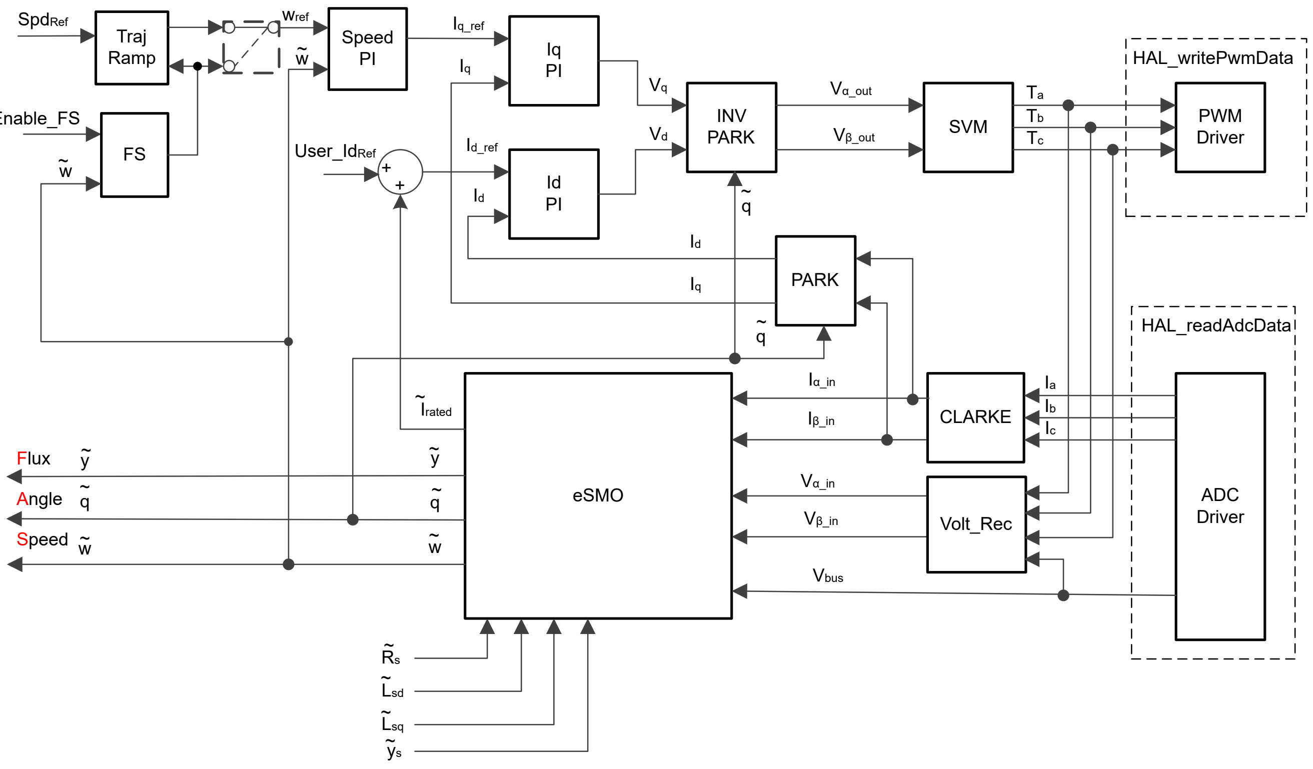


Figure 2-10. Sensorless FOC of PMSM Using eSMO With Flying Start (FS)

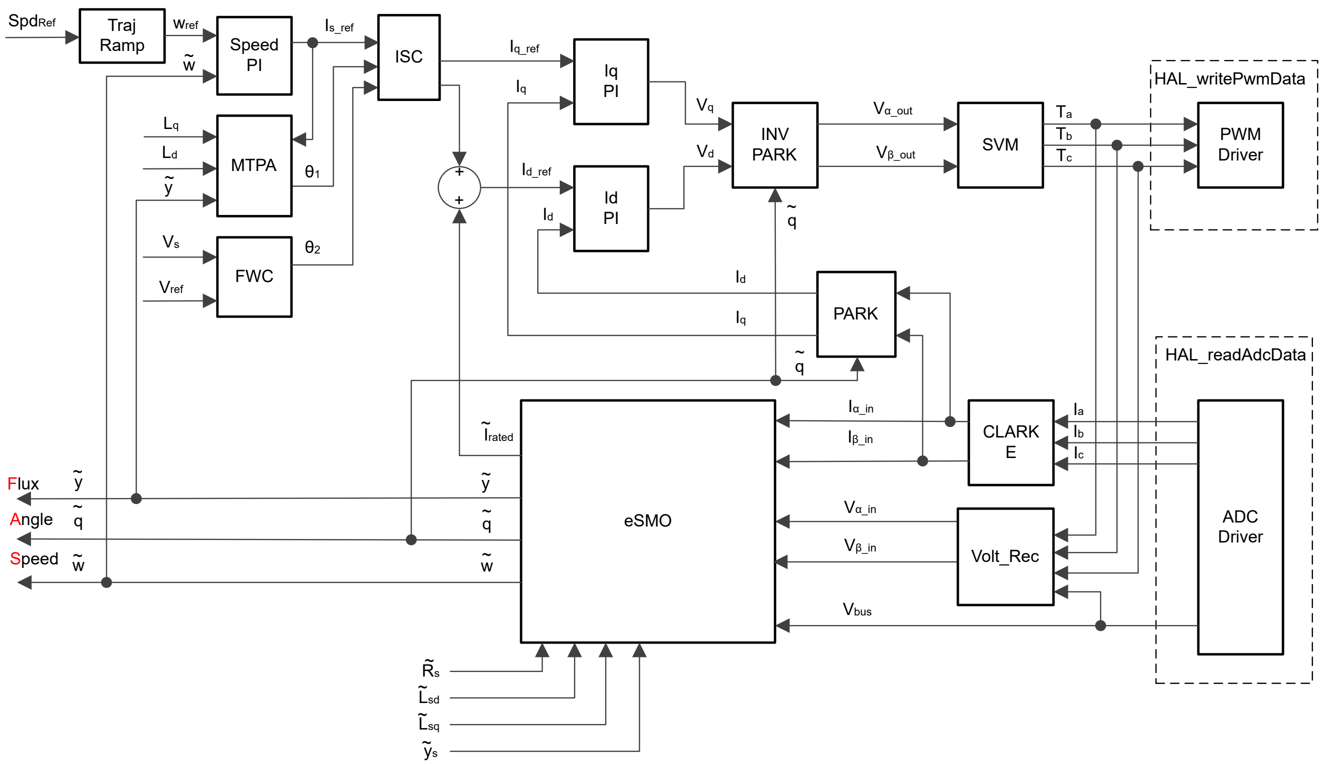


Figure 2-11. Sensorless FOC of PMSM Using eSMO With FWC and MTPA

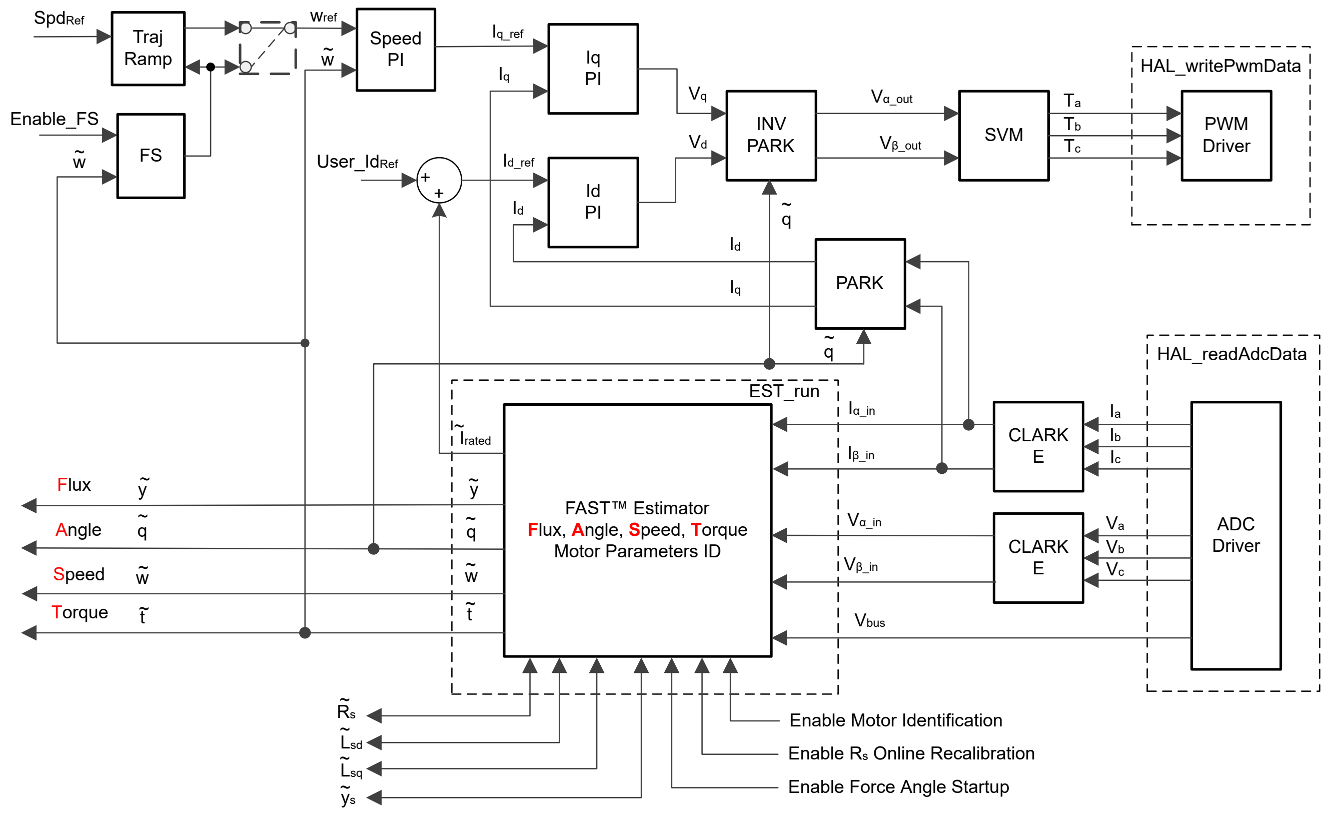


Figure 2-12. Sensorless FOC of PMSM Using FAST With Flying Start (FS)

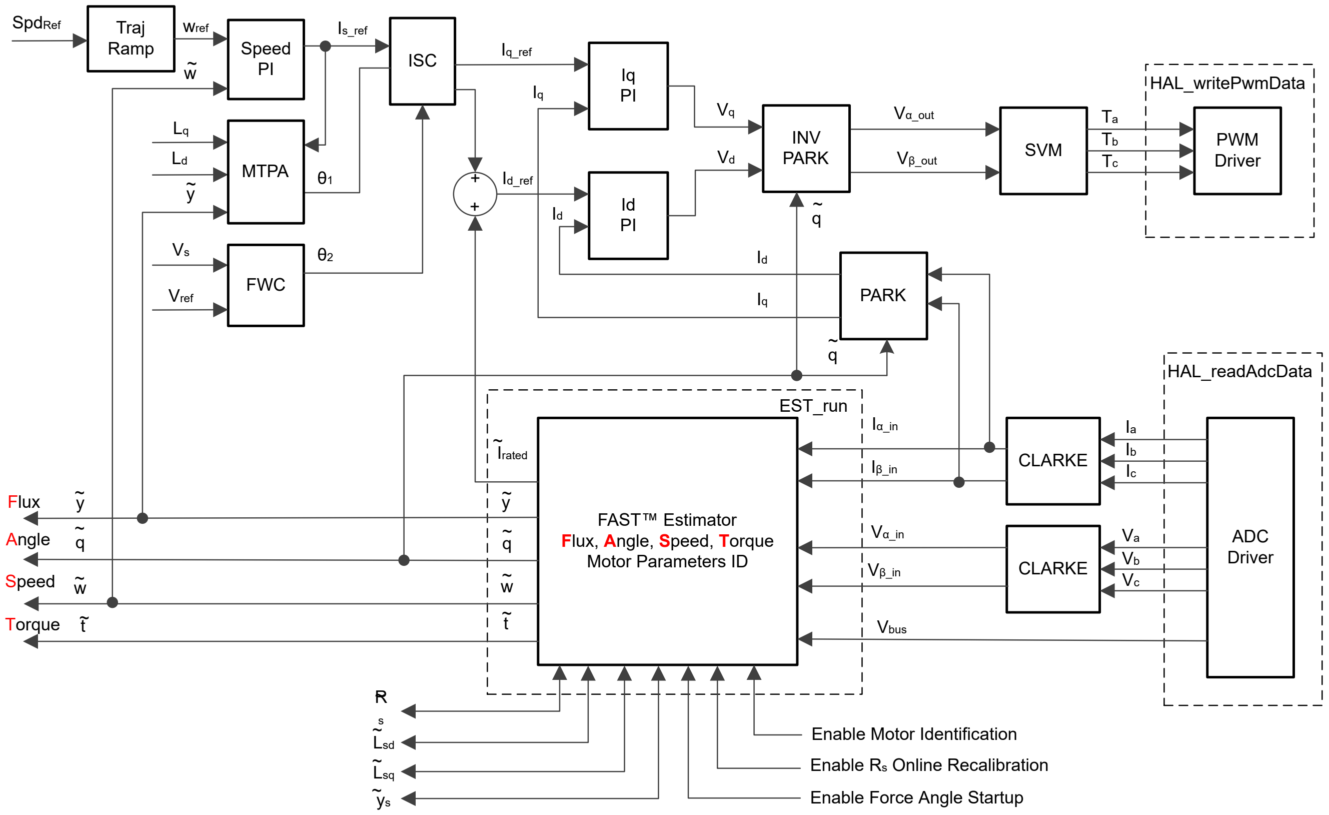


Figure 2-13. Sensorless FOC of PMSM Using FAST With FWC and MTPA

2.3 Sensorless Control of PM Synchronous Motor

In home appliance applications, if the mechanical sensor is used, it will cause increasing cost, size, and reliability problems. To overcome these problems, sensorless control methods are implemented. Several estimation methods to get the rotor speed and position information without mechanical position sensor. The sliding mode observer (SMO) is commonly utilized due to its various attractive features including reliability, desired performance, and robustness against system parameter variations.

2.3.1 Enhanced Sliding Mode Observer with Phase Locked Loop

Model-based method is used to achieve position sensorless control of the IPMSM drive system when the motor runs at middle or high speed. The model method estimates the rotor position by the back-EMF or the flux linkage model. The sliding mode observer is an observer-design method based on sliding mode control. The structure of the system is not fixed but purposefully changed according to the current state of the system, forcing the system to move according to the predetermined sliding mode trajectory. Its advantages include fast response, strong robustness, and insensitivity to both parameter changes and disturbances.

2.3.1.1 Design of ESMO for PMSM

The conventional PLL integrated into the SMO is shown in Figure 2-14.

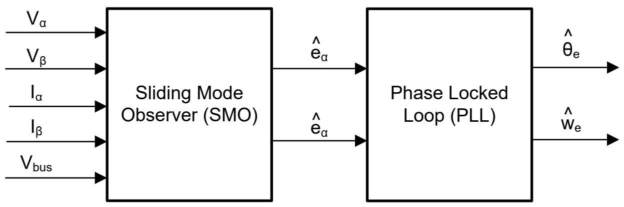


Figure 2-14. Block Diagram of eSMO With PLL for a PMSM

The traditional reduced-order sliding mode observer is constructed, which mathematical model is shown in Figure 2-14 and the block diagram is shown in Figure 2-15.

$$\begin{bmatrix} \dot{\hat{i}}_\alpha \\ \dot{\hat{i}}_\beta \end{bmatrix} = \frac{1}{L_d}\begin{bmatrix} -R_s & -\hat{\omega}_e(L_d - L_q) \\ \hat{\omega}_e(L_d - L_q) & -R_s \end{bmatrix}\begin{bmatrix} \hat{i}_\alpha \\ \hat{i}_\beta \end{bmatrix} + \frac{1}{L_d}\begin{bmatrix} V_\alpha - \hat{e}_\alpha + z_\alpha \\ V_\beta - \hat{e}_\beta + z_\beta \end{bmatrix} \tag{11}$$

where z_α and z_β are sliding mode feedback components and are defined as:

$$\begin{bmatrix} z_\alpha \\ z_\beta \end{bmatrix} = \begin{bmatrix} k_\alpha sign(\hat{i}_\alpha - i_\alpha) \\ k_\beta sign(\hat{i}_\beta - i_\beta) \end{bmatrix} \tag{12}$$

Where k_α and k_β are the constant sliding mode gain designed by Lyapunov stability analysis. If k_α and k_β are positive and significant enough to guarantee the stable operation of the SMO, the k_α and k_β should be large enough to hold $k_\alpha > \max(|e_\alpha|)$ and $k_\beta > \max(|e_\beta|)$.

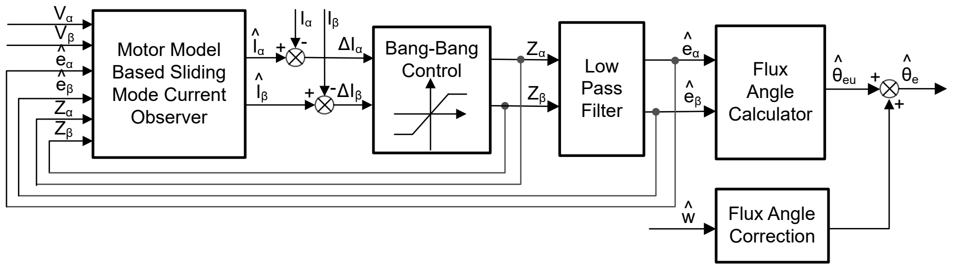


Figure 2-15. Block Diagram of Traditional Sliding Mode Observer

The estimated value of EEMF in α-β axes ($\hat{e}_\alpha$, $\hat{e}_\beta$) can be obtained by low-pass filter from the discontinuous switching signals z_α and z_α :

$$\begin{bmatrix} \hat{e}_\alpha \\ \hat{e}_\beta \end{bmatrix} = \frac{\omega_c}{s + \omega_c}\begin{bmatrix} z_\alpha \\ z_\beta \end{bmatrix} \tag{13}$$

Where $\omega_c = 2\pi f_c$ is the cutoff angular frequency of the LPF, which is usually selected according to the fundamental frequency of the stator current.

Therefore, the rotor position can be directly calculated from arc-tangent the back EMF, defined as follow

$$\hat{\theta}_e = -\tan^{-1}\left(\frac{\hat{e}_\alpha}{\hat{e}_\beta}\right) \tag{14}$$

Low pass filter removes the high-frequency term of the sliding mode function, which leads to occur phase delay resulting. It can be compensated by the relationship between the cut-off frequency ω_c and back EMF frequency ω_e , which is defined as:

$$\Delta\theta_e = -\tan^{-1}\left(\frac{\omega_e}{\omega_c}\right) \tag{15}$$

And then the estimated rotor position by using SMO method is:

$$\hat{\theta}_e = -\tan^{-1}\left(\frac{\hat{e}_\alpha}{\hat{e}_\beta}\right) + \Delta\theta_e \tag{16}$$

In a digital control application, a time discrete equation of the SMO is needed. The Euler method is the appropriate way to transform to a time discrete observer. The time discrete system matrix of Equation 17 in α-β coordinates is given by Equation 17 as:

$$\begin{bmatrix}\hat{i}_\alpha(n+1)\\ \hat{i}_\beta(n+1)\end{bmatrix} = \begin{bmatrix}F_\alpha\\ F_\beta\end{bmatrix}\begin{bmatrix}\hat{i}_\alpha(n)\\ \hat{i}_\beta(n)\end{bmatrix} + \begin{bmatrix}G_\alpha\\ G_\beta\end{bmatrix}\begin{bmatrix}V_\alpha^*(n) - \hat{e}_\alpha(n) + z_\alpha(n)\\ V_\beta^*(n) - \hat{e}_\beta(n) + z_\beta(n)\end{bmatrix} \tag{17}$$

Where the matrix $[F]$ and $[G]$ are given by Equation 18 and Equation 19 as:

$$\begin{bmatrix}F_\alpha\\ F_\beta\end{bmatrix} = \begin{bmatrix}e^{-\frac{R_S}{L_d}}\\ e^{-\frac{R_S}{L_q}}\end{bmatrix} \tag{18}$$

$$\begin{bmatrix}G_\alpha\\ G_\beta\end{bmatrix} = \frac{1}{R_S}\begin{bmatrix}1 - e^{-\frac{R_S}{L_d}}\\ 1 - e^{-\frac{R_S}{L_q}}\end{bmatrix} \tag{19}$$

The time discrete form of Equation 13 is given by Equation 20 as:

$$\begin{bmatrix}\hat{e}_\alpha(n+1)\\ \hat{e}_\beta(n+1)\end{bmatrix} = \begin{bmatrix}\hat{e}_\alpha(n)\\ \hat{e}_\beta(n)\end{bmatrix} + 2\pi f_c\begin{bmatrix}z_\alpha(n) - \hat{e}_\alpha(n)\\ z_\beta(n) - \hat{e}_\beta(n)\end{bmatrix} \tag{20}$$

2.3.1.2 Rotor Position and Speed Estimation With PLL

With the arc tangent method, the accuracy of the position and velocity estimations are affected due to the existence of noise and harmonic components. To eliminate this issue, the PLL model can be used for velocity and position estimations in the sensorless control structure of the IPMSM. The PLL structure used with SMO is illustrated in Section 2.3.1.1. The back-EMF estimations $\hat{e}_\alpha$ and $\hat{e}_\beta$ can be used with a PLL model to estimate the motor angular velocity and position as shown in Figure 2-16.

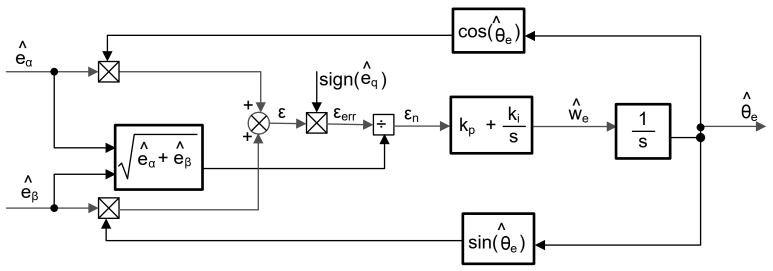


Figure 2-16. Block Diagram of Phase Locked Loop Position Tracker

Since $e_\alpha = E\cos(\theta_e)$, $e_\beta = E\sin(\theta_e)$ and $E = \omega_e\lambda_{pm}$, the position error can be defined as:

$$\varepsilon = \hat{e}_\beta\cos(\hat{\theta}_e) - \hat{e}_\alpha\sin(\hat{\theta}_e) = E\sin(\theta_e)\cos(\hat{\theta}_e) - E\cos(\theta_e)\sin(\hat{\theta}_e) = E\sin(\theta_e - \hat{\theta}_e) \tag{21}$$

Where E is the magnitude of the EEMF, which is proportional to the motor speed ω_e . When $(\theta_e - \hat{\theta}_e) < \frac{\pi}{2}$, the Equation 21 could be simplified as

$$\varepsilon = E(\theta_e - \hat{\theta}_e) \tag{22}$$

Further, the position error after the normalization of the EEMF can be obtained:

$$\varepsilon_n = \theta_e - \hat{\theta}_e \tag{23}$$

According to the analysis, the simplified block diagram of the quadrature phaselocked loop position tracker can be obtained as shown in Figure 2-17. The closed-loop transfer functions of the PLL can be expressed as Equation 24:

$$\frac{\hat{\theta}_e}{\theta_e} = \frac{k_p s + k_i}{s^2 + k_p s + k_i} = \frac{2\xi\omega_n s + \omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2} \tag{24}$$

where the k_p and k_i are the proportional and the integral gains of the standard PI regulator, its natural frequency ω_n and the damping ratio ξ is given:

$$k_p = 2\xi\omega_n, \qquad k_i = \omega_n^2 \tag{25}$$

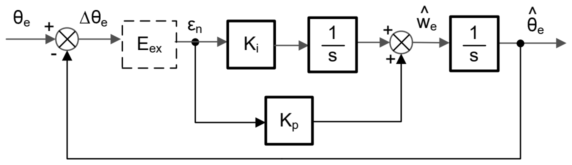


Figure 2-17. Simplified Block Diagram of Phase Locked Loop Position Tracker

2.4 Hardware Prerequisites for Motor Drive

The algorithm for controlling the motor makes use of sampled measurements of the motor conditions, including dc bus power supply voltage, the voltage on each motor phase, the current of each motor phase. There are a few hardware dependent parameters that need to be set correctly to identify the motor properly and run the motor effectively using Field Oriented Control (FOC). The following sections show how to calculate the current scale value, voltage scale value and voltage filter pole for compressor and fan motors control with FAST or eSMO.

2.4.1 Motor Phase Voltage Feedback

Motor phase voltage feedback is needed in the FAST estimator to allow the best performance at the widest speed range, the phase voltages are measured directly from the motor phases instead of a software estimate. The eSMO relies on software estimation values to represent the voltage phases without using the motor phase voltage sensing circuit. This software value (USER_ADC_FULL_SCALE_VOLTAGE_V) depends on the circuit that senses the voltage feedback from the motor phases. Figure 2-35 shows how the motor voltage is filtered and scaled for the ADC input range using a voltage feedback circuit based on resistor dividers. The similar circuit is used to measure all three of both compressor and fan motors, and dc bus.

The maximum phase voltage feedback measurable by the microcontroller in this reference design can be calculated as given in Equation 92, considering the maximum voltage for the ADC input is 3.3 V.

2.5 Additional Control Features

2.5.1 Field Weakening (FW) and Maximum Torque Per Ampere (MTPA) Control

Permanent magnet synchronous motor (PMSM) is widely used in home appliance applications due to its high power density, high efficiency, and wide speed range. The PMSM includes two major types: the surface-mounted PMSM (SPM), and the interior PMSM (IPM). SPM motors are easier to control due to their linear relationship between the torque and q-axis current. However, the IPMSM has electromagnetic and reluctance torques due to a large saliency ratio. The total torque is non-linear with respect to the rotor angle. As a result, the MTPA technique can be used for IPM motors to optimize torque generation in the constant torque region. The aim of the field weakening control is to optimize to reach the highest power and efficiency of a PMSM drive. Field weakening control can enable a motor operation over its base speed, expanding its operating limits to reach speeds higher than rated speed and allow optimal control across the entire speed and voltage range.

The voltage equations of the mathematical model of an IPMSM can be described in d-q coordinates as shown in Equation 26 and Equation 27.

$$v_d = L_d \frac{di_d}{dt} + R_s i_d - p\omega_m L_q i_q \tag{26}$$

$$v_q = L_q \frac{di_q}{dt} + R_s i_q + p\omega_m L_d i_d + p\omega_m \psi_m \tag{27}$$

The dynamic equivalent circuit of an IPM synchronous motor is shown in Figure 2-18.

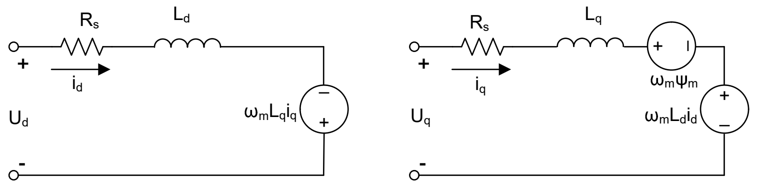


Figure 2-18. Equivalent Circuit of an IPM Synchronous Motor

The total electromagnetic torque generated by the IPMSM can be expressed as Equation 28 that the produced torque is composed of two distinct terms. The first term corresponds to the mutual reaction torque occurring between torque current i_q and the permanent magnet ψ_m , while the second term corresponds to the reluctance torque due to the differences in d-axis and q-axis inductance.

$$T_e = \frac{3}{2} p \left[\psi_m i_q + \left(L_d - L_q\right) i_d i_q \right] \tag{28}$$

In most applications, IPMSM drives have speed and torque constraints, mainly due to inverter or motor rating currents and available DC link voltage limitations respectively. These constraints can be expressed with the mathematical equations Equation 29 and Equation 30.

$$I_a = \sqrt{i_d^2 + i_q^2} \le I_{max} \tag{29}$$

$$V_a = \sqrt{v_d^2 + v_q^2} \le V_{max} \tag{30}$$

Where $\mathrm{V_{max}}$ and $\mathrm{I_{max}}$ are the maximum allowable voltage and current of the inverter or motor. In a two-level three-phase Voltage Source Inverter (VSI) fed machine, the maximum achievable phase voltage is limited by the DC link voltage and the PWM strategy. The maximum voltage is limited to the value as shown in Equation 31 if Space Vector Modulation (SVPWM) is adopted.

$$\sqrt{v_d^2 + v_q^2} \le v_{max} = \frac{v_{dc}}{\sqrt{3}} \tag{31}$$

Usually the stator resistance R_s is negligible at high speed operation and the derivate of the currents is zero in steady state, thus Equation 32 is obtained as shown.

$$\sqrt{L_d^2 \left(i_d + \frac{\psi_{pm}}{L_d} \right)^2 + L_q^2 i_q^2} \le \frac{V_{max}}{\omega_m} \tag{32}$$

The current limitation of Equation 29 produces a circle of radius $\mathrm{I_{max}}$ in the d-q plane, and the voltage limitation of Equation 31 produces an ellipse whose radius $\mathrm{V_{max}}$ decreases as speed increases. The resultant d-q plane current vector must be controlled to obey the current and voltage constraints simultaneously. According to these constraints, three operation regions for the IPMSM can be distinguished as shown in Figure 2-19.

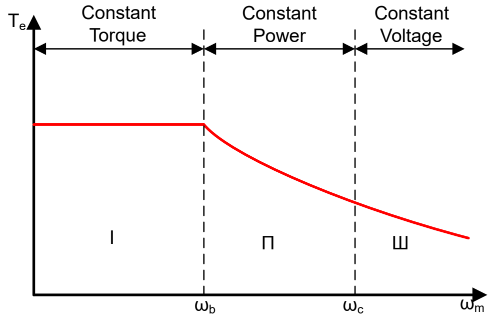


Figure 2-19. IPMSM Control Operation Regions

1. Constant Torque Region: MTPA can be implemented in this operation region to ensure maximum torque generation.
2. Constant Power Region: Field weakening control must be employed and the torque capacity is reduced as the current constraint is reached.
3. Constant Voltage Region: In this operation region, deep field weakening control keeps a constant stator voltage to maximize the torque generation.

In the constant torque region, according to Equation 28, the total torque of an IPMSM includes the electromagnetic torque from the magnet flux linkage and the reluctance torque from the saliency between L_d and L_q . The electromagnetic torque is proportional to the q-axis current i_q , and the reluctance torque is proportional to the multiplication of the d-axis current i_d , the q-axis current i_q , and the difference between L_d and L_q .

Conventional vector control systems of a SPM motors only utilizes electromagnetic torque by setting the commanded i_d to zero for non-field weakening modes. But an IPMSM will utilize the reluctance torque of the motor, d-axis current should be controlled as well. The aim of the MTPA control is to calculate the reference currents i_d and i_q to maximize the ratio between produced electromagnetic torque and reluctance torque. The relationship between i_d and i_q , and the vectorial sum of the stator current $\mathrm{I_s}$ is shown in the following equations.

$$I_s = \sqrt{i_d^2 + i_q^2} \tag{33}$$

$$I_d = I_s \cos\beta \tag{34}$$

$$I_q = I_s \sin\beta \tag{35}$$

Where β is the stator current angle in the synchronous (d-q) reference frame. Equation 28 can be expressed as Equation 36 where $\mathrm{I_s}$ substituted for i_d and i_q .

Equation 36 shows that motor torque depends on the angle of the stator current vector; as such:

$$T_e = \frac{3}{2} p I_s \sin\beta \left[\psi_m + \left(L_d - L_q\right) I_s \cos\beta \right] \tag{36}$$

The maximum efficiency point can be calculated when the motor torque differential is equal to zero. The MTPA point can be found when this differential, $\frac{\mathrm{dT_e}}{\mathrm{d\beta}}$ is zero as given in Equation 37.

$$\frac{dT_e}{d\beta} = \frac{3}{2} p \left[\psi_m I_s \cos\beta + \left(L_d - L_q\right) I_s^2 \cos 2\beta \right] = 0 \tag{37}$$

Following, the current angle of the MTPA control can be derived as in Equation 38.

$$\beta_{mtpa} = \cos^{-1}\frac{-\psi_m + \sqrt{\psi_m^2 + 8^*\left(L_d - L_q\right)^2 {}^*I_s^2}}{4^*\left(L_d - L_q\right)^*I_s} \tag{38}$$

Thus, the effective d-axis and q-axis reference currents can be expressed by Equation 39 and Equation 40 using the current angle of the MTPA control.

$$I_d = I_s{}^*\cos\beta_{mtpa} \tag{39}$$

$$I_q = I_s{}^*\sin\beta_{mtpa} \tag{40}$$

However, as shown in Equation 38, the angle of the MTPA control, β_{mtpa} is related to d-axis and q-axis inductance. This means that the variation of inductance will impede the ability to find the optimal MTPA point. To improve the efficiency of a motor drive, the d-axis and q-axis inductance should be estimated online, but the parameters L_d and L_q are not easily measured online and are influenced by saturation effects. A robust Look-Up Table (LUT) method ensures controllability under electrical parameter variations. Usually, to simplify the mathematical model, the coupling effect between d-axis and q-axis inductance can be neglected. Thus, assumes that L_d changes with i_d only, and L_q changes with i_q only. Consequently, d- and q-axis inductance can be modeled as a function of their d-q currents respectively, as shown in Equation 41 and Equation 42.

$$L_d = f_1\left(i_d, i_q\right) = f_1\left(i_d\right) \tag{41}$$

$$L_q = f_2\left(i_q, i_d\right) = f_2\left(i_q\right) \tag{42}$$

To reduce the ISR calculation burden by simplifying Equation 38. The motor-parameter-based constant, K_{mtpa} is expressed instead as Equation 44, where K_{mtpa} is computed in the background loop using the updated L_d and L_q .

$$K_{mtpa} = \frac{\psi_m}{4^*\left(L_q - L_d\right)} = 0.25^*\frac{\psi_m}{\left(L_q - L_d\right)} \tag{43}$$

$$\beta_{mtpa} = cos^{-1}\left(K_{mtpa}/I_s - \sqrt{\left(K_{mtpa}/I_s\right)^2 + 0.5}\right) \tag{44}$$

A second intermediate variable, G_{mtpa} described in Equation 45, is defined to further simplify the calculation. Using G_{mtpa} , the angle of the MTPA control, β_{mtpa} can be calculated as Equation 46. These two calculations are performed in the ISR to achieve a real current angle β_{mtpa} .

$$G_{mtpa} = K_{mtpa}/I_s \tag{45}$$

$$\beta_{mtpa} = cos^{-1}\left(G_{mtpa} - \sqrt{G_{mtpa}^2 + 0.5}\right) \tag{46}$$

In all cases, the magnetic flux can be weakened to extend the achievable speed range by acting on the direct axis current i_d . As a consequence of entering this constant power operating region, field weakening control is chosen instead of the MTPA control used in constant power and voltage regions. Since the maximum inverter voltage is limited, PMSM motors cannot operate in such speed regions where the back-electromotive force, almost proportional to the permanent magnet field and motor speed, is higher than the maximum output voltage of the inverter. The direct control of magnet flux is not an option in PM motors. However, the air gap flux can be weakened by the demagnetizing effect due to the d-axis armature reaction by adding a negative i_d .

Considering the voltage and current constraints, the armature current and the terminal voltage are limited as Equation 29 and Equation 30. The inverter input voltage (DC-Link voltage) variation limits the maximum output of the motor. Furthermore, the maximum fundamental motor voltage also depends on the PWM method used. In Equation 32, the IPMSM has two factors: one is a permanent magnet value and the other is made by inductance and current of flux.

Figure 2-20 shows the typical control structure is used to implement field weakening. β_{fw} is the output of the field weakening (FW) PI controller and generates the reference i_d and i_q . Before the voltage magnitude reaches its limit, the input of the PI controller of FW is always positive and therefore the output is always saturated at 0.

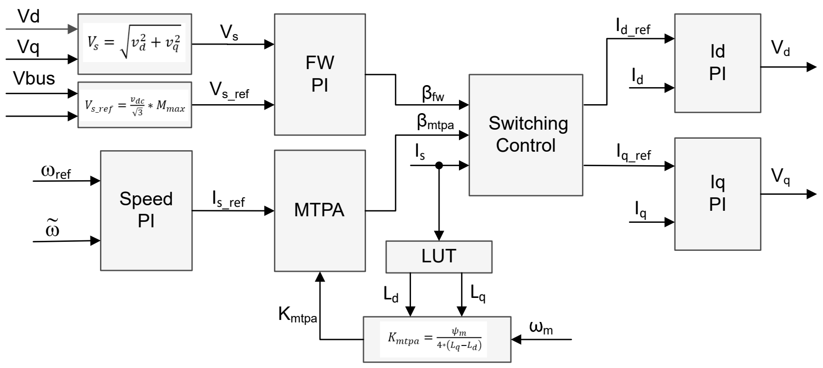


Figure 2-20. Block Diagram of Field-Weakening and Maximum Torque per Ampere Control

Figure 2-11 and Figure 2-13 show the implementation of FAST or eSMO-based FOC block diagram. The block diagrams provide an overview of the FOC system's functions and variables. There are two control modules in the motor drive FOC system: one is MTPA control and the other one is field weakening control. These two modules generate current angle β_{mtpa} and β_{fw} , respectively, based on input parameters as show in Figure 2-21.

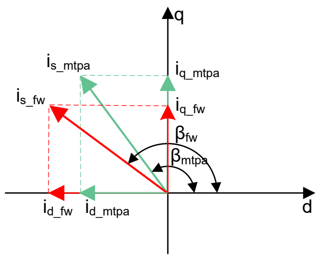


Figure 2-21. Current Phasor Diagram of an IPMSM During FW and MTPA

The switching control module is used to decide which angle should be applied, and then calculate the reference i_d and i_q as shown in Equation 34 and Equation 35. The current angle is chosen as following Equation 47 and Equation 48.

$$\beta = \beta_{fw}\, if\ \beta_{fw} > \beta_{mtpa} \tag{47}$$

$$\beta = \beta_{mpta}\, if\ \beta_{fw} < \beta_{mtpa} \tag{48}$$

Figure 2-22 is the flowchart that shows the steps required to run InstaSPIN-FOC with FW and MPTA in the main loop and interrupt.

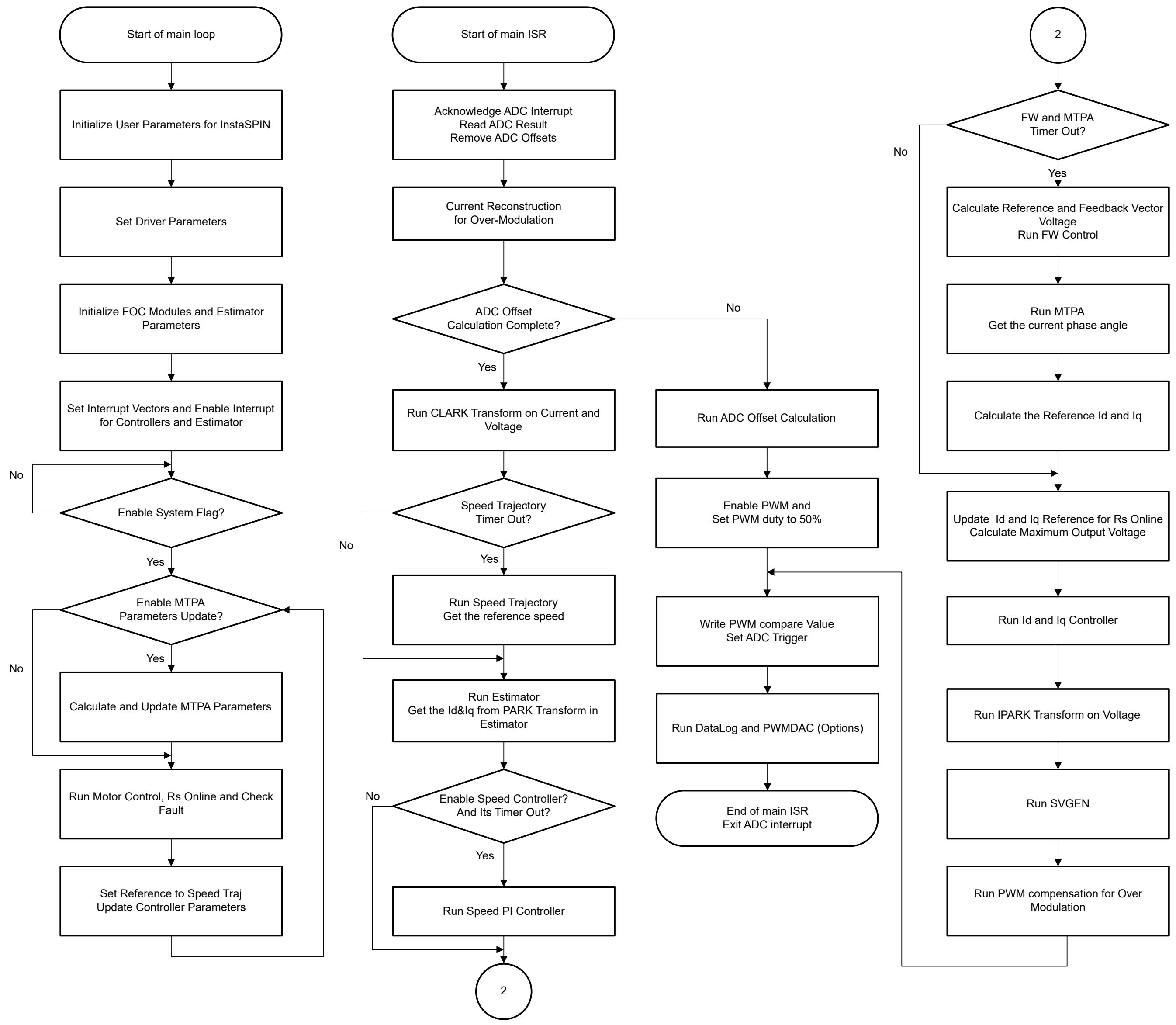

Start of main loop
Initialize User Parameters for InstaSPIN
Set Driver Parameters
Initialize FOC Modules and Estimator Parameters
Set Interrupt Vectors and Enable Interrupt for Controllers and Estimator
Enable System Flag?
No
Yes
Enable MTPA Parameters Update?
No
Yes
Calculate and Update MTPA Parameters
Run Motor Control, Rs Online and Check Fault
Set Reference to Speed Traj Update Controller Parameters
Start of main ISR
Acknowledge ADC Interrupt Read ADC Result Remove ADC Offsets
Current Reconstruction for Over-Modulation
ADC Offset Calculation Complete?
No
Yes
Run CLARK Transform on Current and Voltage
Speed Trajectory Timer Out?
No
Yes
Run Speed Trajectory Get the reference speed
Run Estimator Get the Id&Iq from PARK Transform in Estimator
Enable Speed Controller? And Its Timer Out?
No
Yes
Run Speed PI Controller
2
Run ADC Offset Calculation
Enable PWM and Set PWM duty to 50%
Write PWM compare Value Set ADC Trigger
Run DataLog and PWMDAC (Options)
End of main ISR Exit ADC interrupt
2
FW and MTPA Timer Out?
No
Yes
Calculate Reference and Feedback Vector Voltage Run FW Control
Run MTPA Get the current phase angle
Calculate the Reference Id and Iq
Update Id and Iq Reference for Rs Online Calculate Maximum Output Voltage
Run Id and Iq Controller
Run IPARK Transform on Voltage
Run SVGEN
Run PWM compensation for Over Modulation

Figure 2-22. Flowchart for an InstaSPIN-FOC Project With FW and MTPA

2.5.2 Flying Start

Flying start is a feature that allows the drive to determine the speed and direction of a spinning motor and begin the output voltage and frequency at that speed and direction. Without flying start, the drive will begin its output at zero volts and zero speed and attempt to ramp to the commanded speed. If the inertia or direction of rotation of a load requires the motor to produce a large amount of torque, excess current can result and overcurrent trips might occur on the drive. These problems can be eliminated with flying start.

Flying start is the capacity to start control at any speed other than **ZERO**, which is an important function in air-condition application for fan drive.

When a motor is started in its normal mode, the control initially applies a frequency of 0 Hz and ramps to the desired frequency. If the drive is started in this mode with the motor already spinning with non-zero frequency, large currents are generated. An over current trip can result if the current limiter cannot react quickly enough. Even if the current limiter is fast enough to prevent an over current trip, it can take an unacceptable amount of time for synchronization to occur and for the motor to reach its desired frequency. In addition, larger mechanical stress is placed on the application.

In flying start mode, the drive’s response to a start command is to synchronize with the motor’s speed (frequency and phase) and voltage. The motor then accelerates to the commanded frequency. This process prevents an over current trip and significantly reduces the time for the motor to reach its commanded frequency. Because the drive synchronizes with the motor at its rotating speed and ramps to the proper speed, little or no mechanical stress are present.

The flying start function implements an algorithm that searches for the rotor speed. The algorithm searches for a motor voltage that corresponds with the excitation current applied to the motor

When the motor is spinning, the speed and position information can be estimated from the BEMF voltages. Since the stator voltage is measured in InstaSPIN drive, the speed and position are easily obtained by switching the inverter. A zero torque current is applied to the motor and the generated current and stator voltage is measured, then InstaSPIN-FOC module uses these signals to estimate rotor position and speed.

The block diagram of FOC with flying start is shown in Figure 2-23, the flying start module outputs a flag to enable or disable speed close loop control. A zero reference torque current is set and the speed PI controller output is disabled while flying start is operating.

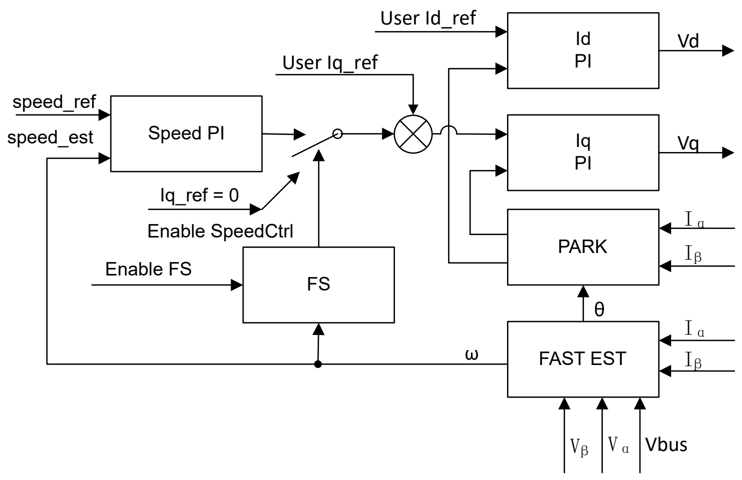


Figure 2-23. Flying Start Control Block Diagram

As shown in Figure 2-24, the module routine disables speed close loop control, sets the reference Iq to zero, and enables the FOC module during starting run the motor. After the phase currents and voltages are measured, the routine runs InstaSPIN-FOC and the real motor speed can be estimated. The program re-enables speed closed loop control and sets the speed reference value after flying start is completed.

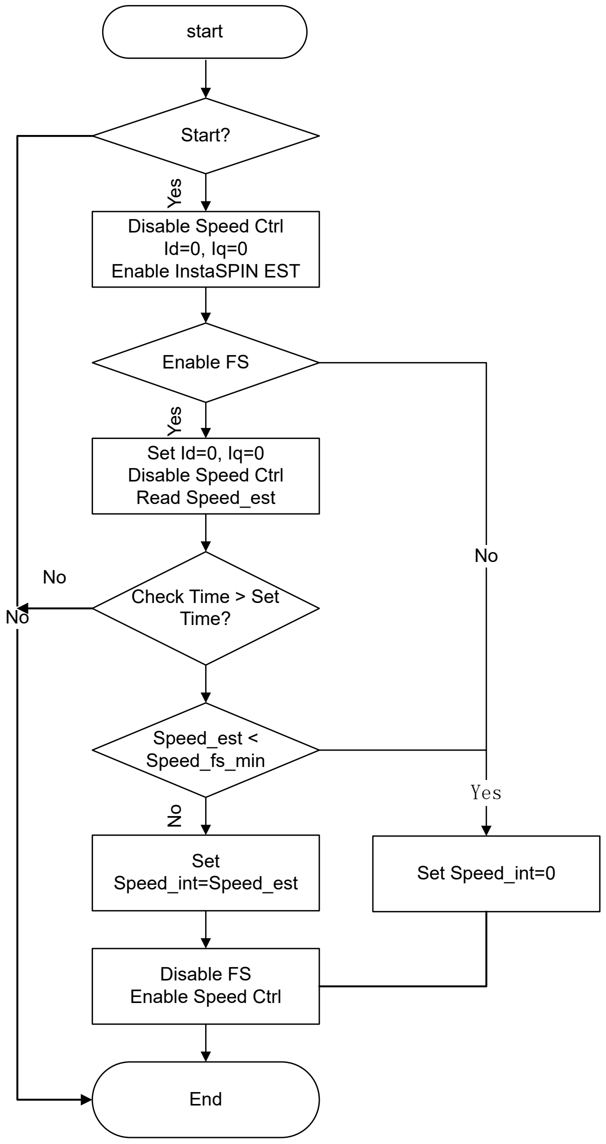


Figure 2-24. Flying Start Module Program Flowchart

3 Running the Universal Lab on TI Hardware Kits

3.1 Supported TI Motor Evaluation Kits

The TMS320F28002x (F28002x), TMS320F28003x (F28003x), or TMS320F280013x (F280013x) is a member of the C2000™ real-time Microcontroller family with IEEE 754 Floating-Point Unit (FPU) and Trigonometric Math Unit (TMU). The user can use one of these LaunchPad™ development kits or controlCARDs with the relevant motor drive evaluation board to evaluate this lab for motor control.

Table 3-1 lists the current evaluation kits that are supported for this universal motor control lab project in MotorControl SDK.

