

**ABSTRACT**

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

Table of Contents

1 Introduction	3
2 Running the Universal Lab on TI Hardware Kit	3
2.1 Supported TI Motor Evaluation Kits	3
2.2 Hardware Board Setup	5
2.3 Lab Software Implementation	18
2.4 Monitoring Feedback or Control Variables	26
2.5 Running the Project with Incremental Build	30
3 Building Custom Board	54
3.1 Building New Custom Board	54
3.2 Supporting New BLDC Motor Driver Board	67
3.3 Porting Reference Code to New C2000 MCU	70
4 References	71

List of Figures

Figure 2-1. F28002x LaunchPad Board Overview and Switches Setting	6
Figure 2-2. F28002x controlCARD and Switches Setting	7
Figure 2-3. TMDSDAP180TO100 Adapter and Switches Setting	7
Figure 2-4. LAUNCHXL-F280025C Connects BOOSTXL-DRV8323RH and DAC128S085EVM	8
Figure 2-5. LAUNCHXL-F280025C Board Settings For Connecting BOOSTXL-DRV Board	9
Figure 2-6. BOOSTXL-DRV8323Rx Modification for Using Single Shunt Technology	9
Figure 2-7. LAUNCHXL-F280025C Connects BOOSTXL-DRV8323RS and DAC128S085EVM	10
Figure 2-8. LAUNCHXL-F280025C Connects DRV8353RS-EVM and DAC128S085EVM	11
Figure 2-9. LAUNCHXL-F280025C Connects BOOSTXL-3PHGANINV and DAC128S085EVM	12
Figure 2-10. LAUNCHXL-F280025C Connects DRV8316 REVM and DAC128S085EVM	13
Figure 2-11. TMDSHVMTRINSPIN Connects TMDSCNCD280025C and TMDSDAP180TO100	14
Figure 2-12. TMDSHVMTRINSPIN Kit Jumpers and Connectors Diagram	16
Figure 2-13. Select the Right Build Configurations within CCS	20
Figure 2-14. Select the Right Pre-define Symbols in Project Properties	20
Figure 2-15. Project Structure Overview	21
Figure 2-16. Project Explorer View of the Example Lab	22
Figure 2-17. Project Software Flowchart Diagram	23
Figure 2-18. DATALOG Module Block Diagram	26
Figure 2-19. Graph window settings	27
Figure 2-20. PWMDAC Module Block Diagram	28
Figure 2-21. External RC Low-Lass Filter Connecting to PWM Pin in C2000 MCU	28
Figure 2-22. DAC128S Module Block Diagram	29
Figure 2-23. DAC128S085EVM Evaluation Board	29
Figure 2-24. Build Level 1 Software Block Diagram - Offset Validation	30
Figure 2-25. Build Level 1: Variables in Expressions Window	33
Figure 2-26. Build Level 1: PWM Output Waveforms	34
Figure 2-27. Build Level 2 Software Block Diagram - Open Loop Control	34
Figure 2-28. Build Level 2: Variables in Expressions Window	37
Figure 2-29. Build Level 2: Current Protection Setting	37
Figure 2-30. Build Level 2: Motor Phase Current Waveforms	38

Figure 2-31. Build Level 2: Motor Phase Voltage Waveforms Using Common SVM Mode.....	38
Figure 2-32. Build Level 2: Motor Phase Voltage Waveforms Using Minimum SVM Mode.....	39
Figure 2-33. Build Level 2: Motor Rotor Angle and Phase Current Waveforms.....	39
Figure 2-34. Build Level 2: Motor Phase Current Waveforms with Graph Tool.....	40
Figure 2-35. Build Level 2: Motor Phase Voltage Waveforms with Graph Tool.....	40
Figure 2-36. Build Level 2: Motor Rotor Angle Waveforms With Graph Tool.....	41
Figure 2-37. Build Level 3 Software Block Diagram - Current Close Loop Control.....	42
Figure 2-38. Build Level 3: Variables in Expressions Window.....	44
Figure 2-39. Build Level 3: Motor Rotor Angle and Phase Current Waveforms on Oscilloscope.....	45
Figure 2-40. Build Level 4 Software Block Diagram - Speed and Current Close Loop Control.....	45
Figure 2-41. Build Level 4: Variables in Expressions Window.....	48
Figure 2-42. Build Level 4: Rotor Angle with FAST, Phase Current Waveforms at Forward Move.....	49
Figure 2-43. Build Level 4: Rotor Angle with FAST and eSMO, Phase Current Waveforms at Forward Rotation.....	50
Figure 2-44. Build Level 4: Rotor Angle With FAST and eSMO, Phase Current Waveforms at Reversal Rotation.....	50
Figure 2-45. Build Level 4: Rotor Angle with FAST and Encoder, Phase Current Waveforms at Forward Rotation.....	51
Figure 2-46. Build Level 4: Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Forward Rotation.....	52
Figure 2-47. Build Level 4: Motor Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Reversal Rotation.....	52
Figure 2-48. Build Level 4: Rotor Angle with eSMO and Encoder, Phase Current Waveforms at Forward Rotation.....	53
Figure 3-1. HAL Configuration and Motor Control Setting State Machines.....	54
Figure 3-2. PWM Connection Diagram.....	56
Figure 3-3. ADC Connection Diagram.....	58
Figure 3-4. CMPSS Connection Diagram.....	59

List of Tables

Table 2-1. InstaSPIN-FOC Based Evaluation Kits Supported by Motor Control SDK.....	4
Table 2-2. Motor Phase, Encoder, or Hall Sensors Connections for Reference Kits and Motors.....	5
Table 2-3. Key Jumpers, Connectors Explanation.....	17
Table 2-4. The Supporting Algorithms, Functions and Motors Matrix in Example Lab.....	19
Table 2-5. Using Motor Control Modules in Lab Project.....	24
Table 2-6. Motor Control Modules Used per Incremental Build.....	25
Table 2-7. Supporting Estimator Algorithms in Lab Project.....	26

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 to also use as a reference for your own design. The universal motor control solution, as well as the lab project, is delivered within the MotorControl SDK.

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

The FAST library 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 use FAST or InstaSPIN-FOC.

In the lab project, you will learn how to modify "user_mtr1.h," 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 "user_mtr1.h" 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 "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 button, potentiometer, or CAN interface.

The Motor Control Universal Lab project is built in the MotorControl SDK and on top of 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 device-specific drivers and support software to complete example system applications.

The Universal MotorControl Lab requires:

- Code Composer Studio™ v10.4.0 or newer
- C2000 Compiler v20.2.5.LTS or newer
- C2000Ware MotorControl SDK V3.03.00 or newer

2 Running the Universal Lab on TI Hardware Kit

2.1 Supported TI Motor Evaluation Kits

The TMS320F28002x (F28002x) 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 LaunchPad™ development kits or controlCARDS with the relevant motor drive evaluation board.

Table 2-1 lists the current supporting evaluation kits with the motor control lab project in MotorControl SDK.

Table 2-1. InstaSPIN-FOC Based 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				
BOOSTXL-DRV8323RH	6~54V, 15A 3-ph inverter with CSD88599Q5DC NexFET™ power blocks	LAUNCHXL-F280025C	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 NexFET™ power blocks	LAUNCHXL-F280025C	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	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 NexFET™ power blocks	LAUNCHXL-F280025C	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-3PHGANINV	12~60V, 3.5A 3-ph GaN inverter	LAUNCHXL-F280025C	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 + TMDADAP180TO100	Three low-side current shunt	FAST estimator based sensorless-FOC eSMO observer based sensorless-FOC QEP encoder based sensored-FOC	HVPMSMMTR (Encoder Embedded) HVBLCMTR (Hall Sensors Embedded)

It is important to ensure that the physical connections are done correctly if the lab is set to use Encoder or Hall based sensed-FOC. If the motor, encoder or hall wires are connected in the wrong order, the lab will not function properly. It will result in the motor being unable to move. For the motor it is important to ensure that the motor phases are connected to the right phase on the controller. Phase connections for the motors that are provided with the TI Motor Control Reference Kits are provided in [Table 2-2](#).

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. Please refer to the information for your board in order to wire your encoder correctly.

It is important for the setup and configuration of the ENC module that the number of lines on the encoder be provided. This allows the ENC module to correctly convert encoder counts into an angle. This value is represented by USER_MOTOR1_NUM_ENC_SLOTS. This value needs to be defined in user.h as part of the user motor definitions. This value must be updated to the correct value for your encoder. If this value is not correct the motor will spin faster or slower depending on if the value you set. It is important to note that this value should be set to the number of lines on the encoder, not the resultant number of counts after figuring the quadrature accuracy.

Table 2-2. 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	BLACK (J4-1)	N/A, No support encoder based sensed-FOC	BLACK	Not support encoder based sensed-FOC
	+5V	RED (J4-2)		RED	
	I	BROWN (J4-3)		YELLOW	
	B	ORANGE (J4-4)		GREEN	
	A	BLUE (J4-1)		BLUE	
Hall Sensors	GND	BLACK (J10-1)	BLACK	Not support hall based sensed-FOC	BLACK
	+5V	RED (J10-2)	RED		RED
	A	GRAY-WHITE (J10-3)	BLUE		BLUE
	B	GREEN-WHITE (J10-4)	GREEN		GREEN
	C	GREEN (J10-5)	WHITE		WHITE

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

- Step 1: Order the power inverter board, C2000 development kits and motors as shown in [Table 2-1](#).
- Step 2: Download and always use latest version of [MotorControl SDK](#).
- Step 3: Download and always use latest version of [Code Composer Studio IDE](#).
- Step 4: Follow the instructions to setup the hardware and run the example lab in the following sections.
- Step 5: Search existing answers or ask your own question to get the quick design help you need in the [TI C2000 E2E forum](#).

2.2 Hardware Board Setup

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

2.2.1 LAUNCHXL-F280025C Setup

[LAUNCHXL-F280025C](#) is a low-cost development board for 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.

- Hardware files are in <install_location>\boards\LaunchPads\LAUNCHXL_F280025C folder of [C2000Ware](#).
- For more details about [LAUNCHXL-F280025C](#), see [F28002x Series LaunchPad™ Development Kit](#).
- Make sure that switches on [LAUNCHXL-F280025C](#) are set as described below shown in [Figure 2-1](#).
 - S5, position Q1->J12 for encoder interface on J12, position Q2->J13 for Hall sensor interface on J13.
 - S3, position GPIO24->0, position GPIO32->1 to put F280025C into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.

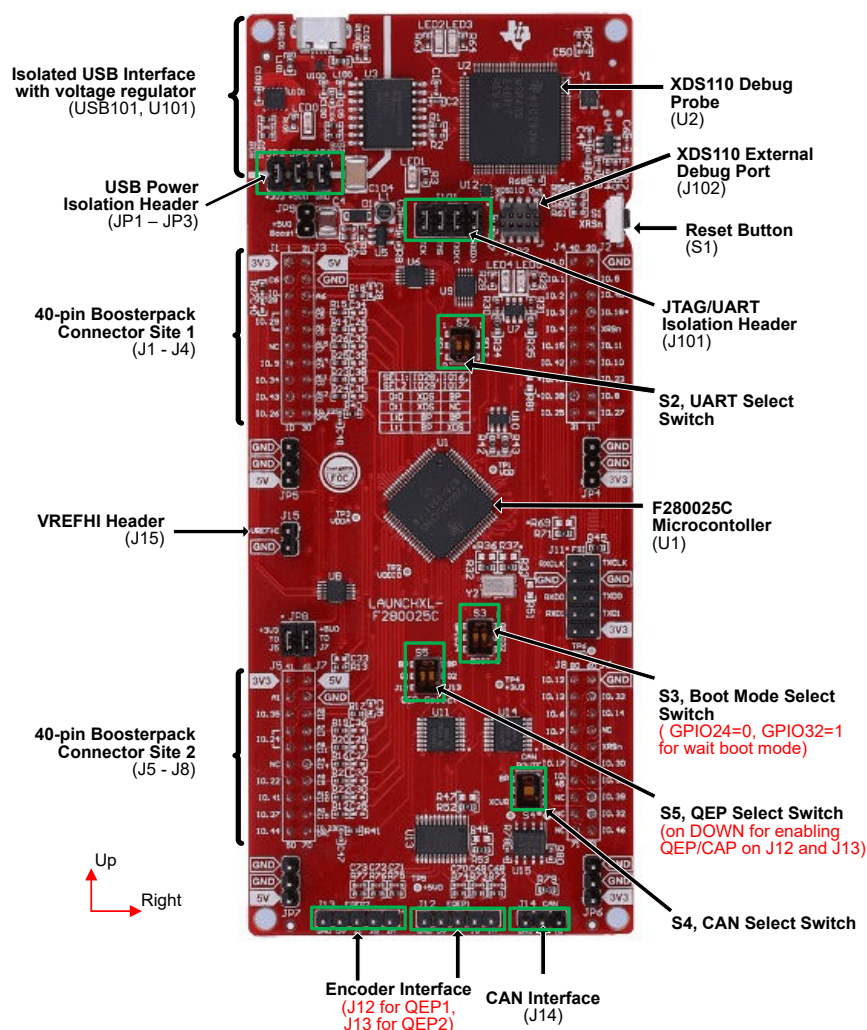


Figure 2-1. F28002x LaunchPad Board Overview and Switches Setting

2.2.2 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 [TMDSHVMTRINSPI](#) with [TMDSADAP180TO100](#) adapter

- Hardware Files are in <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 2-2**.
 - S1:A, both positions are ON(up) for using the on-Card XDS100v2 emulator
 - S4, position_1 (GPIO24)->0(down), position_2(GPIO32)->1(up) to put F280025C into wait boot mode for reducing the risk of connectivity issues or a previous loaded code execution.
 - S5, position_1->down, A8/C11 goes to HSEC pin 30, Position_2->up, enables the internal voltage reference.
 - S3, ON(up), enables external crystal.

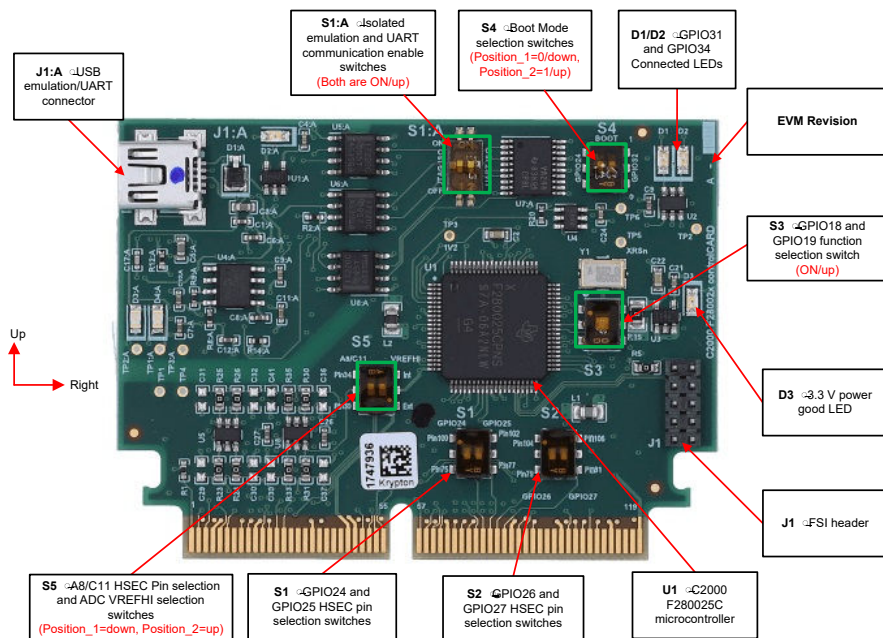


Figure 2-2. F28002x controlCARD and Switches Setting

2.2.3 TMD5ADAP180TO100 Setup

The **TMD5ADAP180TO100** adapter allows the use of 180-Pin C2000 controlCARDS with existing 100-Pin DIMM based evaluation tools. The **TMDSCNCD280025C** needs **TMD5ADAP180TO100** to be used on **TMDSHVMTRINSPIN**.