Table 3-1. Motor Drive Evaluation Kits Supported by Motor Control SDK

Motor Drive Evaluation Board		C2000 MCU Evaluation Module	Current Sensing Topology	Rotor Position Sensing Method	Tested Motors
Part Number	Description				
DRV8329AEVM	4.5~60V, 30A 3-ph inverter with CSD18536KTTT NexFET™	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Single shunt dc-link current	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC Sensorless trapzoidalcontrol	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)
BOOSTXL-DRV8323RH	6~54V, 15A 3-ph inverter with CSD88599Q5DC NexFET™ power blocks	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Three low-side current shunt	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)
BOOSTXL-DRV8323RS	6~54V, 15A 3-ph inverter with CSD88599Q5DC NexFETTM power blocks	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Three low-side current shunt	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)
DRV8316REVM	4.5~35V, 8A peak current 3-ph inverter integrated MOSFET	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Integrated CSAs for three-phase low-side current	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)
DRV8353RS-EVM	9~95V, 15A 3-ph inverter with CSD19532Q5B	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Three low-side current shunt	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)

Table 3-1. Motor Drive Evaluation Kits Supported by Motor Control SDK (continued)

Motor Drive Evaluation Board		C2000 MCU Evaluation Module	Current Sensing Topology	Rotor Position Sensing Method	Tested Motors
Part Number	Description				
BOOSTXL-3PHGANINV	12~60V, 3.5A 3-ph GaN inverter	LAUNCHXL-F280025C LAUNCHXL-F280039C LAUNCHXL-F2800137	Three shunt-based inline motor phase current sensing	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC Hall sensors based sensored-FOC	LVSERVOMTR (Encoder Embedded) LVBLDCMTR (Hall Sensors Embedded)
TMDSHVMTRINSPIN	400V, 10A 3-ph inverter	TMDSCNCD280025C, TMDSCNCD280039C, TMDSCNCD2800137, with TMDSADAP180TO100	Three low-side current shunt	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC	HVPMSMMTR (Encoder Embedded) HVBLDCMTR (Hall Sensors Embedded)

If the lab is set to use Encoder or Hall based sensored-FOC, it is important to ensure that the physical connections are connected in the correct order. If the motor, encoder, or hall wires are connected in the wrong order, the lab will not function properly, potentially resulting in the motor being unable to spin. For the motor phase wires, it is important to ensure that the motor phases are connected to the right phase on the inverter board. For the motors that are provided with the TI Motor Control Reference Kits, the correct phase connections are provided as shown in Table 3-2.

For the encoder, it is important to ensure that A is connected to A, B to B, and I to I. For the Hall sensor, it is important to ensure that A is connected to A, B to B, and C to C. Often +5V dc and ground connections are required as well. If you are using Hall sensors or encoders that are different than the ones specifically listed in Table 2-2, please refer to the users manual for the Hall sensor or encoder you are using to ensure that you properly connect the wires.

It is important for the setup and configuration of the ENC module that the number of slots per rotation for the encoder be provided. This allows the ENC module to correctly convert the encoder signal into an angle. The USER_MOTOR1_NUM_ENC_SLOTS constant that is defined in the *user_mtr1.h* file needs to be updated to the correct value for your encoder. If this value is not correct, the motor will spin faster or slower depending on the value that was set. It is important to note that this value should be set to the number of slots on the encoder, not the resulting number of counts after figuring the quadrature accuracy.

Table 3-2. Motor Phase, Encoder, or Hall Sensors Connections for Reference Kits and Motors

		LVSERVOMTR	LVBLDCMTR	HVPMSMMTR	HVBLDCMTR
Motor Phase Lines	U	BLACK (16AWG)	YELLOW	RED	YELLOW
	V	RED (16AWG)	RED	BLUE/BLACK	RED
	W	WHITE (16AWG)	BLACK	WHITE	BLACK
Encoder	GND (J12-1 of LAUNCHXL-F280025C/39C/137)	BLACK (J4-1)	N/A, Not support for encoder based sensored-FOC	BLACK	Not support for encoder based sensored-FOC
	+5V	RED (J4-2)		RED	
	I (1I, J12-3 of LAUNCHXL-F280025C/39C/137)	BROWN (J4-3)		YELLOW	
	B (1B, J12-4 of LAUNCHXL-F280025C/39C/137)	ORANGE (J4-4)		GREEN	
	A (1A, J12-5 of LAUNCHXL-F280025C/39C/137)	BLUE (J4-1)		BLUE	

Table 3-2. Motor Phase, Encoder, or Hall Sensors Connections for Reference Kits and Motors (continued)

		LVSERVOMTR	LVBLDCMTR	HVPMSMMTR	HVBLDCMTR
Hall Sensors (LAUCHXL_F2800137 only has J12, Hall sensors share the J12 with Encoder)	GND	BLACK (J10-1)	BLACK	Not support for Hall sensor based sensored-FOC	BLACK
	+5V	RED (J10-2)	RED		RED
	A (2I, J13-3 of LAUNCHXL-F280025C/39C)	GRAY-WHITE (J10-3)	BLUE		BLUE
	B (2B, J13-4 of LAUNCHXL-F280025C/39C)	GREEN-WHITE (J10-4)	GREEN		GREEN
	C (2A, J13-5 of LAUNCHXL-F280025C/39C)	GREEN (J10-5)	WHITE		WHITE

Get started with C2000™ Real-Time Control Microcontrollers (MCUs) to implement motor control.

1. Step 1: Order the desired motor drive evaluation board, C2000 MCU evaluation module, and motor as shown in Table 3-1.
2. Step 2: Download the latest version of MotorControl SDK.
3. Step 3: Download the latest version of Code Composer Studio IDE.
4. Step 4: Follow the instructions in Section 3.2 to setup the hardware and run the example labs described in the following sections.
5. Step 5: For answers to any design questions that you may have, you can search existing answers or ask your own question using the TI C2000 E2E forum.

3.2 Hardware Board Setup

This section describes how to set up hardware boards for motor control when combining the motor driver evaluation board with the C2000 development tools. The following sections show the detailed operation procedure on different motor driver evaluation boards.

3.2.1 LAUNCHXL-F280025C Setup

LAUNCHXL-F280025C is a low-cost development board for the TI C2000 real-time microcontrollers series of F28002x devices. This LaunchPad™ kit offers extra pins for development and supports the connection of two BoosterPack™ plug-in modules.

- The hardware files are in the `<install_location>\boards\LaunchPads\LAUNCHXL_F280025C` folder of C2000Ware.
- For more details about the LAUNCHXL-F280025C, see the *F28002x Series LaunchPad™ Development Kit User's Guide*.
- Make sure that the switches on the LAUNCHXL-F280025C are set as described below shown in Figure 3-1.
  - Install jumpers on JP1, JP2, JP3 and J101 for the power supply and debug JTAG. And jumpers on JP8 for the power supply of DAC128S board if used.
  - For **S2**, position the SEL1 (LEFT) switch **UP** (1) to route GPIO28 and GPIO29 to the BoosterPack connector, and position the SEL2 (RIGHT) switch **UP** (1) to rout GPIO16 and GPIO17 to the virtual COM port of the XDS110 debugger.
  - For **S3**, position the SEL1(LEFT) switch **DOWN** to pull GPIO24 low to logic 0, and position the SEL2(RIGHT) switch **UP** to pull GPIO32 high to logic 1 to put the F280025C into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
  - For **S4**, set S4 to **DOWN** (on), GPIO32 and GPIO33 are routed to the CAN transceiver to J14 CAN interface if the pre-define symbols "CMD_CAN_EN" is set in project properties.
  - For **S5**, position the QEP1 SEL(LEFT) switch **DOWN** to route GPIO44/37/43 to eQEP1 for the encoder interface on J12, and position the QEP2 SEL (RIGHT) switch **DOWN** to route GPIO14/25/26 to eQEP2 for the Hall sensor interface on J13.

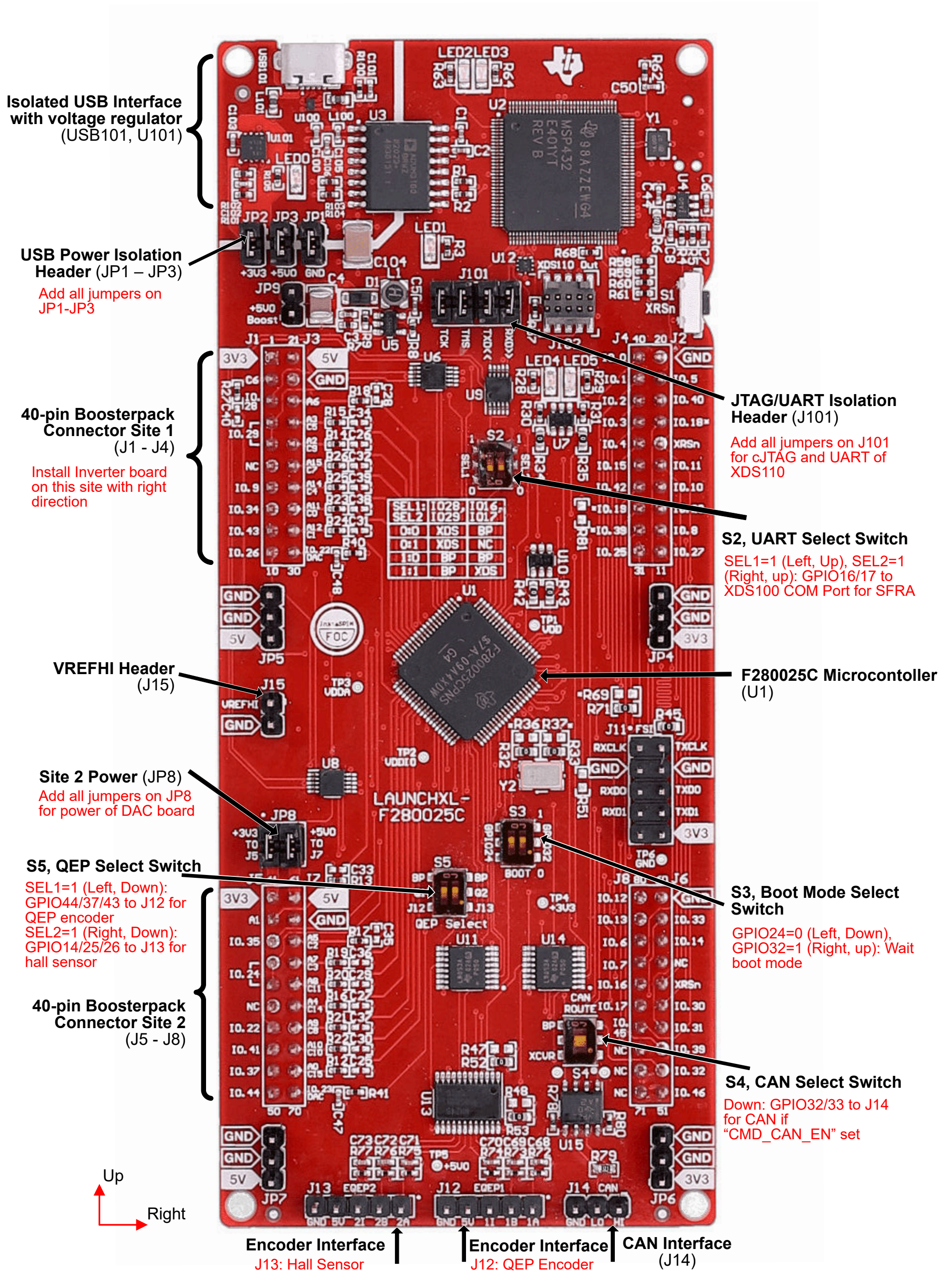

Isolated USB Interface with voltage regulator (USB101, U101)
USB Power Isolation Header (JP1 – JP3)
Add all jumpers on JP1-JP3
40-pin Boosterpack Connector Site 1 (J1 - J4)
Install Inverter board on this site with right direction
VREFHI Header (J15)
Site 2 Power (JP8)
Add all jumpers on JP8 for power of DAC board
S5, QEP Select Switch
SEL1=1 (Left, Down): GPIO44/37/43 to J12 for QEP encoder
SEL2=1 (Right, Down): GPIO14/25/26 to J13 for hall sensor
40-pin Boosterpack Connector Site 2 (J5 - J8)
Up
Right
Encoder Interface
J13: Hall Sensor
Encoder Interface
J12: QEP Encoder
CAN Interface (J14)
JTAG/UART Isolation Header (J101)
Add all jumpers on J101 for cJTAG and UART of XDS110
S2, UART Select Switch
SEL1=1 (Left, Up), SEL2=1 (Right, up): GPIO16/17 to XDS100 COM Port for SFRA
F280025C Microcontoller (U1)
S3, Boot Mode Select Switch
GPIO24=0 (Left, Down), GPIO32=1 (Right, up): Wait boot mode
S4, CAN Select Switch
Down: GPIO32/33 to J14 for CAN if "CMD_CAN_EN" set

Figure 3-1. F280025C LaunchPad Board Overview and Switches Setting

3.2.2 LAUNCHXL-F280039C Setup

LAUNCHXL-F280039C is a low-cost development board for the TI C2000 real-time microcontrollers series of F28003x devices. This LaunchPad™ kit offers extra pins for development and supports the connection of two BoosterPack™ plug-in modules.

- The hardware files are in the `<install_location>\boards\LaunchPads\LAUNCHXL_F280039C` folder of C2000Ware.
- For more details about the LAUNCHXL-F280039C, see the *C2000™ F28003x Series LaunchPad™ Development Kit User's Guide*.

- Make sure that the switches on the LAUNCHXL-F280039C are set as described below shown in Figure 3-2.
  - Install jumpers on JP1, JP2 and J101 for the power supply and debug JTAG. And install jumpers on JP8 for the power supply of DAC128S board if used.
  - For **S2**, position the SEL1 (LEFT) switch **UP** (1) to route GPIO28 and GPIO29 to the BoosterPack connector, and position the SEL2 (RIGHT) switch **UP** (1) to rout GPIO15 and GPIO56 to the virtual COM port of the XDS110 debugger.
  - For **S3**, position the GPIO24 (LEFT) switch **DOWN** to pull GPIO24 low to logic 0, and position the GPIO32 (RIGHT) switch **UP** to pull GPIO32 high to logic 1 to put the F280039C into wait boot mode to reduce the risk of connectivity issues or a previous loaded code execution.
  - For **S4**, set S4 to **DOWN** (on) to route GPIO4 and GPIO5 to the CAN transceiver interface J14 if the pre-define symbol "CMD_CAN_EN" is set in the project properties. Set S4 to **UP** (off) to route GPIO4 and GPIO 5 to the BoosterPack connectors otherwise.
  - For **S5**, position the QEP1 SEL (LEFT) switch **DOWN** to route GPIO40/41/59 to the eQEP1 encoder interface on J12 and position the QEP2 SEL (RIGHT) switch **DOWN** to route GPIO14/55/57 to the eQEP2 Hall sensor interface on J13.

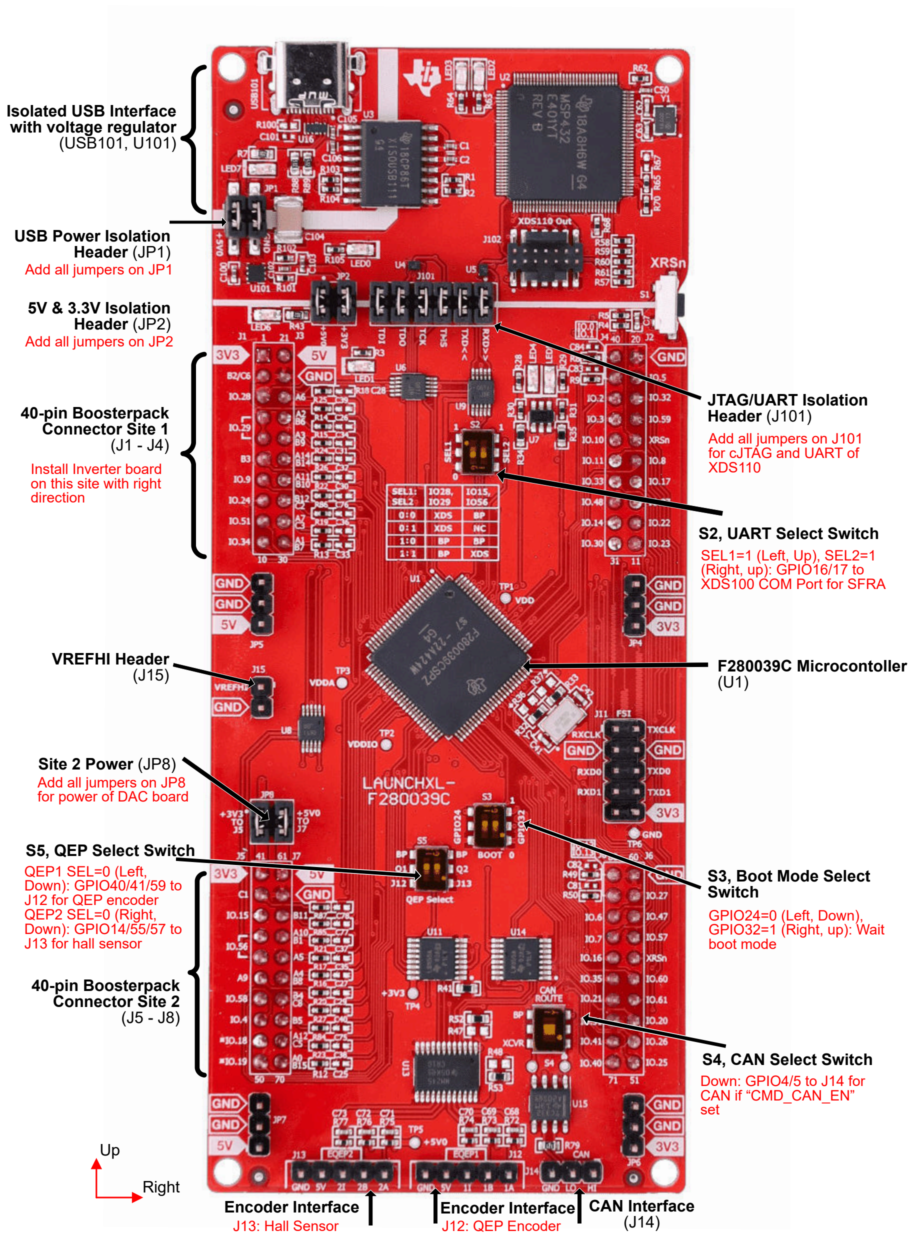

Isolated USB Interface with voltage regulator (USB101, U101)
USB Power Isolation Header (JP1)
Add all jumpers on JP1
5V & 3.3V Isolation Header (JP2)
Add all jumpers on JP2
40-pin Boosterpack Connector Site 1 (J1 - J4)
Install Inverter board on this site with right direction
VREFHI Header (J15)
Site 2 Power (JP8)
Add all jumpers on JP8 for power of DAC board
S5, QEP Select Switch
QEP1 SEL=0 (Left, Down): GPIO40/41/59 to J12 for QEP encoder
QEP2 SEL=0 (Right, Down): GPIO14/55/57 to J13 for hall sensor
40-pin Boosterpack Connector Site 2 (J5 - J8)
Up
Right
Encoder Interface
J13: Hall Sensor
Encoder Interface
J12: QEP Encoder
CAN Interface (J14)
JTAG/UART Isolation Header (J101)
Add all jumpers on J101 for cJTAG and UART of XDS110
S2, UART Select Switch
SEL1=1 (Left, Up), SEL2=1 (Right, up): GPIO16/17 to XDS100 COM Port for SFRA
F280039C Microcontoller (U1)
S3, Boot Mode Select Switch
GPIO24=0 (Left, Down), GPIO32=1 (Right, up): Wait boot mode
S4, CAN Select Switch
Down: GPIO4/5 to J14 for CAN if "CMD_CAN_EN" set
LAUNCHXL-F280039C

Figure 3-2. F280039C LaunchPad Board Overview and Switches Setting

3.2.3 LAUNCHXL-F2800137 Setup

LAUNCHXL-F2800137 is a low-cost development board for the TI C2000 real-time microcontrollers series of F280013x devices. This LaunchPad™ kit offers extra pins for development and supports the connection of two BoosterPack™ plug-in modules.

- The hardware files are in the `<install_location>\boards\LaunchPads\LAUNCHXL_F2800137` folder of C2000Ware.
- For more details about the LAUNCHXL-F2800137, see the *C2000™ F2800137 Series LaunchPad™ Development Kit User's Guide*.
- Make sure that the switches on the LAUNCHXL-F2800137 are set as described below shown in Figure 3-3.
  – Install jumpers on JP1, JP2 and J101 for the power supply and debug JTAG. And install jumpers on JP8 for the power supply of DAC128S board if used.
  – For **S2**, position the SEL1 switch **UP** (1) to route GPIO28 and GPIO29 to the BoosterPack connector. There is only SFRA support on the LAUNCHXL-F2800137 with the BOOSTXL-3PHGANINV via the virtual COM port of the XDS110 debugger by setting SEL1 to **DOWN** (0). The other kits need the GPIO29 to enable the DRV device. This is due to the pin constraints on the LAUNCHXL-F2800137.
  – For **S3**, position the GPIO24 (LEFT) switch **DOWN** to pull GPIO24 low to logic 0, and position the GPIO32 (RIGHT) switch **UP** to pull GPIO32 high to logic 1 to put the F2800137 into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
  – For **S4**, position the CAN ROUTE switch **DOWN** (on) to route GPIO4 and GPIO5 to the J14 CAN interface if the pre-define symbols "CMD_CAN_EN" is set in project properties. Position the CAN ROUTE switch **UP** (off) to route GPIO4 and GPIO 5 to the BoosterPack connectors otherwise.
  – For **S5**, position the QEP1 SEL switch **DOWN** to route GPIO40/41/39 to eQEP1 for the encoder interface on J12. (There is no J13 on LAUNCHXL-F2800137. Hall sensor reading is unavailable by default.)

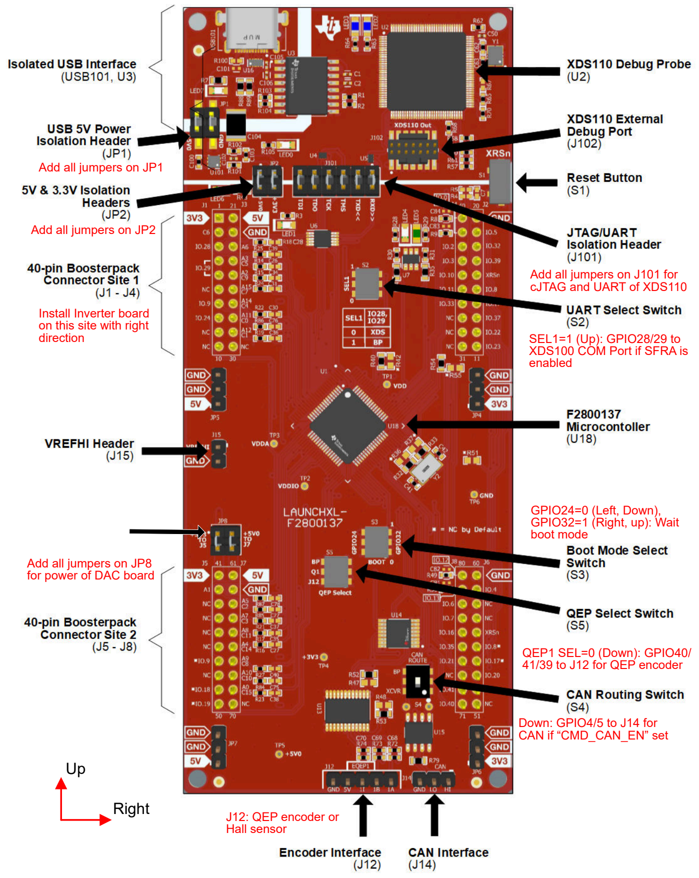
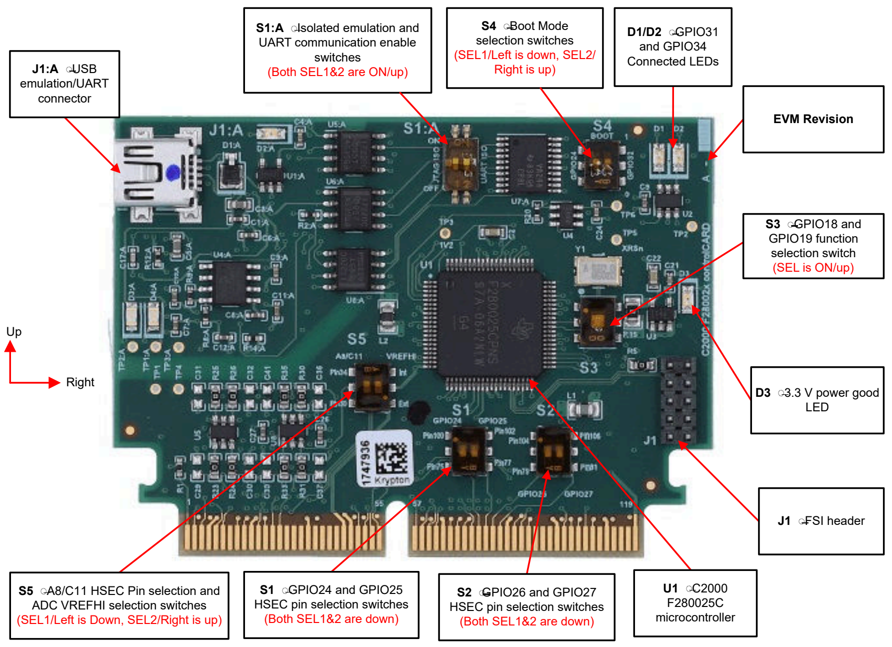

Isolated USB Interface (USB101, U3)
USB 5V Power Isolation Header (JP1)
Add all jumpers on JP1
5V & 3.3V Isolation Headers (JP2)
Add all jumpers on JP2
40-pin Boosterpack Connector Site 1 (J1 - J4)
Install Inverter board on this site with right direction
VREFHI Header (J15)
Add all jumpers on JP8 for power of DAC board
40-pin Boosterpack Connector Site 2 (J5 - J8)
Up
Right
J12: QEP encoder or Hall sensor
Encoder Interface (J12)
CAN Interface (J14)
XDS110 Debug Probe (U2)
XDS110 External Debug Port (J102)
Reset Button (S1)
JTAG/UART Isolation Header (J101)
Add all jumpers on J101 for cJTAG and UART of XDS110
UART Select Switch (S2)
SEL1=1 (Up): GPIO28/29 to XDS100 COM Port if SFRA is enabled
F2800137 Microcontoller (U18)
GPIO24=0 (Left, Down), GPIO32=1 (Right, up): Wait boot mode
Boot Mode Select Switch (S3)
QEP Select Switch (S5)
QEP1 SEL=0 (Down): GPIO40/41/39 to J12 for QEP encoder
CAN Routing Switch (S4)
Down: GPIO4/5 to J14 for CAN if "CMD_CAN_EN" set
LAUNCHXL-F2800137

Figure 3-3. F2800137 LaunchPad Board Overview and Switches Setting

3.2.4 TMDSCNCD280025C Setup

TMDSCNCD280025C is a low-cost evaluation and development board for TI C2000™ MCU series of TMS320F28002x devices. It comes with a HSEC180 (180-pin High Speed Edge Connector) and can be used on existing 100-Pin DIMM based TMDSHVMTRINSPIN with TMDSADAP180TO100 adapter

- Hardware Files are in the `<install_location>\boards\controlCARDs\TMDSCNCD280025C` folder of C2000Ware.
- For more details about on TMDSCNCD280025C , see TMS320F280025C controlCARD Information Guide.
- Make sure that switches on TMDSCNCD280025C are set as described below shown in Figure 3-4.
  - For S1:A, position both SEL1 (left) and SEL2 (right) switches **UP** for using the on-Card XDS100v2 emulator and ISO UART communication.
  - For S1, position both SEL1 (left) and SEL2 (right) switches **DOWN** for routing GPIO24 and GPIO25 to HSEC connector.
  - For S2, position both SEL1 (left) and SEL2 (right) switches **DOWN** for routing GPIO26 and GPIO27 to HSEC connector.
  - For S3, position the switch **UP** to enable the external crystal.
  - For S4, position the SEL1 (left) switch **DOWN** to pull GPIO24 low to logic 0, and position the SEL2 (right) switch **UP** to pull GPIO32 high to logic 1 to put the F280025C into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
  - For S5, position the SEL1 (left) switch **DOWN** so that A8/C11 is routed to HSEC pin 30, and position the SEL2 (right) switch **UP** to enable the internal voltage reference.



J1:A –USB emulation/UART connector
S1:A –Isolated emulation and UART communication enable switches
(Both SEL1&2 are ON/up)
S4 –Boot Mode selection switches
(SEL1/Left is down, SEL2/Right is up)
D1/D2 –GPIO31 and GPIO34 Connected LEDs
EVM Revision
S3 –GPIO18 and GPIO19 function selection switch
(SEL is ON/up)
Up
Right
D3 –3.3 V power good LED
J1 –FSI header
S5 –A8/C11 HSEC Pin selection and ADC VREFHI selection switches
(SEL1/Left is Down, SEL2/Right is up)
S1 –GPIO24 and GPIO25 HSEC pin selection switches
(Both SEL1&2 are down)
S2 –GPIO26 and GPIO27 HSEC pin selection switches
(Both SEL1&2 are down)
U1 –C2000 F280025C microcontroller

Figure 3-4. F28002x controlCARD and Switches Setting

3.2.5 TMDSCNCD280039C Setup

TMDSCNCD280039C is a low-cost evaluation and development board for TI C2000 MCU series of TMS320F28003x devices. It comes with a HSEC180 (180-pin High Speed Edge Connector) and can be used on existing 100-Pin DIMM based TMDSHVMTRINSPIN with TMDSADAP180TO100 adapter

- Hardware Files are in the `<install_location>\boards\controlCARDs\TMDSCNCD280039C` folder of C2000Ware.
- For more details about on TMDSCNCD280039C , see TMS320F280039C controlCARD Information Guide.
- Make sure that switches on TMDSCNCD280039C are set as described below shown in Figure 3-5.
  - For S1:A, position both SEL1 (up) and SEL2 (down) switches **LEFT** for using the on-Card XDS110 emulator and ISO UART communication.
  - For S1, position the switch **UP** to enable the external crystal.
  - For S2, position the SEL1 (left) switch **DOWN** to pull GPIO24 low to logic 0, and position the SEL2 (right) switch **UP** to pull GPIO32 high to logic 1 to put the F280039C into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
  - For S3, position the switch **DOWN** to enable the internal voltage reference.

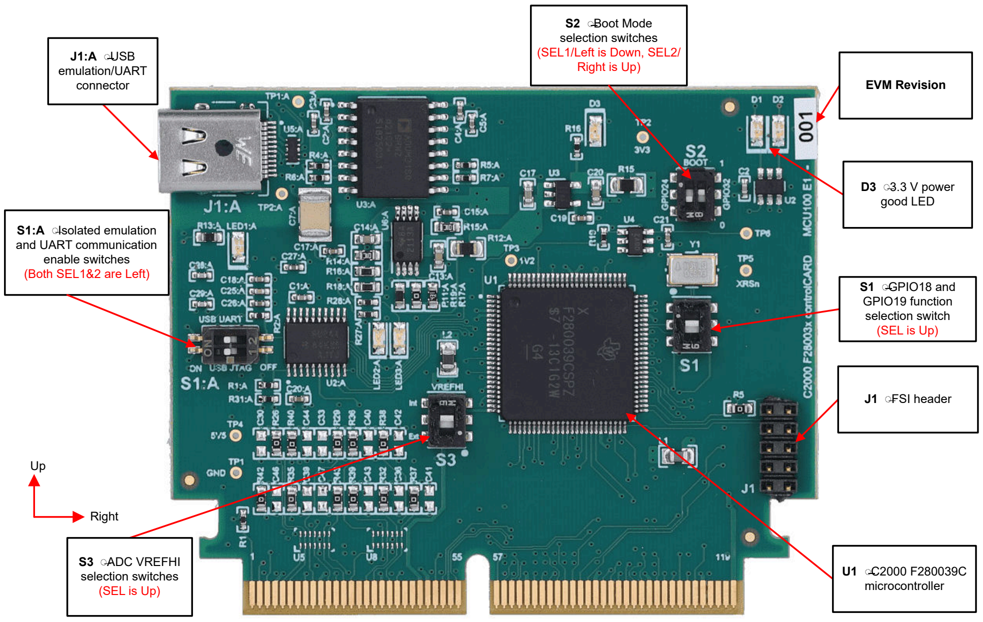

J1:A USB emulation/UART connector
S2 Boot Mode selection switches (SEL1/Left is Down, SEL2/Right is Up)
EVM Revision
D3 3.3 V power good LED
S1:A Isolated emulation and UART communication enable switches (Both SEL1&2 are Left)
S1 GPIO18 and GPIO19 function selection switch (SEL is Up)
J1 FSI header
Up
Right
S3 ADC VREFHI selection switches (SEL is Up)
U1 C2000 F280039C microcontroller

Figure 3-5. F280039C controlCARD and Switches Setting

3.2.6 TMDSCNCD2800137 Setup

TMDSCNCD2800137 is a low-cost evaluation and development board for TI C2000 MCU series of TMS320F280013x devices. It comes with a HSEC180 (180-pin High Speed Edge Connector) and can be used on existing 100-Pin DIMM based TMDSHVMTRINSPIN with TMDSADAP180TO100 adapter

- Hardware Files are in the `<install_location>\boards\controlCARDs\TMDSCNCD2800137` folder of C2000Ware.
- For more details about on TMDSCNCD2800137 , see TMS320F2800137 controlCARD Information Guide.
- Make sure that switches on TMDSCNCD2800137 are set as described below shown in Figure 3-6.
  - For S1:A, position both SEL1 (up) and SEL2 (down) switches **LEFT** for using the on-Card XDS110 emulator and ISO UART communication.
  - For S1, position the switche **UP** for routing GPIO35 and GPIO37 to HSEC connector.
  - For S2, position the switch **UP** to enable the external crystal.
  - For S3, position the SEL1 (left) switch **DOWN** to pull GPIO24 low to logic 0, and position the SEL2 (right) switch **UP** to pull GPIO32 high to logic 1 to put the F2800137 into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
  - For S4, position the SEL1 (left) switch **DOWN** so that A8/C11 is routed to HSEC pin 30, and position the SEL2 (right) switch **UP** to enable the internal voltage reference.

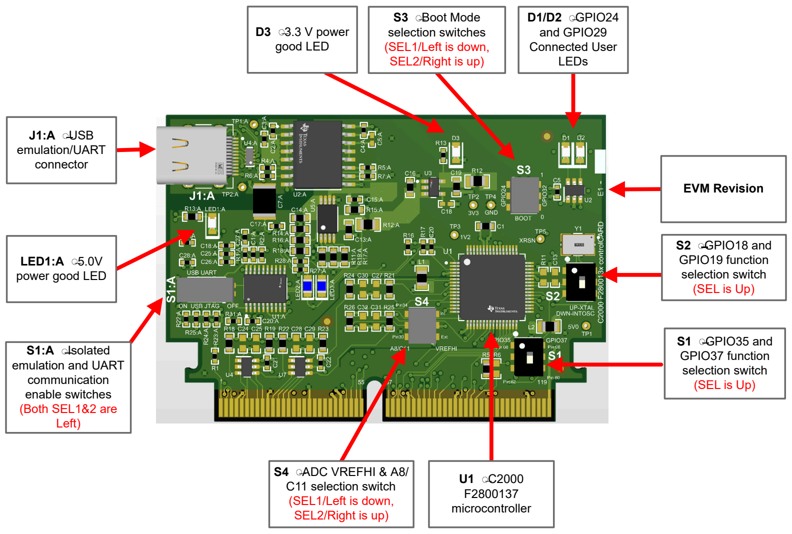

D3 – 3.3 V power good LED
S3 – Boot Mode selection switches (SEL1/Left is down, SEL2/Right is up)
D1/D2 – GPIO24 and GPIO29 Connected User LEDs
J1:A – USB emulation/UART connector
EVM Revision
LED1:A – 5.0V power good LED
S2 – GPIO18 and GPIO19 function selection switch (SEL is Up)
S1:A – Isolated emulation and UART communication enable switches (Both SEL1&2 are Left)
S1 – GPIO35 and GPIO37 function selection switch (SEL is Up)
S4 – ADC VREFHI & A8/C11 selection switch (SEL1/Left is down, SEL2/Right is up)
U1 – C2000 F2800137 microcontroller

Figure 3-6. F2800137 controlCARD and Switches Setting

3.2.7 TMDSADAP180TO100 Setup

The TMDSADAP180TO100 adapter allows the use of 180-Pin C2000 controlCARDs with existing 100-Pin DIMM based evaluation tools. The TMDSCNCD280025C, TMDSCNCD280039C or TMDSCNCD2800137 controlCARD needs TMDSADAP180TO100 to be used on TMDSHVMTRINSPIN.

- Hardware files are located in the `<install_location>\boards\controlCARDs\TMDSADAP180TO100` folder of C2000Ware.
- Make sure that switches TMDSADAP180TO100 are set as described below or shown in Figure 3-7.
  - The S1 switch needs to be positioned to the **RIGHT**, and S2, S3, and S4 switches need to be positioned to the **LEFT**.

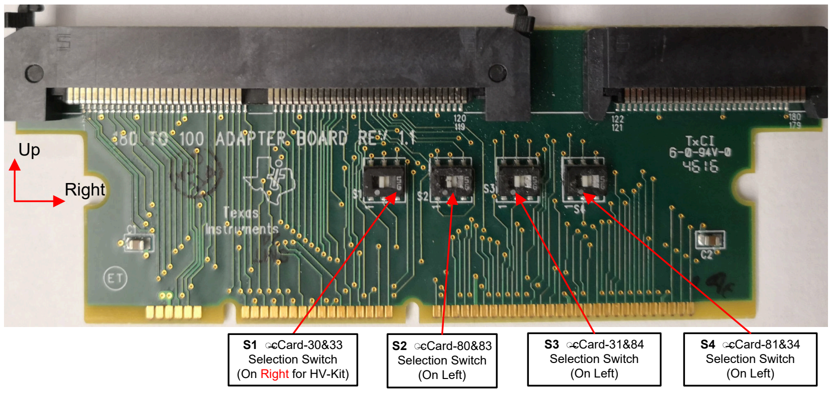

Up
Right
S1 cCard-30&33 Selection Switch (On Right for HV-Kit)
S2 cCard-80&83 Selection Switch (On Left)
S3 cCard-31&84 Selection Switch (On Left)
S4 cCard-81&34 Selection Switch (On Left)

Figure 3-7. TMDSADAP180TO100 Adapter and Switches Setting

3.2.8 DRV8329AEVM Setup

DRV8329AEVM is a 4.5V-60V, 30-A, 3-phase brushless DC drive stage based on the DRV8329A gate driver and CSD18536KT MOSFET for BLDC/PMSM motors . The module has an individual DC bus and three-phase voltage sensing as well as single shunt dc-link current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad development kits suitable for use with the sensorless InstaSPIN-FOC algorithm. The drive stage is fully protected with short circuit, thermal, shoot-through, and under voltage protection and easily configurable via the device's GPIO interfacing with C2000 MCUs.

- The DRV8329AEVM Hardware Design Files are on the DRV8329AEVM page within ti.com.
- For more details about the DRV8329AEVM, see the User's Guide for the EVM.
- Make sure that the following items are completed as described below, and then connect the DRV8329AEVM to site 1 (J1/J3 and J4/J2) of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as shown in Figure 3-8.
  - The DRV8329AEVM connects on the top of the launchpad, so it is necessary to use male to female adapters to connect the headers on the DRV8329AEVM to the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 headers.
  - If using the DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables.
- Connect the motor, encoder, and Hall sensors to the DRV8329AEVM and the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as described in Table 3-2 and shown in Figure 3-8.
- Connect a supply voltage ranging from 9 to 54 V from a battery or a DC voltage source to the voltage supply pins. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

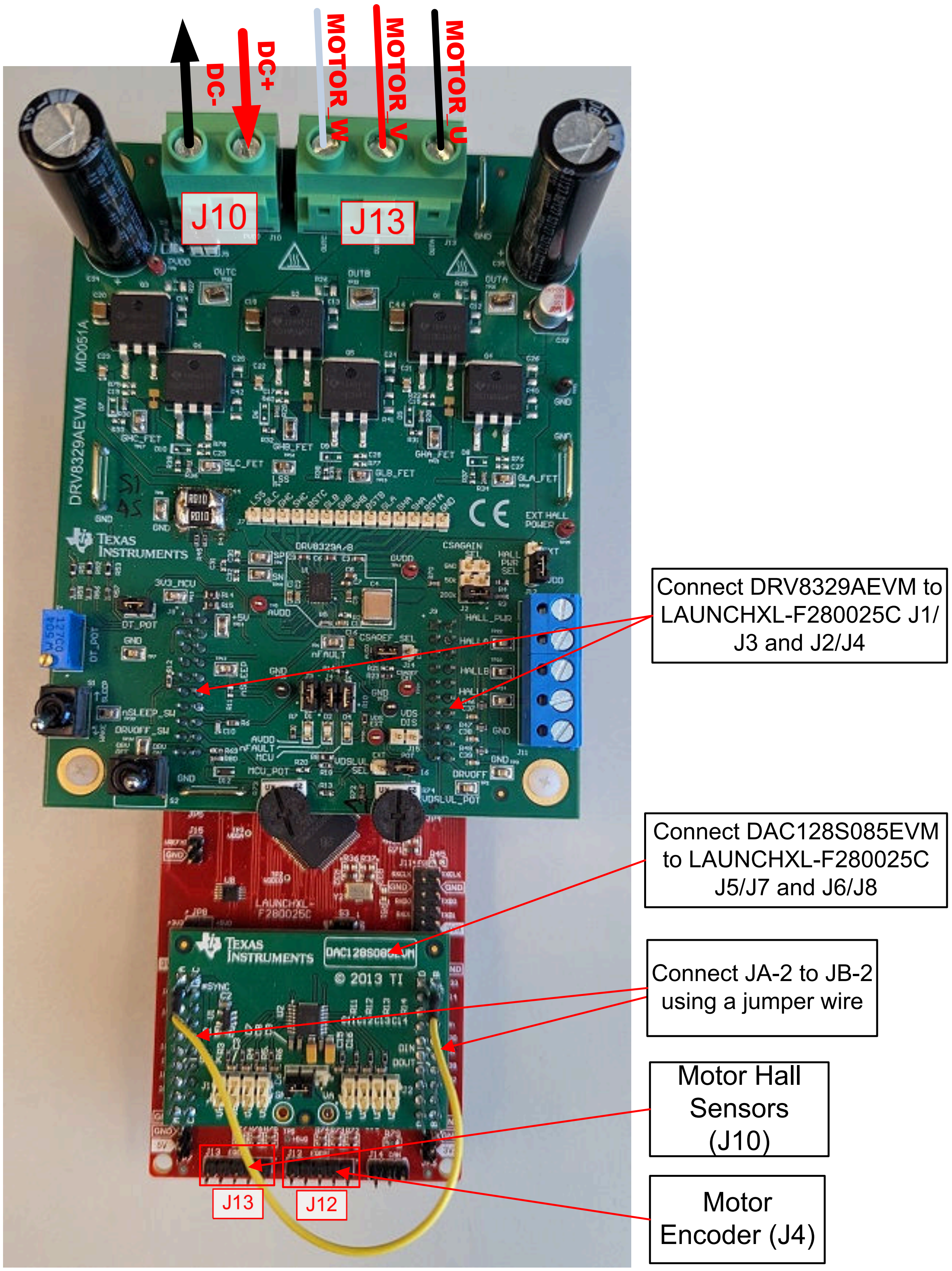

DC-
DC+
MOTOR_W
MOTOR_V
MOTOR_U
J10
J13
Connect DRV8329AEVM to LAUNCHXL-F280025C J1/J3 and J2/J4
Connect DAC128S085EVM to LAUNCHXL-F280025C J5/J7 and J6/J8
Connect JA-2 to JB-2 using a jumper wire
Motor Hall Sensors (J10)
Motor Encoder (J4)
J13
J12

Figure 3-8. LAUNCHXL-F280025C Connected to the DRV8329AEVM and DAC128S085EVM

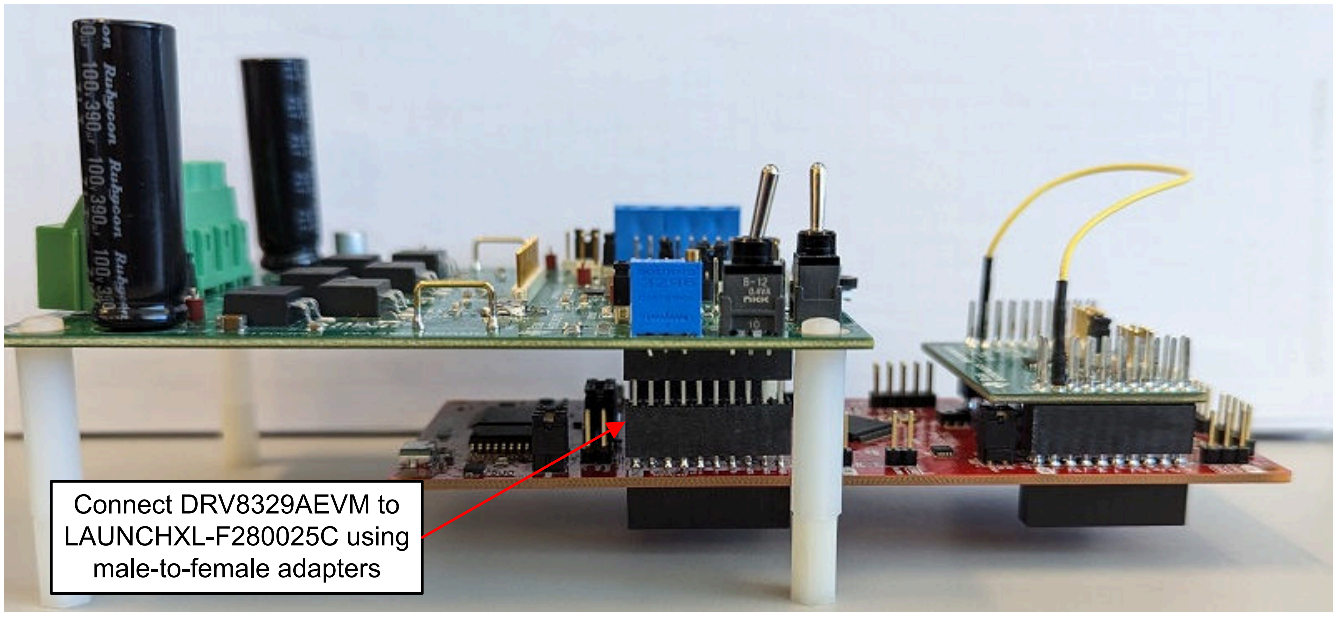

Connect DRV8329AEVM to LAUNCHXL-F280025C using male-to-female adapters

Figure 3-9. LAUNCHXL-F280025C Connected to the DRV8329AEVM and DAC128S085EVM, Side View

3.2.9 BOOSTXL-DRV8323RH Setup

BOOSTXL-DRV8323RH is a 15A, three-phase brushless DC drive stage based on the DRV8323RH gate driver and CSD88599Q5DC NexFETTM power blocks. The module has an individual DC bus and three-phase voltage sensing as well as three low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with the sensorless InstaSPIN-FOC algorithm.

- The BOOSTXL-DRV8323RH Hardware Files are on the BOOSTXL-DRV8323RH page within ti.com.
- For more details about the BOOSTXL-DRV8323RH, see the User's Guide for the EVM.
- Make sure that the following items are completed as described below, and then connect the BOOSTXL-DRV8323RH to J1/J3 and J4/J2 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as shown in Figure 3-10.
  - Populate C9, C10, and C11 with a **47nF** capacitor.
  - Bend J3-29 and J3-30 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90° so that they are not connected to the BOOSTXL-DRV8323RH as shown in Figure 3-11.
  - If using the DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables.
  - Use a jumper wire to connect J3-29 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 to J3-11 of the BOOSTXL-DRV8323RH if the potentiometer functionality is desired to be used on the BOOSTXL-DRV8323RH for setting the reference speed.
- Connect the motor, encoder, and Hall sensors to the BOOSTXL-DRV8323RH and LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 boards as described in Table 3-2 and shown in Figure 3-10.
  - Connect a supply voltage ranging from 6 to 40 V from a battery or a DC voltage source to the voltage supply pins as shown in Figure 3-10. The maximum supply voltage rating for the BOOSTXL-DRV8323RH is 65V, so care must be taken to ensure that the voltage does not exceed this value during operation. This is particularly important while slowing down or braking the motor, which can cause the supply voltage to rise substantially. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

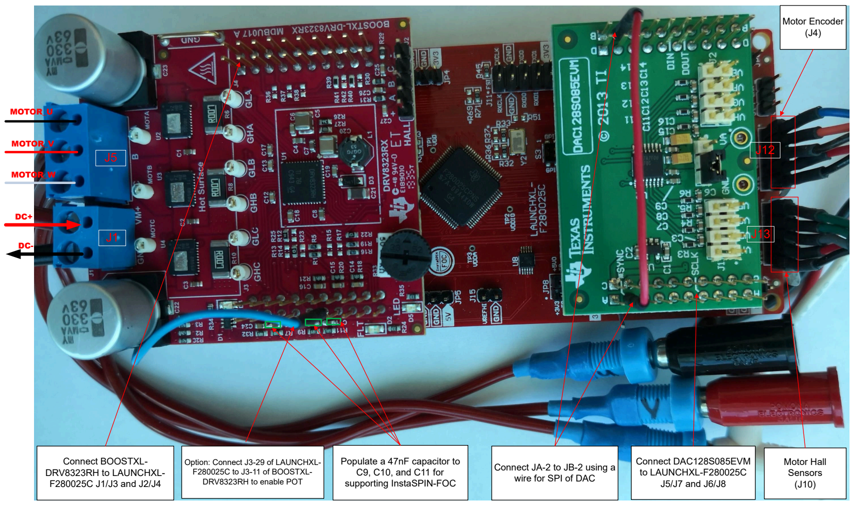

Motor Encoder (J4)
MOTOR U
MOTOR V
MOTOR W
DC+
DC-
J5
J1
J12
J13
Connect BOOSTXL-DRV8323RH to LAUNCHXL-F280025C J1/J3 and J2/J4
Option: Connect J3-29 of LAUNCHXL-F280025C to J3-11 of BOOSTXL-DRV8323RH to enable POT
Populate a 47nF capacitor to C9, C10, and C11 for supporting InstaSPIN-FOC
Connect JA-2 to JB-2 using a wire for SPI of DAC
Connect DAC128S085EVM to LAUNCHXL-F280025C J5/J7 and J6/J8
Motor Hall Sensors (J10)

Figure 3-10. LAUNCHXL-F280025C/F280039C/F2800137 Connected to the BOOSTXL-DRV8323RH and the DAC128S085EVM

The switches and connections on LAUNCHXL-F280025C as shown in Figure 3-11 can be used with the BOOSTXL-DRV8323RH, BOOSTXL-DRV8323RS , and DRV8353RS-EVMs.

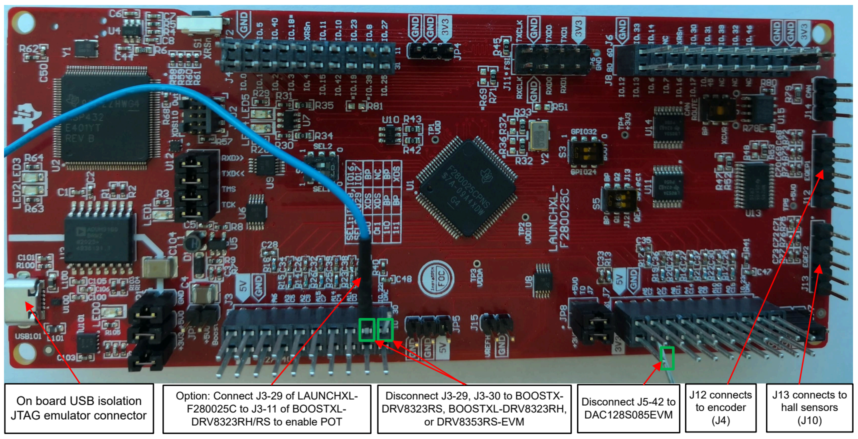

On board USB isolation JTAG emulator connector
Option: Connect J3-29 of LAUNCHXL-F280025C to J3-11 of BOOSTXL-DRV8323RH/RS to enable POT
Disconnect J3-29, J3-30 to BOOSTX-DRV8323RS, BOOSTXL-DRV8323RH, or DRV8353RS-EVM
Disconnect J5-42 to DAC128S085EVM
J12 connects to encoder (J4)
J13 connects to hall sensors (J10)

Figure 3-11. LAUNCHXL-F280025C Board Modifications For Connecting BOOSTXL-DRV Board

3.2.10 BOOSTXL-DRV8323RS Setup

BOOSTXL-DRV8323RS is a 15A, three-phase brushless DC drive stage based on the DRV8323RS gate driver and CSD88599Q5DC NexFETTM power blocks. The module has an individual DC bus and three-phase voltage sensing as well as three low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad development kits suitable for use with the sensorless InstaSPIN-FOC algorithm. The drive stage has IDRIVE configuration, along with a fault pin and protection for short circuit, thermal, shoot-through, and under voltage conditions through configurable SPI by C2000 MCUs.

- The BOOSTXL-DRV8323RS Hardware Files are on the BOOSTXL-DRV8323RS page within ti.com.
- For more details about the BOOSTXL-DRV8323RS, see the User's Guide for the EVM.
- Make sure that the following items are completed as described below, and then connect the BOOSTXL-DRV8323RS to J1/J3 and J4/J2 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as shown in Figure 3-12.
  - Populate C9, C10, and C11 with a 47nF capacitor.
  - Bend J3-29 and J3-30 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees so that they are not connected to the BOOSTXL-DRV8323RS as shown in Figure 3-11.

  - If using the DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables, it's optional and not nessary for a motor control.
  - Bend J2-12 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees so that they are not connected to the BOOSTXL-DRV8323RS. For LAUNCHXL-F280025C and LAUNCHXL-F280039C use a jumper wire to connect J4-18(nSCS/GAIN) and J4-4 of BOOSTXL-DRV8323RS to use SPIA-STE of the C2000 device directly as shown in Figure 3-11. For LAUNCHXL-F2800137 use a jumper wire to connect J4-18(nSCS/GAIN) and J4-16 of BOOSTXL-DRV8323RS.
  - Use a jumper wire to connect J3-29 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 to J3-11 of the BOOSTXL-DRV8323RS if the potentiometer functionality is desired to be used on the BOOSTXL-DRV8323RS for setting the reference speed.
- Connect the motor, encoder, and Hall sensors to the BOOSTXL-DRV8323RS and LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 boards as described in Table 3-2 and shown in Figure 3-12.
  - Connect a supply voltage ranging from 6 to 40 V from a battery or a DC voltage source to the voltage supply pins as shown in Figure 3-12. The ABS max supply voltage rating for the DRV8323RS is 65V, so care must be taken to ensure that the voltage does not exceed this value during operation. This is particularly important while slowing down or braking the motor, which can cause the supply voltage to rise substantially. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

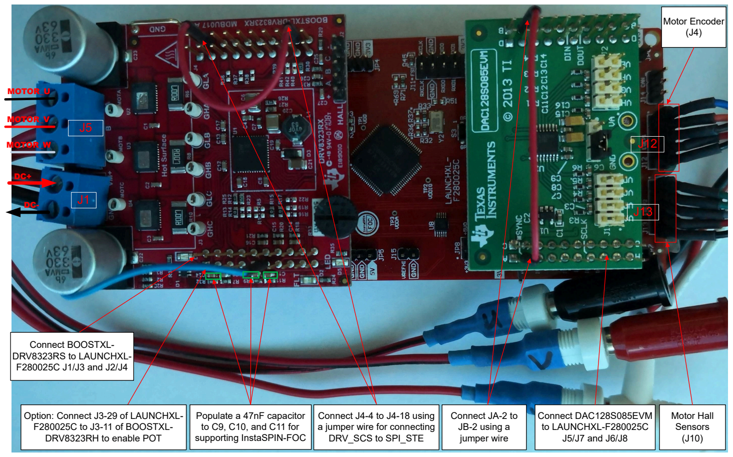

Motor Encoder (J4)
MOTOR_U
MOTOR_V
MOTOR_W
J5
DC+
DC-
J1
J12
J13
Connect BOOSTXL-DRV8323RS to LAUNCHXL-F280025C J1/J3 and J2/J4
Option: Connect J3-29 of LAUNCHXL-F280025C to J3-11 of BOOSTXL-DRV8323RH to enable POT
Populate a 47nF capacitor to C9, C10, and C11 for supporting InstaSPIN-FOC
Connect J4-4 to J4-18 using a jumper wire for connecting DRV_SCS to SPI_STE
Connect JA-2 to JB-2 using a jumper wire
Connect DAC128S085EVM to LAUNCHXL-F280025C J5/J7 and J6/J8
Motor Hall Sensors (J10)

Figure 3-12. LAUNCHXL-F280025C/F280039C/F2800137 Connected to the BOOSTXL-DRV8323RS and DAC128S085EVM

3.2.11 DRV8353RS-EVM Setup

DRV8353RS-EVM is a 15A, three-phase brushless DC drive stage based on the DRV8353RS gate driver and CSD19532Q5B NexFETTM power blocks. The module has an individual DC bus and three-phase voltage sensing as well as three low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad development kits suitable for use with the sensorless InstaSPIN-FOC algorithm. The drive stage is fully protected with short circuit, thermal, shoot-through, and under voltage protection and easily configurable via the device's SPI registers interfacing with C2000 MCUs.

- The DRV8353Rx-EVM Design Files are on the DRV8353RS-EVM page within ti.com.
- For more details about the DRV8353RS-EVM, see the User's Guide for the EVM.
- Make sure that the following items are completed as described below, and then connect the DRV8353RS-EVM to site 1 (J1/J3 and J4/J2) of the LAUNCHXL-F280025C as shown in Figure 3-13.
  - The DRV8353RS-EVM connects on the bottom of the launchpad, so it is necessary to use male to male adapters to connect the headers on the DRV8353RS-EVM to the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 headers. The male adapter pins corresponding to pins J1-17 and J1-19 of the DRV8353RS-EVM need to be disconnected from the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137. This can be accomplished by bending these pins 90 degrees.

  - Do not connect J3-29 and J3-30 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 to the DRV8353RS-EVM as shown in Figure 3-13.
  - Use a jumper wire to connect J2-19 to J2-12 of LAUNCHXL-F280025C andLAUNCHXL-F280039C or J2-13 to J2-12 of LAUNCHXL-F2800137 to make proper connection of the C2000's SPI-STE to the DRVx device as shown in Figure 3-13.
  - If using the DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 with 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables.
- Connect the motor, encoder, and Hall sensors to the DRV8353RS-EVM and the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as described in Table 3-2 and shown in Figure 3-13.
- Connect a supply voltage ranging from 9 to 54 V from a battery or a DC voltage source to the voltage supply pins. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

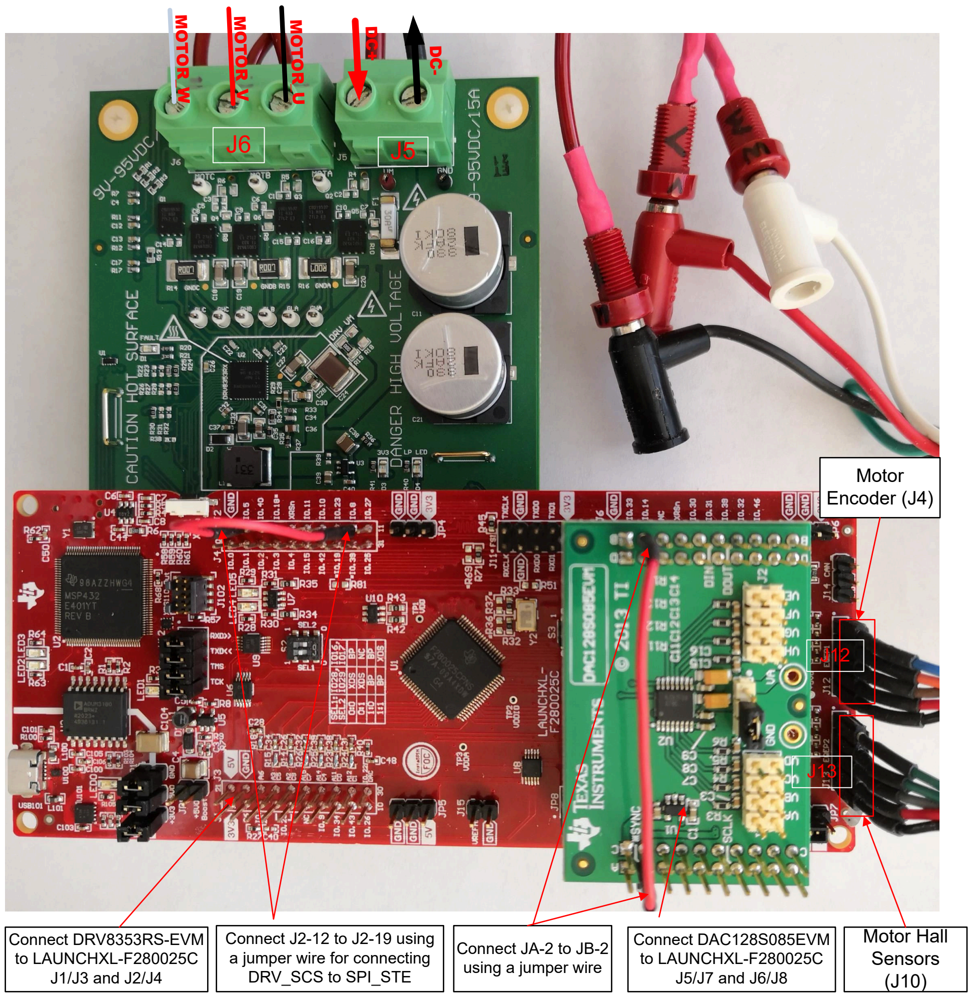

MOTOR_W
MOTOR_V
MOTOR_U
DC+
DC-
J6
J5
Motor Encoder (J4)
J12
J13
Connect DRV8353RS-EVM to LAUNCHXL-F280025C J1/J3 and J2/J4
Connect J2-12 to J2-19 using a jumper wire for connecting DRV_SCS to SPI_STE
Connect JA-2 to JB-2 using a jumper wire
Connect DAC128S085EVM to LAUNCHXL-F280025C J5/J7 and J6/J8
Motor Hall Sensors (J10)

Figure 3-13. LAUNCHXL-F280025C/F280039C/F2800137 Connected to the DRV8353RS-EVM and DAC128S085EVM

3.2.12 BOOSTXL-3PHGANINV Setup

The BOOSTXL-3PHGANINV evaluation module features a 48-V/10-A three-phase GaN inverter with precision in-line shunt-based phase current sensing for accurate control of precision drives such as servo drives. The module also has an individual DC bus and three-phase voltage sensing, making this board for BLDC/ PMSM control with C2000 LaunchPad™ development kits suitable for use with the sensorless InstaSPIN-FOC algorithm.

- The hardware files and more details are available on the BOOSTXL-3PHGANINV page within ti.com.
- For more details about the BOOSTXL-3PHGANINV, see the corresponding *DRV8353Rx-EVM User's Guide*.
- Make sure that the following items are completed as described below, and then connect the BOOSTXL-3PHGANINV to J1/J3 and J4/J2 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as shown in Figure 3-14.
  - If using the DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables.
- Connect the motor, encoder, and Hall sensors to the BOOSTXL-3PHGANINV and the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as described in Table 3-2 and shown in Figure 3-14.
- Connect a supply voltage ranging from 12 to 54 V from a battery or a DC voltage source to the voltage supply pins. Follow the operation instructions in Section 3.5 to turn on the power source.

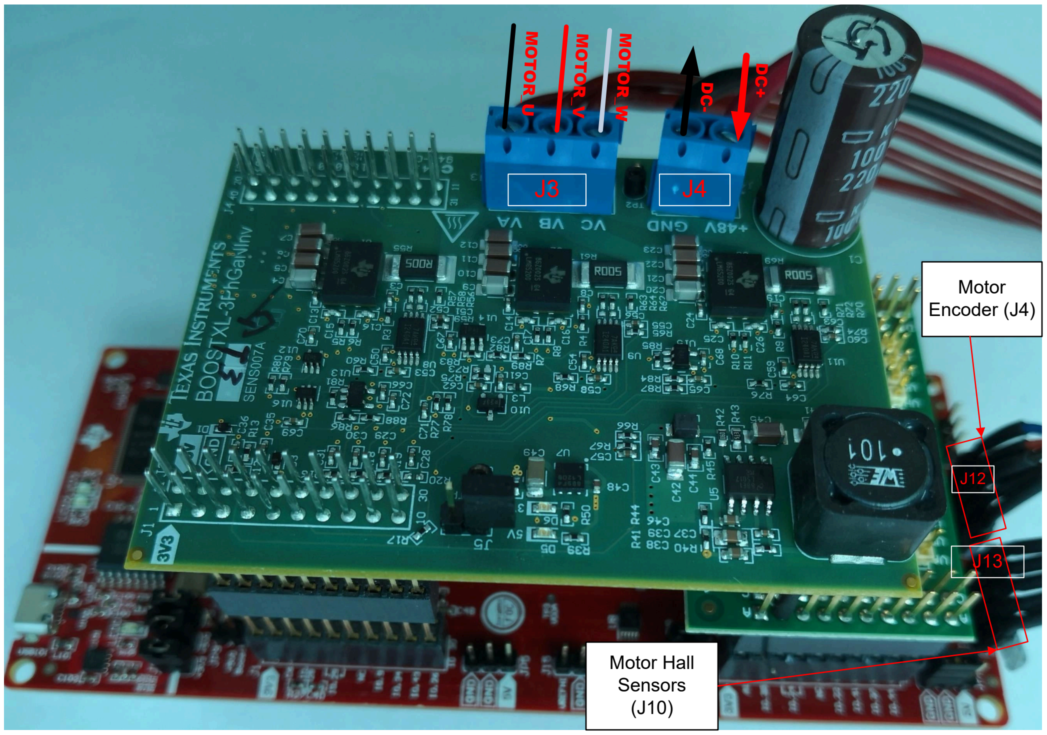


Figure 3-14. LAUNCHXL-F280025C/F280039C/F2800137 Connected to the BOOSTXL-3PHGANINV and DAC128S085EVM

3.2.13 DRV8316REVM Setup

The DRV8316REVM provides three half-H-bridge integrated MOSFET drivers for driving a three-phase brushless DC (BLDC) motor with 8-A Peak current drive. The DRV8316 device integrates current-sense amplifiers (CSA) for three-phase low-side current measurement and individual DC bus and three-phase voltage sensing on the module that makes this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with the sensorless InstaSPIN-FOC algorithm

- The *DRV8316REVM Hardware Design Files* are on the DRV8316REVM page within ti.com.
- For more details about the DRV8316REVM, see the *DRV8316REVM Evaluation Module*.
- Make sure that the following items are completed as described below, and then connect the DRV8316REVM to J1/J3 and J4/J2 site of LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as shown in Figure 3-15.
  - If using DAC128S085EVM board, bend J5-42 of the LAUNCHXL-F280025C 90 degrees to avoid connecting that pin to the DAC128S085EVM board. This EVM board is only used for debugging purposes to monitor various software variables.
- Connect the motor, encoder, and Hall sensors to the DRV8316REVM and LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 as described in Table 3-2 and shown in Figure 3-15.
- Connect a supply voltage ranging from 4.5 to 24V from a battery or a DC voltage source to the voltage supply pins as shown in Figure 3-15. The ABS max supply voltage rating for the DRV8316REVM is 40V, so care must be taken to ensure that the voltage does not exceed this value during operation. This is particularly important while slowing down or braking the motor, which can cause the supply voltage to rise substantially. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

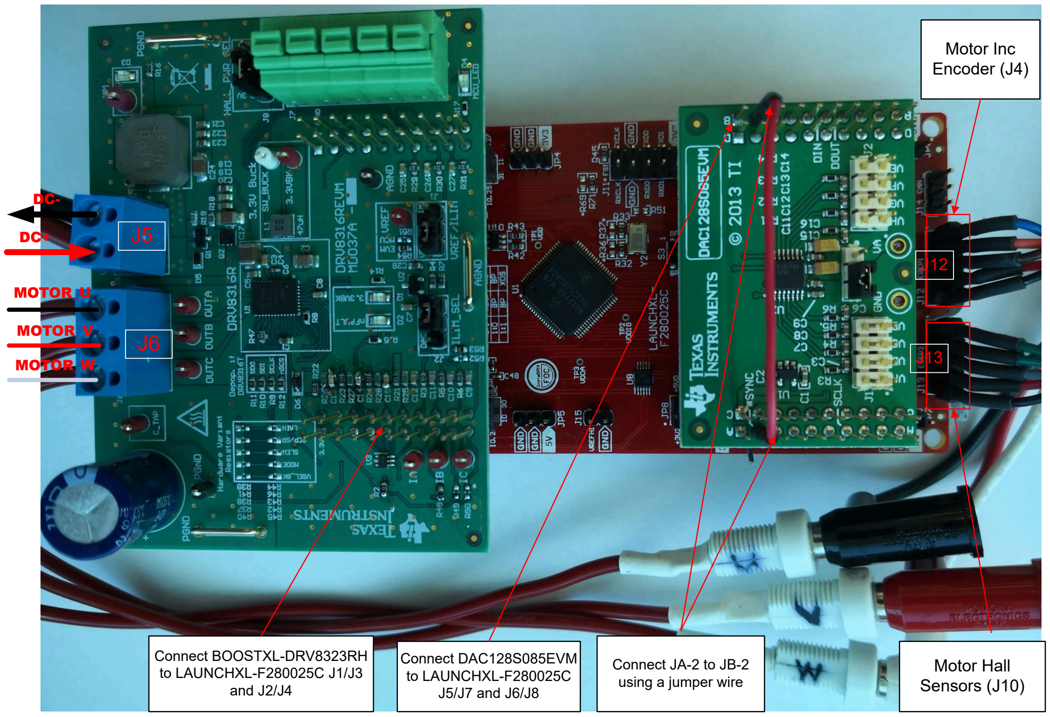

Motor Inc Encoder (J4)
DC-
DC+
J5
MOTOR U
MOTOR V
MOTOR W
J6
J12
J13
Connect BOOSTXL-DRV8323RH to LAUNCHXL-F280025C J1/J3 and J2/J4
Connect DAC128S085EVM to LAUNCHXL-F280025C J5/J7 and J6/J8
Connect JA-2 to JB-2 using a jumper wire
Motor Hall Sensors (J10)

Figure 3-15. LAUNCHXL-F280025C Connected to the DRV8316 REVM and DAC128S085EVM

3.2.14 TMDSHVMTRINSPIN Setup

WARNING

- This EVM is meant to be operated in a lab environment only and is not considered by TI to be a finished end-product fit for general consumer use.
- This EVM must be used only by qualified engineers and technicians familiar with risks associated with handling high voltage electrical and mechanical components, systems and subsystems.
- This EVM operates at voltages and currents that can result in electrical shock, fire hazard and/or personal injury if not properly handled. Equipment must be used with necessary caution and appropriate safeguards must be employed to avoid personal injury or property damage.
- Always use caution when using the EVM electronics due to presence of high voltages! DC bus Capacitors will remain charged for a long time after the mains supply is disconnected.
- The EVM can accept power from the AC Mains/wall power supply, only uses the live and neutral line from the wall supply, the protective earth is unconnected (floating). The power ground is floating from the protective earth ground, all of the ground planes are the same. Hence appropriate caution must be taken and proper isolation requirements must be met before connecting scopes and other test equipment to the board. Isolation transformers must be used when connecting grounded equipment to the EVM.
- The power stages on the board are individually rated. It is the user's responsibility to make sure that these ratings (i.e. voltage, current and power levels) are well understood and complied with, prior to connecting these power blocks together and energizing the board. When energized, the EVM or components connected to the EVM should not be touched.

TMDSHVMTRINSPIN is a DIMM100 controlCARD based motherboard evaluation module showcasing control of the most common types of high voltage, three-phase motors including AC induction (ACI), brushless DC (BLDC), and permanent magnet synchronous motors (PMSM). The High Voltage Motor Control Kit has individual DC bus and three-phase voltage sensing making this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with the sensorless InstaSPIN-FOC algorithm.

- The hardware Files are in the `<install_location>\solutions\tmdshvmtrinspin\hardware` folder of C2000WARE-MOTORCONTROL-SDK.

This section explains the steps needed to run the TMDSHVMTRINSPIN with the software supplied through MotorControl SDK. The kit ships with the jumper and switch settings correctly positioned for connecting with the controlCARD. Make sure that these settings are valid on the board as described below, and then insert the controlCARD with the TMDSADAP180TO100 adapter into the TMDSHVMTRINSPIN board as shown in Figure 3-16.

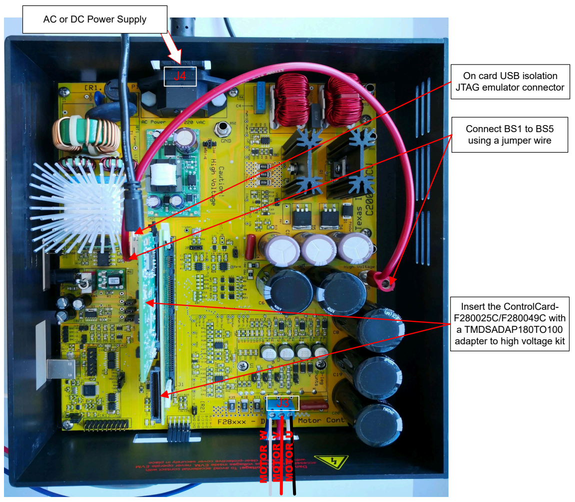

AC or DC Power Supply
On card USB isolation JTAG emulator connector
Connect BS1 to BS5 using a jumper wire
Insert the ControlCard-F280025C/F280049C with a TMDSADAP180TO100 adapter to high voltage kit

Figure 3-16. TMDSHVMTRINSPIN Connected to the TMDSCNCD280025C, TMDSCNCD280039C, or TMDSCNCD2800137 with TMDSADAP180TO100

CAUTION

Do not apply AC power to the board before you have verified these settings!

- Make sure nothing is connected to the board, and no power is being supplied to the board.
- Insert the Control card with TMDSADAP180TO100 adapter into the [Main]-J1 controlCARD connector if not already populated.
- Make sure the following jumpers & connector settings are correctly implemented as shown in Figure 3-17.
  - [Main]-J3, J4, J5 and J8 are populated.
  - [Main]-J9 and [M3]-J5 are not populated for using a controlCARD with its own onboard emulation to disable the XDS100 on HVKIT.
  - [Main]-J7 is populated between pins 2-3 (pins furthest from the DIMM 100 socket).
  - Banana cable between [Main]-BS1 and [Main]-BS5 is installed to bypass the PFC.
  - Make sure that the DC Fan shipped with the kit is connected to the DC Fan Jumper [Main]-J17 when operating the motor under load > 150W.
- Two options to get DC Bus power are as below, recommend using the external 15V DC power supply.
  - [Main]-J2 is not populated if using the +15V from an external 15V DC power supply. Ensure that [M6]-SW1 is in the “Off” position, connect 15V DC power supply to [M6]-JP1.
  - [Main]-J2 is populated with a jumper between bridge and the middle pin if using the +15V power supply from aux power supply module.
- Turn on [M6]-SW1. Now [M6]-LD1 should turn on. Notice the control card LED would light up as well indicating the control card is receiving power from the board.
- Connect [Main]-BS1 and BS5 to each other using banana plug cord, and then connect one end of the AC mains power or DC power to [Main]-P1.
- Connect the motor, encoder, and Hall sensors to the kits as described in Table 3-2 and shown in Figure 3-17.
- Connect a supply voltage from an AC or a DC voltage source to the voltage supply pins. Power should only be applied when instructed to do so in Section 3.5, keep disconnected otherwise.

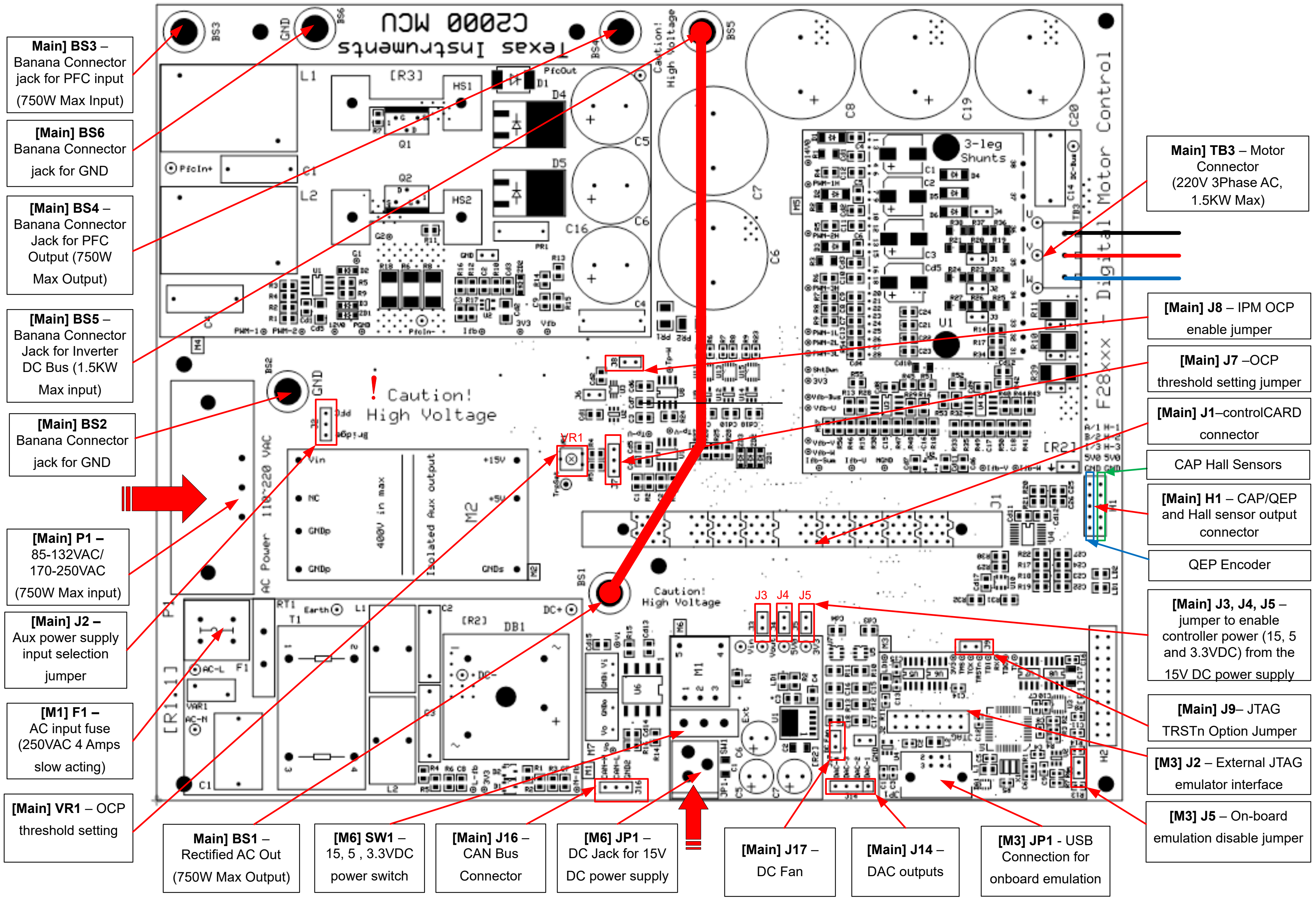

Main] BS3 – Banana Connector jack for PFC input (750W Max Input)
[Main] BS6 Banana Connector jack for GND
[Main] BS4 – Banana Connector Jack for PFC Output (750W Max Output)
[Main] BS5 – Banana Connector Jack for Inverter DC Bus (1.5KW Max input)
[Main] BS2 Banana Connector jack for GND
[Main] P1 – 85-132VAC/ 170-250VAC (750W Max input)
[Main] J2 – Aux power supply input selection jumper
[M1] F1 – AC input fuse (250VAC 4 Amps slow acting)
[Main] VR1 – OCP threshold setting
Main] BS1 – Rectified AC Out (750W Max Output)
[M6] SW1 – 15, 5 , 3.3VDC power switch
[Main] J16 – CAN Bus Connector
[M6] JP1 – DC Jack for 15V DC power supply
[Main] J17 – DC Fan
[Main] J14 – DAC outputs
[M3] JP1 - USB Connection for onboard emulation
[M3] J5 – On-board emulation disable jumper
[M3] J2 – External JTAG emulator interface
[Main] J9– JTAG TRSTn Option Jumper
[Main] J3, J4, J5 – jumper to enable controller power (15, 5 and 3.3VDC) from the 15V DC power supply
QEP Encoder
[Main] H1 – CAP/QEP and Hall sensor output connector
CAP Hall Sensors
[Main] J1–controlCARD connector
[Main] J7 –OCP threshold setting jumper
[Main] J8 – IPM OCP enable jumper
Main] TB3 – Motor Connector (220V 3Phase AC, 1.5KW Max)
C2000 MCU
Texas Instruments
Caution! High Voltage
Digital Motor Control – F28xxx
VR1
J3
J4
J5

Figure 3-17. TMDSHVMTRINSPIN Kit Jumpers and Connectors Diagram

Table 3-3 shows the various connections available on the board. The location of these connections on the board are shown in Figure 3-17.

Table 3-3. Key Jumpers, Connectors Explanation

[Main]-P1	AC input connector (110V – 220V AC)
[Main]-TB3	Terminal Block to connect motor
[Main]-BS1	Banana Jack for Output from AC Rectifier
[Main]-BS2, BS6	Banana Jack for GND Connection
[Main]-BS3	Banana Jack for connecting an input voltage for the PFC stage, this would typically be rectified AC voltage from the [Main]-BS1 connector.
[Main]-BS4	Banana Jack for connecting a load to the output from the PFC stage, When using PFC+Motor project the output of the PFC stage would connect to the input for the inverter bus i.e. [Main]-BS5
[Main]-BS5	Banana Jack for input of DC bus voltage for the inverter
[Main]-J2	Aux power supply module input voltage selection jumper, • When jumper connected to Bridge position the aux power supply module sources power from the AC rectifier bridge output. • When Jumper connected to PFC position the aux power supply module sources power from the output of the PFC stage.
[Main]-J3, J4, J5	Jumpers J3,J4 and J5 are used for sourcing 15V, 5V and 3.3V power respectively for the board from the 15V DC Power supply.
[Main]-J7	J7 is used to select the over current protection threshold source
[Main]-J8	J8 is used to enable/disable the IPM over current protection
[Main]-J9	JTAG TRSTn disconnect jumper, populating the jumper enables JTAG connection to the Microcontroller. The jumpers need to be unpopulated when no JTAG connection is required such as when booting from FLASH.
[Main]-J14	PWMDAC outputs: Gives voltage outputs that result from a PWM being attached to a first-order low-pass filter. Pins 1,2,3 and 4 are attached to low pass filtered PWM output pins respectively to observe system variables on an oscilloscope.
[Main]-J16	Isolated CAN bus connector
[Main]-J17	Connector to supply power to the DC fan (shipped with the board) that is attached to the IPM heat sink.
[Main]-H1	QEP connector: connects with a 0-5V QEP sensor to gather information on a motor's speed and position. CAP/Hall effect sensor connector: connects with a 0-5V sensor to gather information on a motor's speed and position.
[M1]-F1	Fuse for the AC input
[M3]-JP1	USB connection for on-board emulation
[M3]-J2	External JTAG interface: this connector gives access to the JTAG emulation pins. If external emulation is desired, place a jumper across [M3]-J5 and connect the emulator to the board. To power the emulation logic a USB connector will still need to be connected to [M3]-JP1.
[M3]-J5	On-board emulation disable jumper: Place a jumper here to disable the on-board emulator and give access to the external interface.

3.3 Lab Software Implementation

1. Download and install Code Composer Studio from the Code Composer Studio (CCS) Integrated Development Environment (IDE) tools folder. Version 10.4 or above is recommended. More details about CCS installation and implementation in CCS User's Guide..
2. Download and install C2000WARE-MOTORCONTROL-SDK software package from the link provided by TI, install this Motor Control SDK software in its default folder. C2000WARE-MOTORCONTROL-SDK can be installed in one of two ways:
   a. Download the software through the C2000Ware MotorControl SDK tools folder.
   b. Go to CCS and under View → Resource Explorer. Under the TI Resource Explorer, go to Software→C2000Ware_MotorControl_SDK, and click on the install button.
3. Once the installation is complete, close CCS, and create a new workspace for importing the project. The software project of this example lab is inside the C2000Ware Motor Control SDK folder at `<install_location>\C2000Ware_MotorControl_SDK_<version>\solutions\universal_motorcontrol_lab`. Follow these steps to build and run this code with different incremental builds as described in the following sections.

3.3.1 Importing and Configuring Project

The example lab is a universal project that has support for various TI EVM motor driver kits that can be used in conjunction with the F280025C, F280039C, or F2800137 C2000 MCU devices. The user can run different TI EVM kits by setting the build configurations and properties of the lab project. In the following sections, the LAUNCHXL-F280025C, LAUNCHXL-F280039C, or LAUNCHXL-F2800137 is used in combination with the BOOSTXL-DRV8323RS lab to show how to import and run the example lab on this kit.

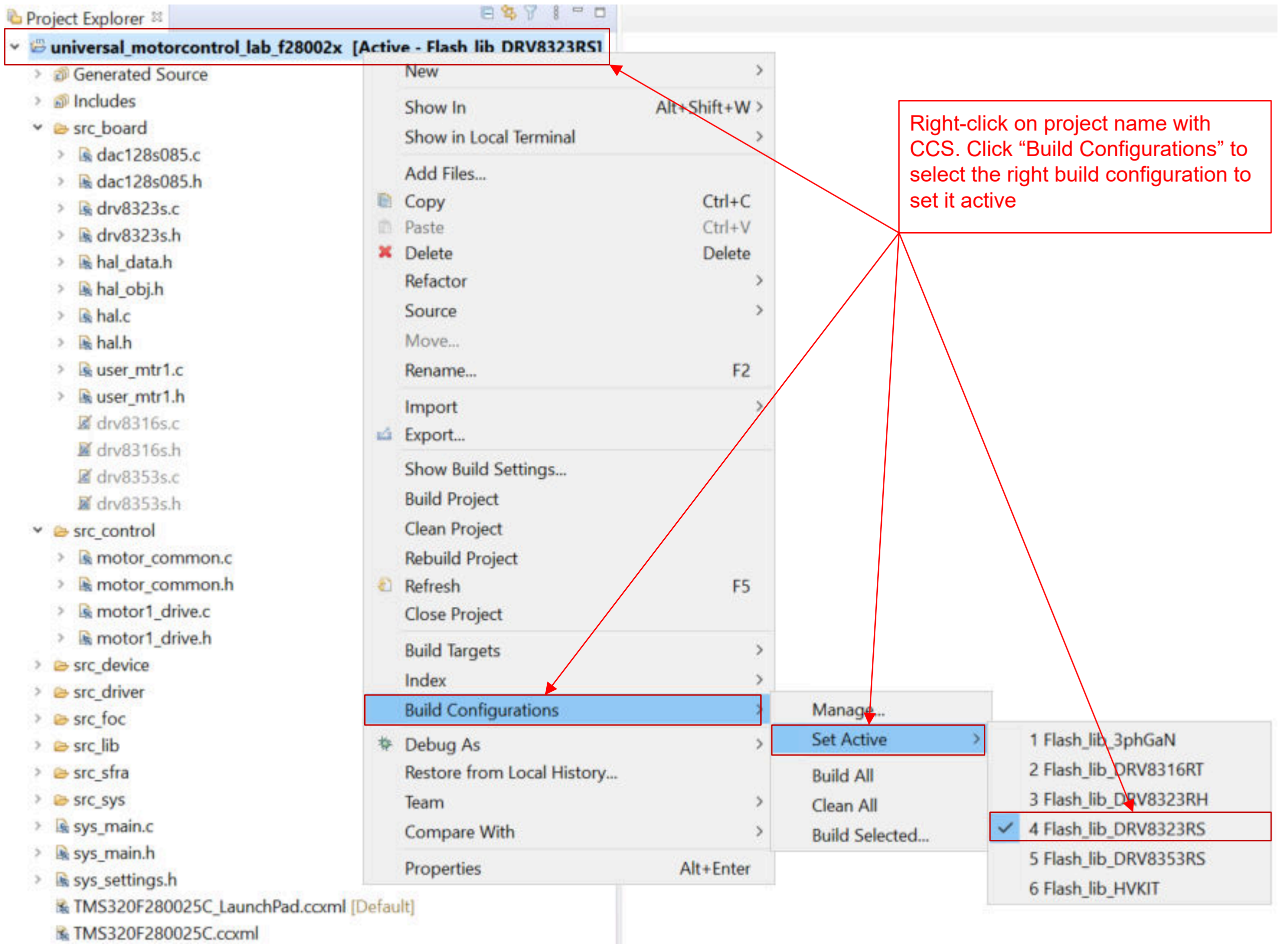
1. Import the project within CCS by clicking "Project" ➔"Import CCS Projects...", and then click "Browse..." button to select search directory at:
   a. F28002x based lab:`<install_location>\solutions\universal_motorcontrol_lab\f28002x\ccs\motor_control\` to select the "universal_motorcontrol_lab_f28002x" project.
   b. F28003x based lab:`<install_location>\solutions\universal_motorcontrol_lab\f28003x\ccs\motor_control\` to select the "universal_motorcontrol_lab_f28003x" project.
   c. F280013x based lab:`<install_location>\solutions\universal_motorcontrol_lab\f280013x\ccs\motor_control\` to select the "universal_motorcontrol_lab_f280013x" project.
2. The lab project can be configured to run on a variety of motor driver kits. You can select one of these kits by right-clicking on the imported project name and selecting the right build configuration (such as **Flash_lib_DRV8323RS_3SC**) as shown in Figure 3-18.
3. Configure the project to select the supporting functions in the project by right-clicking on the imported project name, and then click the "Properties" command to set the pre-define symbols for the project as shown in Figure 3-19.
   a. A pre-define symbol is active or disabled by removing or adding the "_N" in the name. For example, field weakening control is enabled by removing the "_N" in "MOTOR1_FWC_N" to change it to "MOTOR1_FWC", and field weakening control functions are disabled for motor 1 (Compressor) by changing the "MOTOR1_FWC" symbol name to "MOTOR1_FWC_N".
   b. Select the right supporting motor control algorithm based on the motor and hardware board by enabling the related pre-define symbol as described above. The supporting algorithms and related motors matrix are shown in Table 3-4.
   c. Select the right supporting functions by enabling the pre-define symbol/s as shown in Figure 3-19.
4. Select the right target configuration file (.ccxml) as shown in Figure 3-21 by right clicking on the file name to select "Set as Active Target Configuration" and "Set as Default Target Configuration" on the pop-up menu.
   a. TMS320F280025C_LaunchPad.ccxml is for the LAUNCHXL-F280025C based hardware kits.
   b. TMS320F280025C.ccxml is for the TMDSCNCD280025C based hardware kits.

5. Select or define the right motor model in the *user_mtr1.h* and *user_common.h* files. These files are located under the src_board folder located in the project explorer window. The motor defines section in the *user_mtr1.h* file begins on line 921. Uncomment the #define that corresponds with the motor being tested, and ensure that the rest of the #define motors remain commented out. Make sure that the motor parameters in the code match with the specifications of the connecting motor.
6. Set up the hardware kit, connect the motor, encoder, and/or hall sensor to the kit as described in Section 3.2.

Table 3-4. The Supporting Algorithms, Functions and Motors Matrix in Example Lab

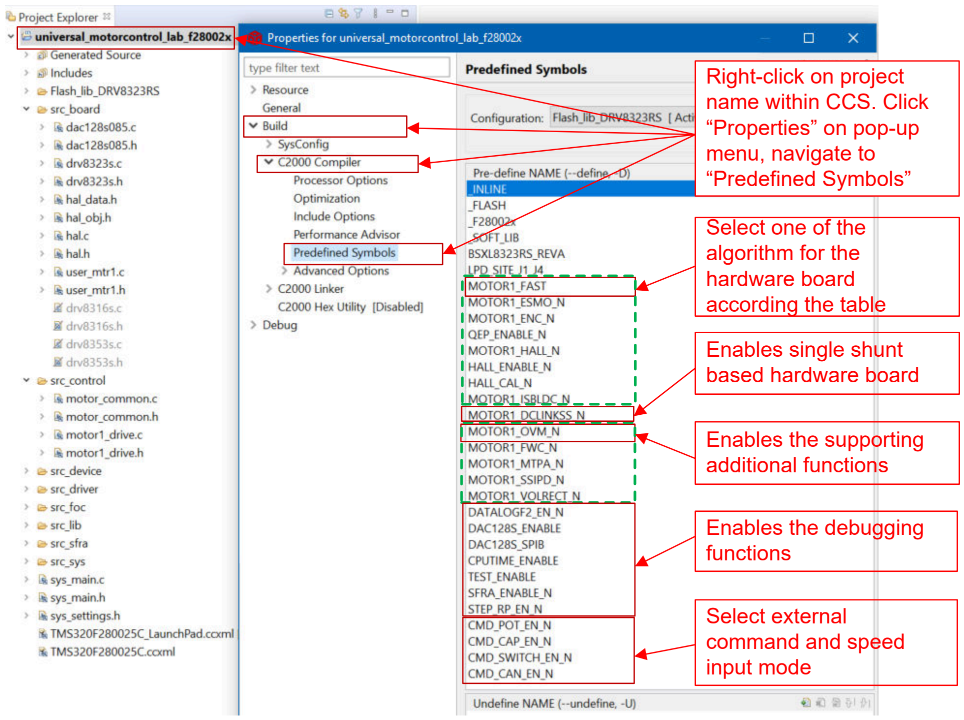
Algorithms or Functions	Pre-Define Symbols	LaunchPad						controlCARD
		DRV8329AEVM	BOOSTXL-DRV8323RH	BOOSTXL-DRV8323RS	DRV8353RS-EVM	BOOSTXL-3PHGANINV	DRV8316REVM	TMDSHVMTRINSPIN
FAST based Sensorless FOC	MOTOR1_FAST	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, HVPMSMMTR
eSMO based Sensorless FOC	MOTOR1_ESMO	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, HVPMSMMTR
QEP Encoder based Sensored FOC	MOTOR1_ENC QEP_ENABLE	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, HVPMSMMTR
Hall Sensors based Sensored FOC	MOTOR1_HALL HALL_ENABLE HALL_CAL	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR, LVBLDCMTR	✓, LVSERVOMTR, LVBLDCMTR	✓, HVBLDCMTR#
Trapezoidal InstaSPIN-BLDC	MOTOR1_ISBLDC	✓, LVSERVOMTR, LVBLDCMTR	✖	✖	✖	✖	✖	✖
Single-Shunt Current Sense	MOTOR1_DCLINKSS	✓, LVSERVOMTR, LVBLDCMTR	✖	✖	✖	✖	✖	✖
Datalog with Graph Tool[1]	DATALOGF2_EN	✓	✓	✓	✓	✓	✓	✓
PWMDAC	EPWMDAC_MODE	✖	✖	✖	✖	✖	✖	✓
External DAC	DAC128S_ENABLE	✓	✓	✓	✓	✓	✓	✖
SFRA Tool	SFRA_ENABLE	✓	✓	✓	✓	✓	✓	✓
Step Response with Graph Tool	STEP_RP_EN	✓	✓	✓	✓	✓	✓	✓

[1] The Datalog implementation in the Universal Motor Control Lab requires DMA. Not all C2000 devices have DMA. Refer to specific device datasheet to determine DMA support.



Right-click on project name with CCS. Click “Build Configurations” to select the right build configuration to set it active

Figure 3-18. Select the Right Build Configurations within CCS


Properties for universal_motorcontrol_lab_f28002x
Build
C2000 Compiler
Predefined Symbols
Configuration: Flash_lib_DRV8323RS [Active]
Pre-define NAME (--define, -D)
Undefine NAME (--undefine, -U)

```
_INLINE
_FLASH
_F28002x
_SOFT_LIB
BSXL8323RS_REVA
LPD_SITE_J1_J4
MOTOR1_FAST
MOTOR1_ESMO_N
MOTOR1_ENC_N
QEP_ENABLE_N
MOTOR1_HALL_N
HALL_ENABLE_N
HALL_CAL_N
MOTOR1_ISBLDC_N
MOTOR1_DCLINKSS_N
MOTOR1_OVM_N
MOTOR1_FWC_N
MOTOR1_MTPA_N
MOTOR1_SSIPD_N
MOTOR1_VOLRECT_N
DATALOGF2_EN_N
DAC128S_ENABLE
DAC128S_SPIB
CPUTIME_ENABLE
TEST_ENABLE
SFRA_ENABLE_N
STEP_RP_EN_N
CMD_POT_EN_N
CMD_CAP_EN_N
CMD_SWITCH_EN_N
CMD_CAN_EN_N
```

Right-click on project name within CCS. Click “Properties” on pop-up menu, navigate to “Predefined Symbols”

Select one of the algorithm for the hardware board according the table

Enables single shunt based hardware board

Enables the supporting additional functions

Enables the debugging functions

Select external command and speed input mode

Figure 3-19. Select the Desired Pre-define Symbols in Project Properties

3.3.2 Lab Project Structure

The general structure of the project is shown in Figure 3-20. The device peripherals configuration is based on C2000Ware driverlib. The user only needs to change the code and definitions in the *hal.c* and *hal.h* files, and the parameters in the *user_mtr1.h* file, if they want to migrate the reference design software to their own custom board or to a different C2000 device.

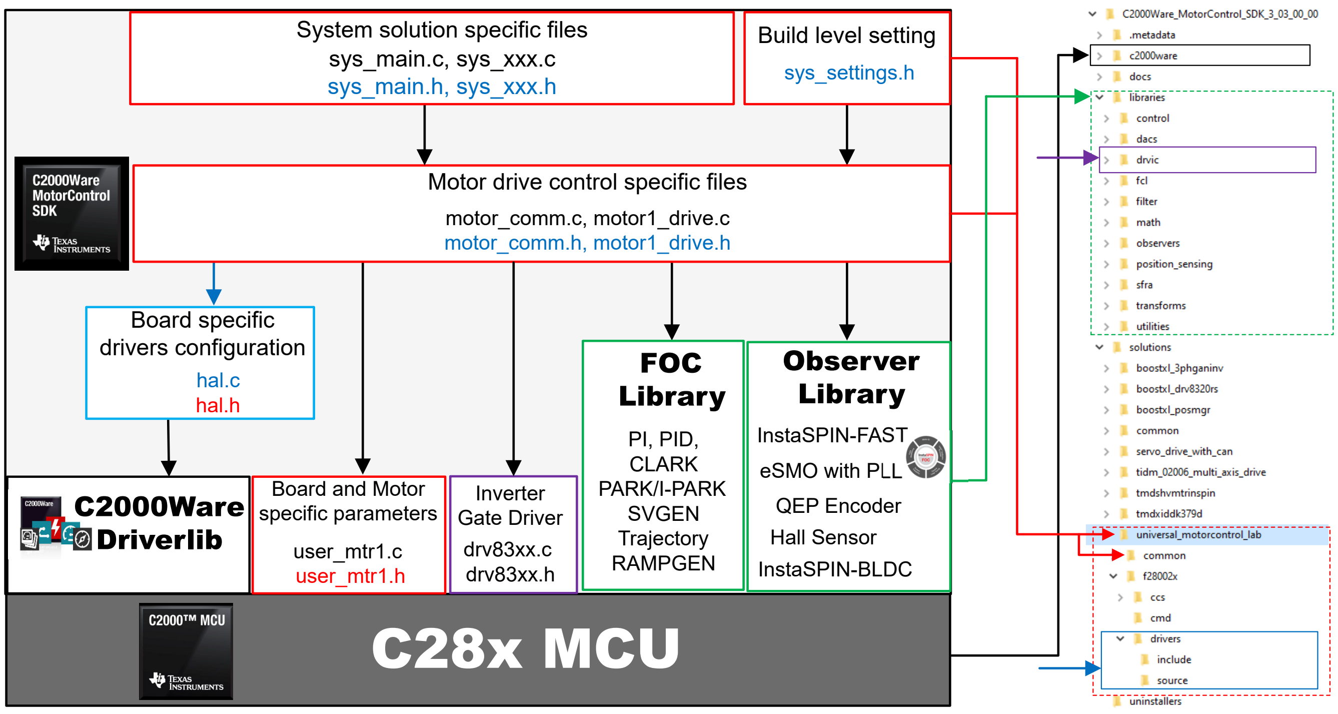


Figure 3-20. Project Structure Overview

Once the project is imported into CCS, the project explorer will appear inside CCS as shown in Figure 3-21.

The *src_foc* folder includes the typical FOC modules including Park, Clark, and inverse Park and Clark transforms, PID, and estimators that consist of the motor drive algorithm and are independent of specific devices or boards.

The *src_lib* folder includes the InstaSPIN-FOC library and math library that is not specific to any particular device or board.