- Hardware files are in <install_location>\boards\controlCARDS\TMD5ADAP180TO100 folder of **C2000Ware**.
- Make sure that switches **TMD5ADAP180TO100** are set as described below or shown in **Figure 2-3**.
 - S1 on the right side. S2, S3, and S4 on the left position

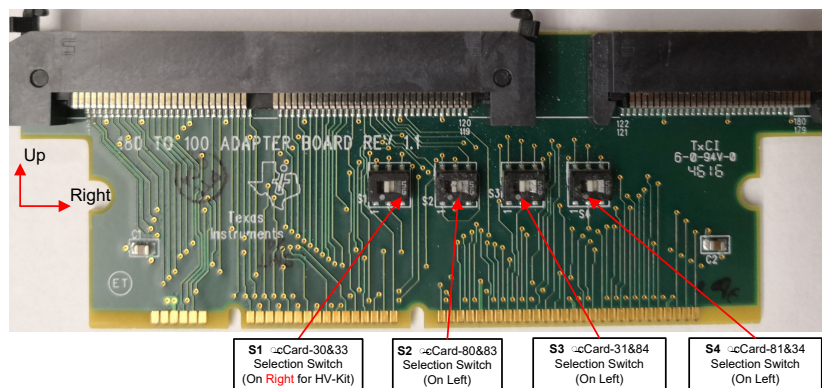


Figure 2-3. TMD5ADAP180TO100 Adapter and Switches Setting

2.2.4 BOOSTXL-DRV8323RH Setup

BOOSTXL-DRV8323RH is a 15A, 3-phase brushless DC drive stage based on the DRV8323RH gate driver and CSD88599Q5DC NexFET™ power blocks. The module has individual DC bus and three-phase voltage sense as well as individual low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with sensorless InstaSPIN-FOC algorithm.

- **BOOSTXL-DRV8323RH Hardware Files** are on the **BOOSTXL-DRV8323RH** page within **ti.com**.
- For more details about **BOOSTXL-DRV8323RH**, see **BOOSTXL-DRV8323Rx EVM User's Guide**.
- Make sure that the following items are set as described below, and then connect **BOOSTXL-DRV8323RH** to J1/J3 and J4/J2 site of **LAUNCHXL-F280025C** as shown in **Figure 2-4**.
 - Populate a 33nF capacitor to C9, C10, and C11.
 - Make the J3-29, and J3-30 on **LAUNCHXL-F280025C** disconnect to **BOOSTXL-DRV8323RH** as shown in **Figure 2-5**.
 - Make the J5-42 on **LAUNCHXL-F280025C** disconnect to **DAC128S085EVM** if have **DAC128S085EVM** board. This EVM board is only used for debugging to monitor the variables.
 - Use a jumper wire connect the J3-29 of **LAUNCHXL-F280025C** to J3-11 of **BOOSTXL-DRV8323RH** if want to use the POT on **BOOSTXL-DRV8323RH** for setting the reference speed.
- Connect the motor, encoder, and hall sensors to the kits as described in **Table 2-2** and shown in **Figure 2-4**.
- Connect a supply voltage ranging from 6 to 54 V from a battery or a DC voltage source to the voltage supply pins as shown in **Figure 2-4**. Power should only be applied when instructed to do so in **Section 2.5**, keep disconnected otherwise.

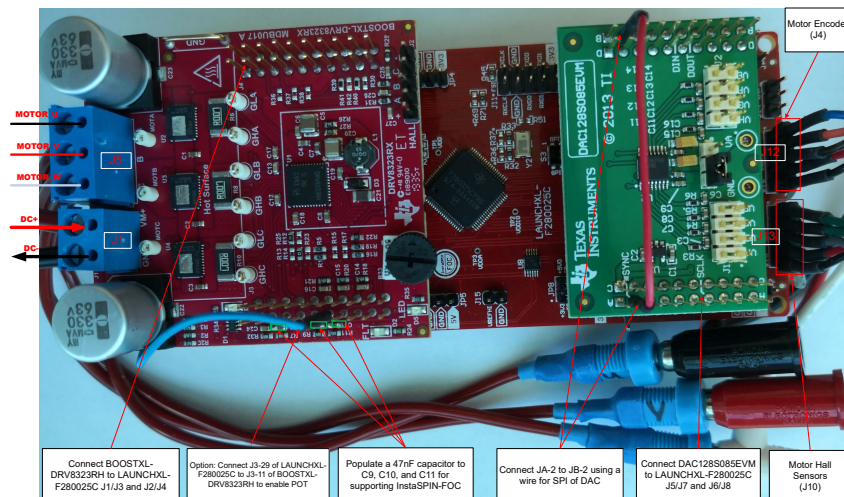


Figure 2-4. LAUNCHXL-F280025C Connects BOOSTXL-DRV8323RH and DAC128S085EVM

The switches and connections on LAUNCHXL-F280025C as shown in [Figure 2-5](#) are fit for [BOOSTXL-DRV8323RH](#), [BOOSTXL-DRV8323RS](#), and [DRV8353RS-EVM](#).

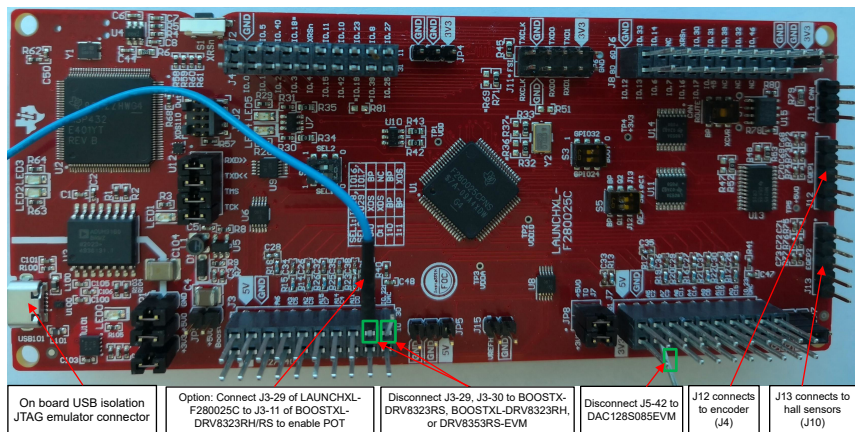


Figure 2-5. LAUNCHXL-F280025C Board Settings For Connecting BOOSTXL-DRV Board

The modification on [BOOSTXL-DRV8323RH](#) as shown in [Figure 2-6](#) is also fit for [BOOSTXL-DRV8323RS](#) to implement single shunt.

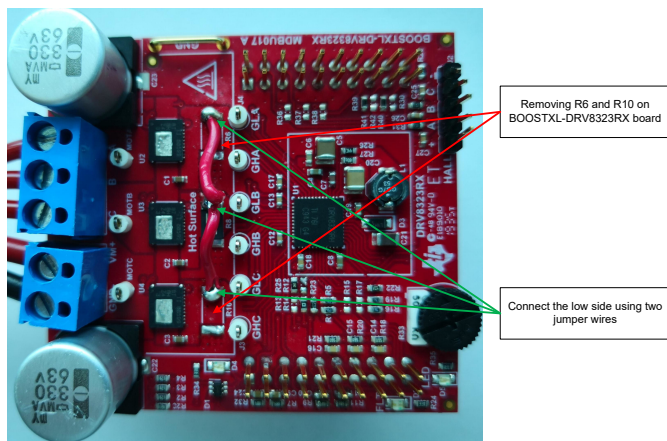


Figure 2-6. BOOSTXL-DRV8323Rx Modification for Using Single Shunt Technology

2.2.5 BOOSTXL-DRV8323RS Setup

BOOSTXL-DRV8323RS is a 15A, 3-phase brushless DC drive stage based on the DRV8323RH gate driver and CSD88599Q5DC NexFET™ power blocks. The module has individual DC bus and three-phase voltage sense as well as individual low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad development kits suitable for use with 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.

- **BOOSTXL-DRV8323RS Hardware Files** are on the **BOOSTXL-DRV8323RS** page within **ti.com**.
- For more details about **BOOSTXL-DRV8323RS**, see the **BOOSTXL-DRV8323Rx EVM User's Guide**.
- Make sure that the following items are set as described below, and then connect **BOOSTXL-DRV8323RS** to J1/J3 and J4/J2 site of **LAUNCHXL-F280025C** as shown in **Figure 2-7**.
 - Populate a 33nF capacitor to C9, C10, and C11.
 - Make the J3-29, and J3-30 on **LAUNCHXL-F280025C** disconnect to **BOOSTXL-DRV8323RS** as shown in **Figure 2-5**.
 - Make the J5-42 on **LAUNCHXL-F280025C** disconnect to **DAC128S085EVM** if have **DAC128S085EVM** board. This is an option for debugging to monitor the variables.
 - Use a jumper wire connect the J3-29 of **LAUNCHXL-F280025C** to J3-11 of **BOOSTXL-DRV8323RS** if want to use the POT on **BOOSTXL-DRV8323RS** for setting the reference speed.
- Connect the motor, encoder, and hall sensors to the kits as described in **Table 2-2** and shown in **Figure 2-7**.
- Connect a supply voltage ranging from 6 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 2.5**, keep disconnected otherwise.

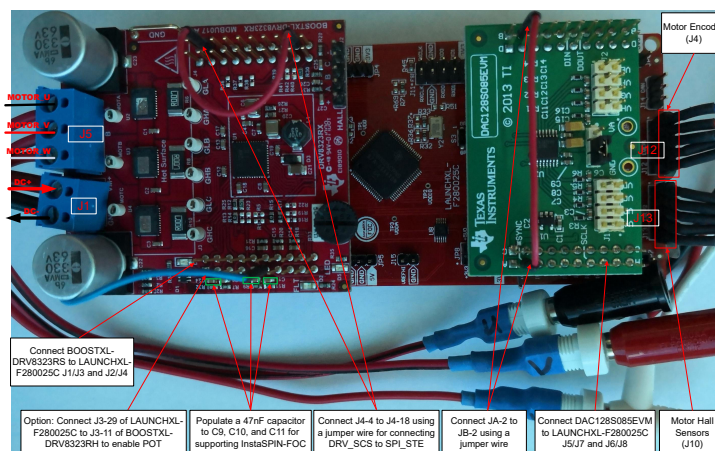


Figure 2-7. LAUNCHXL-F280025C Connects BOOSTXL-DRV8323RS and DAC128S085EVM

2.2.6 DRV8353RS-EVM Setup

DRV8353RS-EVM is a 15A, 3-phase brushless DC drive stage based on the DRV8353RS gate driver and CSD19532Q5B NexFET™ power blocks. The module has individual DC bus and three-phase voltage sense as well as individual low-side current shunt amplifiers, making this board for BLDC/PMSM control with C2000 LaunchPad development kits suitable for use with 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 devices SPI registers written by C2000 MCUs.

- **DRV8353Rx-EVM Design Files** are on the **DRV8353RS-EVM** page within ti.com.
- For more details about **DRV8353RS-EVM**, see **DRV8353Rx-EVM User's Guide**.
- Make sure that the following items are set as described below, and then connect **DRV8353RS-EVM** to J1/J3 and J4/J2 site of **LAUNCHXL-F280025C** as shown in **Figure 2-8**.
 - Make the J1-17 (IDRIVE), and J1-19 (VDS) on **DRV8353RS-EVM** disconnect to **LAUNCHXL-F280025C** as shown in **Figure 2-5**.
 - Make the J5-42 on **LAUNCHXL-F280025C** disconnect to **DAC128S085EVM** if have **DAC128S085EVM** board. This is an option for debugging to monitor the variables.
- Connect the motor, encoder, and hall sensors to the kits as described in **Table 2-2** and shown in **Figure 2-8**.
- Connect a supply voltage ranging from 6 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 2.5**, keep disconnected otherwise.

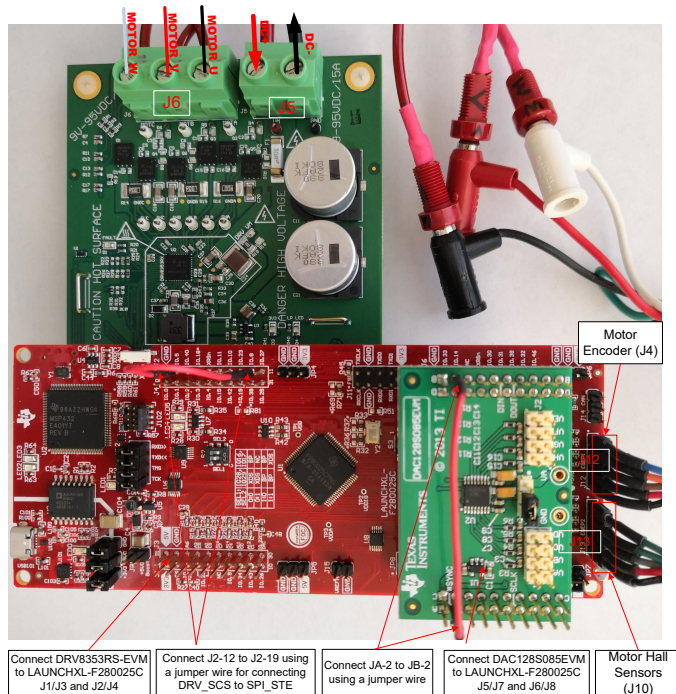


Figure 2-8. LAUNCHXL-F280025C Connects DRV8353RS-EVM and DAC128S085EVM

2.2.7 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 has individual DC bus and three-phase voltage sense making this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with sensorless InstaSPIN-FOC algorithm.

- Hardware files and more details are on **BOOSTXL-3PHGANINV** page within [ti.com](https://www.ti.com).
- Make sure that the following items are set as described below, and then connect **BOOSTXL-3PHGANINV** to J1/J3 and J4/J2 site of **LAUNCHXL-F280025C** as shown in **Figure 2-9**.
 - Make the J5-42 on **LAUNCHXL-F280025C** disconnect to **DAC128S085EVM** if have **DAC128S085EVM** board. This is an option for debugging to monitor the variables.
- Connect the motor, encoder, and hall sensors to the kits as described in **Table 2-2** and shown in **Figure 2-9**.
- Connect a supply voltage ranging from 6 to 54 V from a battery or a DC voltage source to the voltage supply pins. Follow the operation instruction in **Section 2.5** to turn on the power source.

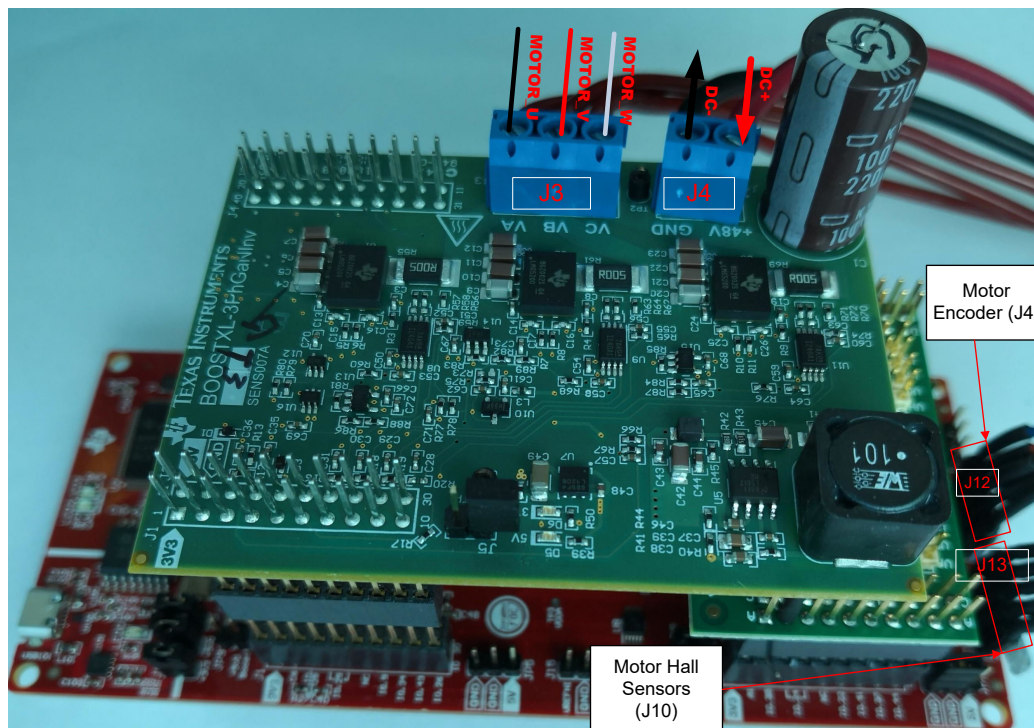


Figure 2-9. LAUNCHXL-F280025C Connects BOOSTXL-3PHGANINV and DAC128S085EVM

2.2.8 DRV8316REVM Setup

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 sense on the module that makes this board for BLDC/PMSM control with C2000 LaunchPad™ development kits suitable for use with sensorless InstaSPIN-FOC algorithm

- [DRV8316REVM Hardware Design Files](#) are on the [DRV8316REVM](#) page within [ti.com](#).
- For more details about [DRV8316REVM](#), see [DRV8316REVM Evaluation Module](#).
- Make sure that the following items are set as described below, and then connect [BOOSTXL-3PHGANINV](#) to J1/J3 and J4/J2 site of [LAUNCHXL-F280025C](#) as shown in [Figure 2-10](#).
 - Make the J5-42 on [LAUNCHXL-F280025C](#) disconnect to [DAC128S085EVM](#) if have [DAC128S085EVM](#) board. This is option for debugging to monitor the variables.
- Connect the motor, encoder, and hall sensors to the kits as described in [Table 2-2](#) and shown in [Figure 2-10](#).
- Connect a supply voltage ranging from 6 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 2.5](#), keep disconnected otherwise.

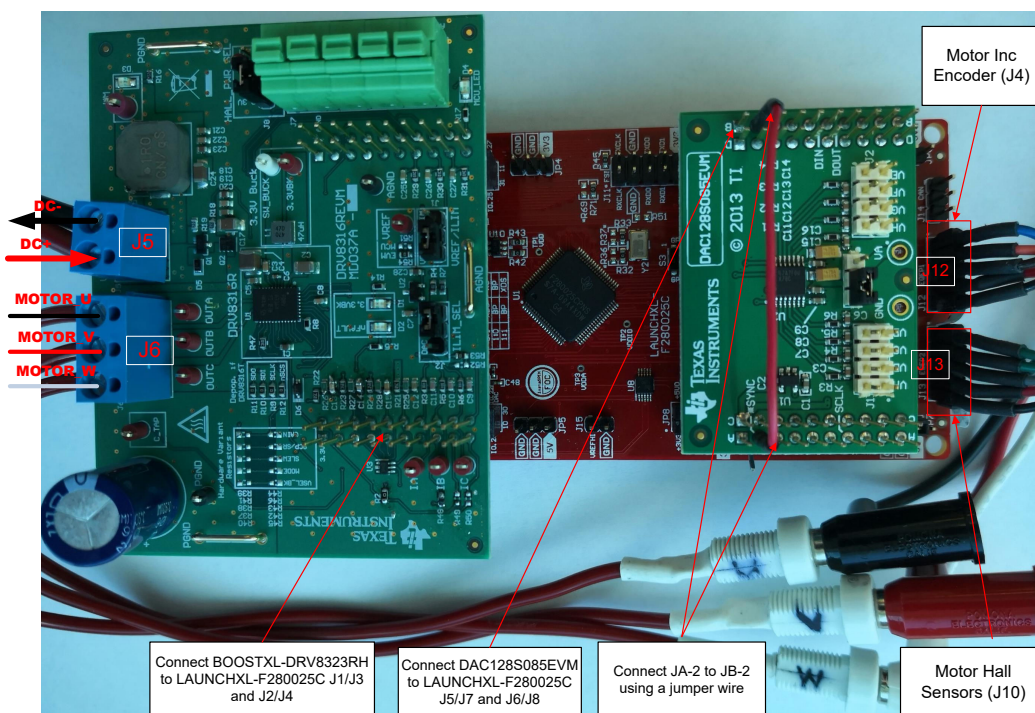


Figure 2-10. LAUNCHXL-F280025C Connects DRV8316 REVM and DAC128S085EVM

2.2.9 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 sense making this board for BLDC/PMSM control with C2000 LaunchPad™ development kits using sensorless InstaSPIN-FOC algorithm.

- Hardware Files are in <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 pre done for connecting with controlCARD. Make sure that ensure that these settings are valid on the board as described below, and then insert controlCARD with [TMDSDAP180TO100](#) adapter into [TMDSHVMTRINSPIN](#) as shown in [Figure 2-11](#).

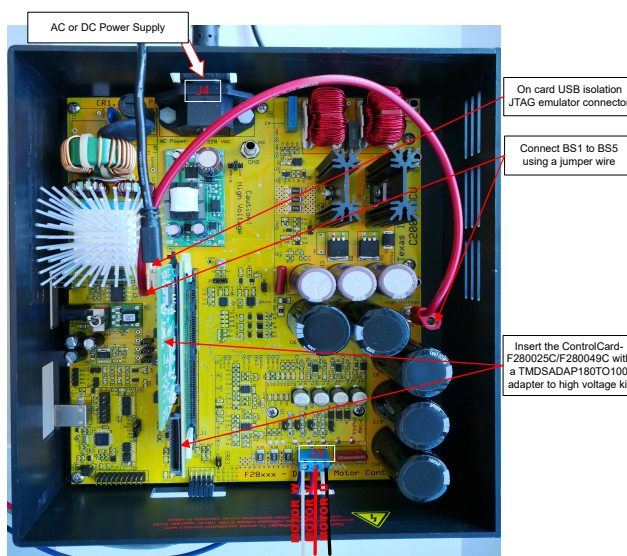


Figure 2-11. TMDSHVMTRINSPIN Connects TMDSCNCD280025C and TMDSDAP180TO100

Do not apply AC power to 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 valid as shown in [Figure 2-12](#).
 - [Main]-J3, J4, J5 and J8 are populated.
 - [Main]-J9 and [M3]-J5 is 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 b/w [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 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 b/w 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 2-2](#) and shown in [Figure 2-12](#).
- 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 2.5](#), keep disconnected otherwise.

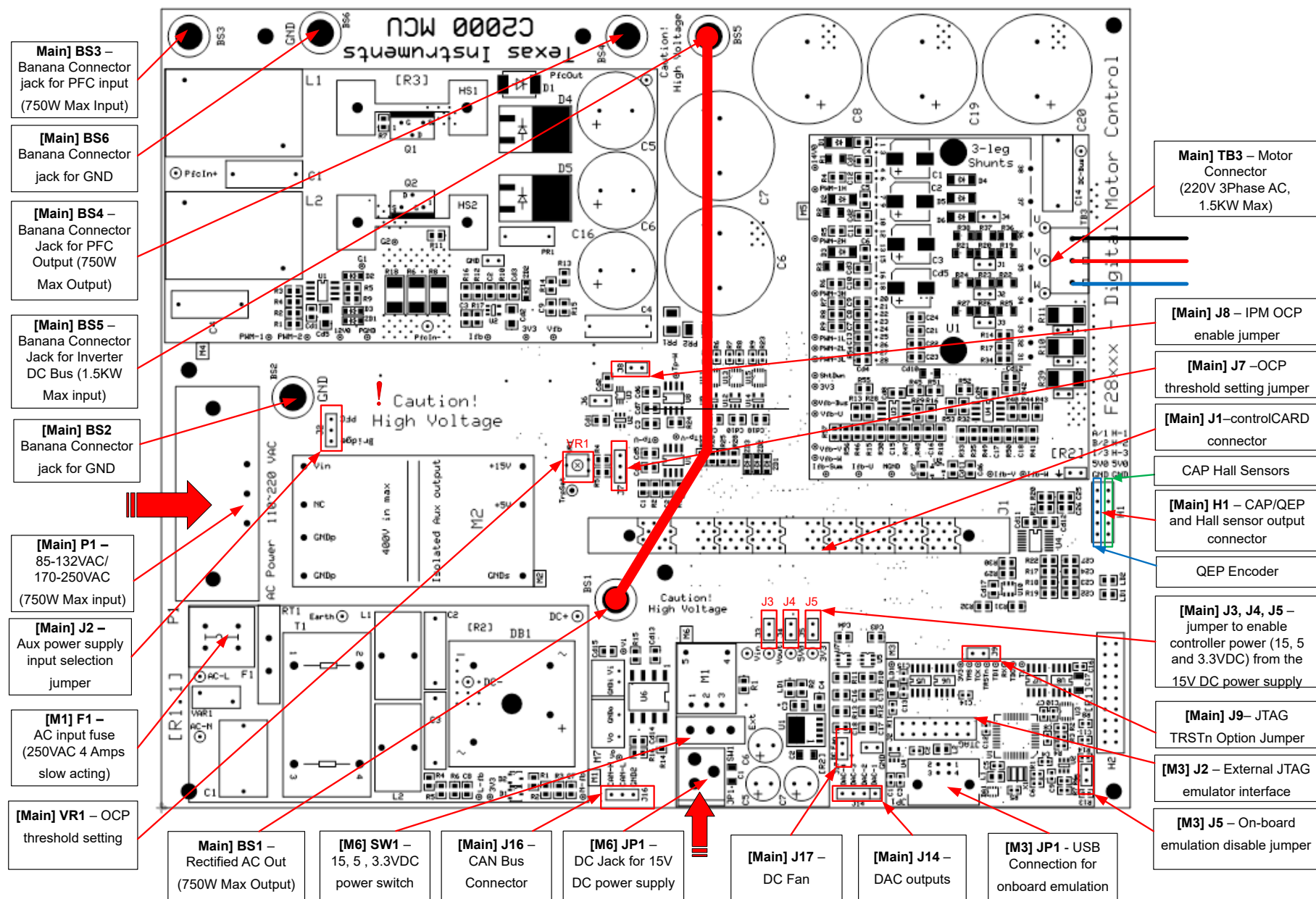


Figure 2-12. TMDSHVMTRINSPIN Kit Jumpers and Connectors Diagram

Table 2-3 shows the various connections available on the board, illustrates the location of these connections on the board with help of a board image as shown in Figure 2-12.

Table 2-3. 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 needs 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.

2.3 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. Install [C2000WARE-MOTORCONTROL-SDK](#) 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 installation complete, close CCS, and create a new workspace for importing the project. The software project of this example lab is inside 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.

2.3.1 Importing and Configuring Project

The example lab is an universal project that can support all the latest C28x CPU with FPU and TMU based C2000 MCUs, and can run on different TI EVM kits by setting the build configurations and properties of the lab project. In the following sections, use [LAUNCHXL-F280025C](#) with [BOOSTXL-DRV8323RS](#) based 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 `<install_location>\solutions\universal_motorcontrol_lab\f28002x\ccs\motor_control_1\` to select the "universal_motorcontrol_lab_f28002x" project.
2. The lab project can be configured to create code and run on multiple kits. You may select one of these kits. Select the right build configurations (such as **Flash_lib_DRV8323RS**) by right-clicking on the imported project name as shown in [Figure 2-13](#).
3. Configure the project to select the supporting functions in project by right-clicking on the imported project name, and then click "Properties" command to set the pre-define symbols for the project as shown in [Figure 2-14](#).
 - a. The pre-define symbol is active or disabled by removing or adding the "_N" in the name. For example, enables field weakening control by removing the "_N" in "MOTOR1_FWC_N" to "MOTOR1_FWC", disables field weakening control functions for motor 1 (Compressor) by changing "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 above. The supporting algorithms and related motors matrix are as shown in [Table 2-4](#).
 - c. Select the right supporting functions by enabling the pre-define symbol as shown in [Figure 2-14](#).
4. Select the right target configuration file (.ccxml) as shown in [Figure 2-16](#) 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 right motor model in the `user_mtr1.h` and `user_common.h`, make sure that the motor parameters are in consonance with the connecting motor.
6. Set up the hardware kit, connect the motor, encoder, and/or hall sensor to the kit as described in the [Section 2.2](#).