The *src_control* folder includes motor drive control files that call motor control core algorithm functions within the interrupt service routines and background tasks.

The *src_sys* folder includes some files reserved for system control that are independent of specific devices or boards. The user can add their own code for system control, communication, and so forth.

Board-specific, motor-specific and device-specific files are in the *src_board* folder. These files consist of device specific drivers to run the solution. If the user wants to migrate the project for their own board or to other C2000 devices, the user only needs to make changes to the *hal.c, hal.h,* and *user_mtr1.h* files based on the usage of device peripherals for the board.

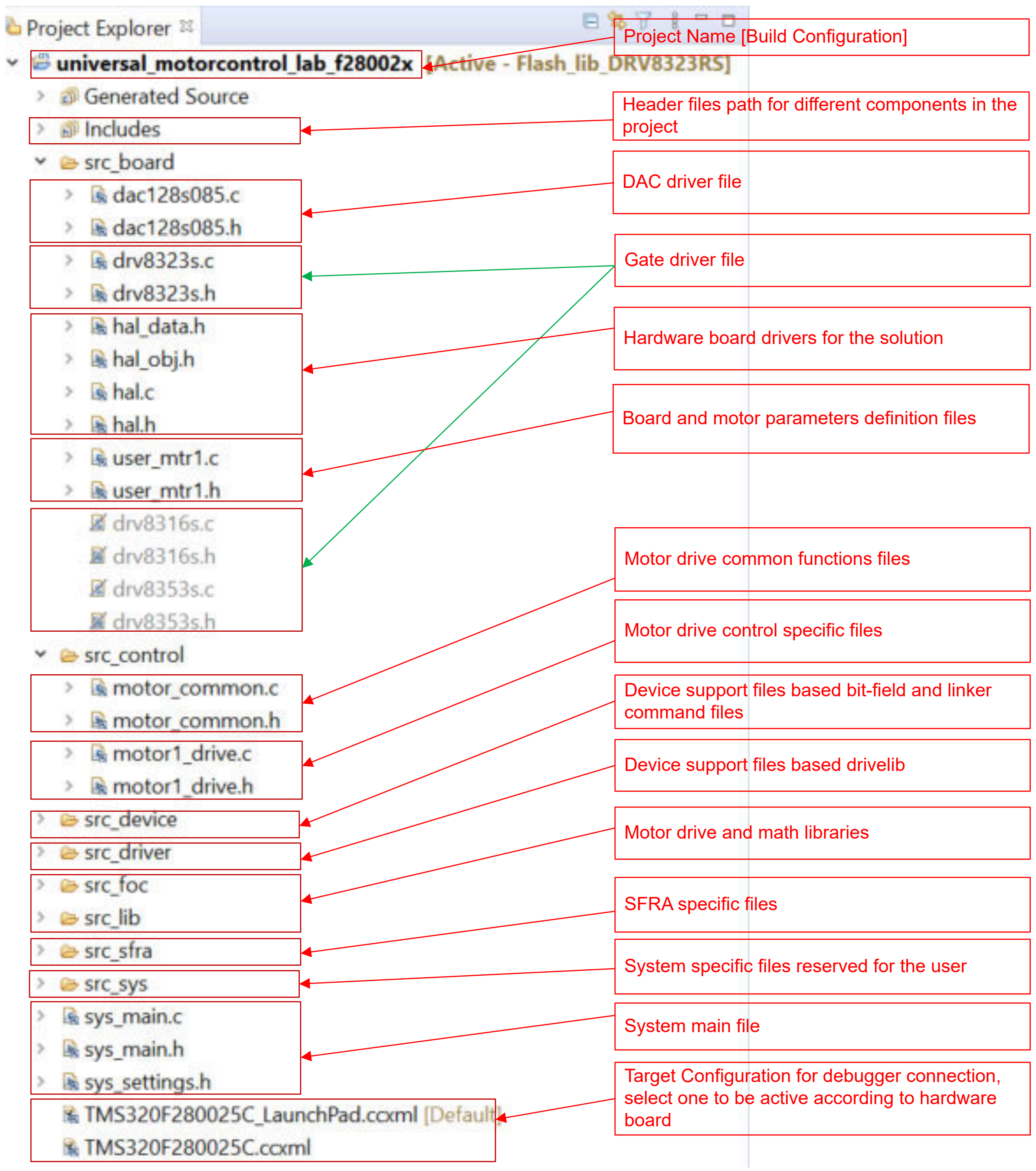

Project Explorer
universal_motorcontrol_lab_f28002x [Active - Flash_lib_DRV8323RS]
Generated Source
Includes
src_board
dac128s085.c
dac128s085.h
drv8323s.c
drv8323s.h
hal_data.h
hal_obj.h
hal.c
hal.h
user_mtr1.c
user_mtr1.h
drv8316s.c
drv8316s.h
drv8353s.c
drv8353s.h
src_control
motor_common.c
motor_common.h
motor1_drive.c
motor1_drive.h
src_device
src_driver
src_foc
src_lib
src_sfra
src_sys
sys_main.c
sys_main.h
sys_settings.h
TMS320F280025C_LaunchPad.ccxml [Default]
TMS320F280025C.ccxml
Project Name [Build Configuration]
Header files path for different components in the project
DAC driver file
Gate driver file
Hardware board drivers for the solution
Board and motor parameters definition files
Motor drive common functions files
Motor drive control specific files
Device support files based bit-field and linker command files
Device support files based drivelib
Motor drive and math libraries
SFRA specific files
System specific files reserved for the user
System main file
Target Configuration for debugger connection, select one to be active according to hardware board

Figure 3-21. Project Explorer View of the Example Lab

3.3.3 Lab Software Overview

Figure 3-22 shows the project software flow diagram of the firmware that includes one ISR for real time motor control and the main loop for motor control parameters that updates in a background loop. The ISR will be triggered by ADC End of Conversion (EOC).

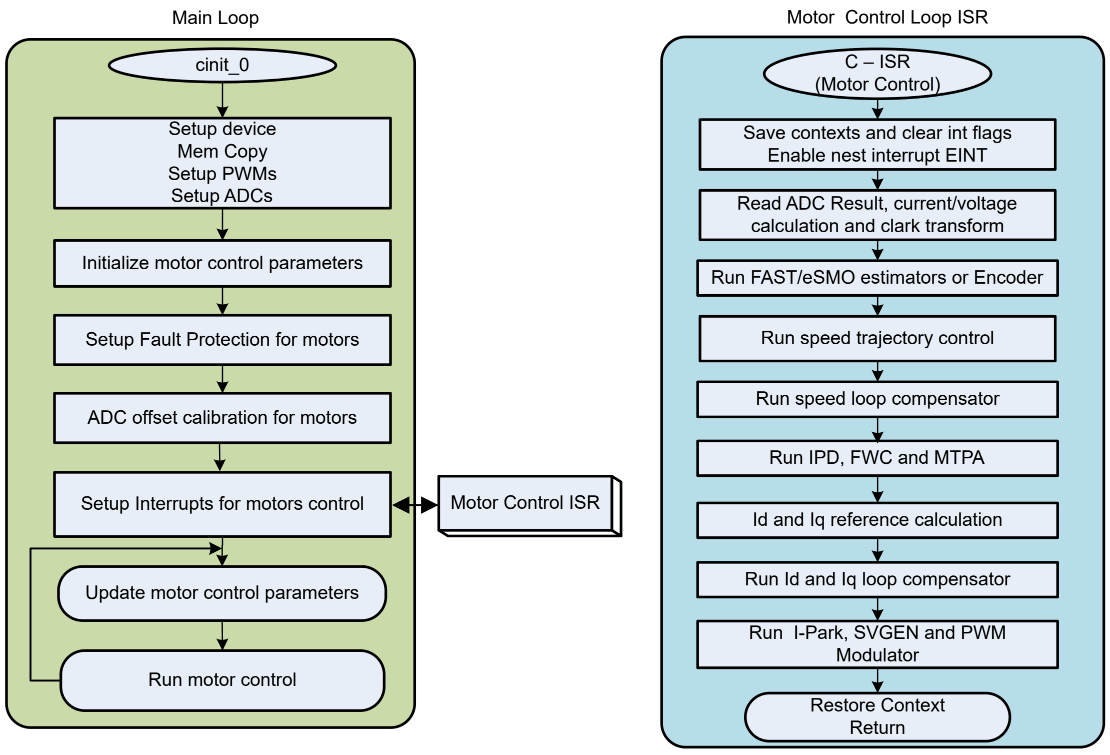

Main Loop
cinit_0
Setup device
Mem Copy
Setup PWMs
Setup ADCs
Initialize motor control parameters
Setup Fault Protection for motors
ADC offset calibration for motors
Setup Interrupts for motors control
Motor Control ISR
Update motor control parameters
Run motor control
Motor Control Loop ISR
C – ISR
(Motor Control)
Save contexts and clear int flags
Enable nest interrupt EINT
Read ADC Result, current/voltage calculation and clark transform
Run FAST/eSMO estimators or Encoder
Run speed trajectory control
Run speed loop compensator
Run IPD, FWC and MTPA
Id and Iq reference calculation
Run Id and Iq loop compensator
Run I-Park, SVGEN and PWM Modulator
Restore Context
Return

Figure 3-22. Project Software Flowchart Diagram

To simplify the system bring up and design, the software is organized in four incremental builds, which makes learning and getting familiar with the board and software easier. This approach is also good for debugging and testing boards.

Table 3-5 lists the framework modules to be used in this lab project.

Table 3-5. Using Motor Control Modules in Lab Project

Module Names	Explanation	Algorithm
ANGLE_GEN_run	Ramp angle generator for open-loop running	eSMO, ENC, HALL
CLARKE_run	Clarke transformation for current or voltage	FAST, eSMO, ENC, HALL
collectRMSData, calculateRMSData	Collect sampling values to calculate the RMS value of phase current and voltage	FAST, eSMO, ENC, HALL
DAC128S_writeData	Converts and send software variables to external DAC with SPI	All Algorithms
DATALOGIF_update	Stores the real-time values into for displaying with graph tool	All Algorithms
ENC_run	Calculate rotor angle based on encoder	ENC
ESMO_run	Enhance Sliding Mode Observer (eSMO) for sensorless-FOC	eSMO
EST_run	FAST estimator for sensorless-FOC	FAST
EST_runTraj	Trajectory generator for current and speed for motor identification	FAST
EST_setupTrajState	Trajectory generator setup for current and speed for motor identification	FAST
HAL_readADCData	Returns ADC conversion values with floating-point format	All Algorithms
HAL_writePWMDACData	Converts software variables into the PWM signals	All Algorithms
HAL_writePWMData	PWM drives for motor	All Algorithms
HALL_run	Calculate rotor angle ans speed based on Hall sensors	HALL
IPARK_run	Inverse Park transformation	FAST, eSMO, ENC, HALL
PARK_run	Park Transformation	FAST, eSMO, ENC, HALL
PI_run	PI Regulators for current and speed	All Algorithms
SPDCALC_run	Speed Measurement based on the angle from encoder signal	ENC
SPDFR_run	Speed measurement based on the angle from observer	eSMO
SVGEN_run	Space Vector PWM with quadrature control	FAST, eSMO, ENC, HALL
TRAJ_run	Trajectory for setting speed reference	All Algorithms
VS_FREQ_run	Generate vector voltage with v/f profile	FAST, eSMO, ENC, HALL

Table 3-6 summarizes the modules tested in each incremental system build.

Table 3-6. Motor Control Modules Used per Incremental Build

Software Module	DMC_LEVEL_1	DMC_LEVEL_2	DMC_LEVEL_3	DMC_LEVEL_4	
	50% PWM duty, verify ADC offset calibration, PWM output, and phase shift	Open loop control to verify the motor current and voltage sensing signals	Closed current loop to validate the current sensing on the board and the current control with the PID	Closed-loop run with estimators/observers	Motor parameters identify with FAST estimators
HAL_readADCData	√√	√ 1	√	√	√
HAL_writePWMData	√√	√	√	√	√
ANGLE_GEN_run		√√	√	√(eSMO, ENC, HALL)*	
VS_FREQ_run		√√			
CLARKE_run (current)		√	√	√	√
CLARKE_run (voltage)		√	√ (FAST)* 2	√ (FAST)*	√ (FAST)*
TRAJ_run		√√	√	√√	
EST_run		√(FAST)*	√ (FAST)*	√√ (FAST)*	√√ (FAST)*
EST_setupTrajState					√√ (FAST)*
EST_runTraj					√√ (FAST)*
ESMO_run		√(eSMO)*	√(eSMO)*	√√ (eSMO)*	
SPDFR_run		√(eSMO)*	√(eSMO)*	√√ (eSMO)*	
ENC_run		√(ENC)*	√(ENC)*	√√(ENC)*	
SPDCALC_run		√(ENC)*	√(ENC)*	√√(ENC)*	
HALL_run		√(HALL)*	√(HALL)*	√√(HALL)*	
PARK_run		√	√	√	√
PI_run (Id)			√√	√	√
PI_run (Iq)			√√	√	√
PI_run (speed)				√√	√
IPARK_run		√√	√	√	√
SVGEN_run		√√	√	√	√
HAL_writePWMDAC Data		√** 3	√**	√**	√**
DATALOGIF_update		√	√	√	√
DAC128S_writeData		√**	√**	√**	√**

1. √ means this module is used. √√ means this module is being tested.
2. √(FAST)* means this module is only used by FAST. √ (eSMO)* means this module is only used by eSMO. √ (ENC)* means this module is only used by ENC. √ (HALL)* means this module is only used by HALL.
3. √** means this module is supported by some hardware kit as shown in Table 3-1.

The universal lab project can use one of the FOC algorithms separately for motor control, or use two of the FOC algorithms simultaneously as shown in Table 3-7. The estimator in use can be switched smoothly on the fly if two algorithms are implemented in the lab project.

Table 3-7. Supporting Estimator Algorithms in Lab Project

	FAST(MOTOR1_FAST)	eSMO (MOTOR1_ESMO)	ENCODER (MOTOR1_ENC)	HALL (MOTOR1_HALL)	ISBLDC (MOTOR1_ISBLDC)
FAST	√ 1	√	√	√	×
eSMO	√	√	√	×	×
ENCODER	√	√	√	×	×
HALL	√	×	×	√	×
ISBLDC	×	×	×	×	√

1. √ means these two algorithms can be used simultaneously in project. × means these two algorithms can not be used simultaneously in project.

3.4 Monitoring Feedback or Control Variables

The continuous feedback or control variables can be monitored by using multiple methods, such as datalog with graph tool, PWMDAC with an oscilloscope, external DAC with an oscilloscope.

3.4.1 Using DATALOG Function

The DATALOG module stores the real-time values of two user selectable software variables in the data RAM provided on the C2000 MCU as shown in Figure 3-23. The two variables are selected by configuring the module inputs, iptr[0] and iptr[1] to the address of the two variables. The starting addresses of the two RAM buffer locations, where the data values are stored, are stored in datalogBuff1[0] and datalogBuff1[1]. These Datalog buffers are large arrays that contain value-triggered data that can then be displayed to a graph. The datalog prescalar is configurable, which will allow the dlog function to only log one out of every prescalar samples. The default prescalar is set to 10, but can be changed by modifying the value of the DATA_LOG_SCALE_FACTOR define in the *datalogIF.h* file. Direct Memory Access (DMA) is used to transfer the values of the selected software variables to the datalog buffer in RAM.

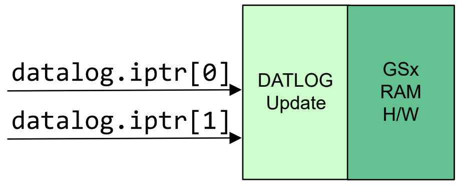


Figure 3-23. DATALOG Module Block Diagram

In order to enable the datalog functionality, the predefine symbol DATALOGF2_EN must be added in the project properties as shown in Figure 3-19.

The code below shows the declaration of one DATALOG object and handle. This code is located in the *datalogIF.c* file.

```
DATALOG_Obj datalog;
DATALOG_Handle datalogHandle;       //!< the handle for the Datalog object
```

The code below shows the initialization and setting up of the datalog object, handle and parameters. This code is located in the *sys_main.c* file.

```
// Initialize Datalog
datalogHandle = DATALOGIF_init(&datalog, sizeof(datalog));
DATALOG_Obj *datalogObj = (DATALOG_Obj *)datalogHandle;

HAL_setupDMAforDLOG(halHandle, 0, &datalogBuff1[0], &datalogBuff1[1]);
HAL_setupDMAforDLOG(halHandle, 1, &datalogBuff2[0], &datalogBuff2[1]);
```

The code below shows the configuration of the two module inputs, iptr[0] and iptr[1], to point to the address of two variables. The datalog module inputs point to different system variables depending on the build level. This code is located in the *sys_main.c* file:

```
datalogObj->iptr[0] = &motorVars_M1.adcData.I_A.value[0];
datalogObj->iptr[1] = &motorVars_M1.adcData.I_A.value[1];
```

The code below shows the periodic updating of the datalog buffer with the new data during the execution of the **motor1ctrlISR()** interrupt. This code is located in the *motor1_drive.c* file.

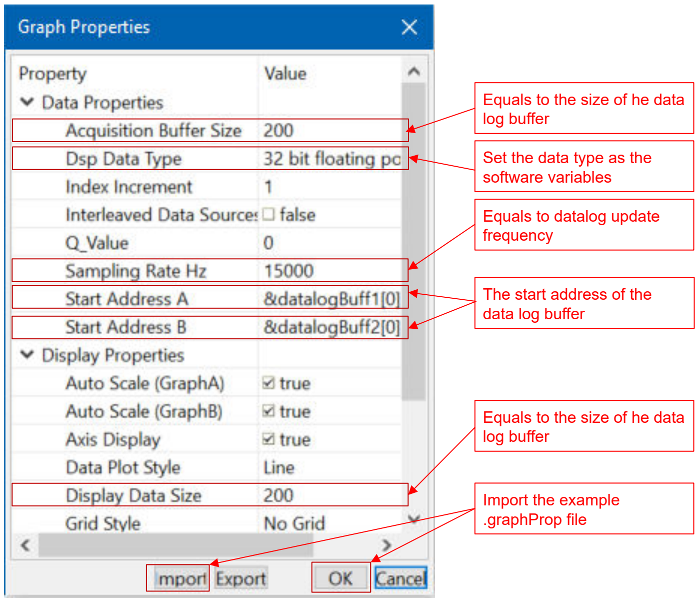
```
if(DATALOGIF_enable(datalogHandle) == true)
{
    DATALOGIF_updateWithDMA(datalogHandle);

    // Force trig DMA channel to save the data
    HAL_trigDMAforDLOG(halHandle, 0);
    HAL_trigDMAforDLOG(halHandle, 1);
}
```

Note

If there is not enough RAM, the datalog will be not used in the software on the device.

The datalog module is used with the graph tool, which provides a means to visually inspect the variables and judge system performance. The graph tool is available in CCS, which can display arrays of data in various graphical types. The arrays of data are stored in a device's memory in various formats.

While the project is in debug mode, open and setup time graph windows to plot the data log buffers as shown in Figure 3-24. Alternatively, the user can import the graph configurations files that are located in the project folder. In order to import them, Click: Tools -> Graph -> DualTime… and select import and browse to the following location `<install_location>\solutions\universal_motorcontrol_lab\common\debug` and select *motor_datalog_fp2.graphProp* file. Hit OK, this should add the Graphs to your debug perspective. Click on Continuous Refresh button on the top left corner of the graph tab.



Property	Value
Data Properties	
Acquisition Buffer Size	200
Dsp Data Type	32 bit floating po
Index Increment	1
Interleaved Data Sources	☐ false
Q_Value	0
Sampling Rate Hz	15000
Start Address A	&datalogBuff1[0]
Start Address B	&datalogBuff2[0]
Display Properties	
Auto Scale (GraphA)	☑ true
Auto Scale (GraphB)	☑ true
Axis Display	☑ true
Data Plot Style	Line
Display Data Size	200
Grid Style	No Grid

Equals to the size of he data log buffer

Set the data type as the software variables

Equals to datalog update frequency

The start address of the data log buffer

Equals to the size of he data log buffer

Import the example .graphProp file

Figure 3-24. Graph window settings

3.4.2 Using PWMDAC Function

The PWMDAC module converts the software variables into PWM signals using ePWM 6A, 6B, 7A, and 7B as shown in Figure 3-25. The PWMDAC module is only supported on the high voltage kit (TMDSHVMTRINSPIN) since it has extra PWM outputs with RC filters available on the board. If the PWMDAC module is used with a motor driver board that does not support the PWMDAC module, then the PWM signals will be routed to spare PWMs on the C2000 LaunchPad and the user would need to add RC filters to those pins in order to utilize the PWMDAC solution.

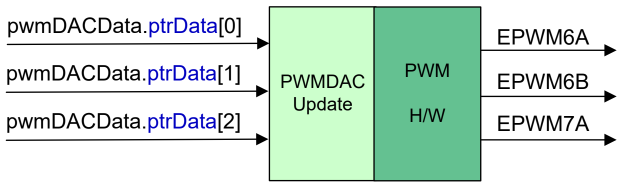


Figure 3-25. PWMDAC Module Block Diagram

The PWMDAC module can be used to view the signal, represented by the variable, at the outputs of the related pins through the external low-pass filters. Therefore, the external low-pass filters are necessary to view the actual signal waveforms as seen in Figure 3-26. The (1st-order) RC low-pass filter is used to filter out the high frequency component embedded in the actual low frequency signals. To select R and C values, the time constant can be expressed in terms of the cut-off frequency (f_c) as shown in the following equations Equation 49 and Equation 50.

$$\tau = RC = \frac{1}{2\pi f_c} \quad (49)$$

$$f_c = 2\pi RC \quad (50)$$

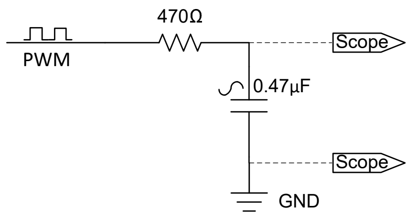


Figure 3-26. External RC Low-Lass Filter Connecting to a PWM Pin of the C2000 MCU

In order to enable the ePWM DAC functionality, the predefined symbol EPWMDAC_MODE must be added in the project properties as shown in Figure 3-19.

The code below shows the declaration of the PWMDAC object. This code is located in the *sys_main.c* file.

```
HAL_PWMDACData_t pwmDACData;
```

The code below shows the initialization and setting up of the PWMDAC object, handle and parameters. Four module inputs, ptrData[0], ptrData[1], ptrData[2], and ptrData[3] are configured to point to the addresses of four variables. The PWMDAC module inputs point to different system variables depending on the build level. This code is located in the *sys_main.c* file.

```
// set DAC parameters
pwmDACData.periodMax =
              PWMDAC_getPeriod(halHandle->pwmDACHandle[PWMDAC_NUMBER_1]);

pwmDACData.ptrData[0] = &motorVars_M1.anglePLL_rad;           // PWMDAC1
... ...
pwmDACData.ptrData[1] = &motorVars_M1.angleENC_rad;           // PWMDAC2
... ...
pwmDACData.ptrData[2] = &motorVars_M1.angleENC_rad;           // PWMDAC3
... ...
pwmDACData.ptrData[3] = &motorVars_M1.adcData.I_A.value[0];   // PWMDAC4

pwmDACData.offset[0] = 0.5f;
pwmDACData.offset[1] = 0.5f;
pwmDACData.offset[2] = 0.5f;
pwmDACData.offset[3] = 0.5f;

pwmDACData.gain[0] = 1.0f / MATH_TWO_PI;
pwmDACData.gain[1] = 1.0f / MATH_TWO_PI;
pwmDACData.gain[2] = 1.0f / MATH_TWO_PI;
pwmDACData.gain[3] = 4096.0f / USER_MOTOR1_OVER_CURRENT_A;
```

The code below shows the updating of the PWM outputs with new data during the execution of the **motor1ctrlISR()** interrupt. This code is located in the *motor1_drive.c* file.

```
// connect inputs of the PWMDAC module. HAL_writePWMDACData(halHandle, &pwmDACData);
```

3.4.3 Using External DAC Board

The DAC128S module converts up to 8 software variables into a 12-bit integer value and transmits the data through SPI to the digital-to-analog converter (DAC) on the DAC128S085EVM as shown in Figure 3-27.

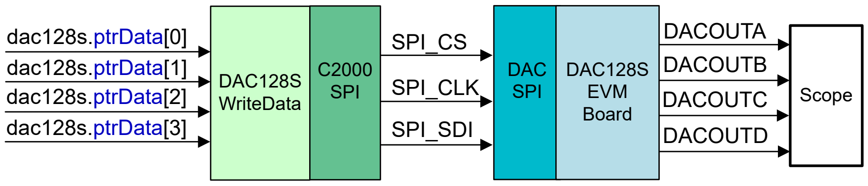


Figure 3-27. DAC128S Module Block Diagram

The DAC128S085EVM can be connected to the LaunchPad as shown in Figure 3-12 . Key connections of the DAC128S are shown in Figure 3-28.

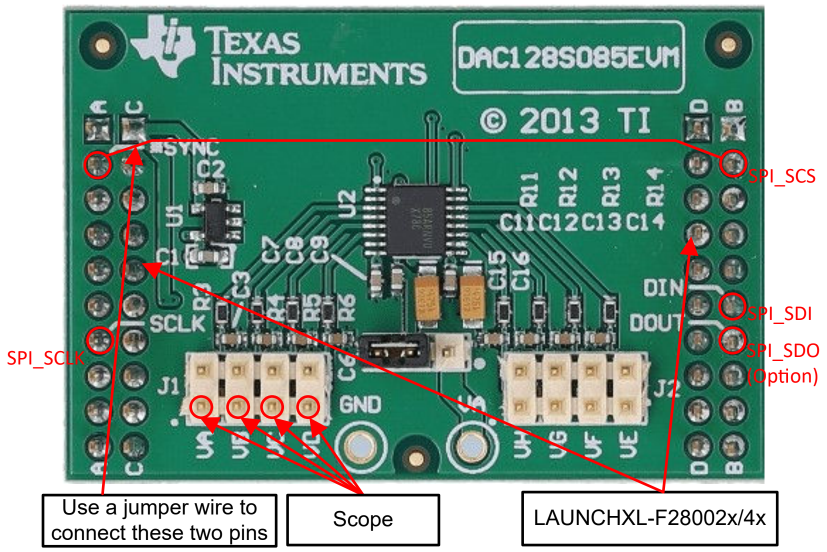


Figure 3-28. DAC128S085EVM Evaluation Board

Table 3-8. Hardware Changes Required for DAC128S085EVM Usage

LaunchPad Device	Hardware Changes Required
F28002x, F28003x	Jumper wire C2000 SPI_STE (SCS) pin JB-2 to SYNC pin JA-2 on the DAC128S085EVM, as shown in Figure 3-28.
F280013x	(1) Device has only 1x SPI module which is shared between Boosterpack site 1 and site 2. This requires populating 0 ohm resistors on the LaunchPad to connect the SPI signals to the Boosterpack site 2, which the DAC128S085EVM connects to. For the specific resistor identifiers, see *SPI Routing* section in the LAUNCHXL-F2800137 Schematic. (2) Jumper wire C2000 SPI_STE (SCS) pin to SYNC pin JA-2 on the DAC128S085EVM. The C2000 SPI_STE pin usage will depend on Inverter boosterpack being used: 1. GPIO19 is used for DRV8323RS, DRV8353RS, DRV8316, & 3-phase GaN inverter EVMs. This will require using the F280013x's Internal Oscillator and making hardware changes. For details, see the *Oscillator* section of the LAUNCHXL-F2800137 Schematic. 2. GPIO37 is used for all other inverter EVMs.

Hardware changes required for DAC128S085EVM usage. In order to enable the DAC128S functionality, the predefined symbol DAC_128S_ENABLE must be added in the project properties as shown in Figure 3-19.

The code below shows the declaration of the DAC128S object. This code is located in the *sys_main.c* file.

```
DAC128S_Handle   dac128sHandle;        //!< the DAC128S interface handle
DAC128S_Obj      dac128s;              //!< the DAC128S interface object
```

The DAC128S085 has an eight channel, 12-bit digital-to-analog converter (DAC) , so the user can set the output number between 1 and 8 by changing the value of the DAC_EN_CH_NUM define in the *dac128s085.h* file. Although the *sys_main.c* file initializes 8 ptrData[] module inputs to 8 different variable addresses, the number of module inputs that will actually be transmitted and used during the execution of the code will be determined by the value of the DAC_EN_CH_NUM constant that is defined in the *dac128s085.h* file (shown below). Since most oscilloscopes only have four probes, using 4 out of the 8 DAC128S085 outputs is the default setting in this example. Using 4 outputs is the default setting in this example lab since most of the oscilloscope only has four probes. More outputs will occupy much more ISR time to convert and transmit the data, which can negatively effect the time to spend on other tasks, which is a factor that must be considered if the user desires to use more than 4 outputs.

```
#define DAC_EN_CH_NUM                        (4)          // 1~8
```

The code below shows the initialization and setting up of the DAC128S object, handle and parameters. The code configures eight module inputs, ptrData[0] - ptrData[7], to point to the addresses of eight different software variables, but the number of module inputs that will actually be used is determined by the value of the DAC_EN_CH_NUM constant that is defined in the *dac128s085.h* file. The dac128s data points to different system variables depending on the build level and the control algorithm that is defined. This code can be found in the *sys_main.c* file.

```
// initialize the DAC128S
dac128sHandle = DAC128S_init(&dac128s);

// setup SPI for DAC128S
DAC128S_setupSPI(dac128sHandle);
... ...
// Build_Level_2 or Level_3, verify the estimator
dac128s.ptrData[0] = &motorVars_M1.angleGen_rad;             // CH_A
dac128s.ptrData[1] = &motorVars_M1.angleEST_rad;             // CH_B
dac128s.ptrData[2] = &motorVars_M1.anglePLL_rad;             // CH_C
dac128s.ptrData[3] = &motorVars_M1.adcData.I_A.value[0];     // CH_D
dac128s.ptrData[4] = &motorVars_M1.adcData.V_V.value[0];     // CH_E, N/A
dac128s.ptrData[5] = &motorVars_M1.adcData.I_A.value[1];     // CH_F, N/A
dac128s.ptrData[6] = &motorVars_M1.adcData.I_A.value[2];     // CH_G, N/A
dac128s.ptrData[7] = &motorVars_M1.adcData.V_V.value[1];     // CH_H, N/A

dac128s.gain[0] = 4096.0f / MATH_TWO_PI;
dac128s.gain[1] = 4096.0f / MATH_TWO_PI;
dac128s.gain[2] = 4096.0f / MATH_TWO_PI;
```

```
dac128s.gain[3] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[4] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V;
dac128s.gain[5] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[6] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[7] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V;

dac128s.offset[0] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[1] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[2] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[3] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[4] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[5] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[6] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[7] = (uint16_t)(0.5f * 4096.0f);
```

The below code shows the updating of the DAC128S board with the new data over SPI periodically during the execution of the **motor1ctrlISR()** interrupt. This code is located in the *motor1_drive.c* file. The number of the DAC outputs that will actually be updated will depend on the value of the DAC_EN_CH_NUM constant.

```
// Write the variables data value to DAC128S085
DAC128S_writeData(dac128sHandle);
```

3.5 Running the Project Incrementally Using Different Build Levels

The system is gradually tested and verified in multiple stages so that the final system can be confidently operated. To select a particular build option, change the value of the DMC_BUILDLEVEL define to the desired DMC_LEVEL_X option in the *sys_settings.h* file. Once the build option is selected, compile the project by right-clicking on the project name and clicking "*Rebuild Project*".

3.5.1 Level 1 Incremental Build

Objectives for this build level:

- Use the HAL object to initialize the peripherals of the MCU for the motor drive hardware.
- Verify the PWM and ADC driver modules
- Verify the ADC Offset validation
- Become familiar with the operation of CCS. More details about CCS can be found in the CCS User's Guide.

In this build level, the board is executed in open loop mode with a fixed PWM duty cycle. The duty cycles are set to 50%. This build level verifies the sensing of feedback values from the power stage and also operation of the PWM gate driver and ensures there are no hardware issues. Additionally, calibration of input and output voltage sensing can be performed in this build level. During this process the motor must remain disconnected. The software block diagram of this build level is shown in Figure 3-29.

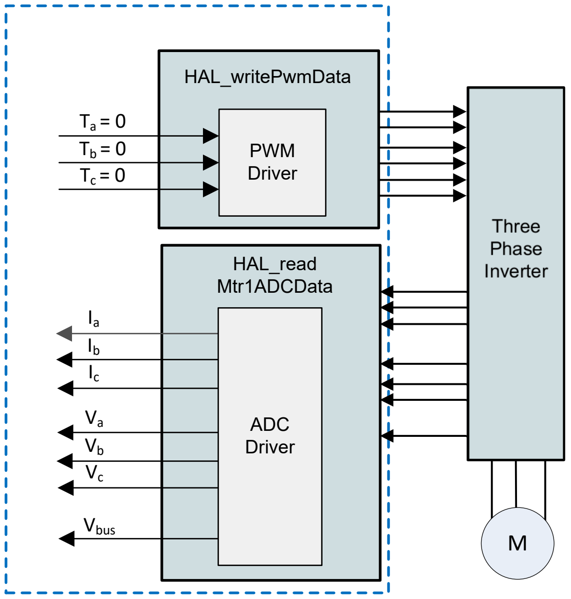


Figure 3-29. Build Level 1 Software Block Diagram - Offset Validation

3.5.1.1 Build and Load Project

1. Set up the motor driver hardware board and the C2000 Launchpad or controlCARD as described in Section 3.2, except the motor should **NOT** be connected to the motor driver board in this build level. Note: it is especially important to ensure that the the S2 switches on the LaunchPad are configured correctly, since this is necessary to connect the enable pin of the driver to GPIO 29 of the Launchpad.
2. Connect a USB cable from the computer to the on-board USB connector on the C2000 Launchpad or controlCARD to enable isolation JTAG emulation to the C2000 device.
3. Power on the motor driver board by applying the appropriate voltage to the bus voltage input terminal as described in Section 3.2.
4. Import the universal motor control lab project into CCS and select the right build configuration as described in Section 3.3.1. Open the *sys_settings.h* file and set DMC_BUILDLEVEL to DMC_LEVEL_1. This will ensure the project is configured to run the first incremental build.
5. In the Project Explorer window, make sure the correct target configuration file is set as Active by right clicking on the desired target configuration file name and selecting *Set as Active Target Configuration*. It is recommended to also set the desired target configuration file as default by right clicking on the file name and selecting *Set as Default Target Configuration*. One reason for doing this is because there is no visible indicator to show which file is active, but if the file is set to default then the [default] indicator will appear next to the file name in the project explorer window. Setting the file as default will also cause the file to be used by default unless a different configuration file is specifically set to Active. You can also link a target configuration to a project in the workspace by going to *View > Target Configurations* and right clicking on the target configuration name in the Target Configurations view and selecting *Link to Project*.
6. Right click on the project name and click on *Rebuild Project*. Watch the Console window. Any errors in the project will be displayed in the Console window.
7. On successful completion of the build, click on the Debug button or click *Run* → *Debug*. The IDE will now automatically connect to the target, load the output file into the device and change to the Debug perspective. The CCS Debug icon will appear in the upper right-hand corner, indicating that the user is now in the Debug Perspective view. The program should be halted at the start of main().

CAUTION

Do not click *Cancel*, turn off the power of the board, or disconnect the emulator when loading the code to flash.

3.5.1.2 Setup Debug Environment Windows

It is standard debug practice to watch local and global variables while debugging code. There are various methods for doing this in CCS, such as memory views and watch views. Additionally, CCS has the ability to create time (and frequency) domain plots. This ability allows the user to view waveforms using the graph tool. For information on how to set up and configure the graph tool, see Section 3.4.1. For information on setting up the Expressions window, see the instructions below.

1. Setup watch window: *Click View* → *Expressions* on the menu bar to open an Expressions watch window. Variables can be added to the Expressions window by clicking *Add new expression* within the Expressions window and typing the name of the variable and then pressing enter. The number format that the variable value is displayed in is based on the number format associated with the variable when it was declared. You can change the desired number format for a particular variable by right clicking on the variable and navigating to *Number Format* and selecting the desired format.
2. Alternately, a group of variables can be imported into the Expressions window by right clicking within the Expressions window and clicking Import, browse to the directory of the project at `<install_location>\solutions\universal_motorcontrol_lab\common\debug\`, select the *universal_lab_level1.txt* file, and click OK to import the variables shown in Figure 3-30.

Note

Some of the variables have not been initialized at this point in the main code and may contain some useless values.

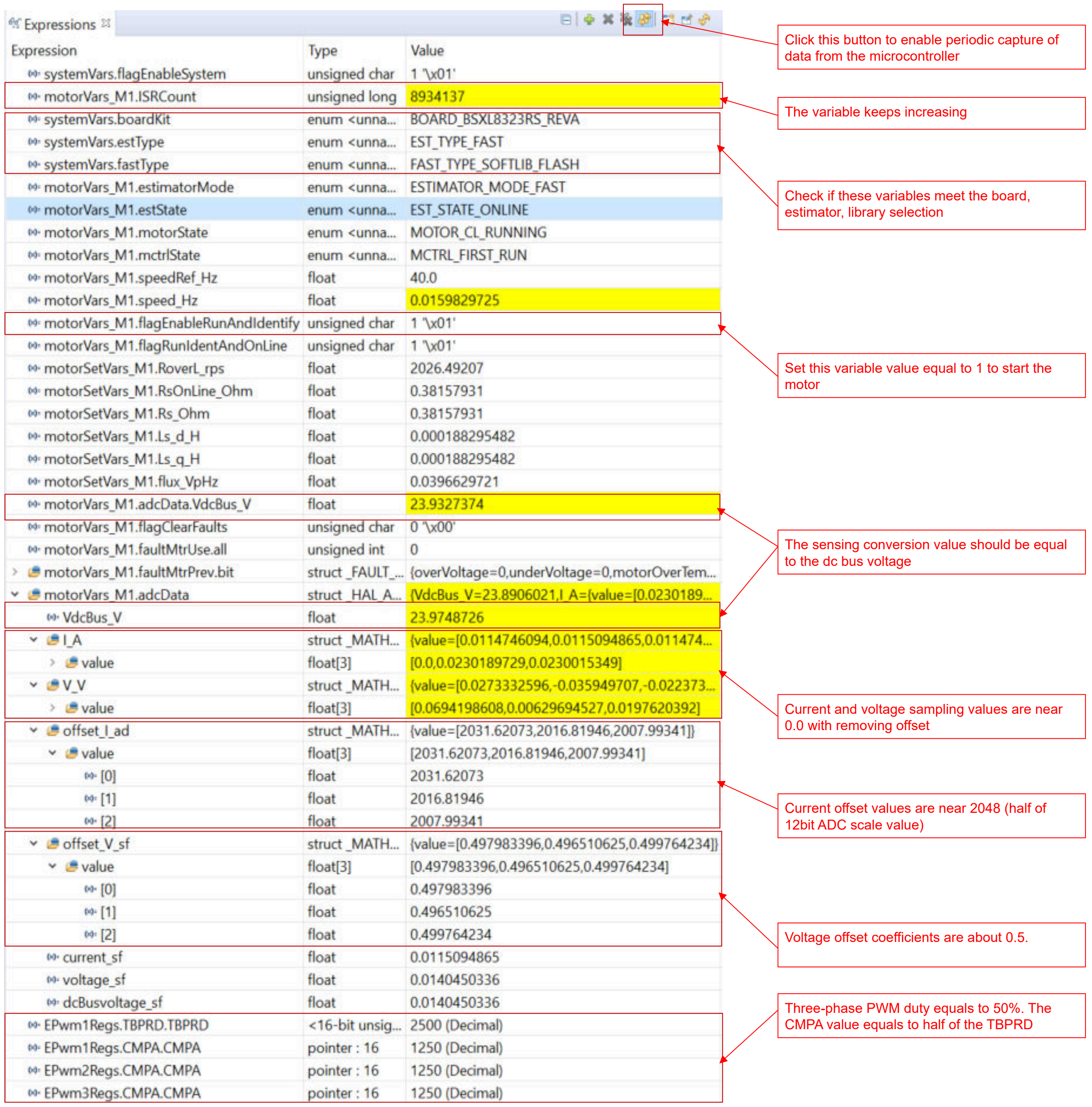
Note

The structure of variables *motorVars_M1* has references to most variables that are related to controlling motor drive.

3. Click on the Continuous Refresh button in the top right corner of the Expressions Window tab to enable periodic capture of data from the Microcontroller. By clicking the "View Menu" button (the three dots in the upper right hand corner of the Expressions window),you can select *Continuous Refresh Interval* and edit the refresh rate of the Expressions window. Note that choosing too fast an interval may affect performance.

3.5.1.3 Run the Code

1. Run the code by pressing on the run button, or click *Run* → *Resume* in the Debug tab.
2. The project should now run, and the values in the graph and watch window should keep updating.
3. In the Expressions window, set the variables *motorVars_M1.flagEnableRunAndIdentify* to 1 after *systemVars.flagEnableSystem* was automatically set to 1 in the watch window.
4. The project should now run, and the values in the graphs and expressions window should continuously update as shown in Figure 3-30 while using this project. You may want to re-size the windows according to your preference.
5. In the watch view, the variables *motorVars_M1.flagRunIdentAndOnLine* should be set to 1 automatically if there are no faults. The *ISRCount* should be increasing continuously. If the *ISRCount* is not increasing, ensure that the TEST_ENABLE predefine is defined. If this predefine is not defined, then the *ISRCount* will not increment in the code.
6. Check the calibration offsets of the motor driver board. The offset values of the motor phase current sensing values should be equal to approximately half of the ADC scale current and the phase voltage offsets should be equal to approximately 0.5 as shown in Figure 3-30.
7. If using the graph tool, the variables shown in the graphs are the phase currents for phase u and v. These should be close to 0 in magnitude.
8. Expand and check the *MotorVars_M1.faultMtrPrev.bit* structure to ensure that there are no fault flags set.
9. Using an oscilloscope, probe the PWM outputs that are used for motor drive control. The duty cycles of the three PWMs are set to 50% in this build level. The expected PWM output waveforms are as shown in Figure 3-31. The PWM switching frequency will be the same as the value that was set for the USER_M1_PWM_FREQ_kHz define in the *user_mtr1.h* file.
10. Set the *motorVars_M1.flagEnableRunAndIdentify* variable to 0 to disable the PWMs.
11. If any of the previous steps provide unexpected results, additional debug will be necessary. A few things to check are:
    a. Ensure that the DRV board is properly set up, with the proper capacitors/resistors populated on the EVM.
    b. Ensure that the motor driver board being used is the same as the board selected in the build configurations (see 111).
    c. Ensure that the proper predefines are set.
    d. Ensure that the switches are configured properly on the C2000 MCU Lunchpad/ControlCARD as described in Section 3.2.
12. Once the previous steps are complete, the controller can now be halted, and the debug connection terminated. Halt the controller by first clicking the Halt button on the toolbar or by clicking *Target* → *Halt*. Finally, reset the controller by clicking on the button or clicking *Run* → *Reset*→CPU Reset.
13. Close the CCS debug session by clicking the Terminate Debug Session button or clicking *Run* → *Terminate.* This will halt the program and disconnect Code Composer from the MCU.
14. It is not necessary to terminate the debug session each time the user changes or runs the code again. Instead, the following procedure can be followed. After rebuilding the project, press the button or click *Run* → *Reset*→CPU Reset, and then press the button or click *Run* → *Restart.* The project must be terminated if the target device or the configuration is changed, and before shutting down CCS.



Expressions

Expression	Type	Value
systemVars.flagEnableSystem	unsigned char	1 '\x01'
motorVars_M1.ISRCount	unsigned long	8934137
systemVars.boardKit	enum <unna...	BOARD_BSXL8323RS_REVA
systemVars.estType	enum <unna...	EST_TYPE_FAST
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST
motorVars_M1.estState	enum <unna...	EST_STATE_ONLINE
motorVars_M1.motorState	enum <unna...	MOTOR_CL_RUNNING
motorVars_M1.mctrlState	enum <unna...	MCTRL_FIRST_RUN
motorVars_M1.speedRef_Hz	float	40.0
motorVars_M1.speed_Hz	float	0.0159829725
motorVars_M1.flagEnableRunAndIdentify	unsigned char	1 '\x01'
motorVars_M1.flagRunIdentAndOnLine	unsigned char	1 '\x01'
motorSetVars_M1.RoverL_rps	float	2026.49207
motorSetVars_M1.RsOnLine_Ohm	float	0.38157931
motorSetVars_M1.Rs_Ohm	float	0.38157931
motorSetVars_M1.Ls_d_H	float	0.000188295482
motorSetVars_M1.Ls_q_H	float	0.000188295482
motorSetVars_M1.flux_VpHz	float	0.0396629721
motorVars_M1.adcData.VdcBus_V	float	23.9327374
motorVars_M1.flagClearFaults	unsigned char	0 '\x00'
motorVars_M1.faultMtrUse.all	unsigned int	0
motorVars_M1.faultMtrPrev.bit	struct _FAULT_...	{overVoltage=0,underVoltage=0,motorOverTem...
motorVars_M1.adcData	struct _HAL_A...	{VdcBus_V=23.8906021,I_A={value=[0.0230189...
VdcBus_V	float	23.9748726
I_A	struct _MATH...	{value=[0.0114746094,0.0115094865,0.011474...
value	float[3]	[0.0,0.0230189729,0.0230015349]
V_V	struct _MATH...	{value=[0.0273332596,-0.035949707,-0.022373...
value	float[3]	[0.0694198608,0.00629694527,0.0197620392]
offset_I_ad	struct _MATH...	{value=[2031.62073,2016.81946,2007.99341]}
value	float[3]	[2031.62073,2016.81946,2007.99341]
[0]	float	2031.62073
[1]	float	2016.81946
[2]	float	2007.99341
offset_V_sf	struct _MATH...	{value=[0.497983396,0.496510625,0.499764234]}
value	float[3]	[0.497983396,0.496510625,0.499764234]
[0]	float	0.497983396
[1]	float	0.496510625
[2]	float	0.499764234
current_sf	float	0.0115094865
voltage_sf	float	0.0140450336
dcBusvoltage_sf	float	0.0140450336
EPwm1Regs.TBPRD.TBPRD	<16-bit unsig...	2500 (Decimal)
EPwm1Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)
EPwm2Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)
EPwm3Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)

Click this button to enable periodic capture of data from the microcontroller

The variable keeps increasing

Check if these variables meet the board, estimator, library selection

Set this variable value equal to 1 to start the motor

The sensing conversion value should be equal to the dc bus voltage

Current and voltage sampling values are near 0.0 with removing offset

Current offset values are near 2048 (half of 12bit ADC scale value)

Voltage offset coefficients are about 0.5.

Three-phase PWM duty equals to 50%. The CMPA value equals to half of the TBPRD

Figure 3-30. Build Level 1: Variables in Expressions Window

The C2000 PWM with deadband outputs to the input of the gate drive as shown in Figure 3-31

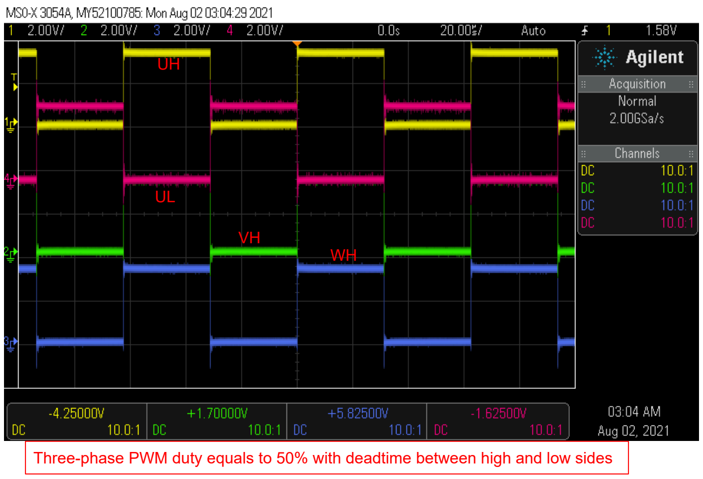

MSO-X 3054A, MY52100785: Mon Aug 02 03:04:29 2021
UH
UL
VH
WH
Agilent
Acquisition
Normal
2.00GSa/s
Channels
-4.25000V
+1.70000V
+5.82500V
-1.62500V
03:04 AM
Aug 02, 2021
Three-phase PWM duty equals to 50% with deadtime between high and low sides

Figure 3-31. Build Level 1: PWM Output Waveforms

3.5.2 Level 2 Incremental Build

Objectives learned in this build level:

- Implements a simple scalar v/f control of motor to drive dual motor for validating current and voltage sensing circuit, and gate driver circuit.
- Test InstaSPIN-FOC FAST or eSMO modules for motor control.

In this build level the system is running with open-loop control, so the ADC values are only used for verification and validation, they are not actually used in the control loop of the motor. The software flow for this build level is shown in Figure 3-32.

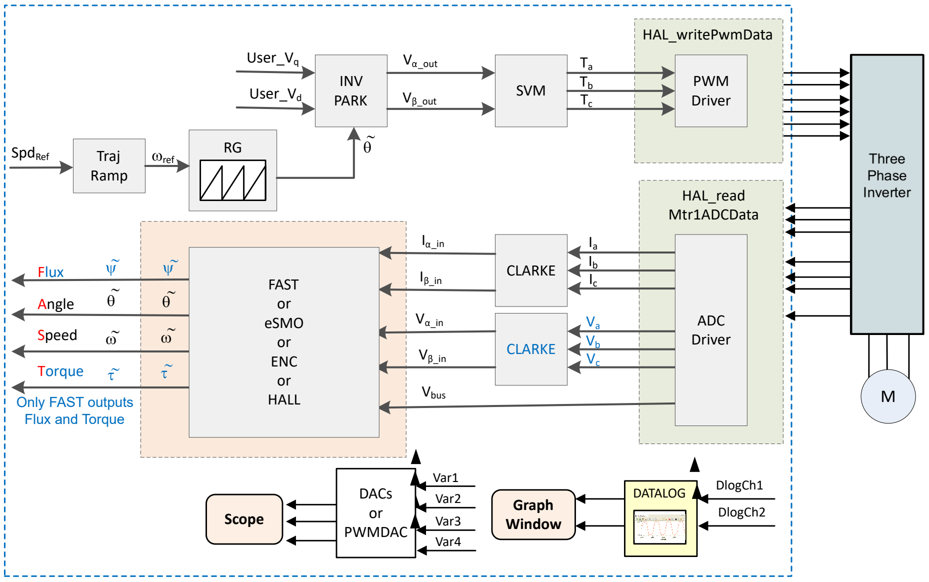


Figure 3-32. Build Level 2 Software Block Diagram - Open Loop Control

3.5.2.1 Build and Load Project

Connect the motor to the appropriate terminals on the motor driver evaluation board. Follow steps 2-7 of Section 3.5.1.1 to build and load the project. In step 4, set the DMC_BUILDLEVEL to DMC_LEVEL_2.

CAUTION
Do not click *Cancel*, turn off the power of the board, or disconnect the emulator when loading the code to flash.

3.5.2.2 Setup Debug Environment Windows

Follow the steps in Section 3.5.1.2 to import the variables into the Expressions window. For build level 2, select the *universal_lab_level2.txt* file. The Expressions window appears as shown in Figure 3-33.

3.5.2.3 Run the Code

1. Power on the appropriate power supply, and gradually increase the output voltage of the power supply to get an appropriate DC-bus voltage.
2. If using the graph tool, lab 2 uses the same graph configurations and parameters as lab 1 to monitor 2 of the phase currents.
3. Run the project by clicking on the button, or click *Run* → *Resume* in the Debug tab. The *systemVars.flagEnableSystem* should be set to *1* after a fixed time, that means the offsets calibration has completed. The fault flags, *motorVars_M1.faultMtrUse.all* should be equal to *0.* If this is not the case, the user should double check the current and voltage sensing circuit in lab 1 as described in Section 3.5.1.3.
4. To verify the current and voltage sensing circuits of the motor driver, set the variable *motorVars_M1.flagEnableRunAndIdentify* to *1* in the Expressions window as shown in Figure 3-33. The motor will run with voltage/frequency (v/f) open loop. If the motor doesn't spin smoothly, tune the v/f profile parameters in the *user_mtr1.h* file as shown below according to the specification of the motor. See1 for more details on tuning the v/f profile parameters. Note: modification of these parameters will require rebuilding the project. See step 14 of 14 for more information on rebuilding the project in debug mode.

```
#define USER_MOTOR1_FREQ_LOW_HZ            (5.0)          // Hz
#define USER_MOTOR1_FREQ_HIGH_HZ           (400.0)        // Hz
#define USER_MOTOR1_VOLT_MIN_V             (1.0)          // Volt
#define USER_MOTOR1_VOLT_MAX_V             (24.0)         // Volt
```

5. The *motorVars_M1.speedRef_Hz* variable is used to set the speed reference for the motor. Check the value of the *motorVars_M1.speed_Hz* variable in the Expressions window to ensure that the motor speed (*motorVars_M1.*speed_Hz) is close to the reference speed (*motorVars_M1.speedRef_Hz)* as shown in Figure 3-33.
6. In this build level, the current sensing, voltage sensing, rotor angle estimator, and generator need to be validated. This can be done using either the PWMDAC or the DAC128S module with an oscilloscope as described in Section 3.4.2 or Section 3.4.3. Additionally, the DATALOG module can be used to view these sensing waveforms. For more information on using the DATALOG to view the currents, voltages, and angle signals, see step 7. If using the DAC128S module, configure the dac128s.ptrdata[] inputs to correspond with the code shown in each of the following subsections. The code modification can be done by uncommenting the appropriate code section for the DAC128S initialization and by commenting out the rest of the DAC128S initialization that is not needed in the subsection. It will be necessary to rebuild the project after modifying the code to correspond with each subsection. See step 14 of 14 for more information on rebuilding the project in debug mode.
   a. The first set of parameters to monitor is the phase currents. This is done by uncommenting the portion of code shown below, which sets up the dac128s pointer data to the 3 phase currents as well as the angle estimator variable. This code is found in the *sys_main.c* file. The expected current waveforms should be similar to the waveforms shown on the oscilloscope in Figure 3-35. The current waveforms measured at the DAC128S outputs should be almost the same as the corresponding phase current waveforms capture by a current probe. If this is the case, that indicates the current sensing circuit is good for motor control. If this is not the case, it may be necessary to tune the v/f parameters in the *user_mtr1.h* file as described in 1.

```
dac128s.ptrData[0] = &motorVars_M1.adcData.I_A.value[0];     // CH_A
dac128s.ptrData[1] = &motorVars_M1.adcData.I_A.value[1];     // CH_B
dac128s.ptrData[2] = &motorVars_M1.adcData.I_A.value[2];     // CH_C
dac128s.ptrData[3] = &motorVars_M1.angleGen_rad;             // CH_D
dac128s.ptrData[4] = &motorVars_M1.angleEST_rad;                // CH_E, N/A
dac128s.ptrData[5] = &motorVars_M1.adcData.V_V.value[0];        // CH_F, N/A
dac128s.ptrData[6] = &motorVars_M1.adcData.V_V.value[1];        // CH_G, N/A
dac128s.ptrData[7] = &motorVars_M1.adcData.V_V.value[2];        // CH_H, N/A

dac128s.gain[0] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[1] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[2] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[3] = 4096.0f / MATH_TWO_PI;
dac128s.gain[4] = 4096.0f / MATH_TWO_PI;                          // NA
dac128s.gain[5] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V; // N/A
dac128s.gain[6] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V; // N/A
dac128s.gain[7] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V; // N/A
... ...
dac128s.offset[0] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[1] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[2] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[3] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[4] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[5] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[6] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[7] = (uint16_t)(0.5f * 4096.0f); // N/A
```

b. The second set of parameters to monitor are the phase voltages. To do this, the section of code that sets up the dac128s pointer data to point to the phase voltage should be uncommented as shown below. This code is found in the *sys_main.c* file. The output waveform shape of the phase voltage sensing from the DAC128S module should look similar to the image shown in Figure 3-36 or Figure 3-37. If this is the case, that indicates that the voltage sensing sircuit is working as expected. Note that the amplitude can vary depending on the supply voltage and motor used, but the shape of the waveform should be the same. Note the difference in the phase waveforms depending on the SVM mode used. For the Common SVM mode, there is a voltage dip at the upper and the lower peaks of the waveform, whereas for the Minimum SVM mode, the voltage dip only appears on the upper peak, but stays flat on the lower peak. To change the SVM mode, select either SVM_COM_C or SVM_MIN_C from the *motorVars_M1.svmMode* enumeration in the Expressions window.

```
dac128s.ptrData[0] = &motorVars_M1.adcData.V_V.value[0];     // CH_A
dac128s.ptrData[1] = &motorVars_M1.adcData.V_V.value[1];     // CH_B
dac128s.ptrData[2] = &motorVars_M1.adcData.V_V.value[2];     // CH_C
dac128s.ptrData[3] = &motorVars_M1.angleGen_rad;             // CH_D
dac128s.ptrData[4] = &motorVars_M1.angleEST_rad;                // CH_E, N/A
dac128s.ptrData[5] = &motorVars_M1.adcData.I_A.value[0];        // CH_F, N/A
dac128s.ptrData[6] = &motorVars_M1.adcData.I_A.value[1];        // CH_G, N/A
dac128s.ptrData[7] = &motorVars_M1.adcData.I_A.value[2];        // CH_H, N/A

dac128s.gain[0] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V;
dac128s.gain[1] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V;
dac128s.gain[2] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V;
dac128s.gain[3] = 4096.0f / MATH_TWO_PI;
dac128s.gain[4] = 4096.0f / MATH_TWO_PI;                            // N/A
dac128s.gain[5] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A; // N/A
dac128s.gain[6] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A; // N/A
dac128s.gain[7] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A; // N/A
... ...
dac128s.offset[0] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[1] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[2] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[3] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[4] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[5] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[6] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[7] = (uint16_t)(0.5f * 4096.0f); // N/A
```

c. The third set of parameters to monitor is the angle generator and the angle estimator parameters. To do this, the section of code that sets up the dac128s pointer data to point to the appropriate variables should be uncommented as shown below. This code is found in the *sys_main.c* file. The angle from the angle generator and the estimator waveforms should be similar to the oscilloscope waveforms shown in Figure 3-38. Notice that the angle of the force angle generator is very similar to the estimated rotor angle of the FAST or eSMO estimator. This indicates that the FAST or eSMO estimator works as expected with the motor parameters and the sampling current and voltage signals.

```
dac128s.ptrData[0] = &motorVars_M1.angleGen_rad;            // CH_A
dac128s.ptrData[1] = &motorVars_M1.angleEST_rad;            // CH_B
dac128s.ptrData[2] = &motorVars_M1.anglePLL_rad;            // CH_C
dac128s.ptrData[3] = &motorVars_M1.adcData.I_A.value[0];    // CH_D
dac128s.ptrData[4] = &motorVars_M1.adcData.V_V.value[0];        // CH_E, N/A
dac128s.ptrData[5] = &motorVars_M1.adcData.I_A.value[1];        // CH_F, N/A
dac128s.ptrData[6] = &motorVars_M1.adcData.I_A.value[2];        // CH_G, N/A
dac128s.ptrData[7] = &motorVars_M1.adcData.V_V.value[1];        // CH_H, N/A

dac128s.gain[0] = 4096.0f / MATH_TWO_PI;
dac128s.gain[1] = 4096.0f / MATH_TWO_PI;
dac128s.gain[2] = 4096.0f / MATH_TWO_PI;
dac128s.gain[3] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A;
dac128s.gain[4] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V; // N/A
dac128s.gain[5] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A; // N/A
dac128s.gain[6] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_CURRENT_A; // N/A
dac128s.gain[7] = 2.0f * 4096.0f / USER_M1_ADC_FULL_SCALE_VOLTAGE_V; // N/A

dac128s.offset[0] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[1] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[2] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[3] = (uint16_t)(0.5f * 4096.0f);
dac128s.offset[4] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[5] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[6] = (uint16_t)(0.5f * 4096.0f); // N/A
dac128s.offset[7] = (uint16_t)(0.5f * 4096.0f); // N/A
```

7. If using the DATALOG module with the graph tool to check the current sensing signals, voltage sensing signals, and the angle outputs, follow the steps described below. For more info on the datalog module, see Section 3.4.1. Note: It will be necessary to rebuild the project in between each of the below steps after modifying the code. See step 14 of 14 for more information on rebuilding the project in debug mode.
   a. To test the phase currents for phase U and V using the DATALOG module, the following code must be set up in the *sys_main.c* file.

Note

This code is already configured by default for build level 2. The phase current sampling signals waveform displayed on the graph tool as shown in Figure 3-39.

```
datalogObj->iptr[0] = &motorVars_M1.adcData.I_A.value[0];
datalogObj->iptr[1] = &motorVars_M1.adcData.I_A.value[1];
```

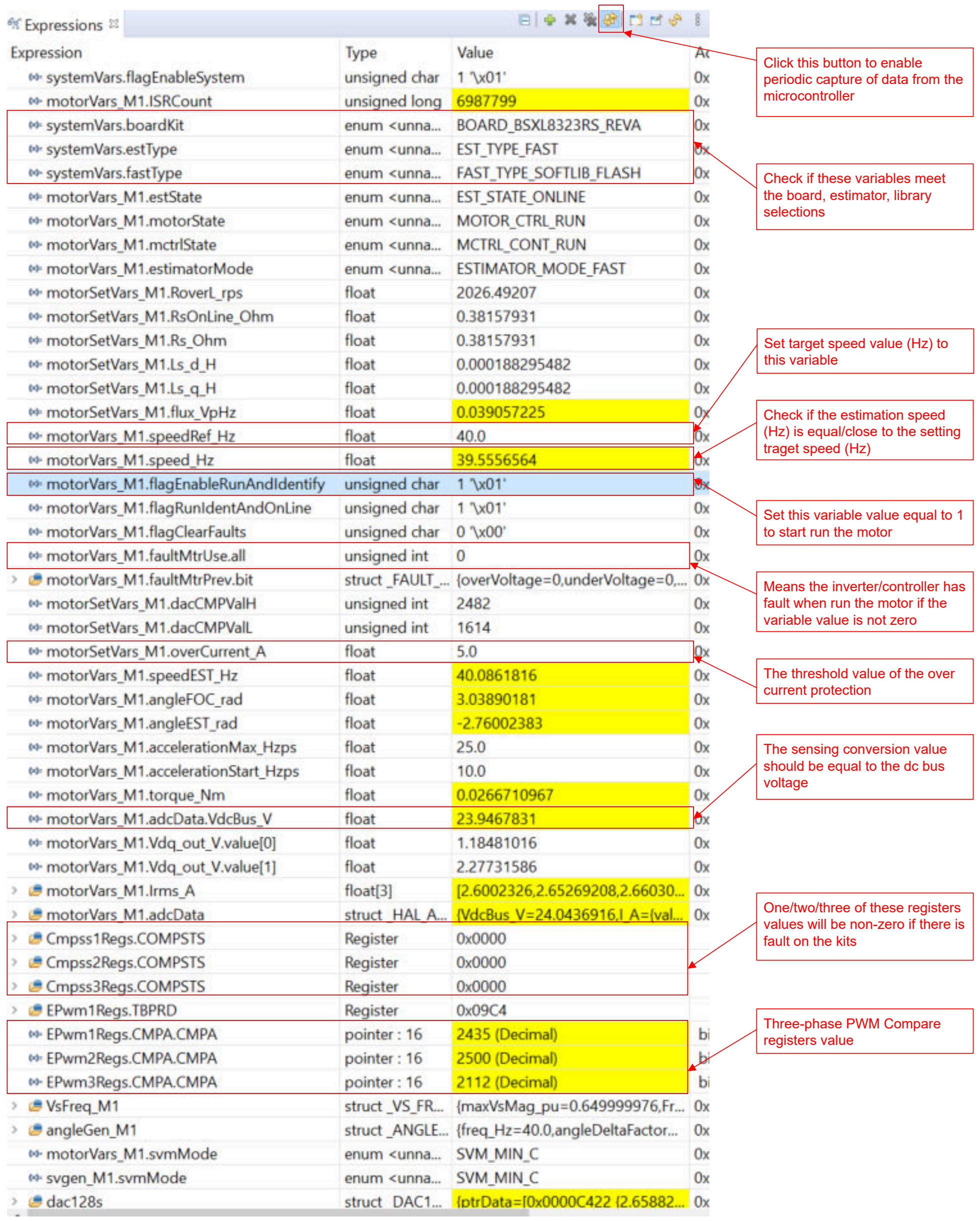
   b. To test the phase voltages for phase U and V using the DATALOG module, the following code must be set up in the *sys_main.c* file and modified to the code shown below. The phase voltage sampling signals waveform on graph tool as shown in Figure 3-40.

```
datalogObj->iptr[0] = &motorVars_M1.adcData.V_V.value[0];
datalogObj->iptr[1] = &motorVars_M1.adcData.V_V.value[1];
```

   c. Configuring DATALOG module four inputs as the following codes. The angle from the force angle generator or estimator waveforms on the graph tool as shown in Figure 3-41. Notice that the angle of the force angle generator is very similar as the estimated rotor angle of the FAST or eSMO estimator.

```
datalogObj->iptr[0] = &motorVars_M1.angleFOC_rad;
datalogObj->iptr[1] = &motorVars_M1.angleEST_rad;
```

8. Verify the over current fault protection by decreasing the value of the variable *motorVars_M1.overCurrent_A*, the over current protection is implemented by the CMPSS modules. The over current fault will be trigger if the *motorVars_M1.overCurrent_A* is set to a value less than the motor phase current actual value, the PWM output will be disabled, the *motorVars_M1.flagEnableRunAndIdentify* is cleared to *0,* and the *motorVars_M1.faultMtrUse.all* will be set to *0x10 (16)* as shown in *Figure 3-34*.
9. Set the variables *motorVars_M1.flagEnableRunAndIdentify* to 0 to stop run the motor.
10. Once complete, the controller can now be halted, and the debug connection terminated. Fully halting the controller by first clicking the Halt button on the toolbar or by clicking *Target → Halt*. Finally, reset the controller by clicking on or clicking *Run → Reset*.
11. Close CCS debug session by clicking on Terminate Debug Session or clicking *Run → Terminate*.
12. Power off the power supply to the inverter kit.



Expression	Type	Value	Ac
systemVars.flagEnableSystem	unsigned char	1 '\x01'	0x
motorVars_M1.ISRCount	unsigned long	6987799	0x
systemVars.boardKit	enum <unna...	BOARD_BSXL8323RS_REVA	0x
systemVars.estType	enum <unna...	EST_TYPE_FAST	0x
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH	0x
motorVars_M1.estState	enum <unna...	EST_STATE_ONLINE	0x
motorVars_M1.motorState	enum <unna...	MOTOR_CTRL_RUN	0x
motorVars_M1.mctrlState	enum <unna...	MCTRL_CONT_RUN	0x
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST	0x
motorSetVars_M1.RoverL_rps	float	2026.49207	0x
motorSetVars_M1.RsOnLine_Ohm	float	0.38157931	0x
motorSetVars_M1.Rs_Ohm	float	0.38157931	0x
motorSetVars_M1.Ls_d_H	float	0.000188295482	0x
motorSetVars_M1.Ls_q_H	float	0.000188295482	0x
motorSetVars_M1.flux_VpHz	float	0.039057225	0x
motorVars_M1.speedRef_Hz	float	40.0	0x
motorVars_M1.speed_Hz	float	39.5556564	0x
motorVars_M1.flagEnableRunAndIdentify	unsigned char	1 '\x01'	0x
motorVars_M1.flagRunIdentAndOnLine	unsigned char	1 '\x01'	0x
motorVars_M1.flagClearFaults	unsigned char	0 '\x00'	0x
motorVars_M1.faultMtrUse.all	unsigned int	0	0x
motorVars_M1.faultMtrPrev.bit	struct _FAULT_...	{overVoltage=0,underVoltage=0,...	0x
motorSetVars_M1.dacCMPValH	unsigned int	2482	0x
motorSetVars_M1.dacCMPValL	unsigned int	1614	0x
motorSetVars_M1.overCurrent_A	float	5.0	0x
motorVars_M1.speedEST_Hz	float	40.0861816	0x
motorVars_M1.angleFOC_rad	float	3.03890181	0x
motorVars_M1.angleEST_rad	float	-2.76002383	0x
motorVars_M1.accelerationMax_Hzps	float	25.0	0x
motorVars_M1.accelerationStart_Hzps	float	10.0	0x
motorVars_M1.torque_Nm	float	0.0266710967	0x
motorVars_M1.adcData.VdcBus_V	float	23.9467831	0x
motorVars_M1.Vdq_out_V.value[0]	float	1.18481016	0x
motorVars_M1.Vdq_out_V.value[1]	float	2.27731586	0x
motorVars_M1.Irms_A	float[3]	[2.6002326,2.65269208,2.66030...	0x
motorVars_M1.adcData	struct _HAL_A...	{VdcBus_V=24.0436916,I_A={val...	0x
Cmpss1Regs.COMPSTS	Register	0x0000	
Cmpss2Regs.COMPSTS	Register	0x0000	
Cmpss3Regs.COMPSTS	Register	0x0000	
EPwm1Regs.TBPRD	Register	0x09C4	
EPwm1Regs.CMPA.CMPA	pointer : 16	2435 (Decimal)	bi
EPwm2Regs.CMPA.CMPA	pointer : 16	2500 (Decimal)	bi
EPwm3Regs.CMPA.CMPA	pointer : 16	2112 (Decimal)	bi
VsFreq_M1	struct _VS_FR...	{maxVsMag_pu=0.649999976,Fr...	0x
angleGen_M1	struct _ANGLE...	{freq_Hz=40.0,angleDeltaFactor...	0x
motorVars_M1.svmMode	enum <unna...	SVM_MIN_C	0x
svgen_M1.svmMode	enum <unna...	SVM_MIN_C	0x
dac128s	struct _DAC1...	{ptrData=[0x0000C422 {2.65882...	0x

Click this button to enable periodic capture of data from the microcontroller

Check if these variables meet the board, estimator, library selections

Set target speed value (Hz) to this variable

Check if the estimation speed (Hz) is equal/close to the setting traget speed (Hz)

Set this variable value equal to 1 to start run the motor

Means the inverter/controller has fault when run the motor if the variable value is not zero

The threshold value of the over current protection

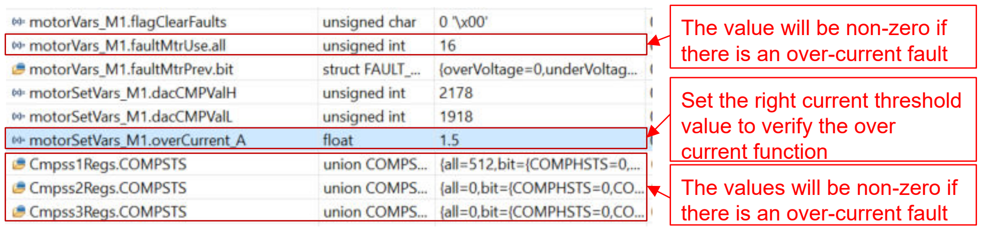
The sensing conversion value should be equal to the dc bus voltage

One/two/three of these registers values will be non-zero if there is fault on the kits

Three-phase PWM Compare registers value

Figure 3-33. Build Level 2: Variables in Expressions Window

Adjust the value of *motorVars_M1.overCurrent_A* in Expression window to trigger the over current fault as shown in Figure 3-34.



motorVars_M1.flagClearFaults	unsigned char	0 '\x00'
motorVars_M1.faultMtrUse.all	unsigned int	16
motorVars_M1.faultMtrPrev.bit	struct FAULT_...	{overVoltage=0,underVoltag...
motorSetVars_M1.dacCMPValH	unsigned int	2178
motorSetVars_M1.dacCMPValL	unsigned int	1918
motorSetVars_M1.overCurrent_A	float	1.5
Cmpss1Regs.COMPSTS	union COMPS...	{all=512,bit={COMPHSTS=0,...
Cmpss2Regs.COMPSTS	union COMPS...	{all=0,bit={COMPHSTS=0,CO...
Cmpss3Regs.COMPSTS	union COMPS...	{all=0,bit={COMPHSTS=0,CO...

The value will be non-zero if there is an over-current fault
Set the right current threshold value to verify the over current function
The values will be non-zero if there is an over-current fault

Figure 3-34. Build Level 2: Current Protection Setting

Use DAC128S085EVM with an oscilloscope to monitor three phase sensing current of the motor and compare the sampling value to the measurement value with a current probe as shown in Figure 3-35.

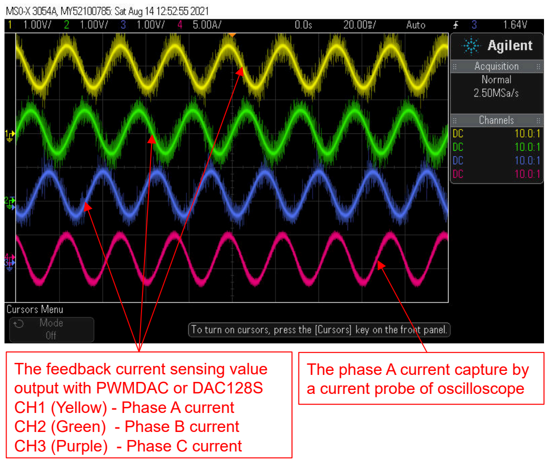

MSO-X 3054A, MY52100785: Sat Aug 14 12:52:55 2021
Agilent
Acquisition
Normal
2.50MSa/s
Channels
Cursors Menu
To turn on cursors, press the [Cursors] key on the front panel.
The feedback current sensing value output with PWMDAC or DAC128S
CH1 (Yellow) - Phase A current
CH2 (Green) - Phase B current
CH3 (Purple) - Phase C current
The phase A current capture by a current probe of oscilloscope

Figure 3-35. Build Level 2: Motor Phase Current Waveforms

Use DAC128S085EVM with an oscilloscope to monitor three phase sensing voltage of the motor, and use common mode SVPWM by setting *motorVars_M1.svmMode* equal to SVM_COM_C as shown in Figure 3-36.

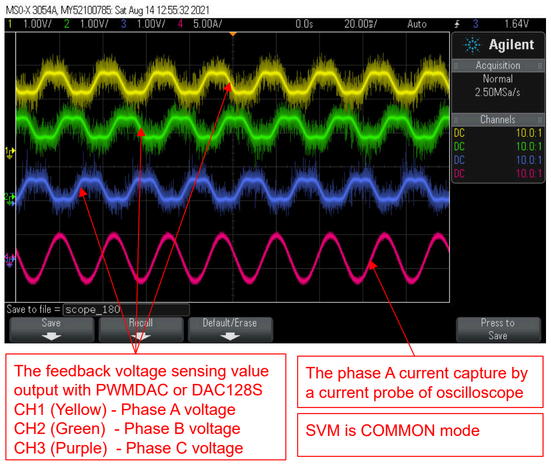

MSO-X 3054A, MY52100785: Sat Aug 14 12:55:32 2021
Agilent
Acquisition
Normal
2.50MSa/s
Channels
Save
Recall
Default/Erase
Press to Save
The feedback voltage sensing value output with PWMDAC or DAC128S
CH1 (Yellow) - Phase A voltage
CH2 (Green) - Phase B voltage
CH3 (Purple) - Phase C voltage
The phase A current capture by a current probe of oscilloscope
SVM is COMMON mode

Figure 3-36. Build Level 2: Motor Phase Voltage Waveforms Using Common SVM Mode

Use DAC128S085EVM with an oscilloscope to monitor three phase sensing voltage of the motor, and use minimum mode SVPWM by setting *motorVars_M1.svmMode* equal to SVM_MIN_C as shown in Figure 3-37.

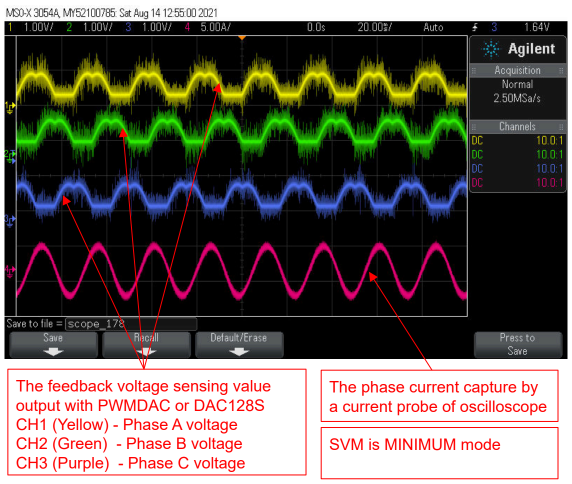

MSO-X 3054A, MY52100785: Sat Aug 14 12:55:00 2021
1 1.00V/ 2 1.00V/ 3 1.00V/ 4 5.00A/ 0.0s 20.00ms/ Auto 3 1.64V
Agilent
Acquisition Normal 2.50MSa/s
Channels DC 10.0:1 DC 10.0:1 DC 10.0:1 DC 10.0:1
Save to file = scope_178
Save
Recall
Default/Erase
Press to Save
The feedback voltage sensing value output with PWMDAC or DAC128S
CH1 (Yellow) - Phase A voltage
CH2 (Green) - Phase B voltage
CH3 (Purple) - Phase C voltage
The phase current capture by a current probe of oscilloscope
SVM is MINIMUM mode

Figure 3-37. Build Level 2: Motor Phase Voltage Waveforms Using Minimum SVM Mode

Use DAC128S085EVM with an oscilloscope to monitor the rotor angle of the motor from the angle generator and the angle from the FAST estimator as shown in Figure 3-38.

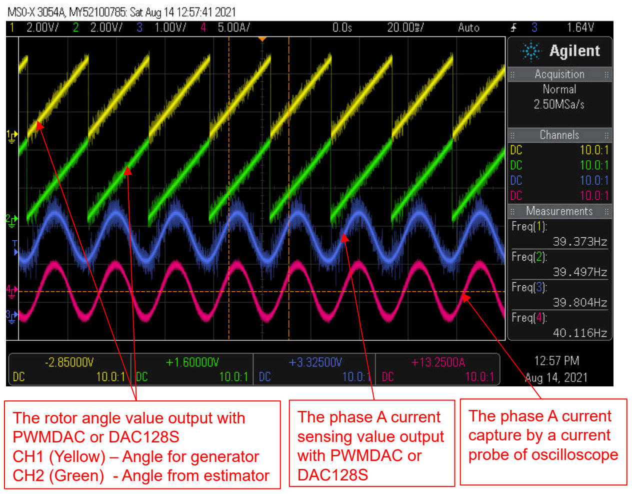

MSO-X 3054A, MY52100785: Sat Aug 14 12:57:41 2021
1 2.00V/ 2 2.00V/ 3 1.00V/ 4 5.00A/ 0.0s 20.00ms/ Auto 3 1.64V
Agilent
Acquisition Normal 2.50MSa/s
Channels DC 10.0:1 DC 10.0:1 DC 10.0:1 DC 10.0:1
Measurements
Freq(1): 39.373Hz
Freq(2): 39.497Hz
Freq(3): 39.804Hz
Freq(4): 40.116Hz
-2.85000V DC 10.0:1
+1.60000V DC 10.0:1
+3.32500V DC 10.0:1
+13.2500A DC 10.0:1
12:57 PM Aug 14, 2021
The rotor angle value output with PWMDAC or DAC128S
CH1 (Yellow) – Angle for generator
CH2 (Green) - Angle from estimator
The phase A current sensing value output with PWMDAC or DAC128S
The phase A current capture by a current probe of oscilloscope

Figure 3-38. Build Level 2: Motor Rotor Angle and Phase Current Waveforms

Use Datalog with Graph Tool to monitor three phase sensing current of the motor as shown in Figure 3-39.

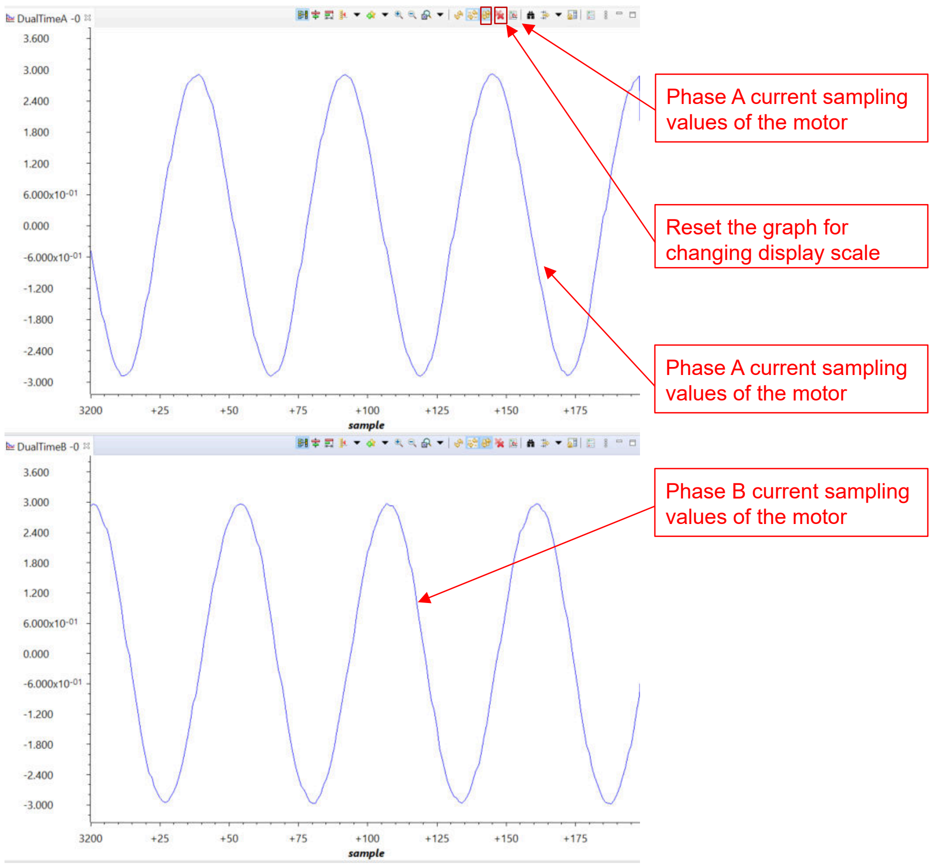

DualTimeA -0
DualTimeB -0
3.600
3.000
2.400
1.800
1.200
6.000x10⁻⁰¹
0.000
-6.000x10⁻⁰¹
-1.200
-1.800
-2.400
-3.000
3200
+25
+50
+75
+100
+125
+150
+175
sample
Phase A current sampling values of the motor
Reset the graph for changing display scale
Phase A current sampling values of the motor
Phase B current sampling values of the motor

Figure 3-39. Build Level 2: Motor Phase Current Waveforms With Graph Tool

Use Datalog with Graph Tool to monitor three phase sensing voltage of the motor as shown in Figure 3-40.

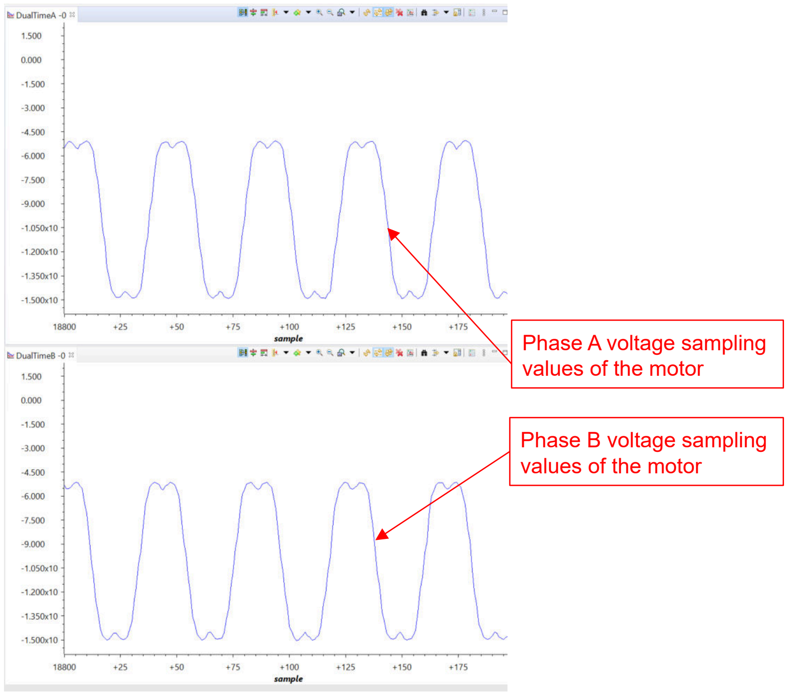


Figure 3-40. Build Level 2: Motor Phase Voltage Waveforms With Graph Tool

Use Datalog with Graph Tool to monitor rotor angle of the motor from the angle generator and angle from the FAST estimator as shown in Figure 3-41.

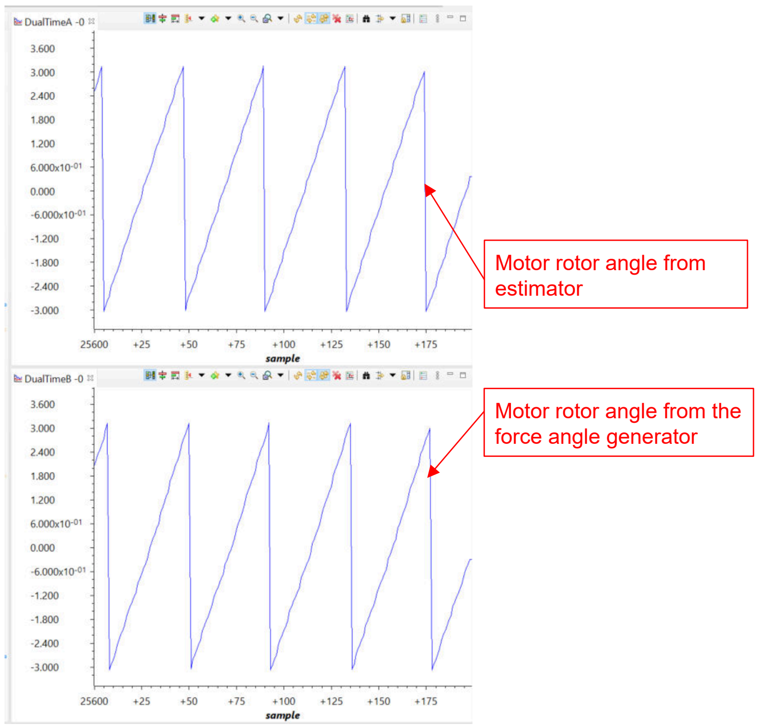


Figure 3-41. Build Level 2: Motor Rotor Angle Waveforms With Graph Tool

3.5.3 Level 3 Incremental Build

Objectives learned in this build level:

- Evaluate the closed current loop operation of the motor.
- Verify the current sensing parameters settings

In this build level, the motor is controlled by using i/f control that the rotor angle is generated from ramp generator module. The software flow for this build level is shown in Figure 3-42.

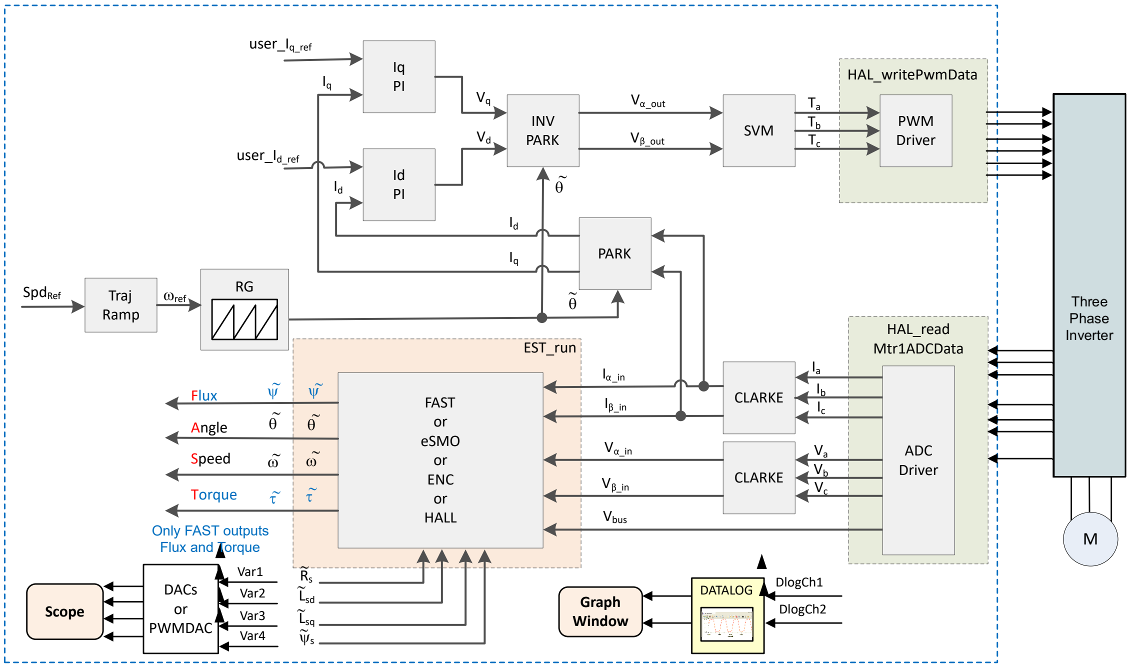


Figure 3-42. Build Level 3 Software Block Diagram - Current Close Loop Control

3.5.3.1 Build and Load Project

Connect the motor to the related terminals on the power inverter board. Follow the operation steps in Section 3.5.1.1 to build and load project by setting DMC_BUILDLEVEL to DMC_LEVEL_3 in the *sys_settings.h* file.

CAUTION
Do not click *Cancel*, turn off the power of the board, or disconnect the emulator when loading the code to flash.

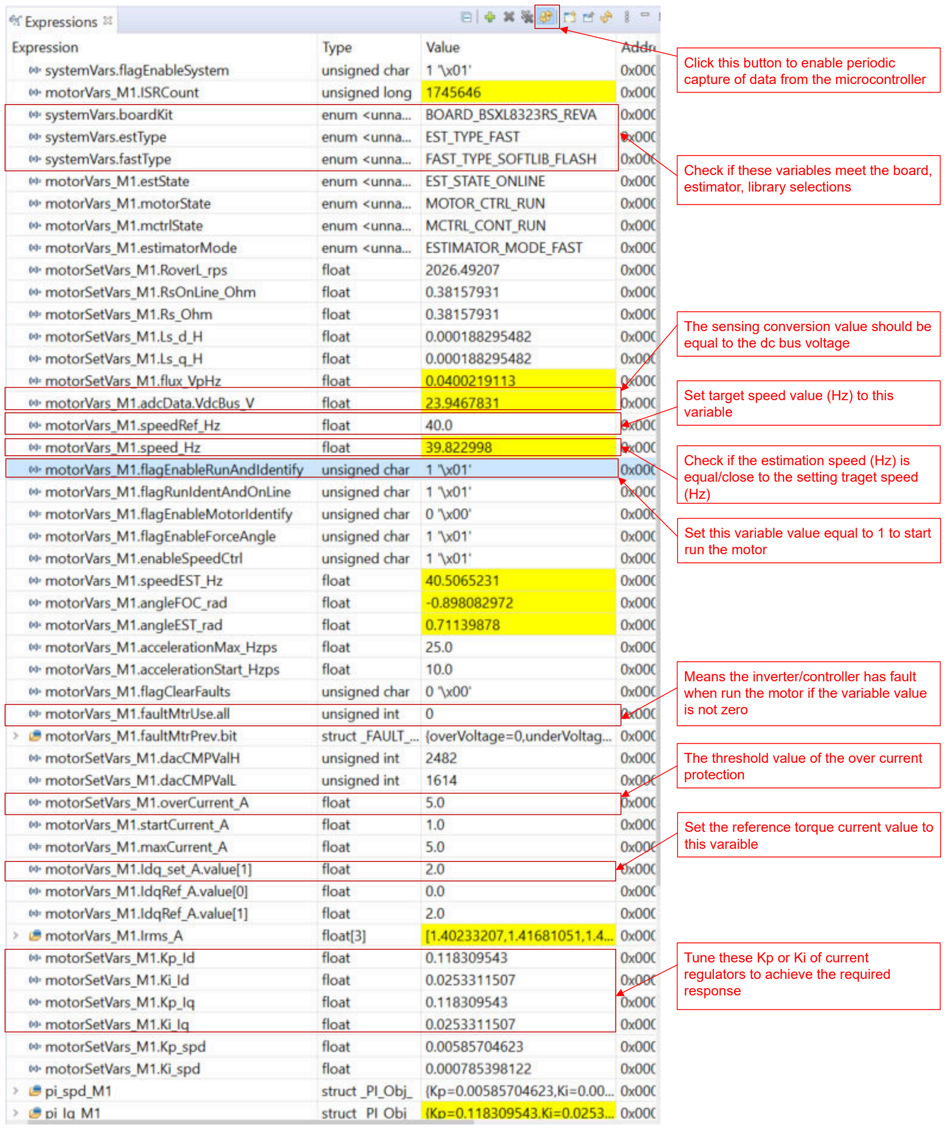
3.5.3.2 Setup Debug Environment Windows

Follow operation steps in Section 3.5.1.2 to import the variables into the Expressions window by picking *universal_lab_level3.txt*. The Expressions window appears as shown in Figure 3-43.

3.5.3.3 Run the Code

1. Power on the AC or DC power supply, gradually increase output voltage at power supply to get an appropriate DC-bus voltage.
2. Run the project by clicking on button , or click *Run* → *Resume* in the Debug tab. The *systemVars.flagEnableSystem* should be set to '*1*' after a fixed time, that means the offsets calibration have been done. The fault flags *motorVars_M1.faultMtrUse.all* should be equal to '*0*' , if not, the user have to check the current and voltage sensing circuit as described in Section 3.5.1.
3. To verify run the motor with current closed-loop control, set the variable *motorVars_M1.flagEnableRunAndIdentify* to '*1*' in the Expressions window as shown in Figure 3-43. The motor will run with a closed-loop control using the angle from the angle generator at a setting speed in the variable *motorVars_M1.speedRef_Hz.* Check the value of *motorVars_M1.speed_Hz* in Expressions window, The values of both variables should be very close.

4. Connect oscilloscope probes to the EPWMDAC or DAC128S outputs and motor phase line to probe the angles and current signals, and current. These waveforms on the oscilloscope appear as shown in Figure 3-44. Change the *motorVars_M1.Idq_set_A.value[1]* in the Expressions window to set the reference torque current, the motor phase current will be increasing with the same percentage accordingly.
5. If the motor cannot run with current-closed loop control and an over current fault appears, check if the sign of *motorVars_M1.adcData.current_sf* and the value of *userParams_M1.current_sf* are set correctly according to the hardware board. The values of both variables are related to the definition constant USER_M1_ADC_FULL_SCALE_CURRENT_A in the *user_mtr1.h* file.
6. Set the variables *motorVars_M1.flagEnableRunAndIdentify* to 0 to stop run the motor.
7. Once complete, the controller can now be halted and the debug connection terminated. Fully halting the controller by first clicking the Halt button on the toolbar or by clicking *Target → Halt*. Finally, reset the controller by clicking on or clicking *Run → Reset*.
8. Close CCS debug session by clicking on Terminate Debug Session or clicking *Run → Terminate*.



Expression	Type	Value	Addr
systemVars.flagEnableSystem	unsigned char	1 '\x01'	0x00
motorVars_M1.ISRCount	unsigned long	1745646	0x00
systemVars.boardKit	enum <unna...	BOARD_BSXL8323RS_REVA	0x00
systemVars.estType	enum <unna...	EST_TYPE_FAST	0x00
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH	0x00
motorVars_M1.estState	enum <unna...	EST_STATE_ONLINE	0x00
motorVars_M1.motorState	enum <unna...	MOTOR_CTRL_RUN	0x00
motorVars_M1.mctrlState	enum <unna...	MCTRL_CONT_RUN	0x00
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST	0x00
motorSetVars_M1.RoverL_rps	float	2026.49207	0x00
motorSetVars_M1.RsOnLine_Ohm	float	0.38157931	0x00
motorSetVars_M1.Rs_Ohm	float	0.38157931	0x00
motorSetVars_M1.Ls_d_H	float	0.000188295482	0x00
motorSetVars_M1.Ls_q_H	float	0.000188295482	0x00
motorSetVars_M1.flux_VpHz	float	0.0400219113	0x00
motorVars_M1.adcData.VdcBus_V	float	23.9467831	0x00
motorVars_M1.speedRef_Hz	float	40.0	0x00
motorVars_M1.speed_Hz	float	39.822998	0x00
motorVars_M1.flagEnableRunAndIdentify	unsigned char	1 '\x01'	0x00
motorVars_M1.flagRunIdentAndOnLine	unsigned char	1 '\x01'	0x00
motorVars_M1.flagEnableMotorIdentify	unsigned char	0 '\x00'	0x00
motorVars_M1.flagEnableForceAngle	unsigned char	1 '\x01'	0x00
motorVars_M1.enableSpeedCtrl	unsigned char	1 '\x01'	0x00
motorVars_M1.speedEST_Hz	float	40.5065231	0x00
motorVars_M1.angleFOC_rad	float	-0.898082972	0x00
motorVars_M1.angleEST_rad	float	0.71139878	0x00
motorVars_M1.accelerationMax_Hzps	float	25.0	0x00
motorVars_M1.accelerationStart_Hzps	float	10.0	0x00
motorVars_M1.flagClearFaults	unsigned char	0 '\x00'	0x00
motorVars_M1.faultMtrUse.all	unsigned int	0	0x00
motorVars_M1.faultMtrPrev.bit	struct _FAULT_...	{overVoltage=0,underVoltag...	0x00
motorSetVars_M1.dacCMPValH	unsigned int	2482	0x00
motorSetVars_M1.dacCMPValL	unsigned int	1614	0x00
motorSetVars_M1.overCurrent_A	float	5.0	0x00
motorSetVars_M1.startCurrent_A	float	1.0	0x00
motorSetVars_M1.maxCurrent_A	float	5.0	0x00
motorVars_M1.Idq_set_A.value[1]	float	2.0	0x00
motorVars_M1.IdqRef_A.value[0]	float	0.0	0x00
motorVars_M1.IdqRef_A.value[1]	float	2.0	0x00
motorVars_M1.Irms_A	float[3]	[1.40233207,1.41681051,1.4...	0x00
motorSetVars_M1.Kp_Id	float	0.118309543	0x00
motorSetVars_M1.Ki_Id	float	0.0253311507	0x00
motorSetVars_M1.Kp_Iq	float	0.118309543	0x00
motorSetVars_M1.Ki_Iq	float	0.0253311507	0x00
motorSetVars_M1.Kp_spd	float	0.00585704623	0x00
motorSetVars_M1.Ki_spd	float	0.000785398122	0x00
pi_spd_M1	struct _PI_Obj_	{Kp=0.00585704623,Ki=0.00...	0x00
pi_Iq_M1	struct _PI_Obj	{Kp=0.118309543,Ki=0.0253...	0x00

Click this button to enable periodic capture of data from the microcontroller

Check if these variables meet the board, estimator, library selections

The sensing conversion value should be equal to the dc bus voltage

Set target speed value (Hz) to this variable

Check if the estimation speed (Hz) is equal/close to the setting traget speed (Hz)

Set this variable value equal to 1 to start run the motor

Means the inverter/controller has fault when run the motor if the variable value is not zero

The threshold value of the over current protection

Set the reference torque current value to this varaible

Tune these Kp or Ki of current regulators to achieve the required response

Figure 3-43. Build Level 3: Variables in Expressions Window

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the angle generator and rotor angle from the FAST estimator, and a phase current of the motor as shown in Figure 3-44.

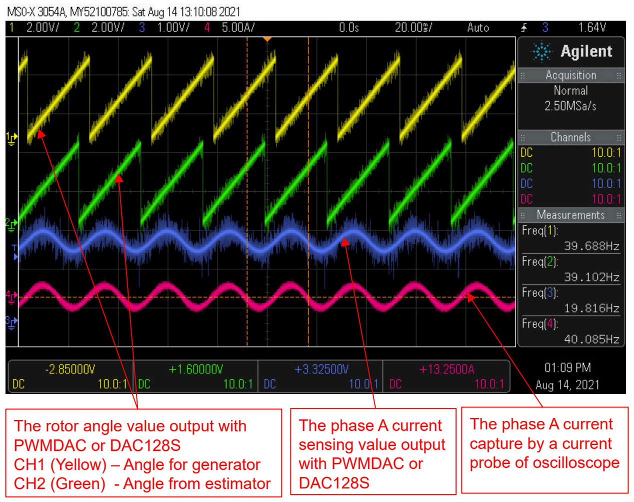

MSO-X 3054A, MY52100785: Sat Aug 14 13:10:08 2021
1 2.00V/ 2 2.00V/ 3 1.00V/ 4 5.00A/
0.0s 20.00ms/ Auto 3 1.64V
Agilent
Acquisition
Normal
2.50MSa/s
Channels
DC 10.0:1
Measurements
Freq(1): 39.688Hz
Freq(2): 39.102Hz
Freq(3): 19.816Hz
Freq(4): 40.085Hz
-2.85000V
+1.60000V
+3.32500V
+13.2500A
01:09 PM Aug 14, 2021
The rotor angle value output with PWMDAC or DAC128S
CH1 (Yellow) – Angle for generator
CH2 (Green) - Angle from estimator
The phase A current sensing value output with PWMDAC or DAC128S
The phase A current capture by a current probe of oscilloscope

Figure 3-44. Build Level 3: Motor Rotor Angle and Phase Current Waveforms on Oscilloscope

3.5.4 Level 4 Incremental Build

Objectives learned in this build level:

- Evaluate motor identification with FAST estimators.
- Evaluate the complete motor drive with Fast based sensorless-FOC, eSMO based sensorless-FOC, encoder based sensored-FOC or hall based sensored-FOC.
- Evaluate the additional features, such as field weakening control, flying start, MTPA, and braking.