Table 2-4. The Supporting Algorithms, Functions and Motors Matrix in Example Lab

Algorithms or Functions	Pre-Define Symbols	LAUNCHXL-F280025C					TMDSCNCD280025C + TMDSDAP180TO100
		BOOSTXL-DRV8323RH	BOOSTXL-DRV8323RS	DRV8353RS-EVM	BOOSTXL-3PHGANI NV	DRV8316REVM	TMDSHVMTRINSPIN
FAST based Sensorless FOC	MOTOR1_FAST	✓, LVSERVOMTR#	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, HVPMSMMTR#
eSMO based Sensorless FOC	MOTOR1_ESMO	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, LVSERVOMTR	✓, HVPMSMMTR
QEP Encoder based Sensored FOC	MOTOR1_ENC QEP_ENABLE	✓, 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	✓, HVBLDCMTR#
Trapezoidal InstaSPIN-BLDC	MOTOR1_ISBLDC	✓* 1, LVSERVOMTR, LVBLDCMTR	✓*, LVSERVOMTR, LVBLDCMTR	✗	✗	✗	✗
Single-Shunt Current Sense	MOTOR1_DCLINKSS	✓** 1	✓**	✗	✗	✗	✗
Datalog with Graph Tool	DATALOGF2_EN	✓	✓	✓	✓	✓	✓
PWMDAC	EPWMDAC_MODE	✗	✗	✗	✗	✗	✓
External DAC	DAC128S_ENABLE	✓	✓	✓	✓	✓	✗
SFRA Tool	SFRA_ENABLE	✓	✓	✓	✓	✓	✓
Step Response with Graph Tool	STEP_RP_EN	✓	✓	✓	✓	✓	✓

1. *, ** means this board needs to be changed by removing two shunt resistors and connecting all of low sides of three phase half bridge with a jumper wire as shown in [Figure 2-6](#).

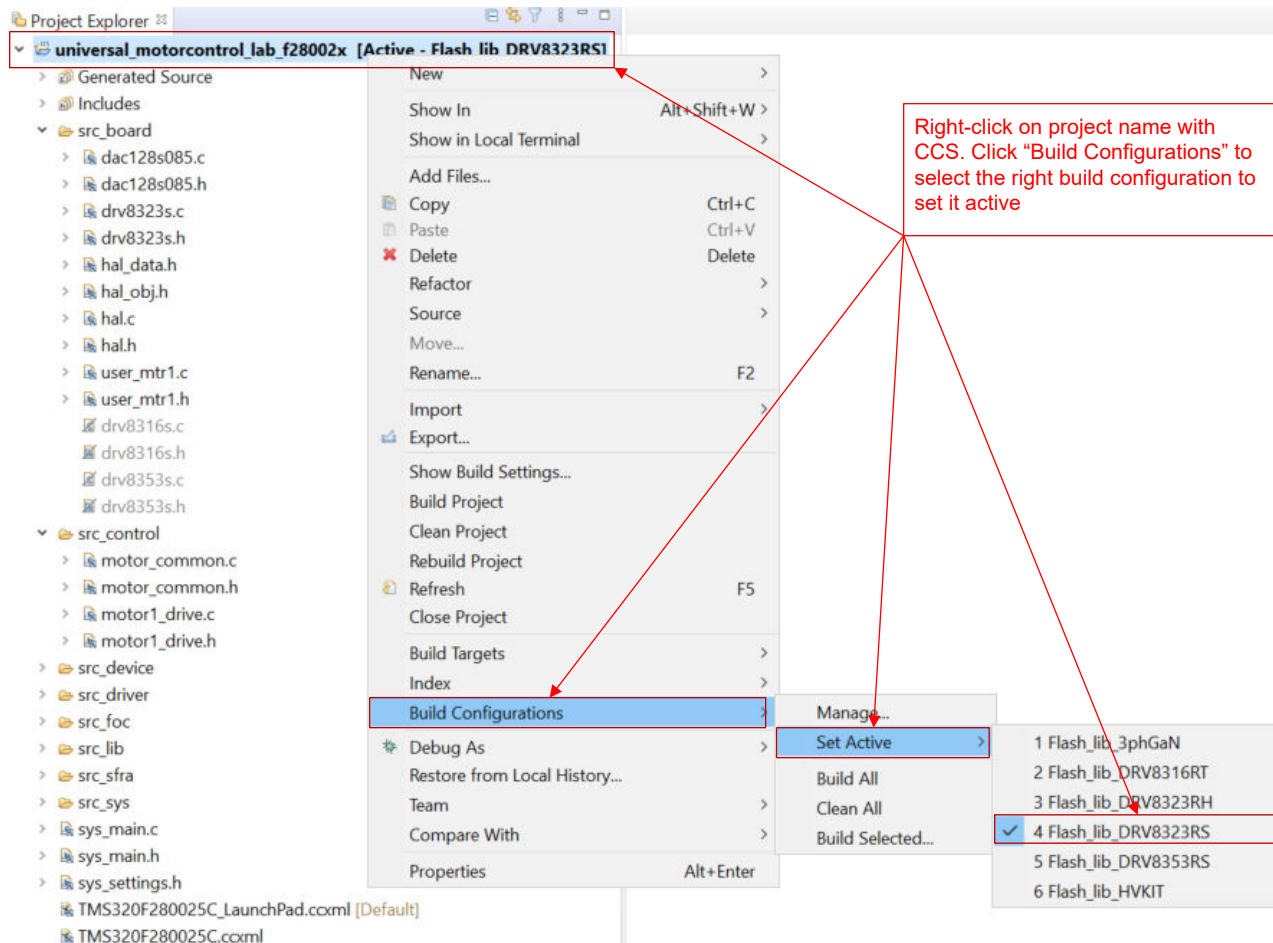


Figure 2-13. Select the Right Build Configurations within CCS

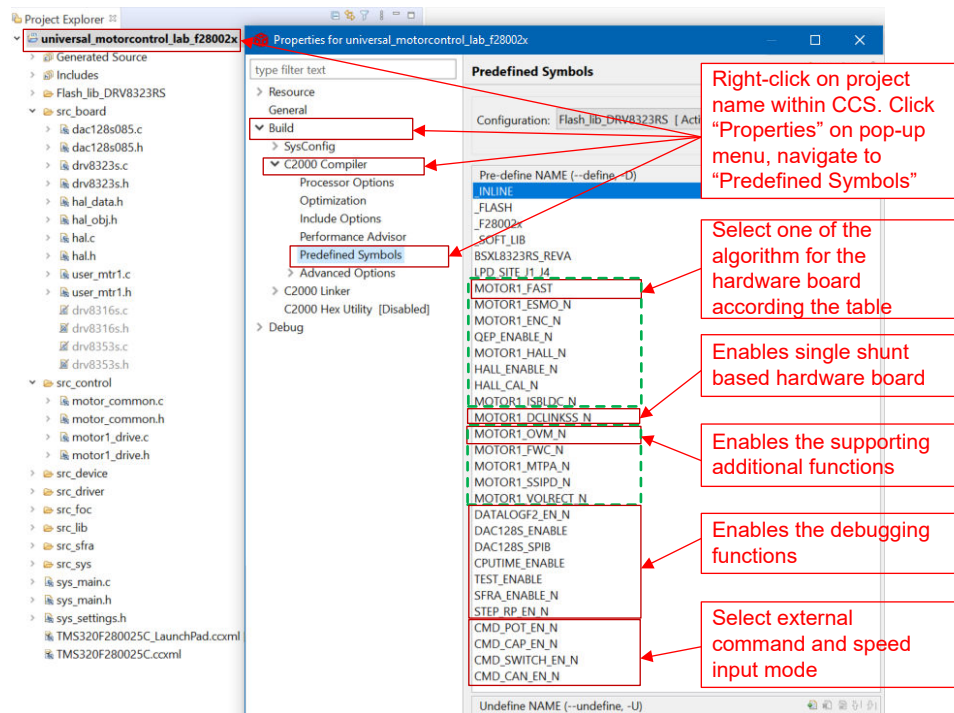


Figure 2-14. Select the Right Pre-define Symbols in Project Properties

2.3.2 Lab Project Structure

The general structure of the project is shown in [Figure 2-15](#). The device peripherals configuration is based on C2000Ware driverlib. The users only need to change the codes and definitions in *hal.c* and *hal.h*, and the parameters in *user_mtr1.h* if they want to migrate the reference design software to their own board.

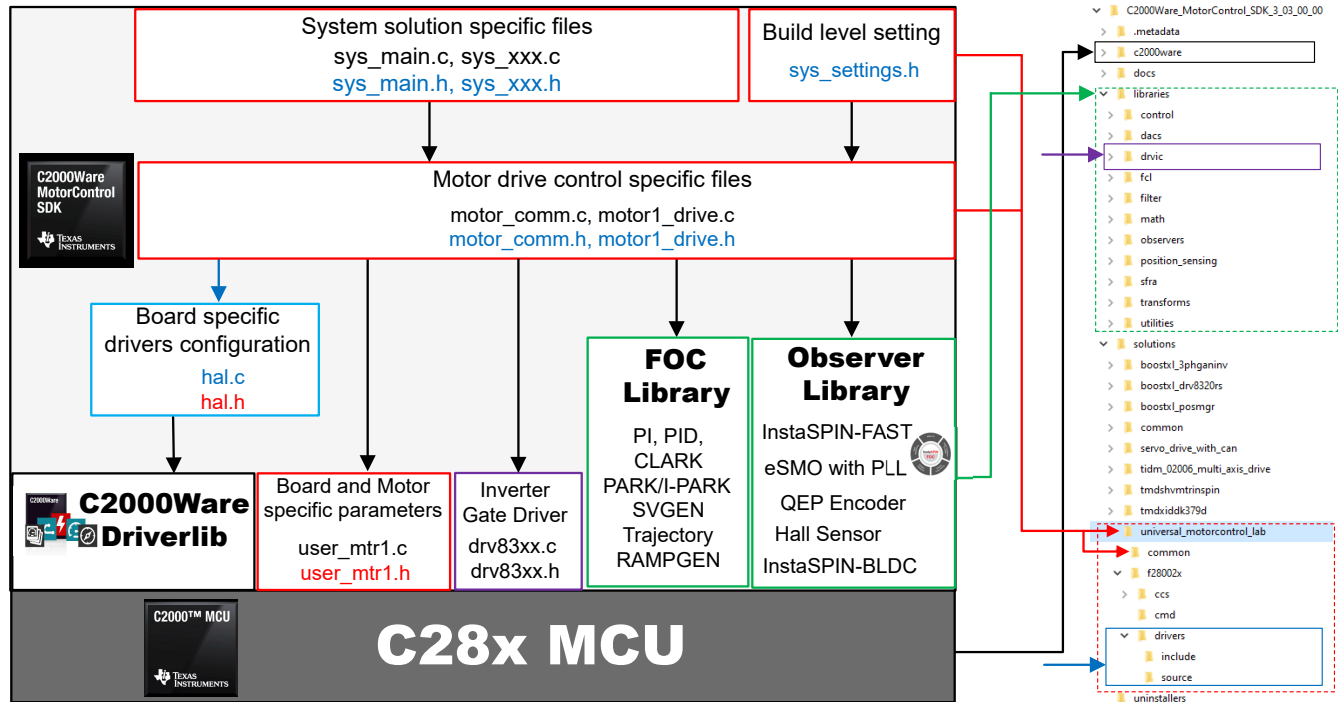


Figure 2-15. Project Structure Overview

Once the project is imported, the project explorer will appear inside CCS as shown in [Figure 2-16](#).

The folder *src_foc* includes the typical FOC modules include transform, PID and estimators that consist of the motor drive algorithm are independent on specific device and board.

The folder *src_lib* includes InstaSPIN-FOC library and math library that is independent of device and board.

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

The folder `src_sys` includes some files reserved for system control that are independent on specific device and board. The user can add their own codes for system control, communication, and so forth.

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

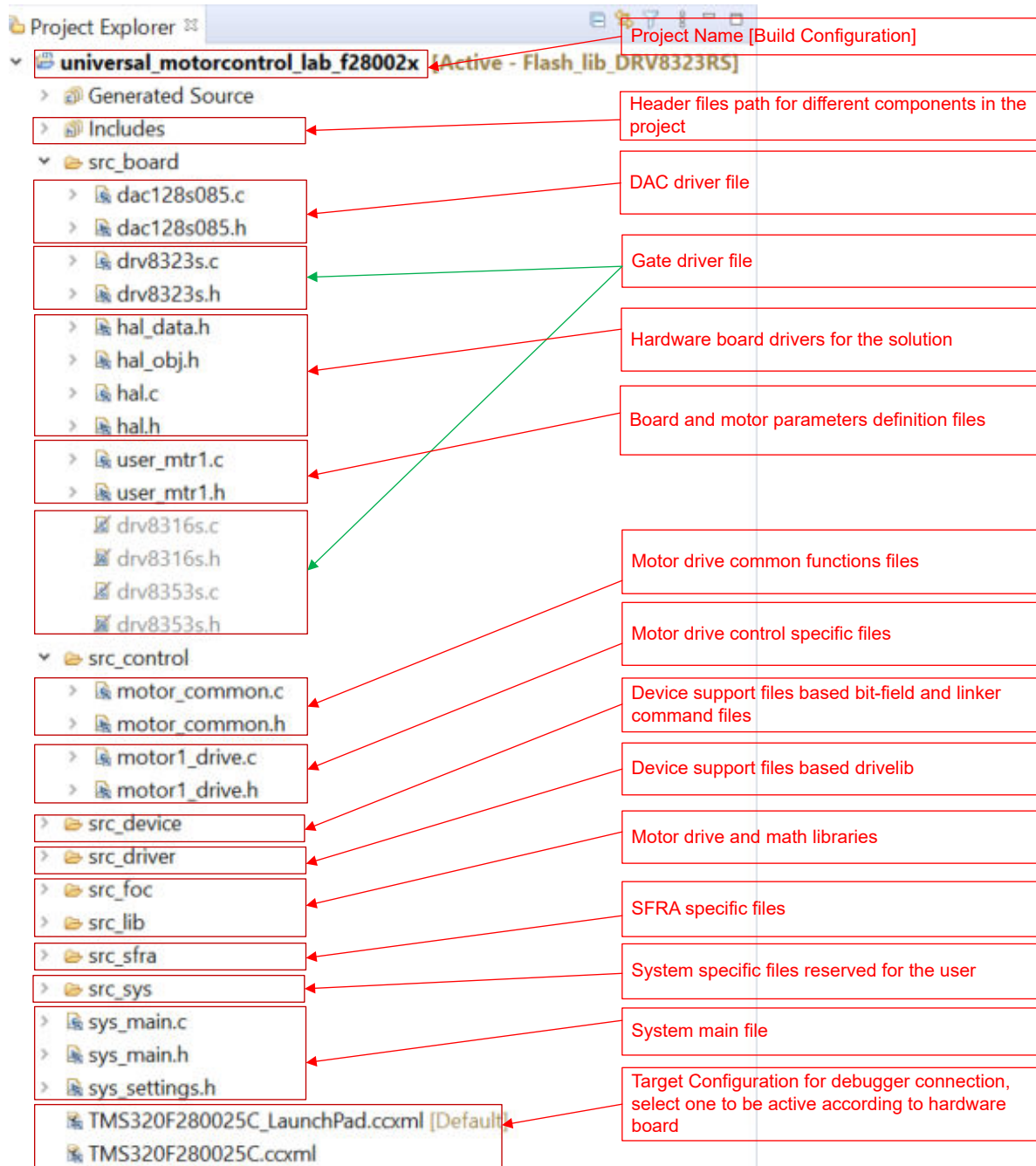


Figure 2-16. Project Explorer View of the Example Lab

2.3.3 Lab Software Overview

Figure 2-17 shows the project software flow diagram of the firmware that includes one ISR for real time motor control, a main loop for motor control parameters update in background loop. The ISR will be triggered by ADC End of Conversion (EOC).