In this build level, the outer speed loop is closed with the inner current loop for motor that the rotor angle is from FAST, eSMO, encoder or Hall sensors modules. The software flow for this build level is shown in Figure 3-45.

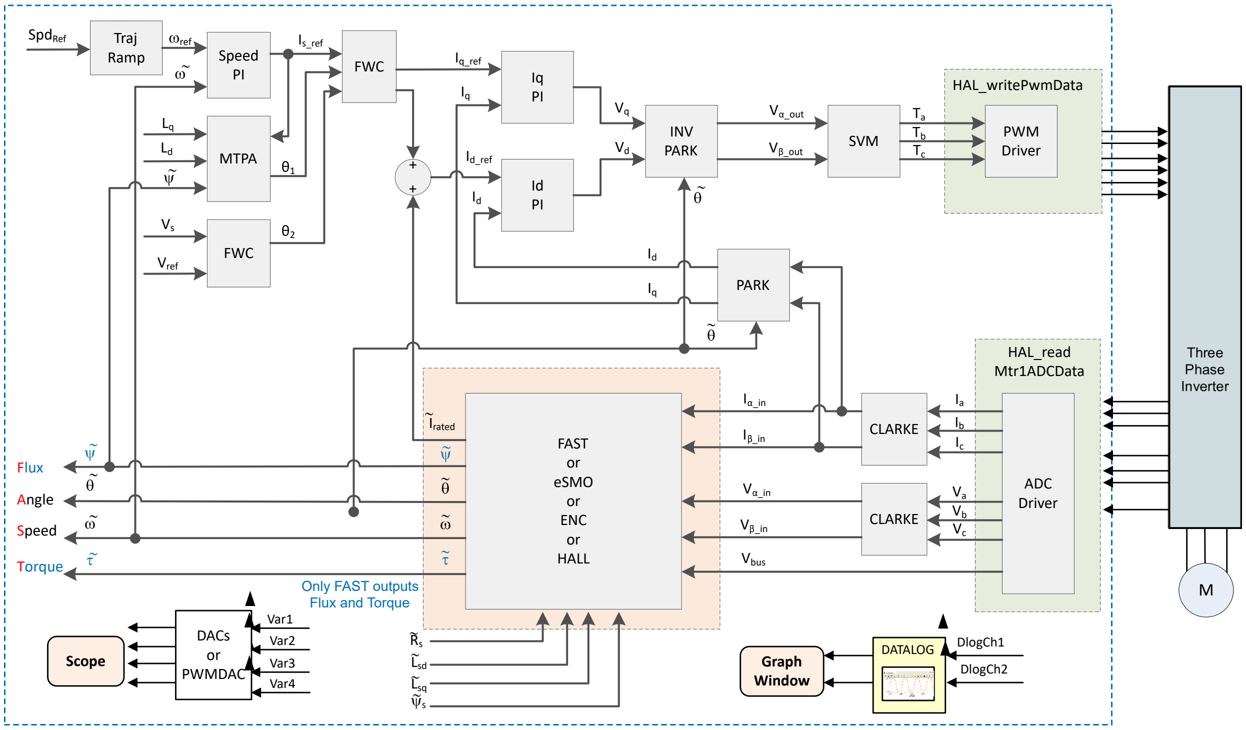


Figure 3-45. Build Level 4 Software Block Diagram - Speed and Current Close Loop Control

3.5.4.1 Build and Load Project

Connect the motor to the related terminals on the power inverter board. Follow the operation steps in Section 3.5.1.1 to build and load project by setting DMC_BUILDLEVEL to DMC_LEVEL_4 in the *sys_settings.h* file.

CAUTION
Do not click *Cancel*, turn off the power of the board, or disconnect the emulator when loading the code to flash.

3.5.4.2 Setup Debug Environment Windows

Follow operation steps in Section 3.5.1.2 to import the variables into the Expressions window by picking *universal_lab_level4.txt*. The Expressions window appears as shown in Figure 3-46.

3.5.4.3 Run the Code

1. Set the AC source output to 0 V at 50/60Hz, turn on the AC power supply, slowly increase the input voltage from 0-V to 220-V AC.
2. The required motor parameters must be defined in the *user_mtr1.h* header file as shown in the following example code. If the motor parameters are not well know by the user, the motor identification can be used to achieve the motor parameters if the FAST estimator is implemented in the example lab.

```
#define USER_MOTOR1_TYPE MOTOR_TYPE_PM
#define USER_MOTOR1_NUM_POLE_PAIRS (4)
#define USER_MOTOR1_Rr_Ohm (NULL)
#define USER_MOTOR1_Rs_Ohm (0.38157931f)
#define USER_MOTOR1_Ls_d_H (0.000188295482f)
#define USER_MOTOR1_Ls_q_H (0.000188295482f)
#define USER_MOTOR1_RATED_FLUX_VpHz (0.0396642499f)
```

3. Changes the *userParams.flag_bypassMotorId* value to "false"in the *sys_main.c* file to enable the motor identification as the following example code.

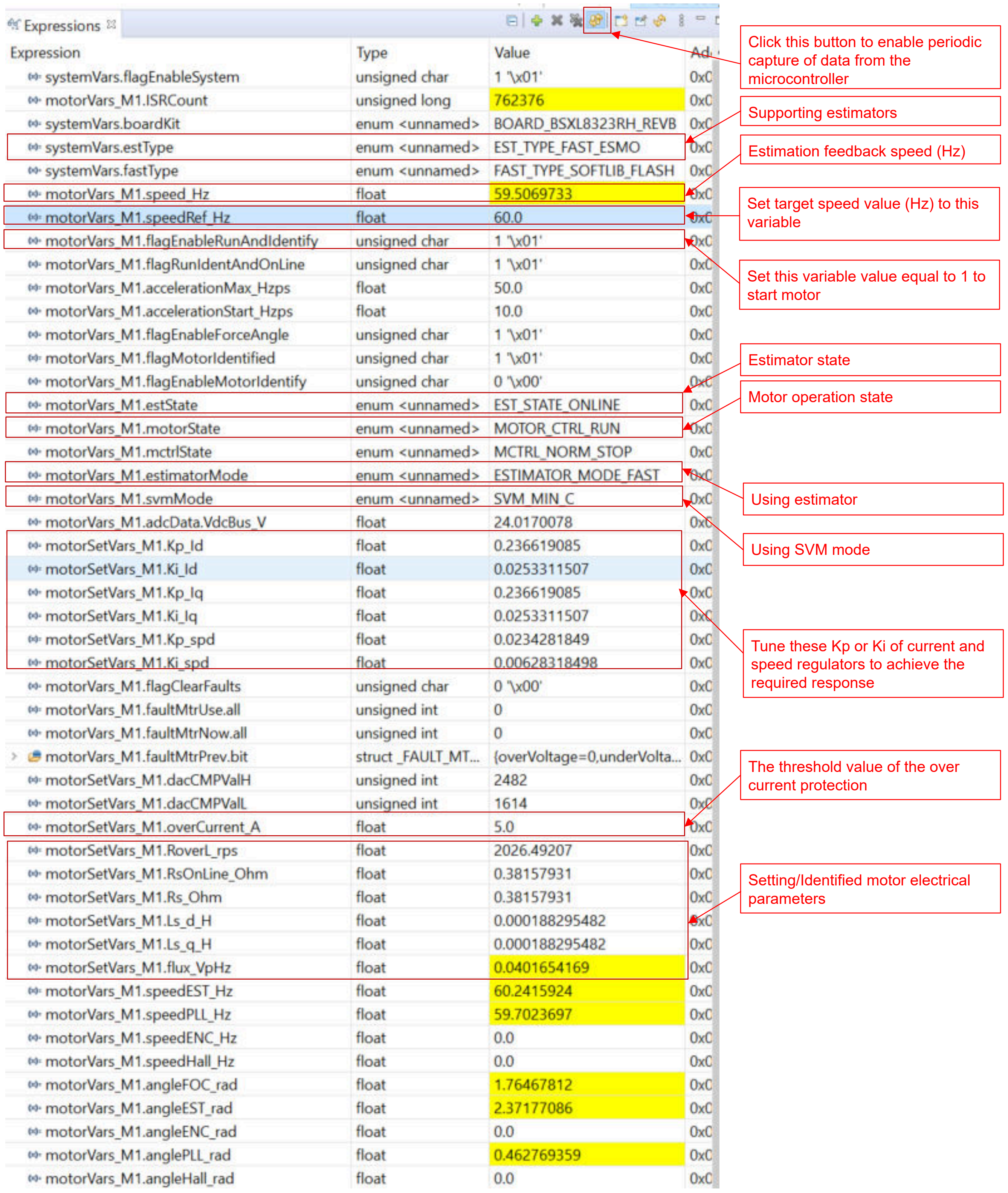
```
// false->enable identification, true->disable identification
    userParams_M1.flag_bypassMotorId = false;
```

4. Set the right identification variables value in the *user_mtr1.h* file according to the motor specification.

```
#define USER_MOTOR1_RES_EST_CURRENT_A       (1.5f)      // A - 10~30% of rated current of the 
motor
#define USER_MOTOR1_IND_EST_CURRENT_A       (-1.0f)     // A - 10~30% of rated current of the 
motor, just enough to enable rotation
#define USER_MOTOR1_MAX_CURRENT_A           (4.5f)      // A - 30~150% of rated current of the 
motor
#define USER_MOTOR1_FLUX_EXC_FREQ_Hz        (40.0f)     // Hz - 10~30% of rated frequency of 
the motor
```

5. Rebuild the project and load the code into the controller, run the project by clicking on button ▶, or click *Run → Resume* in the Debug tab. The *systemVars.flagEnableSystem* should be set to *1* after a fixed time, that means the offsets calibration have been done and the power relay for inrush is turned on. The fault flags *motorVars_M1.faultMtrUse.all* should be equal to '*0*' , if not, the user should check the current and voltage sensing circuit as described in Section 3.5.1.
6. Set the variable *motorVars_M1.flagEnableRunAndIdentify* to' *1* in the Expressions window as shown in Figure 3-46, the motor identification will be executed, the whole process will take about 150s. Once *motorVars_M1.flagEnableRunAndIdentify* is equal to *0* and the motor is stopping, the motor parameters have been identified. Copy the variables value in the watch window to replace the defined motor parameters in the *user_mtr1.h* file as follows:
   - USER_MOTOR1_Rs_Ohm = motorSetVars_M1.Rs_Ohm's value
   - USER_MOTOR1_Ls_d_H = motorSetVars_M1.Ls_d_H's value
   - USER_MOTOR1_Ls_q_H = motorSetVars_M1.Ls_q_H's value
   - USER_MOTOR1_RATED_FLUX_VpHz = motorSetVars_M1.flux_VpHz's value
7. Set *userParams_M1.flag_bypassMotorId* value to 'true' to disable identification after successfully identify the motors parameters, rebuild the project and load the code into the controller.

8. The example can support online identify the motor without reloading the code as the following steps.
   a. Set the *motorVars_M1.flagEnableRunAndIdentify* to' *0*' to stop run the motor.
   b. Set the *motorVars_M1.flagEnableMotorIdentify* to '1' to enable identification.
   c. Set the *motorVars_M1.flagEnableRunAndIdentify* to' *1*' to start identify motor parameters. The *motorVars_M1.flagEnableMotorIdentify* will be set to '0' automatically that means the identification is in processing.
   d. As described in the step 6 above, the new motor parameters will be identified.
9. Once complete motor parameters identification or set the correct the motor parameters in the *user_mtr1.h* file. To start run the motor as the following steps.
   a. Set the variables *motorVars_M1.flagEnableRunAndIdentify* equal to *1* again to start run the motor.
   b. Set the target speed value to the variable *motorVars_M1.speedRef_Hz* and watch how the motor shaft speed will follow the setting speed.
   c. To change the acceleration, enter a different acceleration value for the variable *motorVars_M1.accelerationMax_Hzps*.
   d. Use PWMDAC or DAC128S module to display the monitoring variables as described in Section 3.4.2 or Section 3.4.3. The motor angle and current waveforms are shown in Figure 3-47.
10. The default proportional gain (Kp) and integral gain (Ki) for the current controllers of the FOC system are calculated in the function *setupControllers()*. After *setupControllers*() is called, the global variables motorSetVars_M1.Kp_Id, motorSetVars_M1.Ki_Id, motorSetVars_M1.Kp_Iq, and motorSetVars_M1.Ki_Iq are initialized with the newly calculated Kp and Ki gains. Tune the Kp and Ki value of these four variables in Expressions Watch Window as shown in Figure 3-46 for the current controllers to achieve the expected current control bandwidth and response. The Kp gain creates a zero that cancels the pole of the motor's stator and can easily be calculated. The Ki gain adjusts the bandwidth of the current controller-motor system. When a speed controlled system is needed for a certain damping, the Kp gain of the current controller will relate to the time constant of the speed controlled system.
11. The default proportional gain (Kp) and integral gain (Ki) for the speed controllers of the FOC system are also calculated in the function *setupControllers()*. After *setupControllers*() is called, the global variables motorSetVars_M1.Kp_spd and motorSetVars_M1.Ki_spd are initialized with the newly calculated Kp and Ki gains. Tune the Kp and Ki value of these two variables in Expressions Watch Window as shown in Figure 3-46 for the speed controllers to achieve the expected current control bandwidth and response. Tuning the speed controller has more unknowns than when tuning a current controller, the default calculated Kp and Ki is just a reference value as a starting point.
12. Set the variables *motorVars_M1.flagEnableRunAndIdentify* to '0' to stop run the motor.
13. Once complete, the controller can now be halted and the debug connection terminated. Fully halting the controller by first clicking the Halt button on the toolbar or by clicking *Target* → *Halt*. Finally, reset the controller by clicking on or clicking *Run* → *Reset*.
14. Close CCS debug session by clicking on Terminate Debug Session or clicking *Run* → *Terminate*.


Expressions
Click this button to enable periodic capture of data from the microcontroller
Supporting estimators
Estimation feedback speed (Hz)
Set target speed value (Hz) to this variable
Set this variable value equal to 1 to start motor
Estimator state
Motor operation state
Using estimator
Using SVM mode
Tune these Kp or Ki of current and speed regulators to achieve the required response
The threshold value of the over current protection
Setting/Identified motor electrical parameters

Expression	Type	Value	Ad
systemVars.flagEnableSystem	unsigned char	1 '\x01'	0xC
motorVars_M1.ISRCount	unsigned long	762376	0xC
systemVars.boardKit	enum <unnamed>	BOARD_BSXL8323RH_REVB	0xC
systemVars.estType	enum <unnamed>	EST_TYPE_FAST_ESMO	0xC
systemVars.fastType	enum <unnamed>	FAST_TYPE_SOFTLIB_FLASH	0xC
motorVars_M1.speed_Hz	float	59.5069733	0xC
motorVars_M1.speedRef_Hz	float	60.0	0xC
motorVars_M1.flagEnableRunAndIdentify	unsigned char	1 '\x01'	0xC
motorVars_M1.flagRunIdentAndOnLine	unsigned char	1 '\x01'	0xC
motorVars_M1.accelerationMax_Hzps	float	50.0	0xC
motorVars_M1.accelerationStart_Hzps	float	10.0	0xC
motorVars_M1.flagEnableForceAngle	unsigned char	1 '\x01'	0xC
motorVars_M1.flagMotorIdentified	unsigned char	1 '\x01'	0xC
motorVars_M1.flagEnableMotorIdentify	unsigned char	0 '\x00'	0xC
motorVars_M1.estState	enum <unnamed>	EST_STATE_ONLINE	0xC
motorVars_M1.motorState	enum <unnamed>	MOTOR_CTRL_RUN	0xC
motorVars_M1.mctrlState	enum <unnamed>	MCTRL_NORM_STOP	0xC
motorVars_M1.estimatorMode	enum <unnamed>	ESTIMATOR_MODE_FAST	0xC
motorVars_M1.svmMode	enum <unnamed>	SVM_MIN_C	0xC
motorVars_M1.adcData.VdcBus_V	float	24.0170078	0xC
motorSetVars_M1.Kp_Id	float	0.236619085	0xC
motorSetVars_M1.Ki_Id	float	0.0253311507	0xC
motorSetVars_M1.Kp_Iq	float	0.236619085	0xC
motorSetVars_M1.Ki_Iq	float	0.0253311507	0xC
motorSetVars_M1.Kp_spd	float	0.0234281849	0xC
motorSetVars_M1.Ki_spd	float	0.00628318498	0xC
motorVars_M1.flagClearFaults	unsigned char	0 '\x00'	0xC
motorVars_M1.faultMtrUse.all	unsigned int	0	0xC
motorVars_M1.faultMtrNow.all	unsigned int	0	0xC
motorVars_M1.faultMtrPrev.bit	struct _FAULT_MT...	{overVoltage=0,underVolta...	0xC
motorSetVars_M1.dacCMPValH	unsigned int	2482	0xC
motorSetVars_M1.dacCMPValL	unsigned int	1614	0xC
motorSetVars_M1.overCurrent_A	float	5.0	0xC
motorSetVars_M1.RoverL_rps	float	2026.49207	0xC
motorSetVars_M1.RsOnLine_Ohm	float	0.38157931	0xC
motorSetVars_M1.Rs_Ohm	float	0.38157931	0xC
motorSetVars_M1.Ls_d_H	float	0.000188295482	0xC
motorSetVars_M1.Ls_q_H	float	0.000188295482	0xC
motorSetVars_M1.flux_VpHz	float	0.0401654169	0xC
motorVars_M1.speedEST_Hz	float	60.2415924	0xC
motorVars_M1.speedPLL_Hz	float	59.7023697	0xC
motorVars_M1.speedENC_Hz	float	0.0	0xC
motorVars_M1.speedHall_Hz	float	0.0	0xC
motorVars_M1.angleFOC_rad	float	1.76467812	0xC
motorVars_M1.angleEST_rad	float	2.37177086	0xC
motorVars_M1.angleENC_rad	float	0.0	0xC
motorVars_M1.anglePLL_rad	float	0.462769359	0xC
motorVars_M1.angleHall_rad	float	0.0	0xC

Figure 3-46. Build Level 4: Variables in Expressions Window

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST estimator, feedback speed of the motor, and a phase current of the motor as shown in Figure 3-47 when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

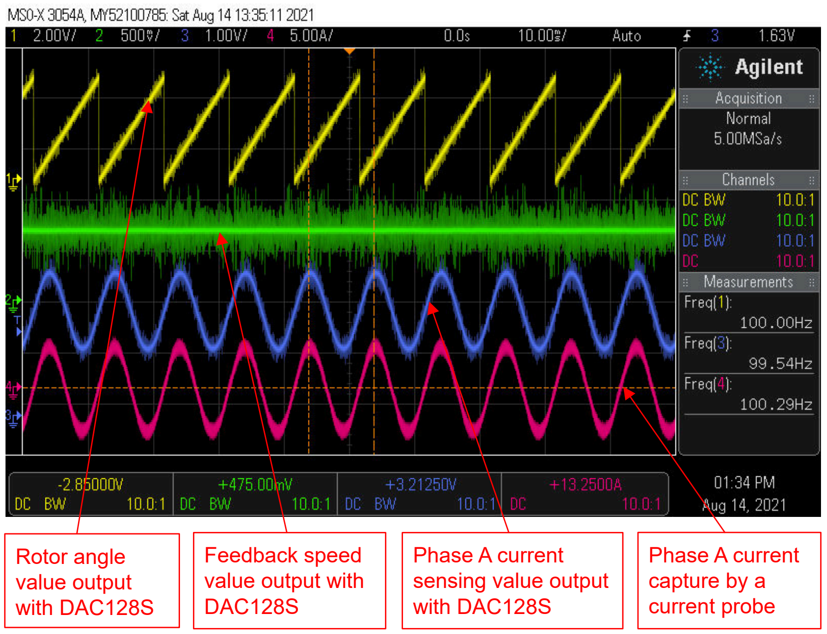


Figure 3-47. Build Level 4: Rotor Angle with FAST, Phase Current Waveforms at Forward Move

As illustrated in Section 3.3.2, multiple FOC algorithms can be supported in the example lab. The user can use one or two algorithm for motor control in the lab project as shown in Table 3-7.

The user can implement FAST and eSMO estimators in the project simultaneously by adding the pre-define name '*MOTOR1_FAST*' and '*MOTOR1_ESMO*' in project properties as described in Section 3.3.1. Rebuild, load and run the project as the operation steps above. The settings will be as shown in Figure 3-46.

- The systemVars.estType value equals to EST_TYPE_FAST_ESMO that means FAST and eSMO estimators are enabled in this project.
- The motorVars_M1.estimatorMode equals to ESTIMATOR_MODE_FAST that means the FAST estimator is using for sensorless-FOC, equals to ESTIMATOR_MODE_ESMO that means the eSMO estimator is using for sensorless-FOC.
- The estimated rotor angles from FAST and eSMO are shown in Figure 3-48. The motor is running with FAST at ***forward*** rotation by setting motorVars_M1.speedRef_Hz to a ***positive*** value.
- The estimated rotor angles from FAST and eSMO are shown in Figure 3-52. The motor is running with FAST at ***reversal*** rotation by setting motorVars_M1.speedRef_Hz to a ***negative*** value.
- The user can change the value to ESTIMATOR_MODE_ESMO to select the eSMO estimator for sensorless-FOC. And also the user can change the value to switch the using estimator on the fly.

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST and eSMO estimator, and a phase current of the motor as shown in Figure 3-48 when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

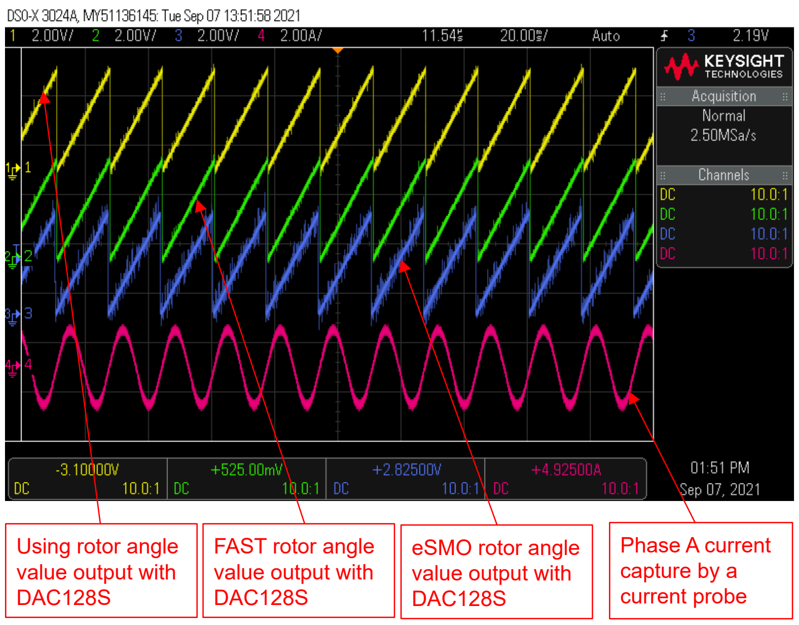

DSO-X 3024A, MY51136145: Tue Sep 07 13:51:58 2021
1 2.00V/ 2 2.00V/ 3 2.00V/ 4 2.00A/ 11.54s 20.00s/ Auto 3 2.19V
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.50MSa/s
Channels
-3.10000V
+525.00mV
+2.82500V
+4.92500A
01:51 PM Sep 07, 2021
Using rotor angle value output with DAC128S
FAST rotor angle value output with DAC128S
eSMO rotor angle value output with DAC128S
Phase A current capture by a current probe

Figure 3-48. Build Level 4: Rotor Angle with FAST and eSMO, Phase Current Waveforms at Forward Rotation

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST and eSMO estimator, and a phase current of the motor as shown in Figure 3-49 when the motor is running at reversal rotation by setting *motorVars_M1.speedRef_Hz* to a negative reference value.

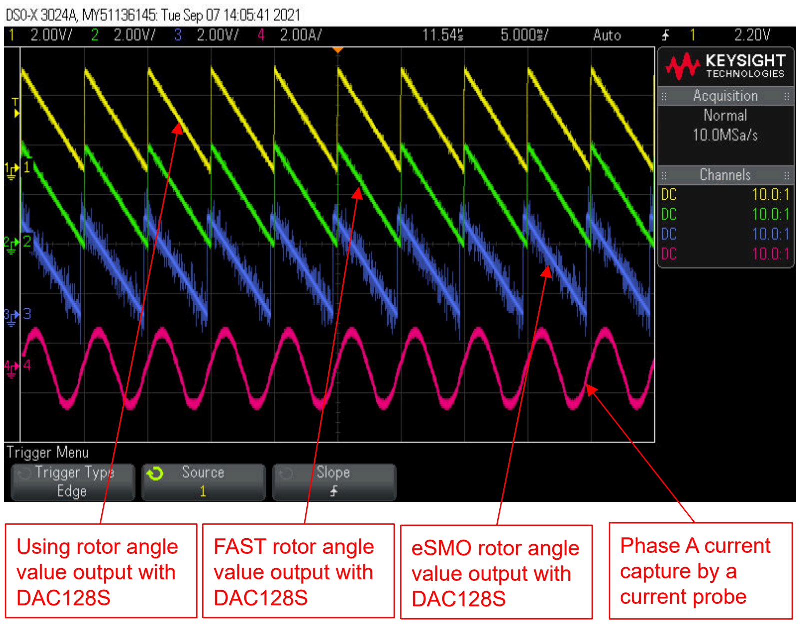

DSO-X 3024A, MY51136145: Tue Sep 07 14:05:41 2021
1 2.00V/ 2 2.00V/ 3 2.00V/ 4 2.00A/ 11.54s 5.000s/ Auto 1 2.20V
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
10.0MSa/s
Channels
Trigger Menu
Trigger Type Edge
Source 1
Slope
Using rotor angle value output with DAC128S
FAST rotor angle value output with DAC128S
eSMO rotor angle value output with DAC128S
Phase A current capture by a current probe

Figure 3-49. Build Level 4: Rotor Angle With FAST and eSMO, Phase Current Waveforms at Reversal Rotation

The user can implement FAST and Encoder estimators in the project simultaneously by adding the pre-define name '*MOTOR1_FAST*' and '*MOTOR1_ENC*' in project properties as described in Section 3.3.1. Rebuild, load and run the project as the operation steps above.

- The systemVars.estType value equals to EST_TYPE_FAST_ENC that means FAST and Encoder estimators are enabled in this project.
- The motorVars_M1.estimatorMode equals to ESTIMATOR_MODE_FAST that means the FAST estimator is using for sensorless-FOC, equals to ESTIMATOR_MODE_ENC that means the encoder estimator is using for sensored-FOC.
- The estimated rotor angles from FAST and Encoder are shown in Figure 3-50. The motor is running with FAST at forward rotation by setting motorVars_M1.speedRef_Hz to a positive value.
- The user can change the value to ESTIMATOR_MODE_ENC to select the Encoder estimator for sensored-FOC. And also the user can change the value to switch the using estimator on the fly.

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST estimator and encoder, and a phase current of the motor as shown in Figure 3-50 when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

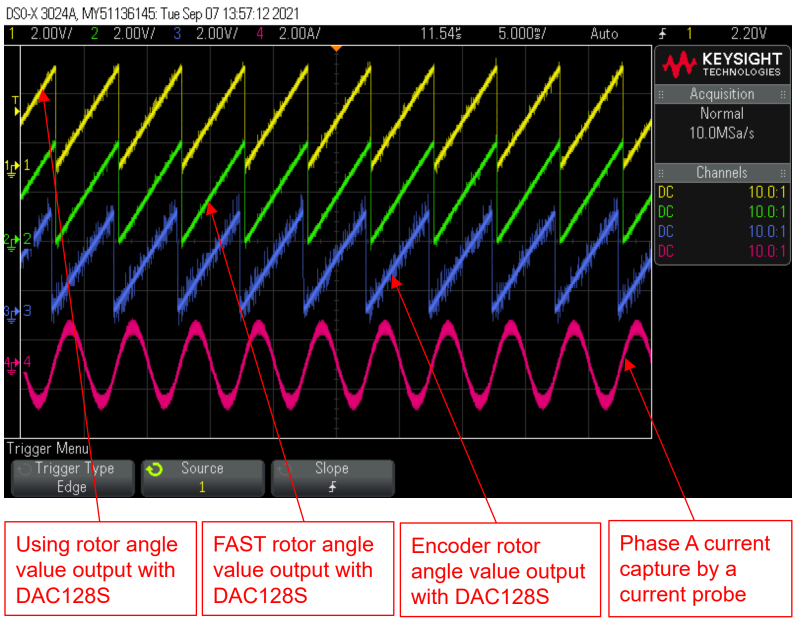


Figure 3-50. Build Level 4: Rotor Angle with FAST and Encoder, Phase Current Waveforms at Forward Rotation

The user can implement FAST and Hall sensor estimators in the project simultaneously by adding the pre-define name '*MOTOR1_FAST*' and '*MOTOR1_HALL*' in project properties as described in Section 3.3.1. Rebuild, load and run the project as the operation steps above.

- The systemVars.estType value equals to EST_TYPE_FAST_HALL that means FAST and Hall sensor estimators are enabled in this project.
- The motorVars_M1.estimatorMode equals to ESTIMATOR_MODE_FAST that means the FAST estimator is using for sensorless-FOC, equals to ESTIMATOR_MODE_HALL that means the Hall sensor estimator is using for sensored-FOC.
- The estimated rotor angles from FAST and Hall sensor are shown in Figure 3-51. The motor is running with FAST at forward rotation by setting motorVars_M1.speedRef_Hz to a positive value.
- The estimated rotor angles from FAST and Hall sensor are shown in Figure 3-51. The motor is running with Hall sensor at reversal rotation by setting motorVars_M1.speedRef_Hz to a negative value.
- The user can change the value to ESTIMATOR_MODE_HALL to select the Hall sensor estimator for sensored-FOC. And also the user can change the value to switch the using estimator on the fly.

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST estimator and Hall sensor, and a phase current of the motor as shown in Figure 3-51 when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

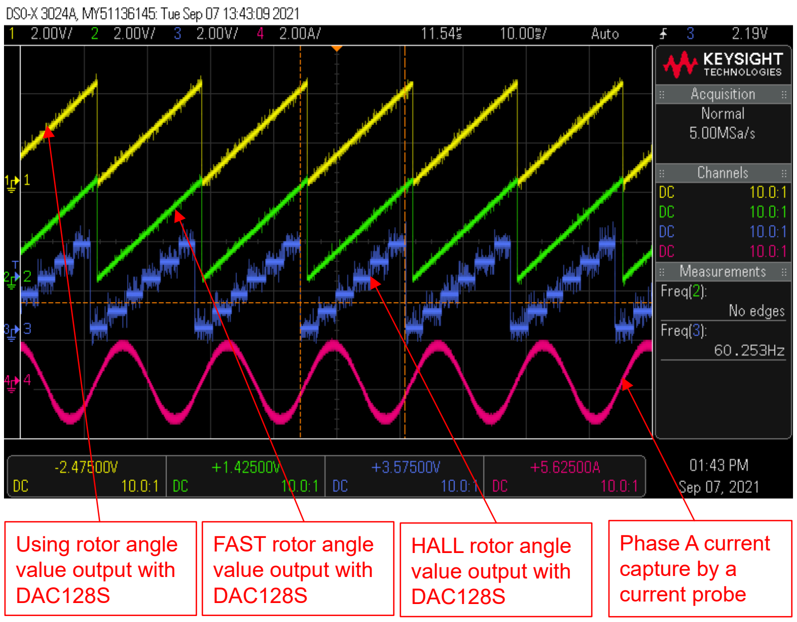


Figure 3-51. Build Level 4: Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Forward Rotation

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the FAST estimator and Hall sensor, and a phase current of the motor as shown in Figure 3-52 when the motor is running at reversal rotation by setting *motorVars_M1.speedRef_Hz* to a negative reference value.

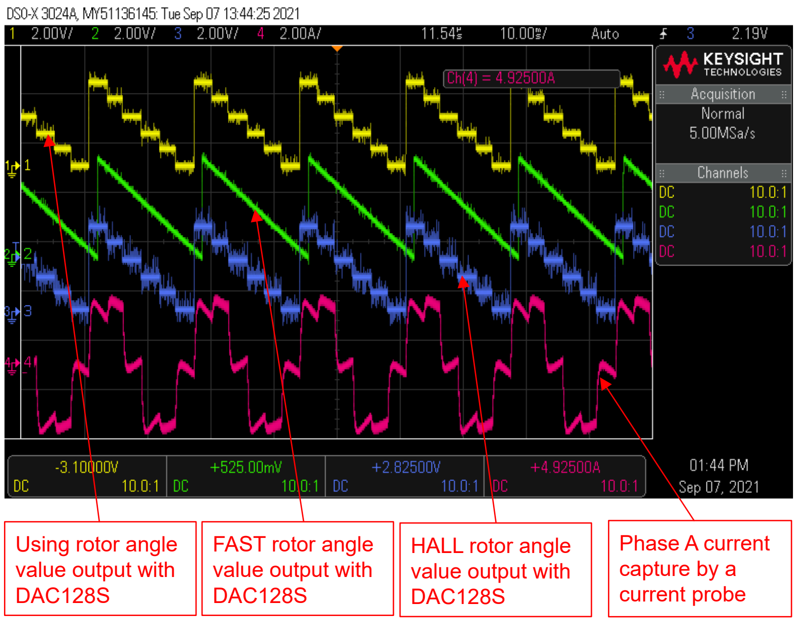


Figure 3-52. Build Level 4: Motor Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Reversal Rotation

The user can implement eSMO and Encoder estimators in the project simultaneously by adding the pre-define name '*MOTOR1_ESMO*' and '*MOTOR1_ENC*' in project properties as described in Section 3.3.1. Rebuild, load and run the project as the operation steps above.

- The systemVars.estType value equals to EST_TYPE_ESMO_ENC that means eSMO and Encoder estimators are enabled in this project.
- The motorVars_M1.estimatorMode equals to ESTIMATOR_MODE_ESMO that means the eSMO estimator is using for sensorless-FOC, equals to ESTIMATOR_MODE_ENC that means the encoder estimator is using for sensored-FOC.
- The estimated rotor angles from eSMO and Encoder are shown in Figure 3-53. The motor is running with eSMO at forward rotation by setting motorVars_M1.speedRef_Hz to a positive value.
- The user can change the value to ESTIMATOR_MODE_ENC to select the Encoder estimator for sensored-FOC. And also the user can change the value to switch the using estimator on the fly.

Use DAC128S085EVM with an oscilloscope to monitor rotor angle of the motor from the eSMO estimator and encoder, and a phase current of the motor as shown in Figure 3-53 when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

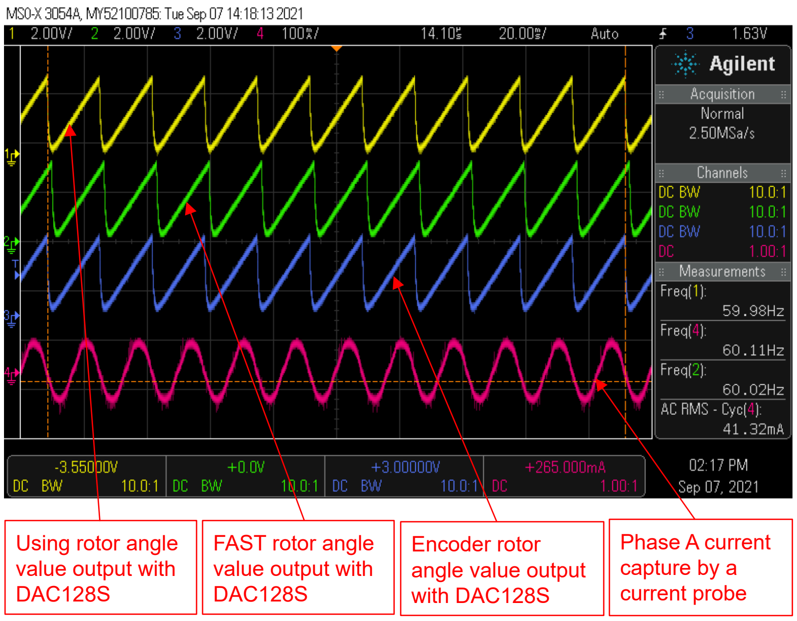

MSO-X 3054A, MY52100785: Tue Sep 07 14:18:13 2021
1 2.00V/ 2 2.00V/ 3 2.00V/ 4 100mA/
14.10s 20.00ms/ Auto 3 1.63V
Agilent
Acquisition
Normal
2.50MSa/s
Channels
DC BW 10.0:1
DC BW 10.0:1
DC BW 10.0:1
DC 1.00:1
Measurements
Freq(1): 59.98Hz
Freq(4): 60.11Hz
Freq(2): 60.02Hz
AC RMS - Cyc(4): 41.32mA
-3.55000V DC BW 10.0:1
+0.0V DC BW 10.0:1
+3.00000V DC BW 10.0:1
+265.000mA DC 1.00:1
02:17 PM Sep 07, 2021
Using rotor angle value output with DAC128S
FAST rotor angle value output with DAC128S
Encoder rotor angle value output with DAC128S
Phase A current capture by a current probe

Figure 3-53. Build Level 4: Rotor Angle with eSMO and Encoder, Phase Current Waveforms at Forward Rotation

4 Building a Custom Board

4.1 Building a New Custom Board

This section discusses how the user can design their own application board to drive a motor, and how to migrate the motor control SDK software for use with their own board.

4.1.1 Hardware Setup

If using a custom board, make sure that the power supply to the C2000 microcontroller and to the gate driver is correct, and that the JTAG emulator can be connected successfully. Modify the reference code to be compatible with the custom board as described in the following sections, and then run the code starting with build level 1 and working the way to build level 4 as illustrated in Section 3.5.

4.1.2 Migrating Reference Code to a Custom Board

To migrate the reference code to a new TI motor driver kit or to a custom board, the user needs to configure the hardware parameters and the motor control parameters in the *user_mtr1.h* file according to the motor driver circuit, and configure the relevant peripherals in the *hal.h* and *hal.c* files as described in the following sections.

The following block diagram summarizes the function calls that are used to configure the motor control settings and the C2000 MCU peripherals (Figure 4-1).

In this lab project, there are several HAL functions that are called only once, related to the configuration of the Hardware. All of these functions deal with the configuration of either a peripheral or of a motor driver IC.

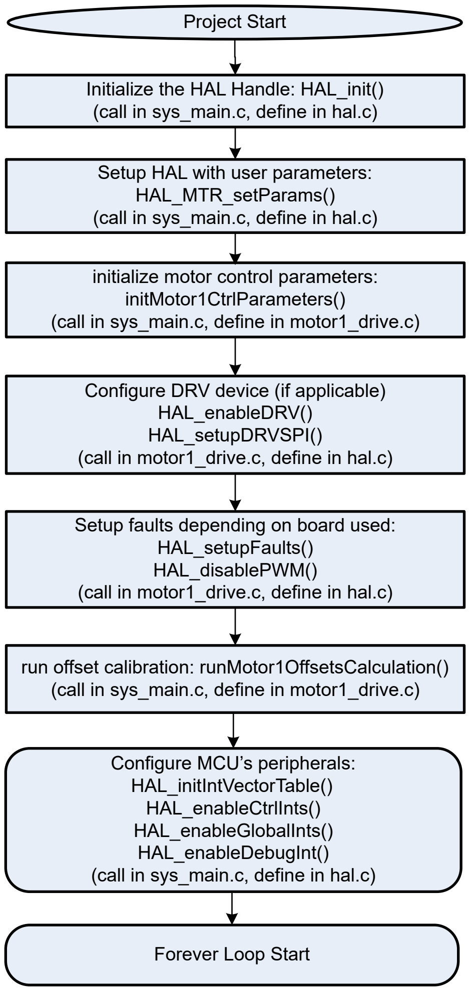

Project Start
Initialize the HAL Handle: HAL_init()
(call in sys_main.c, define in hal.c)
Setup HAL with user parameters:
HAL_MTR_setParams()
(call in sys_main.c, define in hal.c)
initialize motor control parameters:
initMotor1CtrlParameters()
(call in sys_main.c, define in motor1_drive.c)
Configure DRV device (if applicable)
HAL_enableDRV()
HAL_setupDRVSPI()
(call in motor1_drive.c, define in hal.c)
Setup faults depending on board used:
HAL_setupFaults()
HAL_disablePWM()
(call in motor1_drive.c, define in hal.c)
run offset calibration: runMotor1OffsetsCalculation()
(call in sys_main.c, define in motor1_drive.c)
Configure MCU's peripherals:
HAL_initIntVectorTable()
HAL_enableCtrlInts()
HAL_enableGlobalInts()
HAL_enableDebugInt()
(call in sys_main.c, define in hal.c)
Forever Loop Start

Figure 4-1. HAL Configuration and Motor Control Setting Block Diagram

4.1.2.1 Setting Hardware Board Parameters

The *user_mtr1.h* file is where all the user parameters are stored for motor control. The maximum phase current and phase voltage at the input to the AD converter are hardware dependent and should be based on the current and voltage sensing circuitry and scaling at the ADC input. The number of phase current sensors and phase voltage sensors are also defined in the *user_mtr1.h* file. These values are hardware-dependent.

All of the configurable parameters defined in the *user_mtr1.h* file can be calculated using the `Motor Control Parameters Calculation.xlsx` Excel® spreadsheet. This file is included with the project file in the folder: `\solutions\universal_motorcontrol_lab\doc`. Copy parameters marked in ***bold*** to the *user_mtr1.h* file as shown in the following code.

```
//! \brief Defines the maximum voltage at the AD converter
#define USER_M1_ADC_FULL_SCALE_VOLTAGE_V         (57.52845691f)

//! \brief Defines the analog voltage filter pole location, Hz
#define USER_M1_VOLTAGE_FILTER_POLE_Hz           (680.4839141f)     // 47nF

//! \brief Defines the maximum current at the AD converter
#define USER_M1_ADC_FULL_SCALE_CURRENT_A         (47.14285714f)     // gain=10
```

4.1.2.2 Modifying Motor Control Parameters

The parameters provided in the *user_mtr1.h* file for a PMSM/BLDC motor are listed as shown in the below code. The motor parameters can be identified either from the motor data sheet or through the FAST identification process, if the FAST module is utilized.

```
#define USER_MOTOR1_TYPE                    MOTOR_TYPE_PM
#define USER_MOTOR1_NUM_POLE_PAIRS          (4)

#define USER_MOTOR1_Rs_Ohm                  (0.38157931f)
#define USER_MOTOR1_Ls_d_H                  (0.000188295482f)
#define USER_MOTOR1_Ls_q_H                  (0.000188295482f)
#define USER_MOTOR1_RATED_FLUX_VpHz         (0.0396642499f)
#define USER_MOTOR1_MAX_CURRENT_A           (6.0f)
```

4.1.2.3 Changing Pin Assignment

The HAL_setupGPIOs() function located in the *hal.c* file configures the function of the GPIO pins and sets the direction and mode of the specified pin according to the hardware motor driver board/kit that is used. For modifying the code for a custom board, a TI motor driver EVM that doesn't currently have universal lab code support, or for use with a different C2000 MCU, these GPIO assignments will need to be changed to correspond properly with the motor driver board. Be careful to set the proper pad configuration for the specified pin, especially for GPIOs that will be used as PWM outputs. An example configuration of the EPWM1A GPIO0 pin is shown below.

```
// GPIO0->EPWM1A->M1_UH*
GPIO_setPinConfig(GPIO_0_EPWM1_A);
GPIO_setDirectionMode(0, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(0, GPIO_PIN_TYPE_STD);
```

4.1.2.4 Configuring the PWM Module

The HAL module configures the PWM channels. The base addresses of the PWM channels that are used for the motor controller PWM inputs are defined in the *hal.h* file, and the base addresses are assigned to the PWM handles in the *hal.c* file. The connection diagram for the PWM signals between the LAUNCHXL-F280025C and BOOSTXL-DRV8323RS is shown in Figure 4-2.

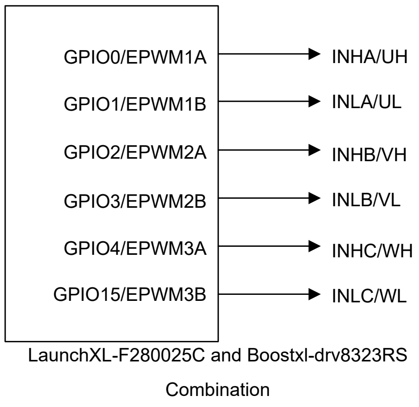


Figure 4-2. PWM Connection Diagram

The code to configure the PWM signals is shown below, taken from the *hal.h* and *hal.c* files that are located in the solutions\universal_motorcontrol_lab\f28002x\drivers\include and \source folder. Motor driver board dependent and MCU dependent changes are highlighted in **bold**.

1. The base addresses of the PWM modules are defined in the *hal.h* file as shown below.

```
//! \ Motor 1
#define MTR1_PWM_U_BASE         EPWM1_BASE
#define MTR1_PWM_V_BASE         EPWM2_BASE
#define MTR1_PWM_W_BASE         EPWM6_BASE
```

2. The GPIOs are set up as PWM outputs in the HAL_setupGpios() function located in the *hal.c* file.

```
// GPIO0->EPWM1A->M1_UH*
GPIO_setPinConfig(GPIO_0_EPWM1_A);
GPIO_setDirectionMode(0, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(0, GPIO_PIN_TYPE_STD);

// GPIO1->EPWM1B->M1_UL*
GPIO_setPinConfig(GPIO_1_EPWM1_B);
GPIO_setDirectionMode(1, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(1, GPIO_PIN_TYPE_STD);

// GPIO2->EPWM2A->M1_VH*
GPIO_setPinConfig(GPIO_2_EPWM2_A);
GPIO_setDirectionMode(2, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(2, GPIO_PIN_TYPE_STD);

// GPIO3->EPWM2B->M1_VL*
GPIO_setPinConfig(GPIO_3_EPWM2_B);
GPIO_setDirectionMode(3, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(3, GPIO_PIN_TYPE_STD);

// GPIO4->EPWM3A->M1_WH*
GPIO_setPinConfig(GPIO_4_EPWM3_A);
GPIO_setDirectionMode(4, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(4, GPIO_PIN_TYPE_STD);

// GPIO15->EPWM3B->M1_WL*
GPIO_setPinConfig(GPIO_15_EPWM3_B);
GPIO_setDirectionMode(15, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(15, GPIO_PIN_TYPE_STD);
```

3. The below code assigns the corresponding base addresses of the PWM modules to the PWM handle in the HAL_MTR1_init() function that is located in the *hal.c* file. The below code does not need to be changed when adapting the code to a new board or C2000 MCU, it is just to show how the PWM handle is initialized in the code.

```
// initialize PWM handles for Motor 1
obj->pwmHandle[0] = MTR1_PWM_U_BASE;        //!< the PWM handle
obj->pwmHandle[1] = MTR1_PWM_V_BASE;        //!< the PWM handle
obj->pwmHandle[2] = MTR1_PWM_W_BASE;        //!< the PWM handle
```

4. The code below shows the configuration of the PWMs that occurs in the HAL_setupPWMs() function that is located in the *hal.c* file. Notice that the desired PWM period (USER_M1_PWM_TBPRD_NUM) for setting the PWM frequency, the SOC event prescale number (USER_M1_PWM_TBPRD_NUM), and the dead-band values (MTR1_PWM_DBRED_CNT, MTR1_PWM_DBFED_CNT) are defined in the *hal.h* file. The value of these defines can be changed according to the hardware board and control requirement. The PWM counter mode and the PWM action qualifier outputs need to be set up based on the hardware board.

```
void HAL_setupPWMs(HAL_MTR_Handle handle)
{
    HAL_MTR_Obj *obj = (HAL_MTR_Obj *)handle;
    uint16_t cnt;
    uint16_t pwmPeriodCycles = (uint16_t)(USER_M1_PWM_TBPRD_NUM);
    uint16_t numPWMTicksPerISRTick = USER_M1_NUM_PWM_TICKS_PER_ISR_TICK;
    ... ...
    for(cnt=0; cnt<3; cnt++)
    {
        // setup the Time-Base Control Register (TBCTL)
        EPWM_setTimeBaseCounterMode(obj->pwmHandle[cnt], EPWM_COUNTER_MODE_UP_DOWN);
        ... ...
        // setup the Action-Qualifier Output A Register (AQCTLA)
        EPWM_setActionQualifierAction(obj->pwmHandle[cnt], EPWM_AQ_OUTPUT_A,
            EPWM_AQ_OUTPUT_HIGH, EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA);
        EPWM_setActionQualifierAction(obj->pwmHandle[cnt], EPWM_AQ_OUTPUT_A,
            EPWM_AQ_OUTPUT_HIGH, EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD);
        EPWM_setActionQualifierAction(obj->pwmHandle[cnt], EPWM_AQ_OUTPUT_A,
            EPWM_AQ_OUTPUT_LOW, EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA);
        EPWM_setActionQualifierAction(obj->pwmHandle[cnt], EPWM_AQ_OUTPUT_A,
            EPWM_AQ_OUTPUT_LOW, EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO);
        ... ...
        // setup the Dead-Band Generator Control Register (DBCTL)
        EPWM_setDeadBandDelayMode(obj->pwmHandle[cnt], EPWM_DB_RED, true);
        EPWM_setDeadBandDelayMode(obj->pwmHandle[cnt], EPWM_DB_FED, true);

        // select EPWMA as the input to the dead band generator
        EPWM_setRisingEdgeDeadBandDelayInput(obj->pwmHandle[cnt], EPWM_DB_INPUT_EPWMA);

        // configure the right polarity for active high complementary config.
        EPWM_setDeadBandDelayPolarity(obj->pwmHandle[cnt], EPWM_DB_RED,
EPWM_DB_POLARITY_ACTIVE_HIGH);
        EPWM_setDeadBandDelayPolarity(obj->pwmHandle[cnt], EPWM_DB_FED,
EPWM_DB_POLARITY_ACTIVE_LOW);

        // setup the Dead-Band Rising Edge Delay Register (DBRED)
        EPWM_setRisingEdgeDelayCount(obj->pwmHandle[cnt], MTR1_PWM_DBRED_CNT);

        // setup the Dead-Band Falling Edge Delay Register (DBFED)
        EPWM_setFallingEdgeDelayCount(obj->pwmHandle[cnt], MTR1_PWM_DBFED_CNT);
        ... ...
    }
    ... ...
    // setup the Event Trigger Selection Register (ETSEL)
    EPWM_setInterruptSource(obj->pwmHandle[0], EPWM_INT_TBCTR_ZERO);
    EPWM_enableInterrupt(obj->pwmHandle[0]);
    EPWM_setADCTriggerSource(obj->pwmHandle[0], EPWM_SOC_A, EPWM_SOC_TBCTR_D_CMPC);
    EPWM_enableADCTrigger(obj->pwmHandle[0], EPWM_SOC_A);
    ... ...

    return;
}  // end of HAL_setupPWMs() function
```

5. The below code is located in the *hal.h* file and defines the source of the ADC start of conversion trigger. It is important that this ePWM SOC trigger corresponds to the same ePWM SOC that was enabled in the code shown in step 4 and the same ePWM that is associated with pwmHandle[0]. This is because these are the ones used in step 4 to set up the trigger source. In this case, EPWM1 A is used as the SOC for the ADC.

```
// Three-shunt
#define MTR1_ADC_TRIGGER_SOC      ADC_TRIGGER_EPWM1_SOCA // EPWM1_SOCA
```

4.1.2.5 Configuring the ADC Module

Similar to the previous PWM section, the ADC connections can also be changed for a custom board or a TI motor control kit or C2000 MCU that is not supported with the universal motor control lab. The HAL module configures the ADC channels to correctly correspond with the motor driver board. As an example, the connection diagram for the LAUNCHXL-F280025C and BOOSTXL-DRV8323RS combination is shown in Figure 4-3. The ADC modules configuration is described in the following steps, with potential board-specific changes highlighted in bold. Steps 1 and 2 are essential for configuring a new motor driver board or a different C2000 MCU to run the motor.