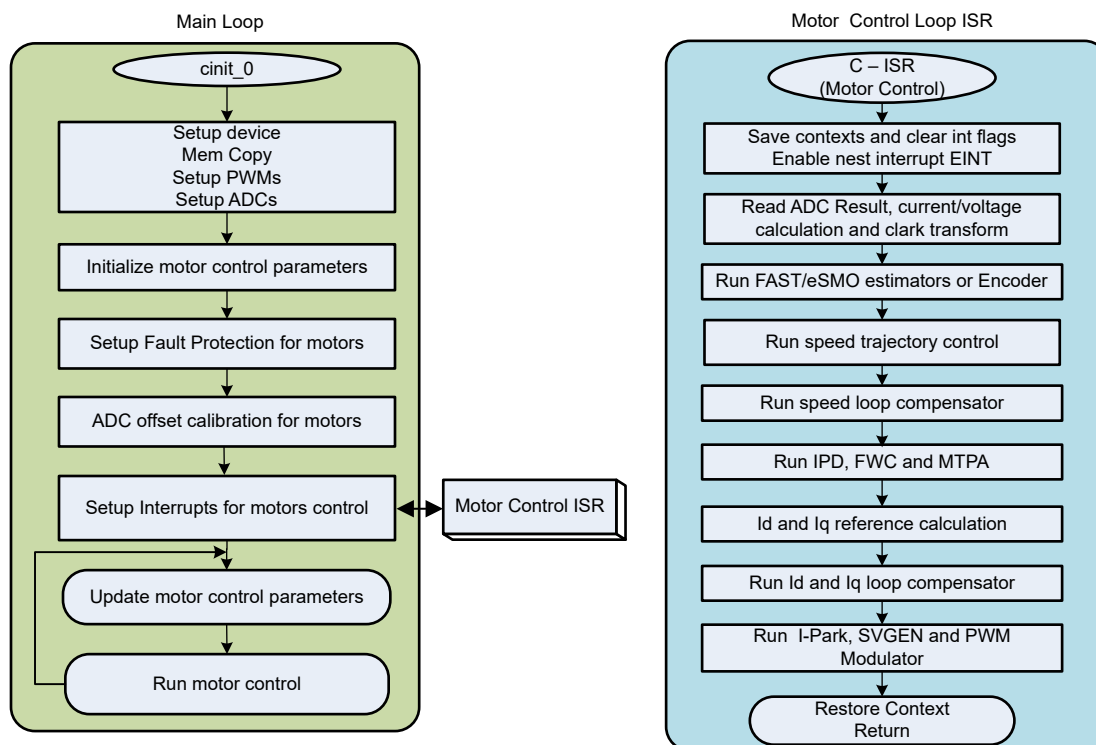


Figure 2-17. Project Software Flowchart Diagram

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 2-5 lists the framework modules to be used in this lab project.

Table 2-5. Using Motor Control Modules in Lab Project

Module Names	Explanation	Algorithm
HAL_readADCData	Returns ADC conversion values with floating-point format	All Algorithms
HAL_writePWMDData	PWM drives for motor	All Algorithms
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
EST_setupTrajState	Trajectory generator setup for current and speed for motor identification	FAST
EST_runTraj	Trajectory generator for current and speed for motor identification	FAST
EST_run	FAST estimator for sensorless-FOC	FAST
ESMO_run	Enhance Sliding Mode Observer (eSMO) for sensorless-FOC	eSMO
ENC_run	Calculate rotor angle based on encoder	ENC
HALL_run	Calculate rotor angle and 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
collectRMSData, calculateRMSData	Collect sampling values to calculate the RMS value of phase current and voltage	FAST, eSMO, ENC, HALL
HAL_writePWMDACData	Converts software variables into the PWM signals	All Algorithms
DATALOGIF_update	Stores the real-time values into for displaying with graph tool	All Algorithms
DAC128S_writeData	Converts and send software variables to external DAC with SPI	All Algorithms

Table 2-6 summarizes the modules tested in each incremental system build.

Table 2-6. 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 check current and voltage sensing signals for the motor	Closed current loop to check the hardware settings	Closed-loop run with estimators/observers	Motor parameters identify with FAST estimators
HAL_readADCData	√√	√ 1	√	√	√
HAL_writePWMDData	√√	√	√	√	√
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 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 2-1.

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 2-7](#). The estimator in use can be switched smoothly on the fly if two algorithm are implemented in lab project.

Table 2-7. Supporting Estimator Algorithms in Lab Project

	FAST(MOTOR1_FAS T)	eSMO (MOTOR1_ESMO)	ENCODER (MOTOR1_ENC)	HALL (MOTOR1_HALL)	ISBLDC (MOTOR1_ISBLDC)
FAST	√ 1	√	√	√	×
eSMO	√	√	√	×	×
ENCODER	√	√	√	×	×
HALL	√	×	×	√	×
ISBLDC	×	×	×		√

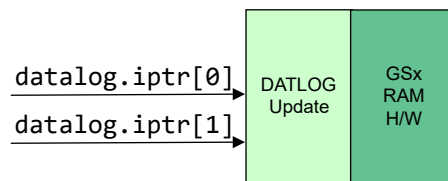
1. √ means these two algorithms can be used simultaneously in project. × means these two algorithms can not be used simultaneously in project.

2.4 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.

2.4.1 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 2-18](#). The two variables are selected by configuring the module inputs, iptr[0], iptr[1] to the address of the two variables. The starting addresses of the two buffer RAM locations, where the data values are stored, are set to datalogBuff1[0], datalogBuff1[1]. These Datalog buffers are large arrays that contain value-triggered data that can then be displayed to a graph. The datalog prescaler is configurable, which will allow the dlog function to only log one out of every prescaler samples. The DMA is used to transfer the values to the previous data values in buffer RAM.


Figure 2-18. DATALOG Module Block Diagram

Instanting one DATALOG object and handle, which is done in "datalogIF.c" file by adding pre-define name "DATALOGF2_EN" in project properperies as shown in [Figure 2-14](#).

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

Initializing and setting the datalog object, handle and parameters:

```
// 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]);
```

Configuring two module inputs, iptr[0] and iptr[1], point to the address of the two variables, the datalog buffers point to different system variables depending on the build level.

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

Call update function for datalog in a periodic interrupt ISR.

```
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 this software on the device.

The datalog module is used with 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.

Open and setup time graph windows to plot the data log buffers as shown in [Figure 2-19](#).

Alternatively, the user can import graph configurations files 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_F12.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.

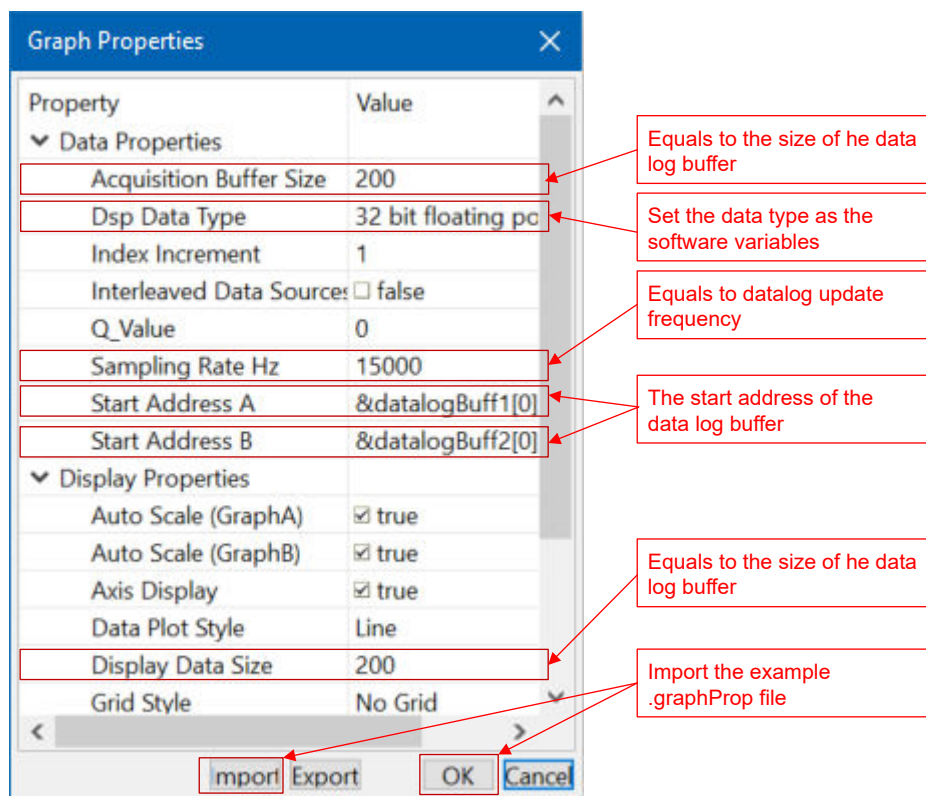


Figure 2-19. Graph window settings

2.4.2 Using PWMDAC Function

The PWMDAC module converts the software variables into the PWM signals in EPWM6A/6B/7A (EPWM channels depending on the hardware kit) for C2000 MCU as shown in [Figure 2-20](#).

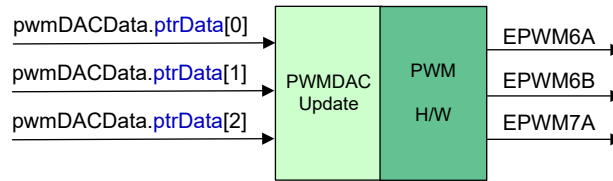


Figure 2-20. PWMDAC Module Block Diagram

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 2-21](#). The (1st-order) RC low-pass filter can be simply used for filter out the high frequency component embedded in the actual low frequency signals. To select R and C values, its time constant can be expressed in term of cut-off frequency (f_c) as following equations [Equation 1](#) and [Equation 2](#).

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

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

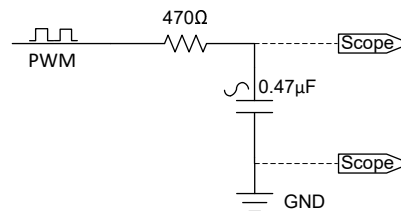


Figure 2-21. External RC Low-Pass Filter Connecting to PWM Pin in C2000 MCU

In the example lab by adding pre-define name "EPWMDAC_MODE" in project properties as shown in [Figure 2-14](#). Instanting one PWMDAC object as the following.

```
HAL_PWMDACData_t pwmDACData;
```

Initializing and setting the PWMDAC object, handle and parameters. Configuring three module inputs, ptrData[0], ptrData[1], and ptrData[2], point to the address of the three variables, the PWMDAC data point to different system variables depending on the build level.

```
// set DAC parameters
pwmDACData.periodMax =

    PWMDAC_getPeriod(halHandle->pwmDACHandle[PWMDAC_NUMBER_1]);

pwmDACData.ptrData[0] = &motorVars_M1.anglePLL_rad;
pwmDACData.ptrData[1] = &motorVars_M1.angleENC_rad;
pwmDACData.ptrData[2] = &motorVars_M1.adcData_I_A.value[0];

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

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

Call update function for PWMDAC in a periodic interrupt ISR.

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

2.4.3 Using External DAC Board

The DAC128S module converts any software variables to 12-bit integer value and transmit the data to the digital-to-analog converter (DAC) on DAC128S085EVM shown in [Figure 2-22](#).

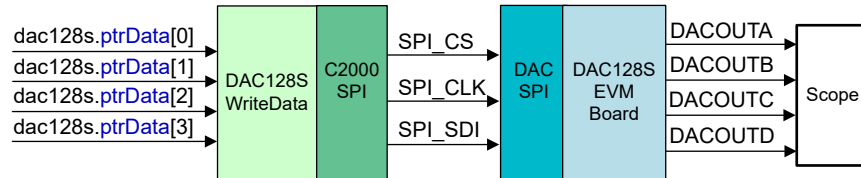


Figure 2-22. DAC128S Module Block Diagram

The [DAC128S085EVM](#) can be connected to the [LAUNCHXL-F280025C](#), or other C2000 LaunchPad™ as shown in [Figure 2-23](#).

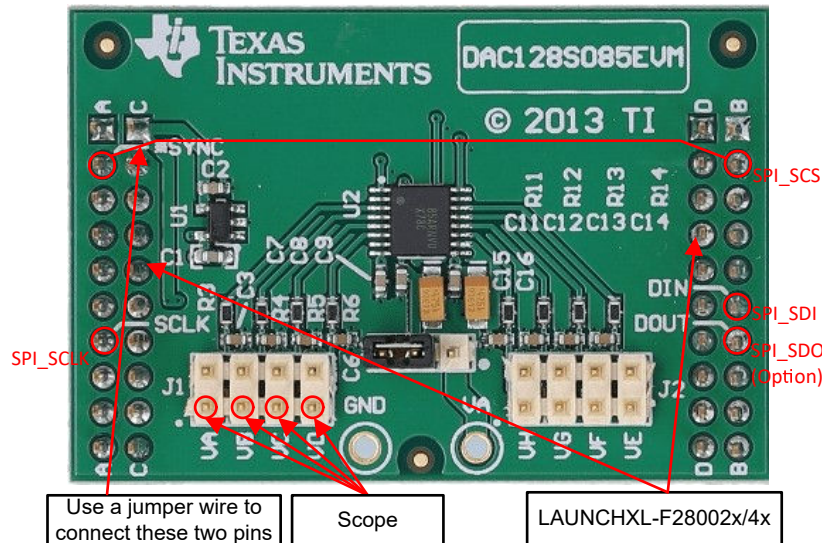


Figure 2-23. DAC128S085EVM Evaluation Board

Instanting one DAC128S object. In the example lab by adding pre-define name "DAC128S_ENABLE" in project properties as shown in [Figure 2-14](#)

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

Initializing and setting the DAC128S object, handle and parameters. Configuring four module inputs, ptrData[0], ptrData[1], ptrData[2] and ptrData[3], point to the address of the two variables, the dac128s data point to different system variables depending on the build level.

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

// setup SPI for DAC128S
DAC128S_setupSPI(dac128sHandle);

dac128s.ptrData[0] = &motorVars_M1.angleGen_rad;    // CH_A
dac128s.ptrData[1] = &motorVars_M1.angleEST_rad;    // CH_B
dac128s.ptrData[2] = &motorVars_M1.adcData.I_A.value[0];    // CH_C
dac128s.ptrData[3] = &motorVars_M1.adcData.V_V.value[0];    // CH_F
```

Call update function for DAC128S in a periodic interrupt ISR.

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

The DAC128S085 has octal 12-bit digital-to-analog converter (DAC) voltage-output, so the user can set the output number between 1 and 8 in "*dac128s085.h*". 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.

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

If change the DAC128S output number, need to configure the related number of the module inputs and parameters.

2.5 Running the Project with Incremental Build

The system is gradually built up in order for the final system can be confidently operated. To select a particular build option, select the corresponding BUILDLEVEL option in *sys_settings.h*. Once the build option is selected, compile the project by right-clicking on the project name and clicking "*Rebuild Project*".

2.5.1 Level 1 Incremental Build

Objectives learned in this build level:

- Use the HAL object to initialize the peripherals of the MCU for inverter 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 [CCS User's Guide](#).

In this build level, the board is executed in open loop mode with a fixed 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 2-24](#).

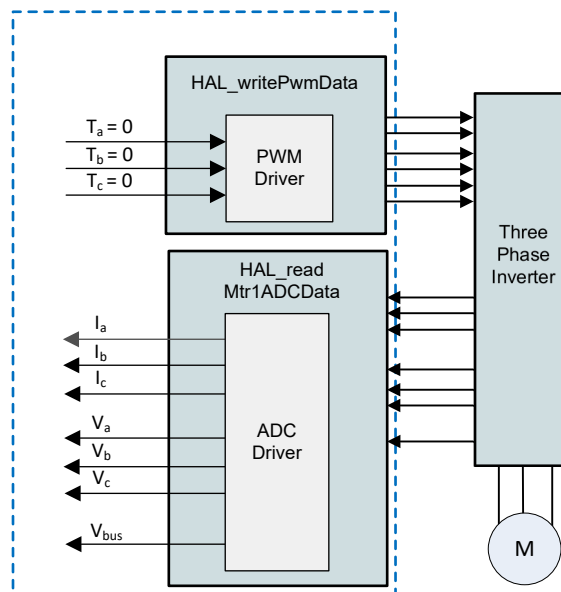


Figure 2-24. Build Level 1 Software Block Diagram - Offset Validation

2.5.1.1 Build and Load Project

1. Set up the hardware board as described in [Section 2.2](#), and connect the power supply to the board. Motor should **NOT** be connected to the terminals in this build level.
2. Connect a USB cable to on-board USB connector to enable isolation JTAG emulation to the C2000 device.
3. Power on the hardware board with apply the right voltage to the bus voltage input terminal as described in [Section 2.2](#).
4. Import the project and select the right build configuration as described in [Section 2.3.1](#). 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 make sure the correct target configuration file is set as Active as shown in [Figure 2-16](#). Do verify this by viewing the ccxml file in the expanded project structure and a [Active/Default] written next to it. By going to "View-> Target Configurations" you may edit existing target configurations or change the default or active configuration. You can also link a target configuration to a project in the workspace by right clicking on the Target configuration name and selecting Link to Project.
6. Now, 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. Notice the CCS Debug icon in the upper right-hand corner, indicating that the user is now in the Debug Perspective view. The program should be stopped 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.

2.5.1.2 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 make time (and frequency) domain plots. This ability allows the user to view waveforms using graph tool.

1. Setup watch window: Click *View* → *Expressions* on the menu bar to open an Expressions watch window . Move the mouse to the Expressions window to view the variables being used in the project. Add variables to the Expressions window as shown in [Figure 2-25](#), it uses the number format associated with variables during declaration, shows an example of the Expressions window. You can select a desired number format for the variable by right clicking on it and choosing.
2. Alternately, a group of variables can be imported into the Expressions window by right clicking within Expressions window and clicking Import, and browse to the directory of the project at <install_location>\solutions\universal_motorcontrol_lab\common\debug\ and pick *universal_lab_level1.txt* and click OK to import the variables shown in [Figure 2-25](#).

Note

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

3. The structure variables *motorVars_M1* has references to most variables that are related to controlling motor drive. By expanding this variable, you can see them all and edit as needed.
4. Click on the Continuous Refresh button  on the top right corner of the Expressions Window tab to enable periodic capture of data from the Microcontroller. By clicking the down arrow in this expressions window, you may select "Customize Continuous Refresh Interval" and edit the refresh rate of the expressions window. Note that choosing too fast an interval may affect performance.

2.5.1.3 Run the Code

1. Run the code by pressing on 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 2-25](#) 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 no any faults on the hardware kits. The *ISRCCount* should be increasing continuously.
6. Check calibration offsets of the motor inverter board, the offset value of the motor phase current sensing value should be equal to approximately half of the scale current of ADC as shown in [Figure 2-25](#).
7. Probe the PWM output for motor drive control with an oscilloscope. The three PWMs duty are set to 50% in this build level, the PWM output waveforms are as shown in [Figure 2-26](#). The PWM switching frequency will be the same as setting value, *USER_M1_PWM_FREQ_kHz* in *user_mtr1.h*.
8. Set the variables *motorVars_M1.flagEnableRunAndIdentify* to 0 to stop run the motor.
9. 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*→CPU Reset.
10. Close CCS debug session by clicking the Terminate Debug Session  or clicking *Run* → *Terminate*. This will halt the program and disconnect Code Composer from the MCU.
11. 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, pressing the button  or clicking *Run* → *Reset*→CPU Reset, and then pressing  or clicking *Run* → *Restart*. Terminate the project if the target device or the configuration is changed, and before shutting down CCS.

Expressions					Click this button to enable periodic capture of data from the microcontroller
Expression	Type	Value			
systemVars.flagEnableSystem	unsigned char	1 '\x01'			
motorVars_M1.ISRCCount	unsigned long	8934137			The variable keeps increasing
systemVars.boardKit	enum <unna...	BOARD_B5XL8323RS_REVA			
systemVars.estType	enum <unna...	EST_TYPE_FAST			
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH			
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST			Check if these variables meet the board, estimator, library selection
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'			Set this variable value equal to 1 to start the motor
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			The sensing conversion value should be equal to the dc bus voltage
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]			Current and voltage sampling values are near 0.0 with removing offset
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			Current offset values are near 2048 (half of 12bit ADC scale value)
[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			Voltage offset coefficients are about 0.5.
[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)			Three-phase PWM duty equals to 50%. The CMPA value equals to half of the TBPRD
EPwm2Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)			
EPwm3Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)			

Figure 2-25. Build Level 1: Variables in Expressions Window

The C2000 PWM with deadband outputs to the input of the gate drive as shown in [Figure 2-26](#)

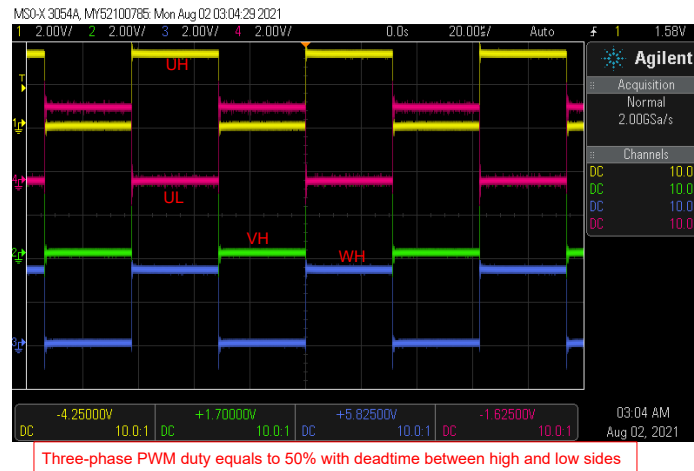


Figure 2-26. Build Level 1: PWM Output Waveforms

2.5.2 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.

This system is running with open-loop control, the ADC measured values are only used for instrumentation purposes in this build level. The software flow for this build level is shown in [Figure 2-27](#).

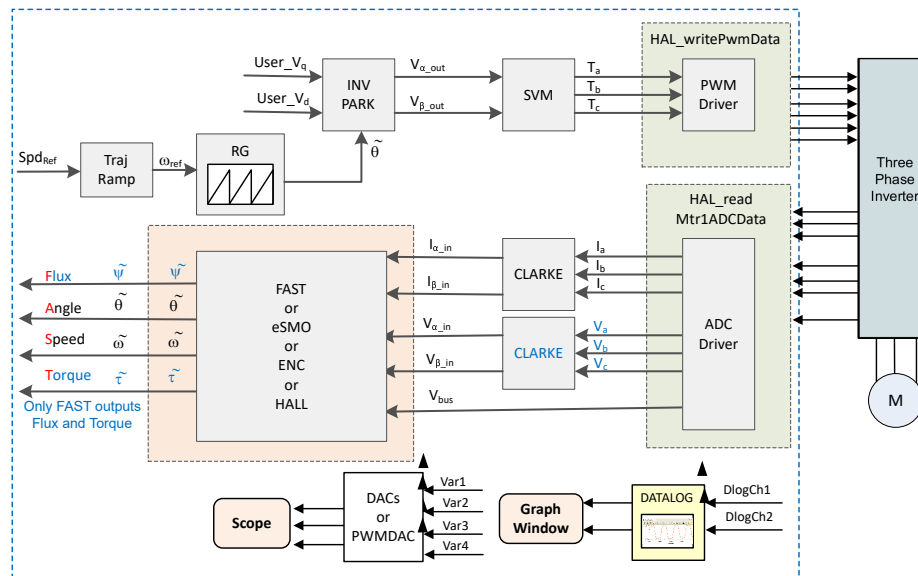


Figure 2-27. Build Level 2 Software Block Diagram - Open Loop Control

2.5.2.1 Build and Load Project

Connect the motor to the related terminals on the power inverter board. Follow the operation steps in [Section 2.5.1.1](#) to build and load project by setting DMC_BUILDLEVEL to DMC_LEVEL_2 in 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.

2.5.2.2 Setup Debug Environment Windows

Follow operation steps in [Section 2.5.1.2](#) to import the variables into the Expressions window by picking *universal_lab_level2.txt*. The Expressions window appears as shown in [Figure 2-28](#).

2.5.2.3 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 has 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 2.5.1.3](#).
3. To verify current and voltage sensing circuit of the inverter, set the variable *motorVars_M1.flagEnableRunAndIdentify* to 1 in the Expressions window as shown in [Figure 2-28](#). The motor will run with v/f open loop. Tune the v/f profile parameters in *user_mtr1.h* as below according to the specification of the motor if the motor doesn't spin smoothly.

```
#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
```

4. After that, the motor spins with a setting speed in the variable *motorVars_M1.speedRef_Hz*, check the value of *motorVars_M1.speed_Hz* in Expressions window, the values of these two variables should be very close as shown in [Figure 2-28](#).
5. Using PWM DAC or DAC128S module to as described in [Section 2.4.2](#) or [Section 2.4.3](#). Connect oscilloscope voltage probes to the outputs of the PWM DAC or DAC128S085EVM board to monitor the angle, current, voltage signals. Connect an oscilloscope current probe to the motor phase current to measure the motor phase current.
 - a. Configuring DAC128S module four inputs as the following codes. The current waveforms on the oscilloscope as shown in [Figure 2-30](#). The sampling current waveform by using a DAC to output on oscilloscope should be the same as the phase current waveform capture by a current probe, that means the current sensing circuit is good for motor control.

```
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
```

- b. Configuring DAC128S module four inputs as the following codes. The sampling voltage waveform by using a DAC to output on oscilloscope should be the similar as shown in [Figure 2-31](#) or [Figure 2-32](#), that means the voltage sensing circuit is good for motor control.

```
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
```

- c. Configuring DAC128S module four inputs as the following codes. The angle from the force angle generator or estimator waveforms on the oscilloscope as shown in [Figure 2-33](#). Notice that the angle of the force angle generator is very similar as the estimated rotor angle of the FAST or eSMO estimator, only a little bit shift error could be between these two angles. That means 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.adcData.I_A.value[0]; // CH_C
dac128s.ptrData[3] = &motorVars_M1.adcData.V_V.value[0]; // CH_D
```

6. Using datalog module with graph tool to check the current, voltage sensing signals, and the angles output as described in [Section 2.4.1](#).
 - a. Configuring DATALOG module four inputs as the following codes. The phase current sampling signals waveform on graph tool as shown in [Figure 2-34](#).


```
datalogObj->iptr[0] = &motorVars_M1.adcData.I_A.value[0];
datalogObj->iptr[1] = &motorVars_M1.adcData.I_A.value[1];
```
 - b. Configuring DATALOG module four inputs as the following codes. The phase voltage sampling signals waveform on graph tool as shown in [Figure 2-35](#).


```
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 2-36](#). 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;
```
7. 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 2-29](#).
8. Set the variables *motorVars_M1.flagEnableRunAndIdentify* to 0 to stop run the motor.
9. 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*.
10. Close CCS debug session by clicking on Terminate Debug Session  or clicking *Run* → *Terminate*.
11. Power off the power supply to the inverter kit.

Expressions				Click this button to enable periodic capture of data from the microcontroller
Expression	Type	Value		
systemVars.flagEnableSystem	unsigned char	1 '\x01'		
motorVars_M1.ISRCCount	unsigned long	8934137		The variable keeps increasing
systemVars.boardKit	enum <unna...	BOARD_B5XL8323RS_REVA		
systemVars.estType	enum <unna...	EST_TYPE_FAST		
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH		
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST		Check if these variables meet the board, estimator, library selection
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'		Set this variable value equal to 1 to start the motor
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		The sensing conversion value should be equal to the dc bus voltage
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]		Current and voltage sampling values are near 0.0 with removing offset
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]		Current offset values are near 2048 (half of 12bit ADC scale value)
[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]		Voltage offset coefficients are about 0.5.
[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)		Three-phase PWM duty equals to 50%. The CMPA value equals to half of the TBPRD
EPwm1Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)		
EPwm2Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)		
EPwm3Regs.CMPA.CMPA	pointer : 16	1250 (Decimal)		

Figure 2-28. 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 2-29.

motorVars_M1.flagClearFaults	unsigned char	0 '\x00'	
motorVars_M1.faultMtrUse.all	unsigned int	16	The value will be non-zero if there is an over-current fault
motorVars_M1.faultMtrPrev.bit	struct_FAULT_...	{overVoltage=0,underVolutag...	
motorSetVars_M1.dacCMPValH	unsigned int	2178	
motorSetVars_M1.dacCMPValL	unsigned int	1918	
motorSetVars_M1.overCurrent_A	float	1.5	Set the right current threshold value to verify the over current function
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 values will be non-zero if there is an over-current fault

Figure 2-29. 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 2-30](#).