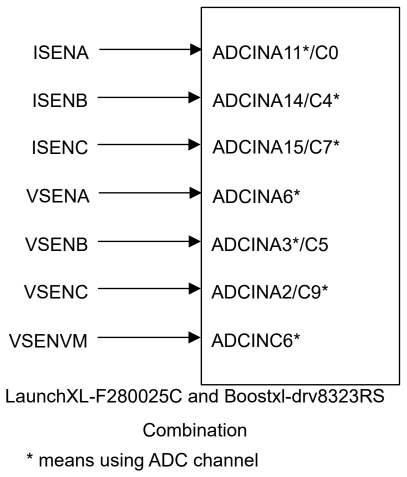


Figure 4-3. ADC Connection Diagram

The code below is taken from the *hal.h* and *hal.c* files located in the solutions\universal_motorcontrol_lab\f28002x\drivers\include and \source folders.

1. The below code shows the defines of the base addresses, assigned channels, and SOCs of the ADC modules in the *hal.h* file. Note that for the SOC number, multiple ADCs can be assotiated with the same SOC number as long as they belong to different ADC modules (in the case below, module A and module C). It is best to try to sample all the currents and all the voltages as close together as possible, so configure the SOC numbers with this in mind.

```
#define MTR1_IU_ADC_BASE          ADCA_BASE              // ADCA-A11*/C0
#define MTR1_IV_ADC_BASE          ADCC_BASE              // ADCC-A14/C4*
#define MTR1_IW_ADC_BASE          ADCC_BASE              // ADCC-A15/C7*
#define MTR1_VU_ADC_BASE          ADCA_BASE              // ADCA-A6*
#define MTR1_VV_ADC_BASE          ADCA_BASE              // ADCC-A3*/C5
#define MTR1_VW_ADC_BASE          ADCC_BASE              // ADCA-A2/C9*
#define MTR1_VDC_ADC_BASE         ADCC_BASE              // ADCC-C6*
#define MTR1_POT_ADC_BASE         ADCA_BASE              // ADCA-A12*/C1

#define MTR1_IU_ADCRES_BASE       ADCARESULT_BASE        // ADCA-A11*/C0
#define MTR1_IV_ADCRES_BASE       ADCCRESULT_BASE        // ADCC-A14/C4*
#define MTR1_IW_ADCRES_BASE       ADCCRESULT_BASE        // ADCC-A15/C7*
#define MTR1_VU_ADCRES_BASE       ADCARESULT_BASE        // ADCA-A6*
#define MTR1_VV_ADCRES_BASE       ADCARESULT_BASE        // ADCC-A3*/C5
#define MTR1_VW_ADCRES_BASE       ADCCRESULT_BASE        // ADCA-A2/C9*
#define MTR1_VDC_ADCRES_BASE      ADCCRESULT_BASE        // ADCC-C6*
#define MTR1_POT_ADCRES_BASE      ADCARESULT_BASE        // ADCA-A12*/C1

#define MTR1_IU_ADC_CH_NUM        ADC_CH_ADCIN11         // ADCA-A11*/C0
#define MTR1_IV_ADC_CH_NUM        ADC_CH_ADCIN4          // ADCC-A14/C4*
```

```
#define MTR1_IW_ADC_CH_NUM       ADC_CH_ADCIN7         // ADCC-A15/C7*
#define MTR1_VU_ADC_CH_NUM       ADC_CH_ADCIN6         // ADCA-A6*
#define MTR1_VV_ADC_CH_NUM       ADC_CH_ADCIN3         // ADCC-A3*/C5
#define MTR1_VW_ADC_CH_NUM       ADC_CH_ADCIN9         // ADCA-A2/C9*
#define MTR1_VDC_ADC_CH_NUM      ADC_CH_ADCIN6         // ADCC-C6*
#define MTR1_POT_ADC_CH_NUM      ADC_CH_ADCIN12        // ADCA-A12*/C1

#define MTR1_IU_ADC_SOC_NUM      ADC_SOC_NUMBER1       // ADCA-A11*/C10-SOC1-PPB1
#define MTR1_IV_ADC_SOC_NUM      ADC_SOC_NUMBER1       // ADCC-A14/C4* -SOC1-PPB1
#define MTR1_IW_ADC_SOC_NUM      ADC_SOC_NUMBER2       // ADCC-A15/C7* -SOC2-PPB2
#define MTR1_VU_ADC_SOC_NUM      ADC_SOC_NUMBER4       // ADCA-A6*     -SOC4
#define MTR1_VV_ADC_SOC_NUM      ADC_SOC_NUMBER5       // ADCC-A3*/C5  -SOC5
#define MTR1_VW_ADC_SOC_NUM      ADC_SOC_NUMBER5       // ADCA-A2/C9*  -SOC5
#define MTR1_VDC_ADC_SOC_NUM     ADC_SOC_NUMBER6       // ADCC-C6*     -SOC6
#define MTR1_POT_ADC_SOC_NUM     ADC_SOC_NUMBER6       // ADCA-A12*/C1 -SOC6

#define MTR1_IU_ADC_PPB_NUM      ADC_PPB_NUMBER1       // ADCA-A11*/C10-SOC1-PPB1
#define MTR1_IV_ADC_PPB_NUM      ADC_PPB_NUMBER1       // ADCC-A14/C4* -SOC1-PPB1
#define MTR1_IW_ADC_PPB_NUM      ADC_PPB_NUMBER2       // ADCC-A15/C7*- SOC2-PPB2
```

2. The below code shows the defines for the interrupt sources for the ISR in the *hal.h* file.

```
// interrupt
#define MTR1_PWM_INT_BASE        MTR1_PWM_U_BASE       // EPWM1

#define MTR1_ADC_INT_BASE        ADCA_BASE             // ADCA-A14 -SOC4
#define MTR1_ADC_INT_NUM         ADC_INT_NUMBER1       // ADCA_INT1-SOC4
#define MTR1_ADC_INT_SOC         ADC_SOC_NUMBER4       // ADCA_INT1-SOC4

#define MTR1_PIE_INT_NUM         INT_ADCA1             // ADCA_INT1-SOC4
#define MTR1_INT_ACK_GROUP       INTERRUPT_ACK_GROUP1  // ADCA_INT1-CPU_INT1
```

3. The ADC modules are set up in the HAL_setupADCs() function that is located in the *hal.c* file. The setup of the ADC that is used to sample the potentiometer is shown below as an example. If the user desires to use extra ADC channels to sample additional signals, the ADC channels will need to be set up in a similar manner as the code below.

```
// POT_M1 ADC_setupSOC(MTR1_POT_ADC_BASE, MTR1_POT_ADC_SOC_NUM, MTR1_ADC_TRIGGER_SOC,
MTR1_POT_ADC_CH_NUM, MTR1_ADC_V_SAMPLEWINDOW);
```

4. The ADC results are read in the HAL_readMtr1ADCData() function that is located in the *hal.h* file. The reading of the ADC result from the POT input is shown as an example below. If the user adds additional ADC channels for additional signal sampling, it will be necessary to add code similar to what is shown below to read the results of the added ADC channels. This will also require modification of the HAL_ADCData_t structure in order to store the data.

```
// read POT adc value
pADCData->potAdc = ADC_readResult(MTR1_POT_ADCRES_BASE, MTR1_POT_ADC_SOC_NUM);
```

4.1.2.6 Configuring the CMPSS Module

The CMPSS module is used for overcurrent monitoring for the phase currents. A threshold is set using the CMPSS DAC, and if the output of the current sense amplifier exceeds that threshold then the CMPSS output will trip.

If using a custom motor driver board, or migrating the code to a C2000 MCU or a TI motor driver EVM that is not supported with the current Universal Motor Control Lab, then the connections between the ADC pins and the CMPSS modules will need to be properly modified in the *hal.h* file based on the motor driver and C2000 MCU connections. For more details on the internal connections of the CMPSS module, see the *Analog Pins and Internal Connections* table in the *TMS320F28002x Real-Time Microcontrollers Technical Reference Manual (Rev. A)*.

The HAL module configures the CMPSS modules according to the motor driver board that is used. For example, the diagram of the connections between the LAUNCHXL-F280025C and the BOOSTXL-DRV8323RS are shown in Figure 4-4. The configuration of the CMPSS modules are described in the following steps (Board-specific or MCU-specific changes are indicated in **bold**).

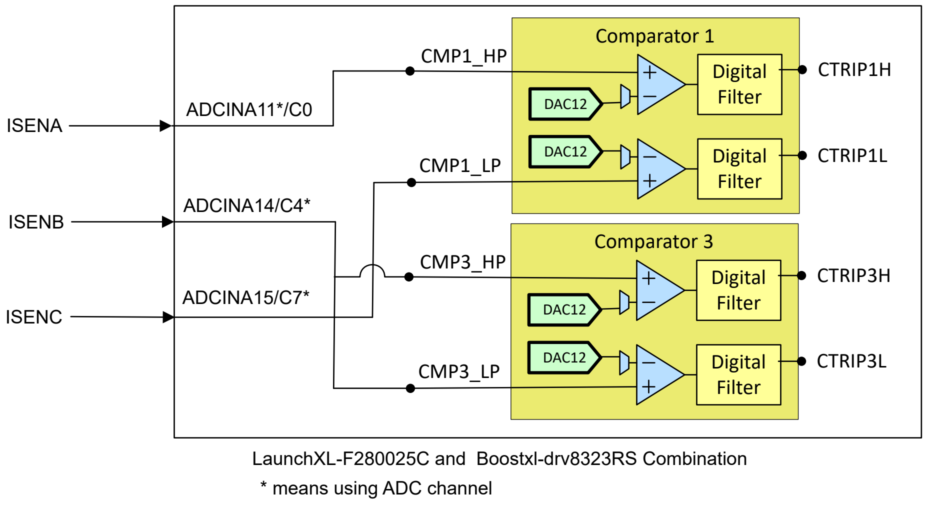


Figure 4-4. CMPSS Connection Diagram

1. The below code shows the defines of the base addresses of the CMPSS modules for the 3 phase currents. The code is located in the *hal.h* file.

```
#define MTR1_CMPSS_U_BASE        CMPSS1_BASE
#define MTR1_CMPSS_V_BASE        CMPSS3_BASE
#define MTR1_CMPSS_W_BASE        CMPSS1_BASE
```

2. The below code shows the defines that are used to assign the desired ADC inputs to the correct CMPSS module. This code is found in the *hal.h* file. Each CMPSS comparator has a high and low comparator, so the signals must be muxed appropriately to the desired input of the desired comparator. For more information on these connections, please refer to the Analog Pins and internal connections table in the technical reference manual of the microcontroller that is being used. Note: for the LAUNCHXL F280025C, the available CMPSS modules for the motor driver current sensing pins are CMPSS3 and CMPSS1, so the phase U current and phase W current both need to share CMPSS1. This results in the limitation that phase U current only trips the CMPSS with a positive overcurrent, and phase W current only trips the CMPSS with a negative overcurrent. Since phase V has a dedicated CMPSS available, overcurrent on this phase will be detected for both postitive and negative currents. If modifying the code to support a C2000 MCU device that has separate CMPSS modules for each phase current inputs (such as in the case of the F280049C Launchpad), then the code can be configured to trip with both high and low overcurrents on each of the phases by muxing the same phase current ADC input to both the high and low inputs of the corresponding CMPSS module.

```
// CMPSS
// For single phase current sensing, DRV8323RH and RS only
#define MTR1_IDC_CMPHP_SEL       ASYSCTL_CMPHPMUX_SELECT_3    // CMPSS3-A14/C4*
#define MTR1_IDC_CMPLP_SEL       ASYSCTL_CMPLPMUX_SELECT_3    // CMPSS3-A14/C4*

#define MTR1_IDC_CMPHP_MUX       4                            // CMPSS3-A14/C4*
#define MTR1_IDC_CMPLP_MUX       4                            // CMPSS3-A14/C4*

// For three-phase current sensing
#define MTR1_IU_CMPHP_SEL        ASYSCTL_CMPHPMUX_SELECT_1    // CMPSS1-A11
#define MTR1_IU_CMPLP_SEL        ASYSCTL_CMPLPMUX_SELECT_1    // CMPSS1-A11, N/A

#define MTR1_IV_CMPHP_SEL        ASYSCTL_CMPHPMUX_SELECT_3    // CMPSS3-C4
#define MTR1_IV_CMPLP_SEL        ASYSCTL_CMPLPMUX_SELECT_3    // CMPSS3-C4

#define MTR1_IW_CMPHP_SEL        ASYSCTL_CMPHPMUX_SELECT_1    // CMPSS1-C7, N/A
#define MTR1_IW_CMPLP_SEL        ASYSCTL_CMPLPMUX_SELECT_1    // CMPSS1-C7

#define MTR1_IU_CMPHP_MUX        1                            // CMPSS1-A11
#define MTR1_IU_CMPLP_MUX        1                            // CMPSS1-A11

#define MTR1_IV_CMPHP_MUX        4                            // CMPSS3-C4
#define MTR1_IV_CMPLP_MUX        4                            // CMPSS3-C4

#define MTR1_IW_CMPHP_MUX        3                            // CMPSS1-C7
#define MTR1_IW_CMPLP_MUX        3                            // CMPSS1-C7
```

3. The below code shows the setup of the CMPSS modules which occurs in the HAL_setupCMPSSs() function that is located in the *hal.c* file. In this example, the CMPSS1_HP is linked to phase U current, and CMPSS1_LP is linked to phase W current, so the CMPSS1 is configured twice to simplify the code. The below code may not need to be modified when modifying the Universal Motor Control Lab, but it is important to ensure that the code in the previous step is properly configured so that the HAL_setupCMPSSs() function sets up the CMPSS modules correctly.

```
void HAL_setupCMPSSs(HAL_MTR_Handle handle)
{
    HAL_MTR_Obj *obj = (HAL_MTR_Obj *)handle;
... ...
    uint16_t cmpsaDACH = MTR1_CMPSS_DACH_VALUE;
    uint16_t cmpsaDACL = MTR1_CMPSS_DACL_VALUE;
... ...
    ASysCtl_selectCMPHPMux(MTR1_IU_CMPHP_SEL, MTR1_IU_CMPHP_MUX);
    ASysCtl_selectCMPLPMux(MTR1_IU_CMPLP_SEL, MTR1_IU_CMPLP_MUX);

    ASysCtl_selectCMPHPMux(MTR1_IV_CMPHP_SEL, MTR1_IV_CMPHP_MUX);
    ASysCtl_selectCMPLPMux(MTR1_IV_CMPLP_SEL, MTR1_IV_CMPLP_MUX);

    ASysCtl_selectCMPHPMux(MTR1_IW_CMPHP_SEL, MTR1_IW_CMPHP_MUX);
    ASysCtl_selectCMPLPMux(MTR1_IW_CMPLP_SEL, MTR1_IW_CMPLP_MUX);

    for(cnt=0; cnt<3; cnt++)
    {
        // Enable CMPSS and configure the negative input signal to come from the DAC
        CMPSS_enableModule(obj->cmpssHandle[cnt]);

        // NEG signal from DAC for COMP-H
        CMPSS_configHighComparator(obj->cmpssHandle[cnt], CMPSS_INSRC_DAC);

        // NEG signal from DAC for COMP-L
        CMPSS_configLowComparator(obj->cmpssHandle[cnt], CMPSS_INSRC_DAC);

        // Configure the output signals. Both CTRIPH and CTRIPOUTH will be fed by
        // the asynchronous comparator output.
        // Dig filter output ==> CTRIPH, Dig filter output ==> CTRIPOUTH
        CMPSS_configOutputsHigh(obj->cmpssHandle[cnt],
                                CMPSS_TRIP_FILTER |
                                CMPSS_TRIPOUT_FILTER);

        // Dig filter output ==> CTRIPL, Dig filter output ==> CTRIPOUTL
        CMPSS_configOutputsLow(obj->cmpssHandle[cnt],
                               CMPSS_TRIP_FILTER |
                               CMPSS_TRIPOUT_FILTER |
                               CMPSS_INV_INVERTED);

        // Configure digital filter. For this example, the maxiumum values will be
        // used for the clock prescale, sample window size, and threshold.
        CMPSS_configFilterHigh(obj->cmpssHandle[cnt], 32, 32, 30);
        CMPSS_initFilterHigh(obj->cmpssHandle[cnt]);

        // Initialize the filter logic and start filtering
        CMPSS_configFilterLow(obj->cmpssHandle[cnt], 32, 32, 30);
        CMPSS_initFilterLow(obj->cmpssHandle[cnt]);

        // Set up COMPHYSCTL register
        // COMP hysteresis set to 2x typical value
        CMPSS_setHysteresis(obj->cmpssHandle[cnt], 1);

        // Use VDDA as the reference for the DAC and set DAC value to midpoint for
        // arbitrary reference
        CMPSS_configDAC(obj->cmpssHandle[cnt],
                   CMPSS_DACREF_VDDA | CMPSS_DACVAL_SYSCLK | CMPSS_DACSRC_SHDW);

        // Set DAC-H to allowed MAX +ve current
        CMPSS_setDACValueHigh(obj->cmpssHandle[cnt], cmpsaDACH);

        // Set DAC-L to allowed MAX -ve current
        CMPSS_setDACValueLow(obj->cmpssHandle[cnt], cmpsaDACL);

        // Clear any high comparator digital filter output latch
        CMPSS_clearFilterLatchHigh(obj->cmpssHandle[cnt]);

        // Clear any low comparator digital filter output latch
        CMPSS_clearFilterLatchLow(obj->cmpssHandle[cnt]);
    }
... ...
    return;
}
```

4.1.2.7 Configuring Fault Protection Function

When certain faults occur, such as the fault pin from the motor driver tripping or an overcurrent event detected on the CMSS modules, action is taken to stop the the output PWMs from the MCU. In order to accomplish this, the ePWM muxes and the ePWM trip zones are configured to take appropriate action. The code that accomplishes this is described in this section. Appropriate modification must be made to this code when using a custom motor driver board, using a TI motor driver EVM that is not already supported for the Universal Motor Control Lab, or if configuring the code for a different C2000 MCU. Possible necessary code changes are highlighted in **bold**. For more detailed information on the X-BAR, see the ePWM X-BAR Mux Configuration Table and OUTPUT X-BAR Mux Configuration Table in the TMS320F28002x Real-Time Microcontrollers Technical Reference Manual.

1. The below code defines the Motor driver Fault GPIO input. This code is found in the *hal.h* file.

```
//! \brief Defines the gpio for the nFAULT of Power Module
#define MTR1_PM_nFAULT_GPIO     34
```

2. The below defines are used to set up the X-BAR to link the signals from the CMPSS modules and the Fault GPIO to the ePWM trip zone sub module. This code is found in the *hal.h* file. Notice in the below code that the phase U current X-BAR ePWM mux is configured for CTRIPL and that the phase W current X-BAR ePWM mux is configured for CTRIPH, but the phase V current X-BAR ePWM mux is configured for CTRIPH_OR_L. This is because the phase W and phase U current inputs share the same CMPSS module (module 1), whereas the phase V current input has its own dedicated CMPSS module (module 3) which allows it to be configured to trip both high or low on the same phase current input. If using a C2000 MCU or motor driver board that allows the current inputs to have their own dedicated CMPSS module, then the ePWM mux configuration for phase U and phase W currents need to be changed to CTRIPH_OR_L, similar to how phase V is currently configured.

```
#define MTR1_XBAR_TRIP_ADDRL    XBAR_O_TRIP7MUX0TO15CFG
#define MTR1_XBAR_TRIP_ADDRH    XBAR_O_TRIP7MUX16TO31CFG

#define MTR1_IDC_XBAR_EPWM_MUX XBAR_EPWM_MUX05_CMPSS3_CTRIPL   // CMPSS3-LP, single shunt only
#define MTR1_IDC_XBAR_MUX       XBAR_MUX05 // CMPSS3-LP, single shunt only

#define MTR1_IU_XBAR_EPWM_MUX XBAR_EPWM_MUX00_CMPSS1_CTRIPH // CMPSS1-HP
#define MTR1_IV_XBAR_EPWM_MUX XBAR_EPWM_MUX04_CMPSS3_CTRIPH_OR_L // CMPSS3-HP&LP
#define MTR1_IW_XBAR_EPWM_MUX XBAR_EPWM_MUX01_CMPSS1_CTRIPL       // CMPSS1-LP

#define MTR1_IU_XBAR_MUX        XBAR_MUX00          // CMPSS1-HP
#define MTR1_IV_XBAR_MUX        XBAR_MUX04          // CMPSS3-HP&LP
#define MTR1_IW_XBAR_MUX        XBAR_MUX01 // CMPSS1-LP

#define MTR1_XBAR_INPUT1 XBAR_INPUT1
#define MTR1_TZ_OSHT1 EPWM_TZ_SIGNAL_OSHT1
#define MTR1_XBAR_TRIP XBAR_TRIP7
#define MTR1_DCTRIPIN EPWM_DC_COMBINATIONAL_TRIPIN7
```

3. The below code configures the ePWM X-BAR and the trip signals for the phase currents and motor fault pin. This is done in the HAL_setupMtrFaults() function in the *hal.c* file. The below code may not need to be modified when modifying the Universal Motor Control Lab, but it is important to ensure that the code in the previous step is properly configured in order to set up the correct mux and trip values.

```
void HAL_setupMtrFaults(HAL_MTR_Handle handle)
{
    ... ...
    // Configure TRIP7 to be CTRIP5H and CTRIP5L using the ePWM X-BAR
    XBAR_setEPWMMuxConfig(MTR1_XBAR_TRIP, MTR1_IU_XBAR_EPWM_MUX);

    // Configure TRIP7 to be CTRIP1H and CTRIP1L using the ePWM X-BAR
    XBAR_setEPWMMuxConfig(MTR1_XBAR_TRIP, MTR1_IV_XBAR_EPWM_MUX);

    // Configure TRIP7 to be CTRIP3H and CTRIP3L using the ePWM X-BAR
    XBAR_setEPWMMuxConfig(MTR1_XBAR_TRIP, MTR1_IW_XBAR_EPWM_MUX);
    // Disable all the mux first
    XBAR_disableEPWMMux(MTR1_XBAR_TRIP, 0xFFFF);

    // Enable Mux 0 OR Mux 4 to generate TRIP
    XBAR_enableEPWMMux(MTR1_XBAR_TRIP, MTR1_IU_XBAR_MUX | MTR1_IV_XBAR_MUX | MTR1_IW_XBAR_MUX);

    ... ...

    // configure the input x bar for TZ2 to GPIO, where Over Current is connected
    XBAR_setInputPin(INPUTXBAR_BASE, MTR1_XBAR_INPUT1, MTR1_PM_nFAULT_GPIO);
    XBAR_lockInput(INPUTXBAR_BASE, MTR1_XBAR_INPUT1);

    for(cnt=0; cnt<3; cnt++)
    {
        EPWM_enableTripZoneSignals(obj->pwmHandle[cnt],
                                   EPWM_TZ_SIGNAL_CBC6);

        //enable DC TRIP combinational input
        EPWM_enableDigitalCompareTripCombinationInput(obj->pwmHandle[cnt],
                                                      MTR1_DCTRIPIN, EPWM_DC_TYPE_DCAH);

        EPWM_enableDigitalCompareTripCombinationInput(obj->pwmHandle[cnt],
                                                      MTR1_DCTRIPIN, EPWM_DC_TYPE_DCBH);

        // Trigger event when DCAH is High
        EPWM_setTripZoneDigitalCompareEventCondition(obj->pwmHandle[cnt],
                                                     EPWM_TZ_DC_OUTPUT_A1,
                                                     EPWM_TZ_EVENT_DCXH_HIGH);

        // Trigger event when DCBH is High
        EPWM_setTripZoneDigitalCompareEventCondition(obj->pwmHandle[cnt],
                                                     EPWM_TZ_DC_OUTPUT_B1,
                                                     EPWM_TZ_EVENT_DCXL_HIGH);

        // Configure the DCA path to be un-filtered and asynchronous
        EPWM_setDigitalCompareEventSource(obj->pwmHandle[cnt],
                                          EPWM_DC_MODULE_A,
                                          EPWM_DC_EVENT_1,
                                          EPWM_DC_EVENT_SOURCE_FILT_SIGNAL);

        // Configure the DCB path to be un-filtered and asynchronous
        EPWM_setDigitalCompareEventSource(obj->pwmHandle[cnt],
                                          EPWM_DC_MODULE_B,
                                          EPWM_DC_EVENT_1,
                                          EPWM_DC_EVENT_SOURCE_FILT_SIGNAL);

        EPWM_setDigitalCompareEventSyncMode(obj->pwmHandle[cnt],
                                            EPWM_DC_MODULE_A,
                                            EPWM_DC_EVENT_1,
                                            EPWM_DC_EVENT_INPUT_NOT_SYNCED);

        EPWM_setDigitalCompareEventSyncMode(obj->pwmHandle[cnt],
                                            EPWM_DC_MODULE_B,
                                            EPWM_DC_EVENT_1,
                                            EPWM_DC_EVENT_INPUT_NOT_SYNCED);

        // Enable DCA as OST
        EPWM_enableTripZoneSignals(obj->pwmHandle[cnt], EPWM_TZ_SIGNAL_DCAEVT1);

        // Enable DCB as OST
        EPWM_enableTripZoneSignals(obj->pwmHandle[cnt], EPWM_TZ_SIGNAL_DCBEVT1);

        // What do we want the OST/CBC events to do?
        // TZA events can force EPWMxA
        // TZB events can force EPWMxB
        EPWM_setTripZoneAction(obj->pwmHandle[cnt],
                               EPWM_TZ_ACTION_EVENT_TZA,
                               EPWM_TZ_ACTION_LOW);

        EPWM_setTripZoneAction(obj->pwmHandle[cnt],
                               EPWM_TZ_ACTION_EVENT_TZB,
                               EPWM_TZ_ACTION_LOW);
    }
    ... ...
    return;
} // end of HAL_setupMtrFaults() function
```

4.1.3 Adding Additional Functionality to Motor Control Project

The example lab project provides several interface functions to start/stop the motor and set the reference speed by using push button, potentiometer, or a communication bus like SCI or CAN.

4.1.3.1 Adding Push Buttons Functionality

It is often useful to read push buttons to allow a motor to run, stop, or simply to change the state of a global variable when pushing a button. As an example, the user can connect GPIO23 to a button to start/stop the motor. To do this, enable the pre-define symbol CMD_SWITCH_EN in project build properties as shown in Figure 3-19. The GPIO state will be assigned to motorVars_M1.flagEnableRunAndIdentify. The detailed steps are as follows.

1. Define the GPIO number in the *hal.h* file

```
#define MTR1_CMD_SWITCH_GPIO    23
```

2. Configure the GPIO in HAL_setupGpios() function of the *hal.c* file to allow this pin to be an input.

```
// GPIO23->Command Switch Button
GPIO_setPinConfig(GPIO_23_GPIO23);
GPIO_setDirectionMode(23, GPIO_DIR_MODE_IN);
GPIO_setPadConfig(23, GPIO_PIN_TYPE_PULLUP);
GPIO_setQualificationMode(23, GPIO_QUAL_3SAMPLE);
GPIO_setQualificationPeriod(23, 4);
```

3. Read the GPIO with a digital filter as follows in updateCmdSwitch() in the *motor_common.c* file.

```
if(GPIO_readPin(MTR1_CMD_SWITCH_GPIO) == 0)
{
    objMtr->cmdSwtich.lowTimeCnt++;

    if(objMtr->cmdSwtich.lowTimeCnt > objMtr->cmdSwtich.delayTimeSet)
    {
        objMtr->cmdSwtich.flagCmdRun = true;
    }

    if(objMtr->cmdSwtich.highTimeCnt > 0)
    {
        objMtr->cmdSwtich.highTimeCnt--;
    }
}
else
{
    objMtr->cmdSwtich.highTimeCnt++;

    if(objMtr->cmdSwtich.highTimeCnt > objMtr->cmdSwtich.delayTimeSet)
    {
        objMtr->cmdSwtich.flagCmdRun = false;
    }

    if(objMtr->cmdSwtich.lowTimeCnt > 0)
    {
        objMtr->cmdSwtich.lowTimeCnt--;
    }
}
```

4. Link the state to the motor start/stop variable in updateCmdSwitch() of the *motor_common.c* file.

```
if((objMtr->cmdSwtich.flagEnablCmd == true) && (objMtr->faultMtrUse.all == 0))
{
    objMtr->flagEnableRunAndIdentify = objMtr->cmdSwtich.flagCmdRun;
}
```

4.1.3.2 Adding Potentiometer Read Functionality

A potentiometer is often used to allow a motor to run, stop and set the reference speed. As an example, the user may connect ADCA12 to a potentiometer. To do this, enable the pre-define symbol CMD_POT_EN in project build properties as shown in Figure 3-19. The results from reading the ADC will be converted to a speed value and assigned to the variables motorVars_M1.flagEnableRunAndIdentify to start/stop the motor and to motorVars_M1.speedRef_Hz to set the reference speed. The detailed steps are as the follows.

1. Define theADC channel for connecting to potentiometer in the *hal.h* file as follows:

```
#define MTR1_POT_ADC_BASE        ADCA_BASE
#define MTR1_POT_ADCRES_BASE     ADCARESULT_BASE
#define MTR1_POT_ADC_CH_NUM      ADC_CH_ADCIN12
#define MTR1_POT_ADC_SOC_NUM     ADC_SOC_NUMBER6
```

2. Configure ADC channel in function HAL_setupAdcs() of the *hal.c* file as follows:

```
// POT_M1
ADC_setupSOC(MTR1_POT_ADC_BASE, MTR1_POT_ADC_SOC_NUM, MTR1_ADC_TIGGER_SOC,
             MTR1_POT_ADC_CH_NUM, MTR1_ADC_V_SAMPLEWINDOW);
```

3. Read the ADC result register and scale the value in HAL_readMtr1ADCData() of the *hal.h* file as follows:

```
// read POT adc value
pADCData->potAdc = ADC_readResult(MTR1_POT_ADCRES_BASE, MTR1_POT_ADC_SOC_NUM);
```

4. Convert the ADC value to a speed value in updateExtCmdPotFreq() of the *motor_common.c* file.

4.1.3.3 Adding CAN Functionality

CAN functionality can be added into the lab project to provide the user a communication bus for sending the start/stop command and getting the feedback running states. To utilize this, enable the pre-define symbol CMD_CAN_EN in project build properties as shown in Figure 3-19. The detailed steps are as the followings.

1. Add the CAN source files to the project. Right-click on the project name in the CCS project explorer window, and select “Add Files.” Next, navigate to the following folder and select “Link to Files”.

   `<install_location>\c2000ware\driverlib\f28002x\driverlib\can.c`
2. Edit the HAL_Obj to add canHandle in the *hal_obj.h* header file.

```
typedef struct _HAL_Obj_
{
  uint32_t       adcHandle[2];      //!< the ADC handles
... ...
  uint32_t       canHandle;         //!< the CAN handle
... ...
} HAL_Obj;
```

3. Initialize the CAN handles in the HAL_init() function in the *hal.c* file.

```
HAL_Handle HAL_init(void *pMemory,const size_t numBytes)
{
... ...
    // initialize CAN handle
    obj->canHandle = CANA_BASE;              //!< the CAN handle
... ...
    return(handle);
} // end of HAL_init() function
```

4. Prototype the CAN setup function in the *communication.h* file as the following code.

```
//! \brief     Sets up the CANA
//! \param[in] handle  The hardware abstraction layer (HAL) handle
extern void HAL_setupCANA(HAL_Handle handle);
```

5. Define the CAN setup functions in the *communication.c* file as the following code.

```
void HAL_setupCANA(HAL_Handle halHandle)
{
    HAL_Obj *obj = (HAL_Obj *)halHandle;

    // Initialize the CAN controller
    CAN_initModule(obj->canHandle);

    // Set up the CAN bus bit rate to 200kHz
    // Refer to the Driver Library User Guide for information on how to set
    // tighter timing control. Additionally, consult the device data sheet
    // for more information about the CAN module clocking.
    CAN_setBitRate(obj->canHandle, DEVICE_SYSCLK_FREQ, 500000, 16);

    // Initialize the transmit message object used for sending CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 1
    //      Message Identifier: 0x1
    //      Message Frame: Standard
    //      Message Type: Transmit
    //      Message ID Mask: 0x0
    //      Message Object Flags: Transmit Interrupt
    //      Message Data Length: 8 Bytes
    CAN_setupMessageObject(CANA_BASE, TX_MSG_OBJ_ID, 0x1, CAN_MSG_FRAME_STD,
                           CAN_MSG_OBJ_TYPE_TX, 0, CAN_MSG_OBJ_TX_INT_ENABLE,
                           MSG_DATA_LENGTH);

    // Initialize the receive message object used for receiving CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 2
    //      Message Identifier: 0x1
    //      Message Frame: Standard
    //      Message Type: Receive
    //      Message ID Mask: 0x0
    //      Message Object Flags: Receive Interrupt
    //      Message Data Length: 8 Bytes
    CAN_setupMessageObject(obj->canHandle, RX_MSG_OBJ_ID, 0x1, CAN_MSG_FRAME_STD,
                           CAN_MSG_OBJ_TYPE_RX, 0, CAN_MSG_OBJ_RX_INT_ENABLE,
                           MSG_DATA_LENGTH);

    // Start CAN module operations
    CAN_startModule(obj->canHandle);

    return;
}  // end of HAL_setupCANA() function
```

Note

The various CAN module parameters are to be initialized according to the system needs, and the above is just a simple reference.

6. Enable the appropriate CAN peripheral clocks in the HAL setup clocks function HAL_setupPeripheralClks() of the *hal.c* file.

```
   SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANA);
```

7. Configure the related GPIOs to CAN function in HAL_setupGpios() function of the *hal.c* file.

```
    // GPIO33->CAN_TX
    GPIO_setPinConfig(GPIO_32_CANA_TX);
    GPIO_setDirectionMode(32, GPIO_DIR_MODE_OUT);
    GPIO_setPadConfig(32, GPIO_PIN_TYPE_STD);
    GPIO_setQualificationMode(32, GPIO_QUAL_ASYNC);

    // GPIO33->CAN_RX
    GPIO_setPinConfig(GPIO_33_CANA_RX);
    GPIO_setDirectionMode(33, GPIO_DIR_MODE_IN);
    GPIO_setPadConfig(33, GPIO_PIN_TYPE_STD);
    GPIO_setQualificationMode(33, GPIO_QUAL_ASYNC);
```

8. Prototype the CAN interrupt setup function in the *communication.h* file.

```
extern void HAL_enableCANInts(HAL_Handle handle);
```

9. Define the CAN interrupt setup function.

```
void HAL_enableCANInts(HAL_Handle handle)
{
    HAL_Obj *obj = (HAL_Obj *)handle;

    // Enable CAN test mode with external loopback
//      CAN_enableTestMode(CANA_BASE, CAN_TEST_EXL);    // Only for debug

    // Enable interrupts on the CAN peripheral.
    CAN_enableInterrupt(obj->canHandle, CAN_INT_IE0 | CAN_INT_ERROR |
                        CAN_INT_STATUS);


    // enable the PIE interrupts associated with the CAN interrupts
    Interrupt_enable(INT_CANA0);


    CAN_enableGlobalInterrupt(obj->canHandle, CAN_GLOBAL_INT_CANINT0);

    // enable the cpu interrupt for CAN interrupts
    Interrupt_enableInCPU(INTERRUPT_CPU_INT9);

    return;
} // end of HAL_enableCANInts() function
```

10. Call the CAN setup function and the CAN interrupt setup function in the *sys_main.c* file.

```
    // setup the CAN
    HAL_setupCANA(halHandle);
    // setup the CAN interrupt
    HAL_enableCANInts(halHandle);
```

11. Prototype the CAN interrupt service (ISR) routine in the *communication.h* file.

```
extern __interrupt void canaISR(void);
```

12. Add the CAN interrupt service (ISR) routine vector to the PIE table in initCANCOM() in the *communication.c* file.

```
    Interrupt_register(INT_CANA0, &canaISR);
```

13. To place the CAN ISR code in flash and run from RAM for accelerating the execution speed by adding the following code in the *communication.c* file.

```
#pragma CODE_SECTION(canaISR, ".TI.ramfunc");
```

14. Define the CAN interrupt routine canaISR() in the *communication.c* file. Define the canaIS() function that has been prototyped and passed to the PIE vector table. The example code below provides a function to receive/transmit the message data with CAN and clears the interrupt in the PIE, allowing the CAN interrupt to be triggered again.

```
__interrupt void canaISR(void)
{
... ...
    // Check if the cause is the transmit message object 1
    // Check if the cause is the transmit message object 1
    else if(status == TX_MSG_OBJ_ID)
    {
        //
        // Getting to this point means that the TX interrupt occurred on
        // message object 1, and the message TX is complete.  Clear the
        // message object interrupt.
        //
        CAN_clearInterruptStatus(CANA_BASE, TX_MSG_OBJ_ID);

        // Increment a counter to keep track of how many messages have been
        // sent.  In a real application this could be used to set flags to
        // indicate when a message is sent.
        canComVars.txMsgCount++;

        // Since the message was sent, clear any error flags.
        canComVars.errorFlag = 0;
    }

    // Check if the cause is the receive message object 2
    else if(status == RX_MSG_OBJ_ID)
    {
        //
        // Get the received message
        //
        CAN_readMessage(halHandle->canHandle, RX_MSG_OBJ_ID,
                        (uint16_t *)(&canComVars.rxMsgData[0]));

        // Getting to this point means that the RX interrupt occurred on
        // message object 2, and the message RX is complete.  Clear the
        // message object interrupt.
        CAN_clearInterruptStatus(halHandle->canHandle, RX_MSG_OBJ_ID);

        canComVars.rxMsgCount++;
        canComVars.flagRxDone = true;

        // Since the message was received, clear any error flags.
        canComVars. errorFlag = 0;
    }
... ...
    // Clear the global interrupt flag for the CAN interrupt line
    CAN_clearGlobalInterruptStatus(halHandle->canHandle, CAN_GLOBAL_INT_CANINT0);

    // Acknowledge this interrupt located in group 9
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);

    return;
}
```

15. Prototype and define updateCANCmdFreq() in the *communication.h* file and the *communication.c* file separately. The CAN bus received and transmitting data is processed and linked to motor control variables in updateCANCmdFreq().

```
void updateCANCmdFreq(MOTOR_Handle handle)
{
...
}
```

16. Call updateCANCmdFreq() in endless loop.

```
updateCANCmdFreq(motorHandle_M1);

if((motorVars_M1.cmdCAN.flagEnablCmd == true) && (motorVars_M1.faultMtrUse.all == 0))
{
    canComVars.flagCmdTxRun = motorVars_M1.cmdCAN.flagCmdRun;
    canComVars.speedSet_Hz = motorVars_M1.cmdCAN.speedSet_Hz;

    if(motorVars_M1.cmdCAN.flagEnablSyncLead == true)
    {
        motorVars_M1.flagEnableRunAndIdentify = motorVars_M1.cmdCAN.flagCmdRun;
        motorVars_M1.speedRef_Hz = motorVars_M1.cmdCAN.speedSet_Hz;
    }
    else
    {
        motorVars_M1.flagEnableRunAndIdentify = canComVars.flagCmdRxRun;
        motorVars_M1.speedRef_Hz = canComVars.speedRef_Hz;
    }
}
```

4.2 Supporting New BLDC Motor Driver Board

C2000 MCUs can be used with BLDC motor drivers for driving three-phase BLDC or PMSM motor applications. This universal lab project can support various pre-defined BLDC motor drivers. The user can refer to the example code in the lab project and follow the steps described in this section to implement newer or otherwise unsupported BLDC motor drivers. This section uses DRV8323RS with SPI as an example.

1. Design the driver file for the new BLDC motor driver EVM board.

   If the BLDC motor driver supports SPI, refer to the existing BLDC motor driver files (*drv8323s.h* and *drv8323s.c)* and change the registers and API function definitions where necessary in the *drv8xxx.h* and *drv8xxx.c* files. The detailed description of the BLDC motor driver register maps can be found in the data sheet of the BLDC motor driver device.

   Create a new set of folders for the driver file, as with the DRV8323 ("\libraries\drvic\drv8323\include" and "\libraries\drvic\drv8323\source").
2. Add the BLDC motor driver source files to the motor control project.

   First, add the BLDC motor driver source files to the project you are working. There are two methods to add the files.

   Using an Editor to open the *universal_motorcontrol_lab.projectspec* projectspec file, add the files to the project as follows.

```
<file action="link" path="SDK_ROOT/libraries/drvic/drv8323/source/drv8323s.c"
targetDirectory="src_board" applicableConfigurations="Flash_lib_DRV8323RS" />
<file action="link" path="SDK_ROOT/libraries/drvic/drv8323/include/drv8323s.h"
targetDirectory="src_board" applicableConfigurations="Flash_lib_DRV8323RS" />
```

   Alternatively, right-click on the project name in the CCS project explorer window, and select “Add Files.” Next, navigate to the following folder and select the designed driver files from " \libraries\drvic\drv8323\source", and then select "Link to Files".
3. Add the header file to the include list in the *hal_obj.h* file.

```
#include "drv8323s.h"
```

   To ensure that the header files can be correctly found, add the directory to the header files in "Project Properties"->"Build"->"C2000 Compiler"->"Include Options"->"Add dir to #include search path".

   Alternatively, add the directory to the header files by adding the following content in the *universal_motorcontrol_lab.projectspec* projectspec file.

```
-I${SDK_ROOT}/libraries/drvic/drv8323/include
```

4. Edit the HAL_Obj to add the drvic interface handle and SPI handle.

   Refer to the DRV8323 files to add the supporting code as follows.

   Add the defines in the *hal_obj.h* file.

```
#define DRAdd the defines in hal_obj.h fileVIC_Obj              DRV8323_Obj
#define DRVIC_VARS_t              DRV8323_VARS_t
#define DRVIC_Handle              DRV8323_Handle
#define DRVICVARS_Handle          DRV8323VARS_Handle

#define DRVIC_init                DRV8323_init
#define DRVIC_enable              DRV8323_enable
#define DRVIC_writeData           DRV8323_writeData
#define DRVIC_readData            DRV8323_readData

#define DRVIC_setupSPI            DRV8323_setupSPI

#define DRVIC_setSPIHandle        DRV8323_setSPIHandle
#define DRVIC_setGPIOCSNumber     DRV8323_setGPIOCSNumber
#define DRVIC_setGPIOENNumber     DRV8323_setGPIOENNumber
```

   Add the drvic interface handle and SPI handle to HAL_Obj:

```
  uint32_t       spiHandle;          //!< the SPI handle

  DRVIC_Handle   drvicHandle;        //!< the drvic interface handle
  DRVIC_Obj      drvic;              //!< the drvic interface object

  uint32_t       gateEnableGPIO;
  // BSXL8353RS_REVA
```

5. Configure SPI for communication with the BLDC motor driver.

   When using a motor driver with SPI, the SPI must be configured correctly from the MCU to match the format needed to communicate with the BLDC motor driver device properly.

   Configure the related GPIOs for SPI function in HAL_setupGPIOs() of the *hal.c* file. Ensure to check the BLDC motor driver data sheet to determine if each SPI pin requires an external pullup or pull down resistor, or if it is configured as a push-pull pin.

```
    // GPIO5->Connect to GPIO5 using a jumper wire->M1_DRV_SCS
    GPIO_setPinConfig(GPIO_5_SPIA_STE);
    GPIO_setDirectionMode(5, GPIO_DIR_MODE_OUT);
    GPIO_setPadConfig(5, GPIO_PIN_TYPE_STD);

    // GPIO09->M1_DRV_SCLK*
    GPIO_setPinConfig(GPIO_9_SPIA_CLK);
    GPIO_setDirectionMode(9, GPIO_DIR_MODE_OUT);
    GPIO_setPadConfig(9, GPIO_PIN_TYPE_PULLUP);

    // GPIO10->SPIA_SOMI->M1_DRV_SDO*
    GPIO_setPinConfig(GPIO_10_SPIA_SOMI);
    GPIO_setDirectionMode(10, GPIO_DIR_MODE_IN);
    GPIO_setPadConfig(10, GPIO_PIN_TYPE_PULLUP);

    // GPIO11->SPIA_SIMO->M1_DRV_SDI*
    GPIO_setPinConfig(GPIO_11_SPIA_SIMO);
    GPIO_setDirectionMode(11, GPIO_DIR_MODE_OUT);
    GPIO_setPadConfig(11, GPIO_PIN_TYPE_PULLUP);
```

   Configure the SPI control registers for baud rate, data frame in HAL_setupSPI() in the *hal.c* file:

```
    // Must put SPI into reset before configuring it
    SPI_disableModule(obj->spiHandle);

    // SPI configuration. Use a 500kHz SPICLK and 16-bit word size, 25MHz LSPCLK
    SPI_setConfig(obj->spiHandle, DEVICE_LSPCLK_FREQ, SPI_PROT_POL0PHA0,
                  SPI_MODE_MASTER, 400000, 16);

    SPI_disableLoopback(obj->spiHandle);

    SPI_setEmulationMode(obj->spiHandle, SPI_EMULATION_FREE_RUN);

    SPI_enableFIFO(obj->spiHandle);
    SPI_setTxFifoTransmitDelay(obj->spiHandle, 0x10);

    SPI_clearInterruptStatus(obj->spiHandle, SPI_INT_TXFF);

    // Configuration complete. Enable the module.
    SPI_enableModule(obj->spiHandle);
```

6. Configure the GPIOs for the other input and output pins, such as ENABLE, nFAULT. You might refer to the example codes in HAL_setupGPIOs() , HAL_setupGate() of the *hal.c* file and the defines in the *hal.h* file as follows:

```
//! \brief Defines the gpio for enabling Power Module
#define MTR1_GATE_EN_GPIO       29

//! \brief Defines the gpio for the nFAULT of Power Module
#define MTR1_PM_nFAULT_GPIO     34
```

7. Call the HAL_setupSPI() and HAL_setupGate() functions in HAL_MTR_setParams() of the *hal.c* file:

```
    // setup the spi for drv8323/drv8353/drv8316
    HAL_setupSPI(handle);

    // setup the drv8323s/drv8353s/drv8316s interface
    HAL_setupGate(handle);
```

8. Call the drivers functions in the *motor1_drive.c* file as the following:

```
    // turn on the DRV8323/DRV8353/DRV8316 if present
    HAL_enableDRV(obj->halMtrHandle);

    // initialize the DRV8323/DRV8353/DRV8316 interface
    HAL_setupDRVSPI(obj->halMtrHandle, &drvicVars_M1);
```

9. Change the default setting value for the BLDC motor driver, if needed.

```
    drvicVars_M1.ctrlReg05.bit.VDS_LVL = DRV8323_VDS_LEVEL_1P700_V;
    drvicVars_M1.ctrlReg05.bit.OCP_MODE = DRV8323_AUTOMATIC_RETRY;
    drvicVars_M1.ctrlReg05.bit.DEAD_TIME = DRV8323_DEADTIME_100_NS;
    drvicVars_M1.ctrlReg06.bit.CSA_GAIN = DRV8323_Gain_10VpV;

    drvicVars_M1.ctrlReg06.bit.LS_REF = false;
    drvicVars_M1.ctrlReg06.bit.VREF_DIV = true;
    drvicVars_M1.ctrlReg06.bit.CSA_FET = false;

    drvicVars_M1.writeCmd = 1;
    HAL_writeDRVData(obj->halMtrHandle, &drvicVars_M1);
```

4.3 Porting Reference Code to New C2000 MCU

The Motor Control Universal Lab project can be ported to other FPU and TMU enabled C2000 MCU controllers. Instructions on how to port the lab code are described in detail in the following steps. The F28004x MCU is used as the example for a new target C2000 MCU. To adapt a SysConfig-enabled version of the lab, replace `universal_motorcontrol_lab` with `universal_motorcontrol_syscfg` in all below instructions.