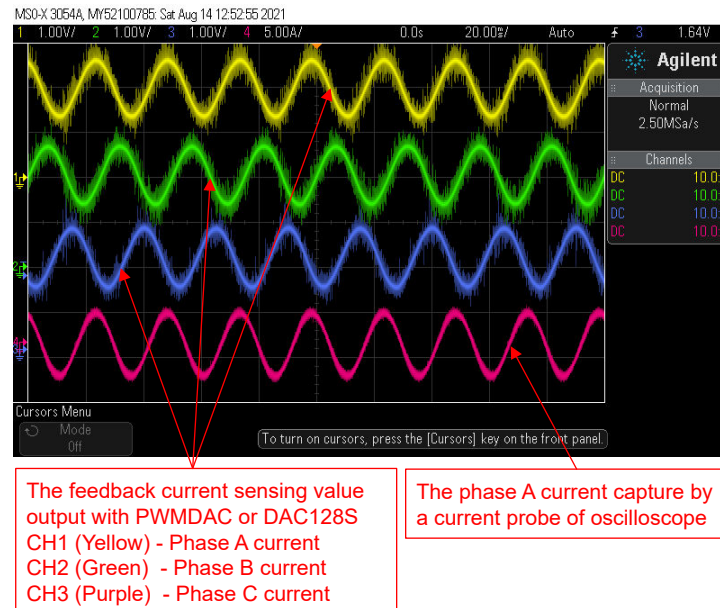


Figure 2-30. Build Level 2: Motor Phase Current Waveforms

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 2-31](#).

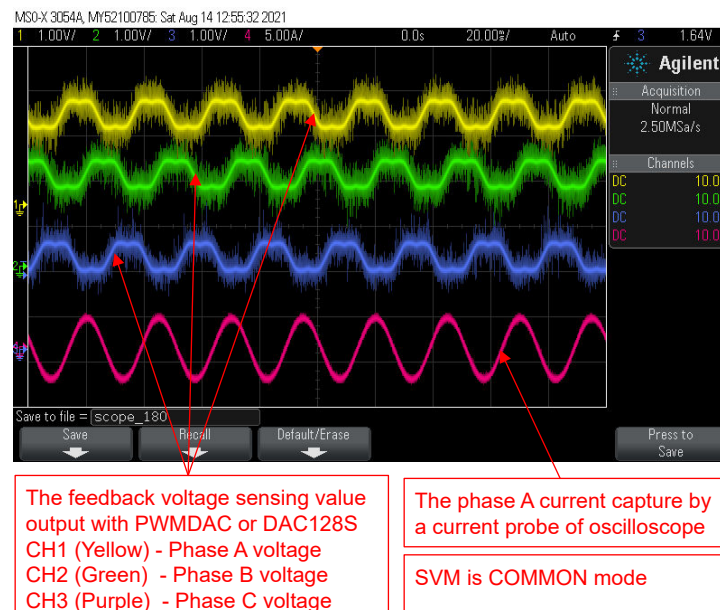


Figure 2-31. Build Level 2: Motor Phase Voltage Waveforms Using Common SVM Mode

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 2-32](#).

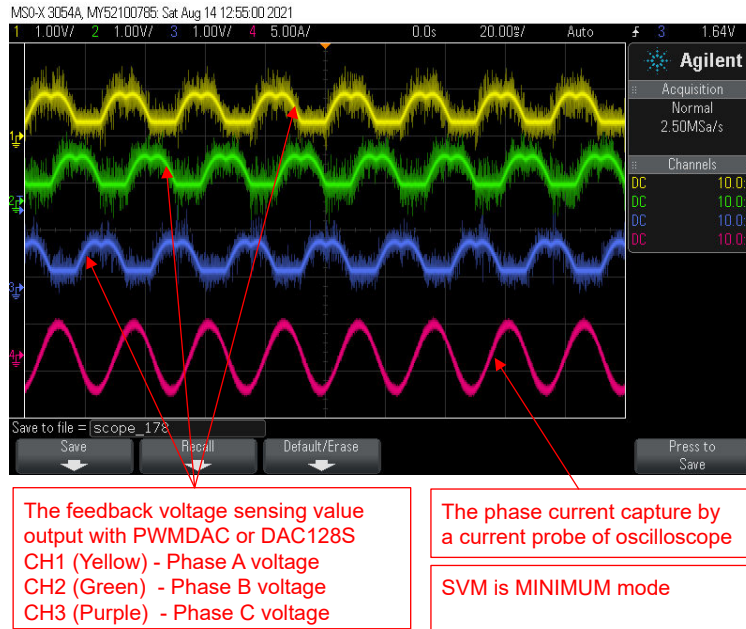


Figure 2-32. Build Level 2: Motor Phase Voltage Waveforms Using Minimum SVM Mode

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 2-33](#).

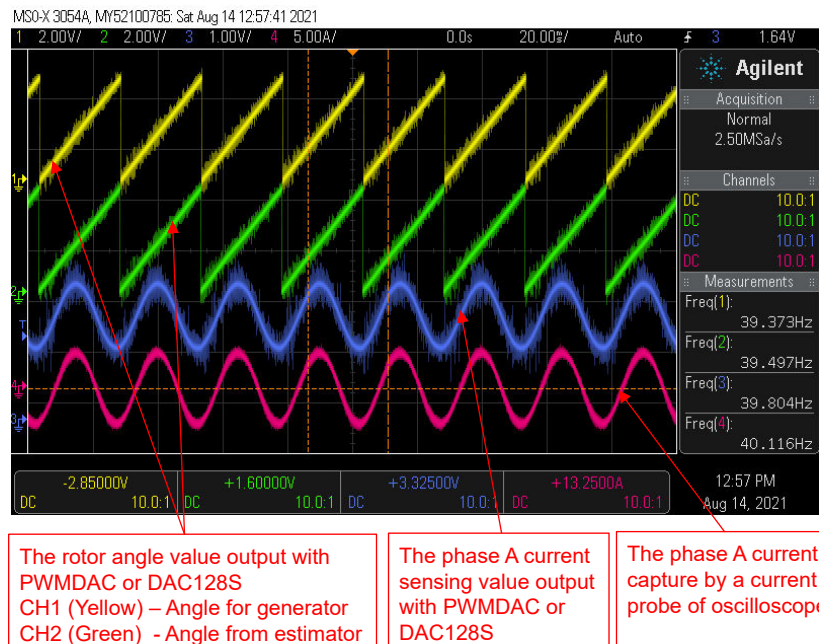


Figure 2-33. Build Level 2: Motor Rotor Angle and Phase Current Waveforms

Use Datalog with Graph Tool to monitor three phase sensing current of the motor as shown in [Figure 2-34](#).

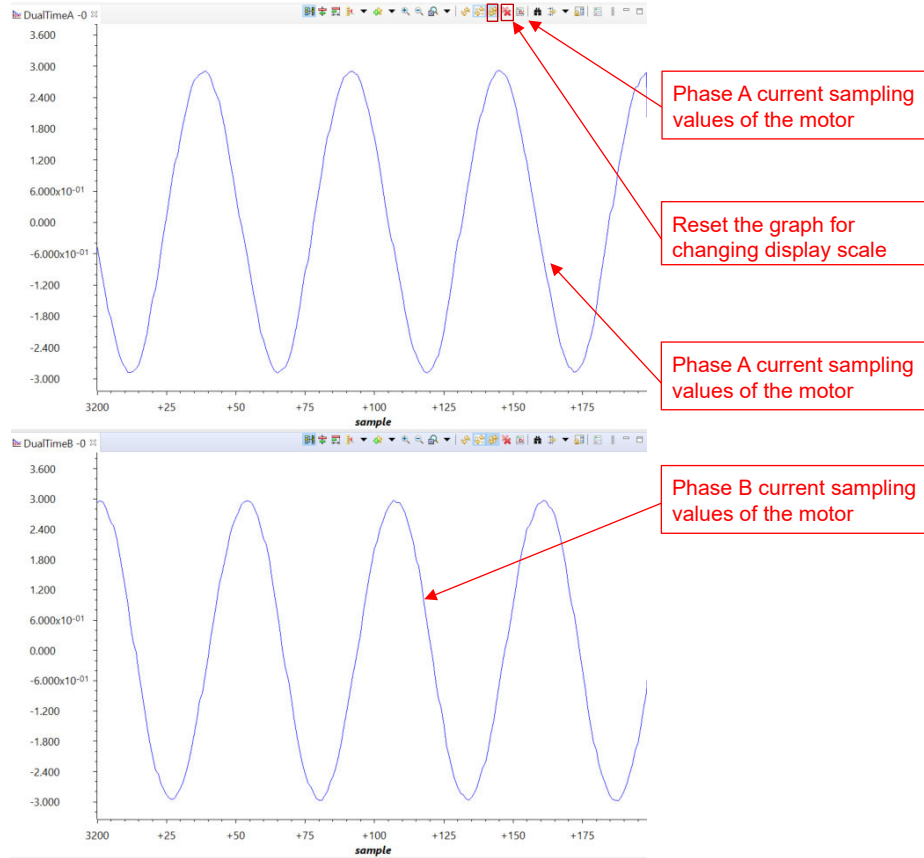


Figure 2-34. Build Level 2: Motor Phase Current Waveforms with Graph Tool

Use Datalog with Graph Tool to monitor three phase sensing voltage of the motor as shown in [Figure 2-35](#).



Figure 2-35. 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 2-36](#).

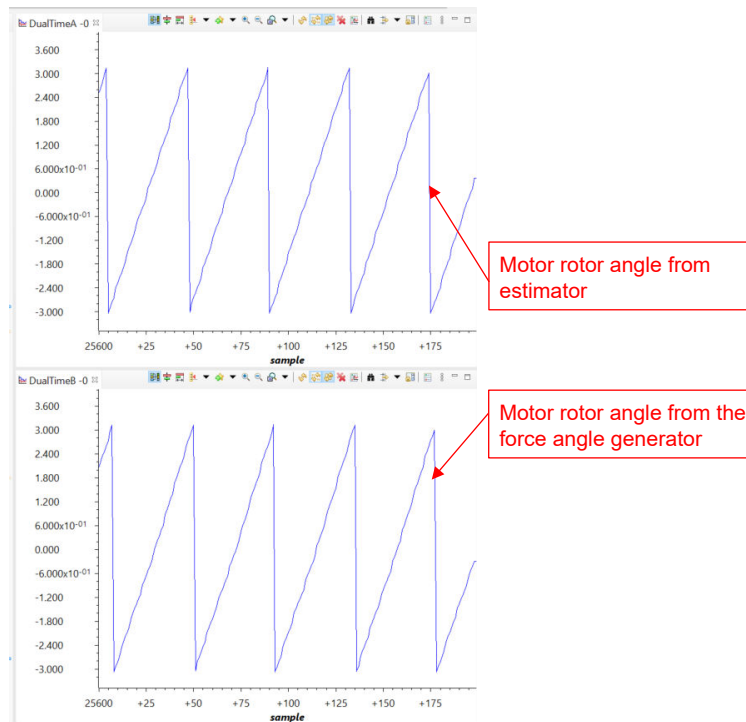


Figure 2-36. Build Level 2: Motor Rotor Angle Waveforms With Graph Tool

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 2-39](#). Change the `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 can not run with current-closed loop and appear a over current fault, 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 `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	Address
systemVars.flagEnableSystem	unsigned char	1 '\x01'	0x00C
motorVars_M1.ISRCCount	unsigned long	1745646	0x00C
systemVars.boardKit	enum <unna...	BOARD_BSL8323RS_REVA	0x00C
systemVars.estType	enum <unna...	EST_TYPE_FAST	0x00C
systemVars.fastType	enum <unna...	FAST_TYPE_SOFTLIB_FLASH	0x00C
motorVars_M1.estState	enum <unna...	EST_STATE_ONLINE	0x00C
motorVars_M1.motorState	enum <unna...	MOTOR_CTRL_RUN	0x00C
motorVars_M1.mctrlState	enum <unna...	MCTRL_CONT_RUN	0x00C
motorVars_M1.estimatorMode	enum <unna...	ESTIMATOR_MODE_FAST	0x00C
motorSetVars_M1.RoverL_rps	float	2026.49207	0x00C
motorSetVars_M1.RsOnLine_Ohm	float	0.38157931	0x00C
motorSetVars_M1.Rs_Ohm	float	0.38157931	0x00C
motorSetVars_M1.Ls_d_H	float	0.000188295482	0x00C
motorSetVars_M1.Ls_q_H	float	0.000188295482	0x00C
motorSetVars_M1.flux_VpHz	float	0.0400219113	0x00C
motorVars_M1.adcData.VdcBus_V	float	23.9467831	0x00C
motorVars_M1.speedRef_Hz	float	40.0	0x00C
motorVars_M1.speed_Hz	float	39.822998	0x00C
motorVars_M1.flagEnableRunAndIdentify	unsigned char	1 '\x01'	0x00C
motorVars_M1.flagRunIdentAndOnLine	unsigned char	1 '\x01'	0x00C
motorVars_M1.flagEnableMotorIdentify	unsigned char	0 '\x00'	0x00C
motorVars_M1.flagEnableForceAngle	unsigned char	1 '\x01'	0x00C
motorVars_M1.enableSpeedCtrl	unsigned char	1 '\x01'	0x00C
motorVars_M1.speedEST_Hz	float	40.5065231	0x00C
motorVars_M1.angleFOC_rad	float	-0.898082972	0x00C
motorVars_M1.angleEST_rad	float	0.71139878	0x00C
motorVars_M1.accelerationMax_Hzps	float	25.0	0x00C
motorVars_M1.accelerationStart_Hzps	float	10.0	0x00C
motorVars_M1.flagClearFaults	unsigned char	0 '\x00'	0x00C
motorVars_M1.faultMtrUse.all	unsigned int	0	0x00C
motorVars_M1.faultMtrPrev.bit	struct _FAULT_...	{overVoltage=0,underVoltage=0}	0x00C
motorSetVars_M1.dacCMPValH	unsigned int	2482	0x00C
motorSetVars_M1.dacCMPValL	unsigned int	1614	0x00C
motorSetVars_M1.overCurrent_A	float	5.0	0x00C
motorVars_M1.startCurrent_A	float	1.0	0x00C
motorVars_M1.maxCurrent_A	float	5.0	0x00C
motorVars_M1.Idq_set_A.value[1]	float	2.0	0x00C
motorVars_M1.IdqRef_A.value[0]	float	0.0	0x00C
motorVars_M1.IdqRef_A.value[1]	float	2.0	0x00C
motorVars_M1.Irms_A	float[3]	[1.40233207,1.41681051,1.4...	0x00C
motorSetVars_M1.Kp_Id	float	0.118309543	0x00C
motorSetVars_M1.Ki_Id	float	0.0253311507	0x00C
motorSetVars_M1.Kp_Iq	float	0.118309543	0x00C
motorSetVars_M1.Ki_Iq	float	0.0253311507	0x00C
motorSetVars_M1.Kp_spd	float	0.00585704623	0x00C
motorSetVars_M1.Ki_spd	float	0.000785398122	0x00C
pi_spd_M1	struct_PI_Obj_...	{Kp=0.00585704623,Ki=0.00...	0x00C
pi_Iq_M1	struct_PI_Obj_...	{Kp=0.118309543,Ki=0.0253...	0x00C

Figure 2-38. 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 2-39](#).