1. Browse to the `<install_location>\solutions\universal_motorcontrol_lab` folder and select one of the existing device-specific lab folders. The "f28002x" folder will be used in this example, but any can be utilized.
2. Create a copy of the selected device-specific lab in the same universal_motorcontrol_lab folder, changing the name to "f28004x". `<install_location>\solutions\universal_motorcontrol_lab\f28004x` will be the location of your final ported lab, and will be referred to as `<f28004x_lab_location>` below.
3. The compiler uses cmd files to map the memory of the C2000 MCU. Browse to the `<f28004x_lab_location>\cmd` folder and update the name of the *f28002x_flash_lib_is.cmd* files to reflect the new f28004x device. Note that there are several other cmd files present in this folder- these are unused by default and can be ignored.
4. The Universal Motor Control Lab utilizes the pin definitions present in the C2000Ware device driver files, *device.c/h.* These must be updated for the new device.
    a. Navigate to the C2000Ware common device support folder for the new C2000 MCU, located at `<install_location>\c2000ware\device_support\f28004x`. Locate the *device.h* file in the `...\common\include` subfolder.
    b. Copy the *device.h* file into the `<f28004x_lab_location>\drivers\include` folder, replacing the existing file.
    c. Navigate back to the C2000Ware common device support folder for the new C2000 MCU. Locate the *device.c* file in the `...\common\source` subfolder.
    d. Copy the *device.c* file into the `<f28004x_lab_location>\drivers\source` folder, replacing the existing file.
5. Browse to the `<f28004x_lab_location>\ccs\motor_control` folder and use an editor to open the projectspec file. This file is used by CCS to generate the project folder in the user workspace and includes references to device-specific C2000Ware source files.
    a. If the SysConfig-enabled version of the lab is being used, update the **bolded text** below to indicate the package of your new C2000 MCU. This line of text can be found within the device definition section of the file, which should be the first section.

```
sysConfigBuildOptions --product ${C2000WARE_ROOT}/.metadata/sdk.json --device F28002x --
package 80QFP --part F28002x_80QFP"
```

b. Some C2000 MCUs have different capabilities than others. Update the file to reflect these differences. For example, the F28002x MCU has Fast Integer Division (FINTDIV) support, while the F28004x does not. The relevant processor option is the 'idiv_support' term. Find and delete all instances of "--idiv_support=idiv0" since the F28004x doesn't support this function.
   i. If you are uncertain what changes may need to be made, refer to the TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS User's Guide, section 2.3 *Changing the Compiler's Behavior with Options,* table 2-1 *Processor Options*, which describes in detail what each option is used for.
   ii. Determine what differences exist between the that device and the new C2000 MCU that you have chosen. For assistance in this process, refer to the C2000 Real-Time Control Peripheral Reference Guide, which describes the differences between devices and peripheral versions.
   iii. Make adjustments as necessary in the projectspec file.

c. Find and replace all instances of "28002x" with "28004x" in the file.

d. Find all instances of "280025C" in the file.
   i. The first result should be near the beginning of the file, specifying the project device. Update the text in bold to correctly show the new C2000 MCU chosen for this project.

```
<project
    name="universal_motorcontrol_lab_f28004x"
    device="TMS320F280025C"
```

   ii. The final two results should be in the final section of the file. The following excerpts can be found in the 'path' for the ccxml file 'copy file' actions.

```
/TMS320F280025C_LaunchPad.ccxml
/TMS320F280025C.ccxml
```

   iii. Update the text in bold to correctly indicate the generic target configuration files for the new C2000 MCU's family of devices, which can be found in the previously referenced device support folder, in the `...\common\targetConfigs` subfolder. For all F28004x devices, the bolded text should be changed to "**TMS320F280049C**".
   iv. All other results for "280025C" should be in comments. Updating these is suggested for documentation accuracy, but is not critical.

6. Import the "universal_motorcontrol_lab_f28004x" project into CCS.
   a. Note that when the project is imported, CCS may present an error indicating that the *f28004x_headers_nonbios.cmd* file was not found. This error does not impact performance, although it may increase difficulty in debugging. The memory allocations performed in this file are utilized only by the debug environment watch window, described in Section 3.5.1.2 in the incremental build stages.
   b. In order to utilize the debug environment watch window fully, follow all instructions in this section related to the *f28002x_flash_lib_is.cmd* file with the *f28002x_headers_nonbios.cmd* file as well.
7. Open the cmd file(s) and change the memory map according to the device chosen. For those entirely unfamiliar with this type of file, refer to the TI Linker Command File Primer for an in-depth introduction and basic usage guide.
   a. It may be easier to adapt one of the generic C2000Ware cmd files, such as the *28004x_generic_flash_lnk.cmd* file, than it is to adapt the cmd file for the old device. These files are located in the `...\common\cmd` subfolder of the device support folder. In this case, the original cmd file for the project should be used as a reference.
   b. If using the *f28004x_headers_nonbios.cmd* file, the generic C2000Ware cmd file is located in the `...\headers\cmd` subfolder of the device support folder.
8. Modify the GPIO, PWM, ADC, and CMPSS modules and defines in the *hal.h* file as described in Section 4.1.2 for the F28004x based hardware kit.
9. Rebuild the lab project. Any errors or warnings in the project will be displayed in the CCS Console window. Follow the message prompts to fix any errors or warnings. There are a few differences in the driverlib APIs between devices which must be accounted for at this point.
10. To add functions to configure and use peripherals that are present in the new C2000 MCU that were not present in the original C2000 MCU source for these files, refer to the example functions in C2000Ware or the MotorControlSDK. For example, the F28004x has a Programmable Gain Amplifier (PGA), while the F28002x does not.
11. Run the project incrementally using the different build levels to test and verify functionality.

12. If this project will be imported multiple times, it is a good idea to update the project's source files so that these changes will only need to be done once. Navigate to the `<f28004x_lab_location>` folder.
    a. In the `...\cmd` subfolder, replace the *f28004x_flash_lib_is.cmd* file with the updated file in your imported F28004x project.
    b. In the `...\drivers\source` subfolder, replace the *hal.c* file with the updated file in your imported F28004x project.
    c. In the `...\drivers\include` subfolder, replace the *hal.h* file with the updated file in your imported F28004x project.

A Appendix A. Motor Control Parameters

The Universal Motor Control Lab has numerous defined parameters that impact system performance. The parameters available for user manipulation in the user_mtr1.h file are listed below. Some of these parameters are described as derived and generally should not be manipulated- the values for these parameters are dependent on one or more other parameters.

- **Hardware Parameters:**
  - **USER_M1_NOMINAL_DC_BUS_VOLTAGE_V**

    This parameter defines the nominal DC bus voltage in Volts (V).

    The value of this parameter must be greater than the rated motor voltage and less than the maximum sensing DC bus voltage of the hardware.
  - **USER_M1_ADC_FULL_SCALE_VOLTAGE_V**

    This parameter defines the maximum voltage at the input to the ADC in Volts (V). This value is represented by the maximum ADC reading.

    This parameter is used to scale the ADC readings, and should be set to the maximum expected voltage at the input to the ADC.
  - **USER_M1_ADC_FULL_SCALE_CURRENT_A**

    This parameter defines the maximum current at the input to the ADC in Amps (A). This value is represented by the maximum ADC reading.

    This parameter is used to scale the ADC readings, and should be set to the maximum expected current at the input to the ADC.
  - **USER_M1_VOLTAGE_FILTER_POLE_Hz**

    This parameter defines the analog filter pole location, in Hz.

    This parameter must be set to match the filter pole location of the hardware voltage feedback filter.
  - **USER_M1_VOLTAGE_FILTER_POLE_rps**

    This derived parameter is a radians per second conversion of USER_M1_VOLTAGE_FILTER_POLE_Hz.
  - **USER_M1_SIGN_CURRENT_SF**

    This parameter defines the sign (positive or negative) of the current scale factor derived from USER_M1_ADC_FULL_SCALE_CURRENT_A.

    If the noninverting (+) pin of the op-amp is grounded in the current feedback circuit, the value of this parameter is -1.0f. If the inverting pin (-) is grounded, the value is +1.0f.
  - **USER_M1_NUM_CURRENT_SENSORS**

    This parameter defines the number of current sensors present in the hardware.

    The calculations in this lab assume that there are either 2 or 3 current sensors. A higher or lower value will therefore cause a compilation error.
  - **USER_M1_NUM_VOLTAGE_SENSORS**

    This parameter defines the number of voltage phase sensors present in the hardware.

    The calculations in this lab assume that there are 3 voltage sensors. A higher or lower value will therefore cause a compilation error.
  - **USER_M1_Ix_OFFSET_AD**

    The USER_M1_Ix_OFFSET_AD parameter, where 'x' is A, B, or C, represents the ADC current offset for each respective phase, as defined by the hardware.

    This value should be close to 2048.

    The lab is capable of automatic offset calibration via the 'flagEnableOffsetCalc' flag. If set, the new value for this parameter is loaded into motorVars_M1.adcData.offset_I_ad.value[2:0] after the calibration process completes. The value in user_mtr1.h must be manually updated.

USER_M1_IA_OFFSET_AD; USER_M1_IB_OFFSET_AD; USER_M1_IC_OFFSET_AD

- **USER_M1_Vx_OFFSET_SF**

  The USER_M1_Vx_OFFSET_SF parameter, where 'x' is A, B, or C, represents the ADC voltage offset for each respective phase, as defined by the hardware. These parameters are used only for InstaSPIN-FOC FAST and InstaSPIN-BLDC.

  This value should be close to 0.5.

  The lab is capable of automatic offset calibration via the 'flagEnableOffsetCalc' flag. If set, the new value for this parameter is loaded into motorVars_M1.adcData.offset_V_sf.value[2:0] after the calibration process completes. The value in user_mtr1.h must be manually updated.

  USER_M1_VA_OFFSET_SF; USER_M1_VB_OFFSET_SF; USER_M1_VC_OFFSET_SF

- **USER_M1_IDC_OFFSET_AD**

  This parameter defines the ADC current offsets for single-shunt hardware.

  As with USER_M1_Ix_OFFSET_AD, this value can be calibrated via the 'flagEnableOffsetCalc' flag. The new value is loaded into motorVars_M1.adcData.offset_Idc_ad.

• **Timing Parameters:**

- **USER_SYSTEM_FREQ_Hz**

  This derived parameter (found in user_common.h) is a floating point conversion of the device SysClk frequency DEVICE_SYSCLK_FREQ in Hz. DEVICE_SYSCLK_FREQ is found in the device.h file.

- **USER_M1_PWM_FREQ_kHz**

  This parameter defines the frequency of the ePWM in kHz. Changing this value alters the control interrupt frequency.

- **USER_M1_PWM_PERIOD_usec**

  This derived parameter is the period of the ePWM frequency USER_M1_PWM_FREQ_kHz in microseconds.

- **USER_M1_ISR_FREQ_Hz**

  This derived parameter defines the control interrupt frequency in Hz.

  This parameter is derived from the PWM frequency USER_M1_PWM_FREQ_kHz.

- **USER_M1_ISR_PERIOD_usec**

  This derived parameter is the period of the control interrupt frequency USER_M1_ISR_FREQ_Hz in microseconds.

- **USER_M1_NUM_PWM_TICKS_PER_ISR_TICK**

  This parameter defines the number of PWM periods per interrupt. This value divides the control interrupt frequency.

  This lab assumes this value is between 1 and 3, inclusive.

- **USER_M1_NUM_ISR_TICKS_PER_SPEED_TICK**

  This parameter defines the number of interrupt ticks per speed controller ticks. This value divides the speed controller trigger rate. This contributes to system response time.

• **Sampling Parameters:**

- **USER_M1_CURRENT_SF**

  This derived parameter calculates the scale of the ADC result bits for current readings, relative to the actual current value in Amps (A). This calculation assumes a 12-bit ADC.

  This parameter is derived from the ADC full scale current USER_M1_ADC_FULL_SCALE_CURRENT_A.

- **USER_M1_VOLTAGE_SF**

  This derived parameter calculates the scale of the ADC result bits for voltage readings, relative to the actual voltage value in Volts (V). This calculation assumes a 12-bit ADC. This parameter is used only by the InstaSPIN-FOC FAST and eSMO estimators.

  This parameter is derived from the ADC full scale voltage USER_M1_ADC_FULL_SCALE_VOLTAGE_V.

- **USER_M1_CURRENT_INV_SF**

  This derived parameter is equivalent to 1 / USER_M1_CURRENT_SF for a 12-bit ADC.

  This parameter is derived from the ADC full scale current USER_M1_ADC_FULL_SCALE_CURRENT_A.
- **USER_M1_DCLINKSS_MIN_DURATION**

  This parameter defines the minimum duration in clock cycles for the single-shunt ADC readings.

  Calculations for the value of this parameter are described in the motor1_drive.c file.
- **USER_M1_DCLINKSS_SAMPLE_DELAY**

  This parameter defines the delay in clock cycles before the single-shunt ADC readings to account for signal propagation.

  Calculations for the value of this parameter are described in the motor1_drive.c file.

- **Motor Properties:**
  - **USER_MOTOR1_TYPE**

    This parameter defines the motor type.

    The two valid values for this parameter are MOTOR_TYPE_PM (for BLDC, PMSM, SMPM, or IPM motors) and MOTOR_TYPE_INDUCTION (for Asynchronous ACI motors)
  - **USER_MOTOR1_NUM_POLE_PAIRS**

    This parameter defines the number of pole pairs in the motor. This value is utilized to calculate motor output power when utilizing the InstaSPIN-FOC FAST estimator.
  - **USER_MOTOR1_Rr_Ohm**

    This parameter defines the rotor resistance of the motor in Ohms for induction motors. For non-induction motors, set to NULL. This parameter is used only by InstaSPIN-FOC FAST.
  - **USER_MOTOR1_Rs_Ohm**

    This parameter defines the stator resistance of the motor in Ohms.
  - **USER_MOTOR1_Ls_d_H**

    This parameter defines the stator inductance of the motor in the direct direction in henries (H). For PM, this is average stator inductance.
  - **USER_MOTOR1_Ls_q_H**

    This parameter defines the stator inductance of the motor in the quadrature direction in henries (H). For PM, this is average stator inductance.
  - **USER_MOTOR1_RATED_FLUX_VpHz**

    This parameter defines the rated flux of the motor in V/Hz (or V*s, Webers).
  - **USER_MOTOR1_MAGNETIZING_CURRENT_A**

    This parameter defines the rated current value of the motor in the direct direction in Amps for induction motors only. For other motors, set to NULL. This parameter is used only by InstaSPIN-FOC FAST.
  - **USER_MOTOR1_MAX_CURRENT_A**

    This parameter defines the maximum current of the motor in A. This contributes to overcurrent handling.
  - **USER_MOTOR1_INERTIA_Kgm2**

    This parameter defines the moment of inertia of the mass rigidly coupled with the motor in kg · m^2. This contributes to the speed controller gain constant calculation.
  - **USER_M1_VD_SF**

    This parameter defines the initial maximum value for Vd for the Id current controller. See USER_M1_MAX_VS_MAG_PU for more detail.

    This value must be between 0.1 and 0.95.
  - **USER_M1_MAX_VS_MAG_PU**

    This parameter defines the maximum magnitude for the Voltage vector Vs in V per-unit.

By the definition of a vector, the relationship between the magnitude of the Voltage vector Vs and the Vd and Vq axis components is as follows;

$Vs = \sqrt{(Vd^2 + Vq^2)}$

Vd is determined by the end application- for some motors, a value of 0 is a valid option. For this example, this is defined differently depending on whether closed or open-loop control is being used. In either case, the calculations for Vs and Vd are provided below.

In open-loop control, Vd is set to 0.3 in the voltage-frequency characterization profile.

In closed-loop control, Vd is dynamically determined by a PI controller. The maximum magnitude of |Vd| is initially set to USER_M1_VD_SF * USER_MOTOR1_RATED_VOLTAGE_V. During motor operation, it is instead set to USER_MOTOR1_RATED_VOLTAGE_V.

In open-loop control, Vs is set to USER_M1_MAX_VS_MAG_PU. All operations are in per-unit.

In closed-loop control, Vs is set to USER_M1_MAX_VS_MAG_PU * obj->adcData.VdcBus_V (the DC Bus Voltage). All operations are in Volts.

– **USER_M1_MAX_ACCEL_Hzps**

  This parameter defines the maximum acceleration magnitude for the estimation speed profiles in Hz per second. Used only during InstaSPIN Motor ID.

  This value should typically be left at the default.
– **InstaSPIN-FOC FAST Parameters:**
– **USER_MOTOR1_RES_EST_CURRENT_A**

  This parameter defines the maximum current value to be used for stator resistance estimation, in Amps.

  This value should be set to 10% to 40% of the rated phase current of the motor. If the motor does not spin during the ramp-up process, increase in 5% increments until the motor is in motion during the entire ramp-up process.
– **USER_MOTOR1_IND_EST_CURRENT_A**

  This parameter defines the maximum current value to use for stator inductance estimation, in Amps.

  This value should be set to 10%-20% of the rated phase current of the motor, just enough to enable rotation.
– **USER_MOTOR1_FLUX_EXC_FREQ_Hz**

  This parameter defines the flux excitation frequency used to estimate the stator inductance and flux during motor identification, in Hz.

  Typically, 5%-20% of the motor rated frequency is high enough for estimation for PMSM motors with ~10-20 microHenries stator inductance or higher. If the inductance is in the single digits of microHenries, then a higher frequency is recommended, up to 60Hz for very low inductances.
– **USER_M1_R_OVER_L_EXC_FREQ_Hz**

  This parameter defines the R/L excitation frequency in Hz, used during motor identification to estimate initial values for the stator resistance and inductance. This is used to calculate current controller gains.

  By default, this parameter is set to 300 Hz.
– **USER_M1_IDRATED_FRACTION_FOR_L_IDENT**

  This parameter defines the fraction of the rated Id to use during inductance estimation.

  This parameter should not be changed from the default value of 0.5.
– **USER_M1_SPEEDMAX_FRACTION_FOR_L_IDENT**

  This parameter defines the fraction of the max speed to use during inductance estimation.

  This parameter should not be changed from the default value of 1.0.
– **USER_M1_R_OVER_L_KP_SF**

  This parameter defines the scale factor for Kp in the FOC current controllers.

This parameter should not be changed from the default value of 0.02.

- **USER_M1_IDRATED_DELTA_A**

  This parameter defines the Id delta current to use during estimation, in A.

  This parameter should not be changed from the default value of 0.0001.

- **USER_M1_FORCE_ANGLE_FREQ_Hz**

  If the force angle startup option is enabled, this parameter defines the force angle startup frequency in Hz. This value is used during motor startup.

  Typical force angle startup speed is +-1 Hz.

- **USER_MOTOR1_FREQ_NEARZEROLIMIT_Hz**

  If the force angle startup option is enabled, this parameter defines the speed range within which the force-angle startup process is used.

  For most applications, the default value of 5Hz is sufficient.

- **USER_M1_PW_GAIN**

  This parameter defines the PowerWarp gain for computing the Id reference value for induction motors.

  This parameter should not be changed from the default value of 1.0.

- **USER_M1_DCBUS_POLE_rps**

  This parameter defines the pole location for the software DC bus filter in radians/second.

  This parameter should not be changed from the default value of 100.

- **USER_M1_SPEED_POLE_rps**

  This parameter defines the pole location for the software frequency estimator filter in radians/second.

  For most applications, the default value of 100rps is sufficient. For high-speed motors, performance may be improved by increasing this value up to 500rps.

- **USER_M1_EST_FLUX_HF_SF**

  This parameter defines the scale factor for flux estimation.

  This parameter can range from 0.1 to 1.25. For most PMSM motors, a value of 1.0 is sufficient. For higher frequency lower inductance motors, a lower value may be used.

- **USER_M1_EST_BEMF_HF_SF**

  This parameter defines the scale factor for Back-EMF (BEMF) estimation.

  If using the InstaSPIN-FOC FAST estimator, this parameter should not be chagned from the default value of 1.

  Otherwise, this parameter can range from from 0.5 to 1.25. For most PMSM motors, a value of 1.0 is sufficient. For higher frequency lower inductance motors, a lower value may be used.

- **USER_MOTOR1_Ls_d_COMP_COEF**

  This parameter defines the Ls d-axis compensation coefficient. Motor inductance decreases with respect to the amplitude of the applied stator current. To simplify calculations, a linear relationship is assumed.

  Where Ls is the motor stator inductance Ld and Ls^1 is the motor stator inductance after compensation,

  Ls^1 = Ls * (1 -(Ls Compensation Coefficient))

  This parameter should be set according to the inductance vs current model provided by the motor manufacturer.

- **USER_MOTOR1_Ls_q_COMP_COEF**

  This parameter is identical to USER_MOTOR1_Ls_d_COMP_COEF, except it refers to Lq instead of Ld.

- **USER_MOTOR1_Ls_MIN_NUM_COEF**

  This parameter defines the minimum inductance of the motor.

This parameter should be set according to the inductance vs current model provided by the motor manufacturer.

- **USER_MOTOR1_RSONLINE_WAIT_TIME**

  Rs online calibration is used to recalibrate the stator resistance Rs while the motor is running in closed-loop. However, given that Rs changes slowly over time, and the Rs online calibration process injects d-axis current that does not generate torque, the process is only enabled intermittently.

  This parameter defines the wait time between Rs Online calibration cycles in 5ms periods.

  This parameter's default value is typically sufficient.
- **USER_MOTOR1_RSONLINE_WORK_TIME**

  This parameter defines the duration of the Rs Online calibration cycle in 5ms periods.

  This parameter's default value is typically sufficient.

• **Open Loop Voltage/Frequency Profile Parameters:**

- **USER_MOTOR1_FREQ_LOW_Hz**

  This parameter defines the low frequency boundary of the open-loop Voltage/Frequency profile in Hz.

  This value defines the lower limits of the linear region of the Voltage/Frequency profile of the motor. Within the linear region, to calculate the applied voltage necessary to achieve a chosen motor speed is incredibly simple. Beyond the lower limit, this relationship is no longer linear.

  This parameter should be set to approximately 10% of the rated motor frequency.
- **USER_MOTOR1_FREQ_HIGH_Hz**

  This parameter defines the high frequency boundary of the open-loop Voltage/Frequency profile in Hz.

  This value defines the upper limits of the linear region of the Voltage/Frequency profile of the motor. Within the linear region, to calculate the applied voltage necessary to achieve a chosen motor speed is incredibly simple. Voltage should not increase beyond the rated motor voltage.

  This parameter should be set to the rated motor frequency.
- **USER_MOTOR1_VOLT_MIN_V**

  This parameter defines the voltage value that generates a speed of USER_MOTOR1_FREQ_LOW_Hz in the motor, in V.

  This parameter should be set to approximately 15% of the rated motor voltage.
- **USER_MOTOR1_VOLT_MAX_V**

  This parameter defines the voltage value that generates a speed of USER_MOTOR1_FREQ_HIGH_Hz in the motor, in V.

  This parameter should be set to the rated motor voltage.

• **Motor Encoder Parameters:**

- **USER_MOTOR1_NUM_ENC_SLOTS**

  This parameter defines the number of encoder slots in the motor quadrature encoder.

  This parameter should be set according to the encoder used.
- **USER_MOTOR1_ENC_POS_MAX**

  This derived parameter defines the maximum value of the position counter for the eQEP module.

  This parameter is derived from the number of encoder slots (USER_MOTOR1_NUM_ENC_SLOTS).
- **USER_MOTOR1_ENC_POS_OFFSET**

  This parameter defines the offset number of the encoder at position zero.

• **eSMO Estimator Parameters:**

- **USER_MOTOR1_KSLIDE_MAX**

  This parameter defines the maximum value of the time-dependent eSMO Kslide gain variable.

This parameter must be set greater than the maximum back-EMF at the maximum motor frequency, in per-unit. The default value is sufficient for most applications. This parameter may vary between 0.1 and 10.

- **USER_MOTOR1_KSLIDE_MIN**

  This parameter defines the initial value of the time-dependent eSMO Kslide gain variable.

  This parameter must be set greater than the maximum back-EMF at the maximum motor frequency, in per-unit. The default value is sufficient for most applications. This parameter may vary between 0.1 and 10.
- **USER_MOTOR1_PLL_KP_MAX**

  This parameter defines the maximum gain for the eSMO PLL.

  This parameter must be set greater than the maximum back-EMF at the maximum motor frequency, in per-unit. The default value is sufficient for most applications. This parameter may vary between 0.1 and 10.
- **USER_MOTOR1_PLL_KP_MIN**

  This parameter defines the minimum gain for the eSMO PLL.

  This parameter must be set greater than the maximum back-EMF at the maximum motor frequency, in per-unit. The default value is sufficient for most applications. This parameter may vary between 0.1 and 5.
- **USER_MOTOR1_PLL_KP_SF**

  This parameter defines the scale factor applied to the per-unit motor speed. This controls the gain of the eSMO PLL.

  This parameter should be set to (USER_MOTOR1_PLL_KP_MAX - USER_MOTOR1_PLL_KP_MIN) / (fscale * fmax). In most cases, the default value is sufficient.
- **USER_MOTOR1_BEMF_THRESHOLD**

  This parameter defines the threshold of estimated current error for the eSMO sliding mode controller.

  This parameter should be equal to the maximum motor Back-EMF (BEMF) divided by the motor's rated voltage, and should range between 0.3 and 0.5. In most cases, the default value of 0.5 is sufficient.
- **USER_MOTOR1_BEMF_KSLF_FC_Hz**

  This parameter defines the cutoff frequency of the low pass filter for the eSMO estimated back-EMF.

  This parameter is equal to the cutoff frequency * 2 * PI() * Ts (timescale). In most cases, the default value is sufficient.
- **USER_MOTOR1_THETA_OFFSET_SF**

  This parameter defines the coefficient applied to the back-EMF filter compensation factor.

  This parameter should be set between 0.5 to 1.5, inclusive. The default value of 1 is sufficient in most cases.
- **USER_MOTOR1_SPEED_LPF_FC_Hz**

  This parameter sets the cutoff frequency of the low-pass filter used to calculate speed. Given that this filter is applied to per-unit values, this parameter only influences the eSMO output in very high noise scenarios.

  The default value of 200Hz is sufficient in most cases. Otherwise, this parameter should be set between 100 and 400.

- **InstaSPIN-BLDC Parameters:**
  - **USER_MOTOR1_RAMP_START_Hz**

    This parameter defines the minimum speed when the motor runs in open-loop, in Hz.

    The specific value for this parameter should be based on motor or system test on performance requirements. This value could be 0.5%-1% of the rated motor speed.
  - **USER_MOTOR1_RAMP_END_Hz**

    This parameter defines when the BEMF zero-cross point for commutation begins to be considered, in Hz.

The specific value for this parameter should be based on motor or system test on performance requirements. This value could be 10%-20% of the rated motor speed.

- **USER_MOTOR1_RAMP_DELAY**

  This parameter defines how long the system runs the motor in open-loop, in seconds.

  The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 5-20s.
- **USER_MOTOR1_ISBLDC_INT_MAX**

  This parameter defines the maximum threshold of integration voltage for commutation point detection, percent of the nominal.

  The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 0.01-0.15.
- **USER_MOTOR1_ISBLDC_INT_MIN**

  This parameter defines the minimum threshold of integration voltage for commutation point detection, percent of the nominal.

  The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 0.005-0.1.
- **USER_MOTOR1_ISBLDC_I_START_A**

  This parameter defines the current to start the motor in open-loop, in A.

  The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 5%-20% of the rated current.
- **USER_MOTOR1_ISBLDC_DUTY_START**

  This parameter defines the PWM duty to start the motor in open-loop, in percent.

  The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 5%-20% duty.

- **Field Weakening Control (FWC) Parameters:**
  - **USER_M1_FWC_KP**

    This parameter defines the Kp gain of the PI regulator for FWC.

    The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 0.01-0.1.
  - **USER_M1_FWC_KI**

    This parameter defines the Ki gain of the PI regulator for FWC.

    The specific value for this parameter should be based on motor or system test on performance requirements. This value could be between 0.01-0.1.
  - **USER_M1_FWC_MAX_ANGLE**

    This parameter defines the maximum vector angle for FWC, in degrees.

    This parameter should be defined between 0 and -45 degrees. A large negative value will set a higher negative current on the d-axis for FWC.
  - **USER_M1_FWC_MIN_ANGLE**

    This parameter defines the minimum vector angle for FWC, in degrees.

    This parameter should not be changed from its default value of 0.
- **Startup and Running State Parameters:**
  - **USER_MOTOR1_RATED_VOLTAGE_V**

    This parameter defines the rated voltage of the motor, in V.

    This parameter could be set to USER_M1_NOMINAL_DC_BUS_VOLTAGE_V / sqrt(2).

- **USER_MOTOR1_FREQ_MAX_Hz**

  This parameter defines the maximum rotational speed of the motor in Hz.

  This parameter is system-dependent, and may be higher than the rated speed of the motor. A value of 100% to 150% of the rated speed of the motor is typical. Specific values should be decided according to system performance requirements.
- **USER_MOTOR1_ALIGN_CURRENT_A**

  This parameter defines the current used to run motor rotor alignment, in A.

  This parameter should be set to 5%-50% of the rated current of the motor. Specific values should be decided according to system performance requirements.
- **USER_MOTOR1_FLUX_CURRENT_A**

  This parameter defines the current used to run the motor in forced open-loop, in A.

  This parameter should be set to 5%-50% of the rated current of the motor. Specific values should be decided according to system performance requirements.
- **USER_MOTOR1_STARTUP_CURRENT_A**

  This parameter defines the current, in A, used to run the motor in closed-loop when the speed is below the defined startup speed (USER_MOTOR1_SPEED_START_Hz).

  This parameter should be set to 10%-100% of the rated motor current.
- **USER_MOTOR1_TORQUE_CURRENT_A**

  This parameter defines the initially assigned value for motor startup current, in A.
- **USER_MOTOR1_SPEED_START_Hz**

  This parameter defines the startup speed threshold of the motor, in Hz.

  This parameter should be set higher than the minimum rotation speed of the motor, and typically falls between 10%-50% of the rated motor speed.
- **USER_MOTOR1_SPEED_FORCE_Hz**

  This parameter defines the force open-loop speed threshold of the motor, in Hz.

  This parameter should be set higher than the minimum rotation speed of the motor, and typically falls between 5%-30% of the rated motor speed.
- **USER_MOTOR1_ACCEL_START_Hzps**

  This parameter defines the motor acceleration during startup.

  This parameter should be set less than the maximum acceleration of the motor (USER_MOTOR1_ACCEL_MAX_Hzps), and otherwise should be set according to system performance requirements.
- **USER_MOTOR1_ACCEL_MAX_Hzps**

  This parameter defines the maximum acceleration of the motor.

  This parameter should be set according to system performance requirements and motor hardware limitations.
- **USER_MOTOR1_SPEED_FS_Hz**

  This parameter defines the flying-start speed threshold of the motor, in Hz.

  This parameter should be set higher than the minimum rotation speed of the motor, and typically falls between 0.1%-5% of the rated motor speed.
- **USER_MOTOR1_BRAKE_CURRENT_A**

  When braking is enabled, this parameter defines the motor current during the braking state.

  This parameter should be set between 10%-50% of the rated motor current, according to system performance requirements.

- **USER_MOTOR1_BRAKE_TIME_DELAY**

  This parameter defines the delay of the motor braking state in 5ms periods.

  This parameter should be set according to system performance requirements, and typically falls between 1-300s, converted to 5ms periods.
- **USER_M1_STOP_WAIT_TIME_SET**

  This parameter defines the minimum waiting time to start the motor from a stopped state, in 5ms periods.

  This parameter should be set higher than 0.1s, converted to 5ms periods.
- **USER_M1_RESTART_WAIT_TIME_SET**

  This parameter defines the minimum waiting time to start the motor from a fault state, in 5ms periods.

  This parameter should be set higher than 0.1s, converted to 5ms periods.
- **USER_M1_START_TIMES_SET**

  This parameter defines the duration of the motor rotor alignment state, in 5ms periods.

  This parameter should be set between 1s-30s, converted to 5ms periods.

- **Motor Protection Parameters:**
  - **USER_M1_IS_OFFSET_AD_DELTA**

    This parameter defines the error threshold for the ADC offset for the phase currents. If offset calibration is enabled, the system checks to ensure that all ADC offset values for current measurement circuits are within this delta from the initially defined parameters (USER_M1_Ix_OFFSET_AD). If the calculated ADC offset is beyond this delta from the initially defined parameter, this indicates a software error or hardware fault, and the system enters a fault handling state. This parameter is not used when utilizing single-shunt measurement.

    This parameter should be set between 0 to 200, inclusive, to allow sufficient range while still catching errors. To achieve a lower error delta, a higher accuracy sensing circuit may be required.
  - **USER_M1_VA_OFFSET_SF_DELTA**

    This parameter defines the error threshold for the ADC offset for the phase voltages. If offset calibration is enabled, the system checks to ensure that all ADC offset values for voltage measurement circuits are within this delta from the initially defined parameters (USER_M1_Vx_OFFSET_SF). If the calculated ADC offset is beyond this delta from the initially defined parameter, this indicates a software error or hardware fault, and the system enters a fault handling state. This parameter is only used for InstaSPIN-FOC FAST and InstaSPIN-BLDC.

    This parameter should be set between 0 to 200, inclusive, to allow sufficient range while still catching errors. To achieve a lower error delta, a higher accuracy sensing circuit may be required.
  - **USER_M1_OVER_VOLTAGE_FAULT_V**

    This parameter defines the over-voltage threshold for the motor and inverter hardware, in Volts. A DC bus voltage greater than this value indicates a software error or hardware fault has resulted in the calculated DC bus voltage going above the user-defined safe or expected range, and the system enters a fault handling state.

    This parameter should be set to a hardware-dependent value less than the full scale ADC voltage (USER_M1_ADC_FULL_SCALE_VOLTAGE_V).
  - **USER_M1_OVER_VOLTAGE_NORM_V**

    This parameter defines the lower threshold that indicates that a DC bus over-voltage fault (see USER_M1_OVER_VOLTAGE_FAULT_V) is no longer present, in V.

    This parameter should be set to a hardware-dependent value less than the over-voltage fault threshold (USER_M1_OVER_VOLTAGE_FAULT_V). Typically, this is several Volts below the over-voltage fault threshold, to ensure that the system has reliably returned to a safe or expected state.

- **USER_M1_UNDER_VOLTAGE_FAULT_V**

  This parameter defines the under-voltage threshold for the motor and inverter hardware, in Volts. A DC bus voltage greater than this value indicates a software error or hardware fault has resulted in the calculated DC bus voltage going below the user-defined safe or expected range, and the system enters a fault handling state.

  This parameter should be set to a hardware-dependent value, typically, a few Volts above the minimum required to spin the motor.
- **USER_M1_UNDER_VOLTAGE_NORM_V**

  This parameter defines the upper threshold that indicates that a DC bus under-voltage fault (see USER_M1_UNDER_VOLTAGE_FAULT_V) is no longer present, in V.

  This parameter should be set to a hardware-dependent value greater than the under-voltage fault threshold (USER_M1_UNDER_VOLTAGE_FAULT_V). Typically, this is a few Volts above the under-voltage fault threshold, to ensure that the system has reliably returned to a safe or expected state.
- **USER_M1_LOST_PHASE_CURRENT_A**

  This parameter defines the current threshold for the lost phase fault, in A. Beyond the minimum speed threshold for this error (USER_M1_FAIL_SPEED_MIN_HZ), the absolute value of any phase currents being beneath this value indicates a software error or hardware fault, and the system enters a fault handling state.

  This parameter should be set to a motor-dependent value according to system performance requirement. Typically, this value is between 0.1% to 10% of the rated current of the motor.
- **USER_M1_UNBALANCE_RATIO**

  This parameter defines the ratio threshold indicating a phase unbalance fault. If the ratio between the largest to smallest RMS phase current value is beyond this ratio, this indicates a software error or hardware fault, and the system enters a fault handling state.

  This parameter should be set to a motor-dependent value according to system performance requirement. Typically, this value is a percent value between 5% and 25%. The default value of 20% should typically be sufficient for this application.
- **USER_MOTOR1_OVER_CURRENT_A**

  This parameter defines the over-current threshold of the motor, in A. This value is used to calculate the CMPSS peripheral's DAC values, which in turn trigger the PWM's trip zone fault handling.

  This parameter should be set to a motor and system-dependent value that is typically 50% to 300% of the rated motor current.

  NOTE: If this value of this parameter is set greater than the maximum peak current value of the motor, defined as 47.5% of the ADC full scale current in this example, this parameter is ignored and the maximum peak current is used instead.
- **USER_M1_OVER_LOAD_POWER_W**

  This parameter defines the over-power threshold for the motor and inverter hardware, in W. In the motor running state, if the calculated power of the motor is above this value, this indicates a software error or hardware fault, and the system enters a fault-handling state.

  This parameter should be set to a motor-dependent value, typically 50% to 200% of the rated motor power. This value must be less than the maximum output power that the motor can produce without exceeding its thermal rating.
- **USER_M1_STALL_CURRENT_A**

  This parameter defines the current threshold for the RMS motor stator current, in A, when the calculated motor speed is beneath the speed threshold (USER_M1_FAIL_SPEED_MIN_HZ). When the motor current is above the fault checking threshold (see USER_M1_FAULT_CHECK_CURRENT_A), this indicates the minimum value for the motor stator current, and values beneath this parameter indicates a failed motor startup fault condition. A value above this parameter always indicates a motor stall fault condition.

This parameter should be set to a motor-dependent value, typically 50% to 300% of the motor rated current. This value should be lower than the maximum peak current of the motor, defined as 47.5% of the ADC full scale current in this example.

Note

The failed motor startup and motor stall conditions should not both be active.

– **USER_M1_FAULT_CHECK_CURRENT_A**

This parameter defines the current threshold for several motor faults, in A. When the RMS motor stator current is below this value, the unbalanced phase current fault, over-speed fault, and failed startup faults are not checked.

This parameter should be set to a motor-dependent value according to system performance requirements. This value is typically 0.1% to 20% of the rated motor current.

– **USER_M1_FAIL_SPEED_MAX_HZ**

This parameter defines the maximum speed of the motor in Hz. If the calculated speed is above this value, this indicates a software error or hardware fault, and the system enters a fault-handling state.

This parameter should be set to a motor-dependent value according to system performance requirements. This value is typically 200% to 500% of the rated speed of the motor.

– **USER_M1_FAIL_SPEED_MIN_HZ**

This parameter defines the minimum speed threshold of the motor, in Hz. Certain faults are only checked when the motor is above or below this value. The stall current and failed motor startup faults are only checked when the motor speed is below this value. The lost phase fault is only checked when the motor speed is above this value.

This parameter should be set to a motor-dependent value according to system performance requirements. This value is typically 1% to 10% of the rated motor speed, and must be higher than the minimum rotation speed of the motor.

– **USER_M1_VOLTAGE_FAULT_TIME_SET**

This parameter defines the duration time to set or clear the over and under-voltage faults, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set. When the fault condition is not present for longer than this period, the error flag is reset.

This parameter should be set according to system performance requirements. This value is typically between 0.02 to 3.0s. The default value should be sufficient for most applications.

– **USER_M1_OVER_LOAD_TIME_SET**

This parameter defines the duration time to set the over load power fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

This parameter should be set according to system performance requirements. This value is typically between 0.01 to 1.0s. The default value should be sufficient for most applications.

– **USER_M1_STALL_TIME_SET**

This parameter defines the duration time to set the motor stall current fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

This parameter should be set according to system performance requirements. This value is typically between 0.01 to 2.0s. The default value should be sufficient for most applications.

– **USER_M1_UNBALANCE_TIME_SET**

This parameter defines the duration time to set the motor unbalanced phase current fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

This parameter should be set according to system performance requirements. This value is typically between 0.05 to 5.0s. The default value should be sufficient for most applications.

– **USER_M1_LOST_PHASE_TIME_SET**

This parameter defines the duration time to set the motor lost phase current fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

This parameter should be set according to system performance requirements. This value is typically between 0.05 to 5.0s. The default value should be sufficient for most applications.

– **USER_M1_OVER_SPEED_TIME_SET**

  This parameter defines the duration time to set the motor over-speed fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

  This parameter should be set according to system performance requirements. This value is typically between 0.05 to 5.0s. The default value should be sufficient for most applications.

– **USER_M1_STARTUP_FAIL_TIME_SET**

  This parameter defines the duration time to set the failed motor startup fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

  This parameter should be set according to system performance requirements. This value is typically between 1.0s to 20.0s. The default value should be sufficient for most applications.

– **USER_M1_OVER_CURRENT_TIMES_SET**

  This parameter defines the duration time to set the motor over-current fault, in 5ms periods. When the fault condition is detected for longer than this period, the error flag is set.

  This parameter should be set according to system performance requirements. This value is typically between 0.01s to 0.5s. The default value should be sufficient for most applications.

- **PI Regulator Gain Tuning Parameters:**

  – *General PI Tuning Notes*

  Typical initial Kp and Ki of the speed and current regulators are calculated based on motor parameters. These runtime values for the gain will be changed according to the motor running speed or vector current after startup.

  The gain of the PI regulator is set according to the following rules:

Kp Gain	Ki Gain	Condition
Kp* = Kp * G(p_start)	Ki* = Ki * G(i_start)	Motor startup
Kp* = Kp * G(p_low)	Ki* = Ki * G(i_low)	After startup and condition low
Kp* = Kp * G(p_high)	Ki* = Ki * G(i_high)	After startup and condition high
Kp* = Kp * (G(p_low) + H(p_low) * (condition state))	Ki* = Ki * (G(i_low) + H(i_low) * (condition state))	No other conditions matched

Where:

- H(p_slope) = (G(p_high) - G(p_low)) / (condition range)
- H(i_slope) = (G(i_high) - G(i_low)) / (condition range)
- Condition low =
  – Speed regulator: ω(e) < ω(e_low)
  – Current regulator: i(s) < i(s_low)
- Condition high =
  – Speed regulator: ω(e) > ω(e_high)
  – Current regulator: i(s) > i(s_high)
- Condition state =
  – Speed regulator: ω(e) - ω(e_low)
  – Current regulator: i(s) -i(s_low)
- Condition range =
  – Speed regulator: ω(e_high) - ω(e_low)
  – Current regulator: i(s_high) -i(s_low)

- **USER_MOTOR1_GAIN_SPEED_LOW_Hz**

  This parameter defines the low speed threshold for adjusting Kp* and Ki* of the speed PI regulator.

  This parameter should be set between 10% to 30% of the rated speed of the motor, according to motor and application requirements.
- **USER_MOTOR1_GAIN_SPEED_HIGH_Hz**

  This parameter defines the high speed threshold for adjusting Kp* and Ki* of the speed PI regulator.

  This parameter should be set between 60% to 100% of the rated speed of the motor, according to motor and application requirements.
- **USER_MOTOR1_Kx_SPD_START_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_start to adjust the Kx* of the speed PI regulator for startup.

  This parameter should be set between 0.1 to 2.0 according to motor and application requirements, and should be lower than Gx_low in most application.

  USER_MOTOR1_Kp_SPD_START_SF; USER_MOTOR1_Ki_SPD_START_SF
- **USER_MOTOR1_Kx_SPD_LOW_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_low to adjust the Kx* of the speed PI regulator.

  This parameter should be set between 0.1 to 10.0 according to motor and application requirements, and should generally be higher than Gx_high.

  USER_MOTOR1_Kp_SPD_LOW_SF; USER_MOTOR1_Ki_SPD_LOW_SF
- **USER_MOTOR1_Kx_SPD_HIGH_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_high to adjust the Kx* of the speed PI regulator.

  This parameter should be set between 0.1 to 5.0 according to motor and application requirements, and should generally be lower than Gx_low.

  USER_MOTOR1_Kp_SPD_HIGH_SF; USER_MOTOR1_Ki_SPD_HIGH_SF
- **USER_MOTOR1_GAIN_IQ_LOW_A**

  This parameter defines the low current threshold is_low to adjust the gains of the q-axis current PI controller.

  This parameter should be set to 10% to 50% of the rated current of the motor, according to system performance, motor, and application requirements.
- **USER_MOTOR1_GAIN_IQ_HIGH_A**

  This parameter defines the high current threshold is_high to adjust the gains of the q-axis current PI controller.

  This parameter should be set to 50% to 100% of the rated current of the motor, according to system performance, motor, and application requirements.
- **USER_MOTOR1_Kx_IQ_START_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_start to adjust the gains of the q-axis current PI controller during startup.

  This parameter should be set to 0.1 to 5.0 according to system performance, motor, and application requirements.

  USER_MOTOR1_Kp_IQ_START_SF; USER_MOTOR1_Ki_IQ_START_SF
- **USER_MOTOR1_Kx_IQ_LOW_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_low to adjust the gains of the q-axis current PI controller during startup.

This parameter should be set to 0.1 to 10.0 according to system performance, motor, and application requirements, and should be higher than Gx_high.

USER_MOTOR1_Kp_IQ_LOW_SF; USER_MOTOR1_Ki_IQ_LOW_SF

- **USER_MOTOR1_Kx_IQ_HIGH_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx_high to adjust the gains of the q-axis current PI controller during startup.

  This parameter should be set to 0.1 to 5.0 according to system performance, motor, and application requirements, and should be lower than Gx_low.

  USER_MOTOR1_Kp_IQ_HIGH_SF; USER_MOTOR1_Ki_IQ_HIGH_SF
- **USER_MOTOR1_Kx_ID_SF**

  Where Kx is either Kp or Ki, this parameter defines the gain coefficient Gx to adjust the gains of the d-axis current PI controller.

  This parameter should be set between 0.1 to 5.0 according to system performance, motor, and application requirements.

  USER_MOTOR1_Kp_ID_SF; USER_MOTOR1_Ki_ID_SF

• **Potentiometer Control Parameters:**
  - **USER_M1_POT_ADC_MIN**

    When potentiometer speed control is enabled, this parameter defines the minimum allowable ADC value for the potentiometer. If the ADC reading is below this value, the motor speed is set to 0.

    This parameter should be set such that ADC noise cannot result in a value above this when the motor is intended to be disabled.
  - **USER_M1_POT_ADC_MAX**

    When potentiometer speed control is enabled, this parameter defines the maximum allowable ADC value for the potentiometer. If the ADC reading is above this value, the motor speed is set to the maximum.

    This parameter should be set to the maximum possible ADC value (4096 in this example) offset by USER_M1_POT_ADC_MIN. In this example, that's 4096U-200U.
  - **USER_M1_POT_SPEED_SF**

    When potentiometer speed control is enabled, this derived parameter defines the exact relationship between ADC reading and motor speed.

    This parameter is derived from the minimum and maximum potentiometer ADC values (USER_M1_POT_ADC_MIN and USER_M1_POT_ADC_MAX), as well as the motor maximum speed (USER_MOTOR1_FREQ_MAX_Hz).
  - **USER_M1_POT_SPEED_MIN_Hz**

    When potentiometer speed control is enabled, this derived parameter defines the minimum frequency of the motor in Hz. When the potentiometer ADC reading results in a nonzero value that is below this speed, the motor speed is set to this value instead.

    This parameter is set to 10% of the motor maximum speed (USER_MOTOR1_FREQ_MAX_Hz).
  - **USER_M1_POT_SPEED_MAX_Hz**

    When potentiometer speed control is enabled, this derived parameter defines the maximum frequency of the motor in Hz. When the potentiometer ADC reading results in a value that is above this speed, the motor speed is set to this value instead.

    This parameter is set to 50% of the motor maximum speed (USER_MOTOR1_FREQ_MAX_Hz).
  - **USER_M1_WAIT_TIME_SET**

    When potentiometer speed control is enabled, this parameter defines the wait period between speed updates in 1ms periods.

    This parameter should be set to a value which allows for rapid adjustments, while not allowing noise to excessively influence calculations. The default value of 500ms is sufficient for most applications.

- **Speed Pulse Control Parameters:**
  - **USER_M1_SPEED_CAP_MIN_Hz**

    When cmd pulse speed control is enabled, allowing the motor speed to be calculated from a speed pulse on an input GPIO, this parameter defines the minimum frequency of the motor in Hz. When the calculated motor speed is below this value, the motor speed is instead set to 0.

    This parameter should be set according to the minimum motor rotation speed, and by default is not derived automatically.
  - **USER_M1_SPEED_CAP_MAX_Hz**

    When cmd pulse speed control is enabled, allowing the motor speed to be calculated from a speed pulse on an input GPIO, this parameter defines the maximum frequency of the motor in Hz. When the calculated motor speed is above this value, the motor speed is instead set to 0.

    This parameter should be set according to the maximum rated motor speed, and by default is not derived automatically.

    Note

    A calculated speed above the maximum does NOT set the speed to the maximum as it does in the potentiometer control mode, instead it disables the motor.
  - **USER_M1_CAP_WAIT_TIME_SET**

    When cmd pulse speed control is enabled, allowing the motor speed to be calculated from a speed pulse on an input GPIO, this parameter defines the wait period between speed updates in 1ms periods.

    This parameter should be set to a value which allows for rapid adjustments, while not allowing noise to excessively influence calculations. The default value of 200ms is sufficient for most applications.
- **Switch Control Parameter:**
  - **USER_M1_SWITCH_WAIT_TIME_SET**

    When cmd switch control is enabled, allowing the motor to be enabled or disabled with a switch, this parameter defines the minimum wait period between switch state updates in 1ms periods.

    This parameter should be set to a value which allows for rapid adjustments, while not allowing noise to excessively influence calculations. The default value of 50ms is sufficient for most applications.

References

- Texas Instruments: *Getting Started With C2000™ Real-Time Control Microcontrollers (MCUs)*
- Texas Instruments: *The Essential Guide for Developing With C2000 Real-Time Microcontrollers*
- Texas Instruments: *Hardware Design Guidelines for TMS320F28xx and TMS320F28xxx*
- Texas Instruments: *C2000 MCU JTAG Connectivity Debug*
- Texas Instruments: *Using PWM Output as a Digital-to-Analog Converter on a TMS320F280x*
- Texas Instruments: *Sensorless-FOC for PMSM With Single DC-Link Shunt*
- Texas Instruments: *C2000™ Software Frequency Response Analyzer (SFRA) Library User's Guide*
- Texas Instruments: *C2000Ware motor control SDK getting started guide*
- General information on C2000 real-time MCUs - C2000™ Overview
- C2000 Products - C2000™ Products
- C2000 Design and Development Resources – C2000™ Design & Development

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (September 2021) to Revision A (April 2024) **Page**

- Added Section 2 ... 4
- Updated Section 3.1: Added support for the following TI hardware: F28003x LP & CC, F280013x LP & CC, and DRV8329AEVM ... 24
- Updated Section 3.2. Added support for the following TI hardware: F28003x LP & CC, F280013x LP & CC, and DRV8329AEVM ... 27
- Updated Section 3.4. Instructions adjusted for clarity ... 55
- Updated Section 3.5. Instructions adjusted for clarity ... 60
- Updated Section 4. Instructions adjusted for clarity ... 84
- Added Appendix A ... 104