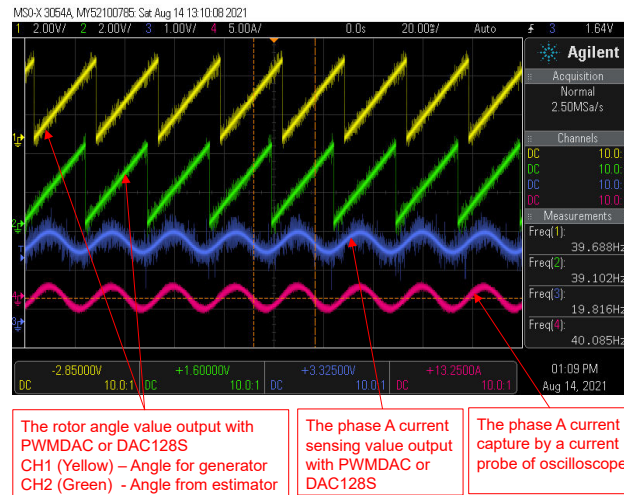


Figure 2-39. Build Level 3: Motor Rotor Angle and Phase Current Waveforms on Oscilloscope

2.5.4 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 sensed-FOC or hall based sensed-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 2-40](#).

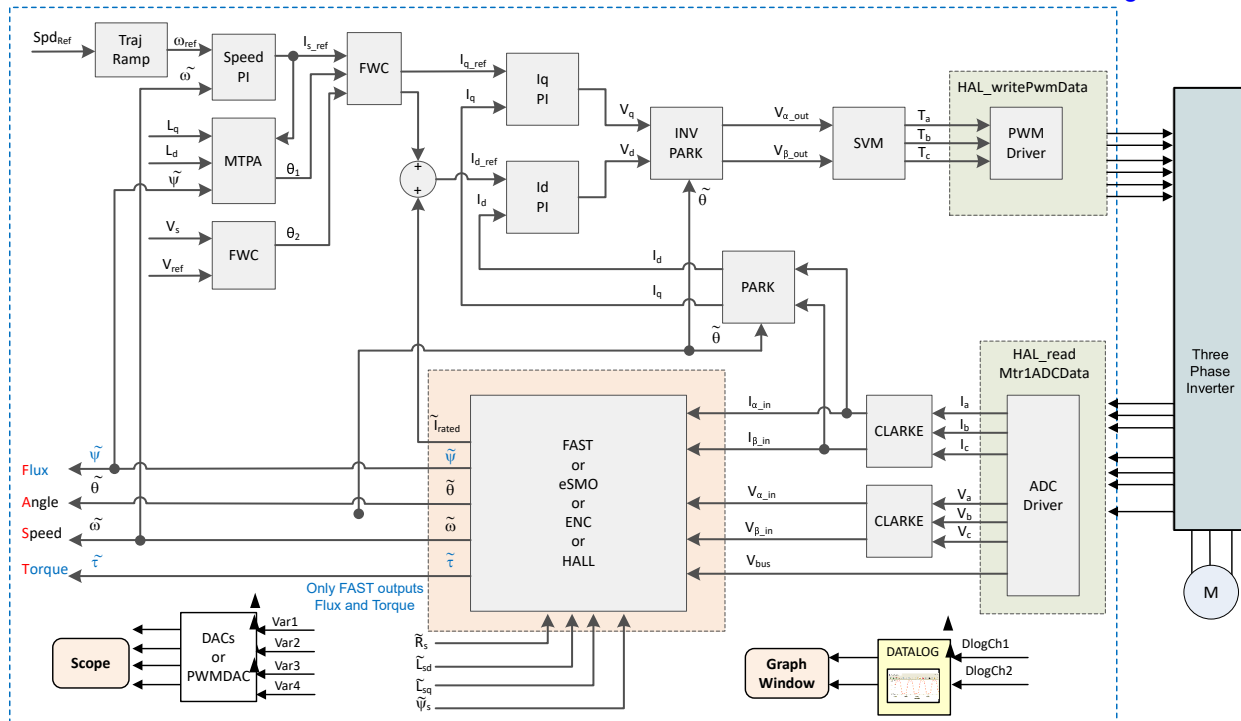


Figure 2-40. Build Level 4 Software Block Diagram - Speed and Current Close Loop Control

2.5.4.1 Build and Load Project

Connect the motor to the related terminals on the power inverter board. Follow the operation steps in [Section 2.5.1.1](#) to build and load project by setting DMC_BUILDLEVEL to DMC_LEVEL_4 in 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.

2.5.4.2 Setup Debug Environment Windows

Follow operation steps in [Section 2.5.1.2](#) to import the variables into the Expressions window by picking *universal_lab_level4.txt*. The Expressions window appears as shown in [Figure 2-41](#).

2.5.4.3 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 header files *user_mtr1.h* as shown in the following example codes. 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" to enable the motor identification as the following example code.

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

4. Set the right identification variables value in the *user_mtr1.h* 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 2.5.1](#).
6. Set the variable *motorVars_M1.flagEnableRunAndIdentify* to '1' in the Expressions window as shown in [Figure 2-41](#), 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 *user_mtr1.h* 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 `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 PWM DAC or DAC128S module to display the monitoring variables as described in [Section 2.4.2](#) or [Section 2.4.3](#). The motor angle and current waveforms are shown in [Figure 2-42](#).
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 2-41](#) 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 2-41](#) 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`.

Expression	Type	Value	Ad
systemVars.flagEnableSystem	unsigned char	1 '\x01'	0xC
motorVars_M1.ISRCount	unsigned long	762376	0xC
systemVars.boardKit	enum <unnamed>	BOARD_BSL8323RH_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

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

Figure 2-41. 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 2-42](#) when the motor is running at forward rotation by setting `motorVars_M1.speedRef_Hz` to a positive reference value.

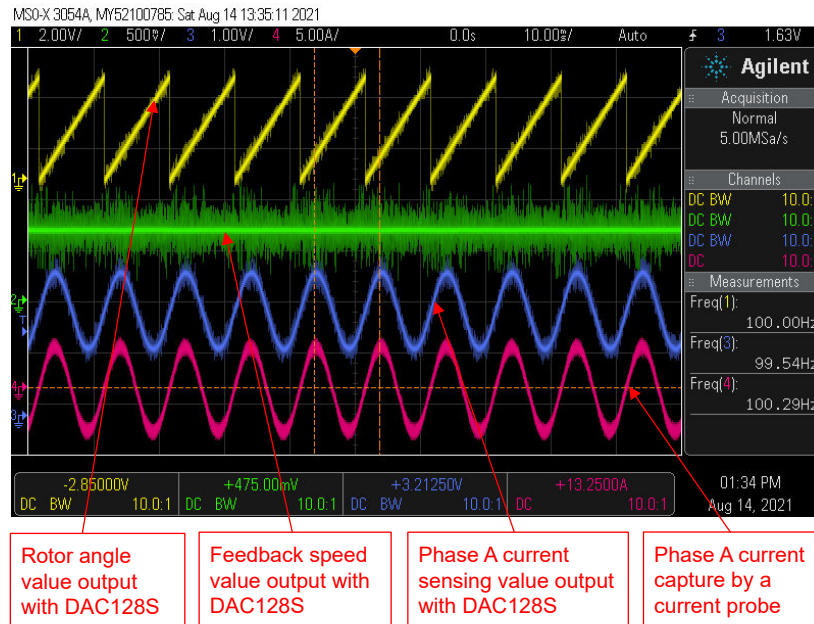


Figure 2-42. Build Level 4: Rotor Angle with FAST, Phase Current Waveforms at Forward Move

As illustrated in [Section 2.3.4](#), 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 2-7](#).

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 2.3.1](#). Rebuild, load and run the project as the operation steps above. The settings will be as shown in [Figure 2-41](#).

- 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 2-43](#). 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 2-47](#). 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 2-43](#) when the motor is running at forward rotation by setting `motorVars_M1.speedRef_Hz` to a positive reference value.

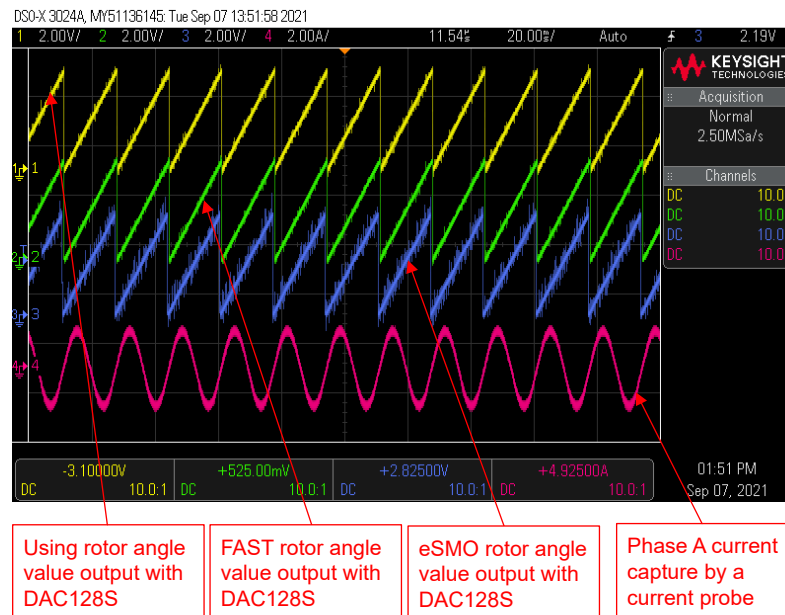


Figure 2-43. Build Level 4: Rotor Angle with FAST and eSMO, Phase Current Waveforms at Forward Rotation

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 2-44](#) when the motor is running at reversal rotation by setting `motorVars_M1.speedRef_Hz` to a negative reference value.

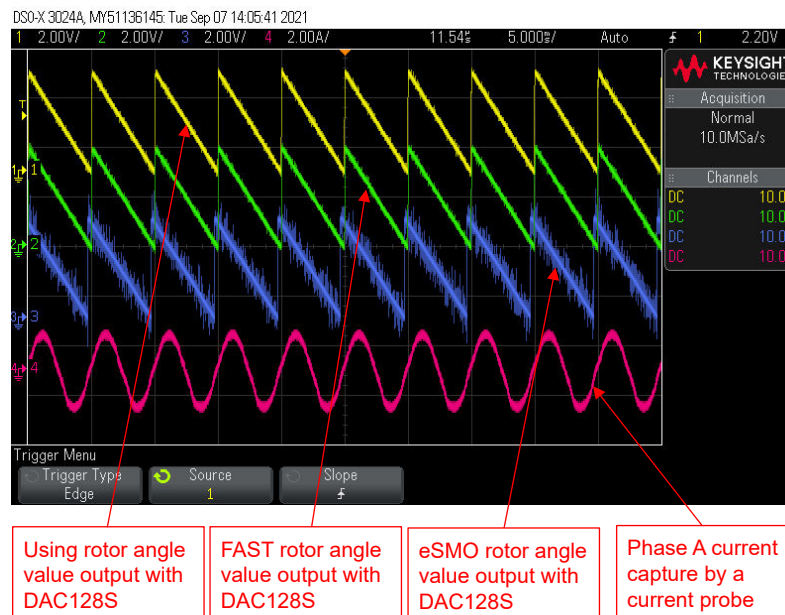


Figure 2-44. Build Level 4: Rotor Angle With FAST and eSMO, Phase Current Waveforms at Reversal Rotation

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 2.3.1](#). 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 sensed-FOC.
- The estimated rotor angles from FAST and Encoder are shown in [Figure 2-45](#). 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 sensed-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 2-45](#) when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

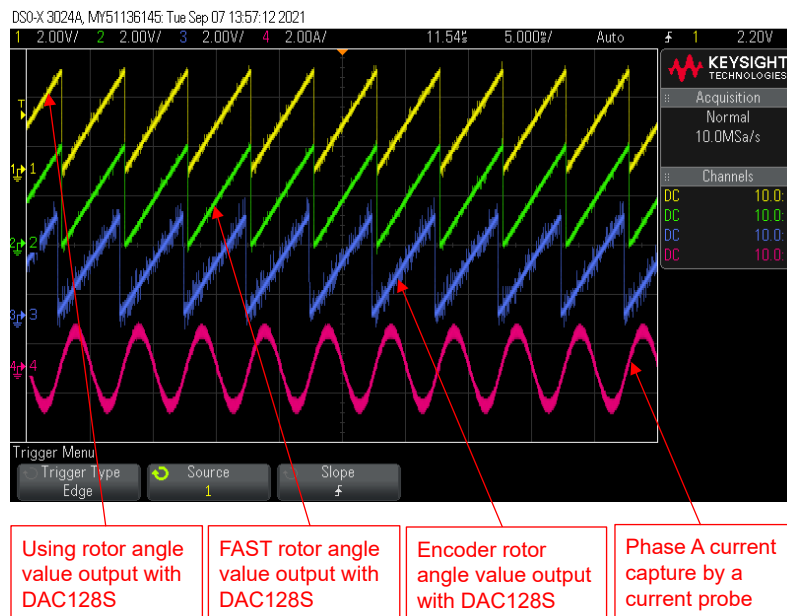


Figure 2-45. Build Level 4: Rotor Angle with FAST and Encoder, Phase Current Waveforms at Forward Rotation

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 2.3.1](#). 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 sensed-FOC.
- The estimated rotor angles from FAST and Hall sensor are shown in [Figure 2-46](#). 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 2-46](#). 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 sensed-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 2-46](#) when the motor is running at forward rotation by setting `motorVars_M1.speedRef_Hz` to a positive reference value.

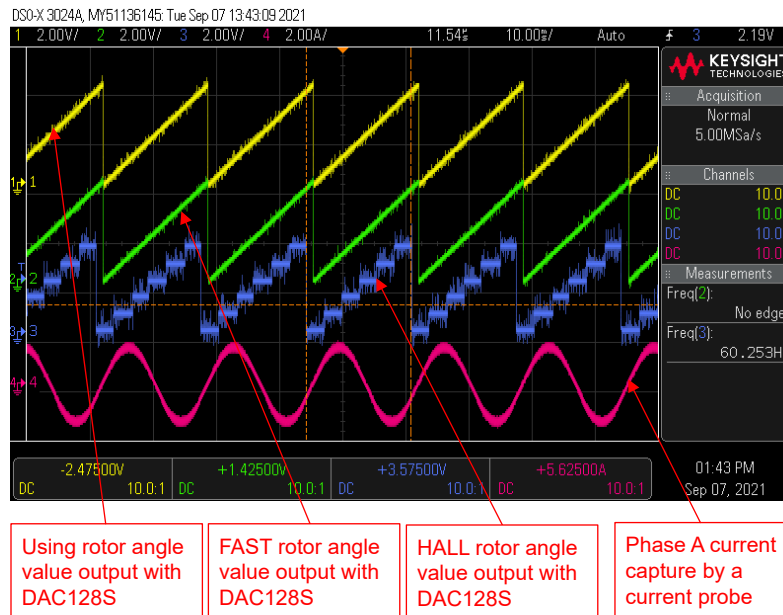


Figure 2-46. Build Level 4: Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Forward Rotation

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 2-47](#) when the motor is running at reversal rotation by setting `motorVars_M1.speedRef_Hz` to a negative reference value.

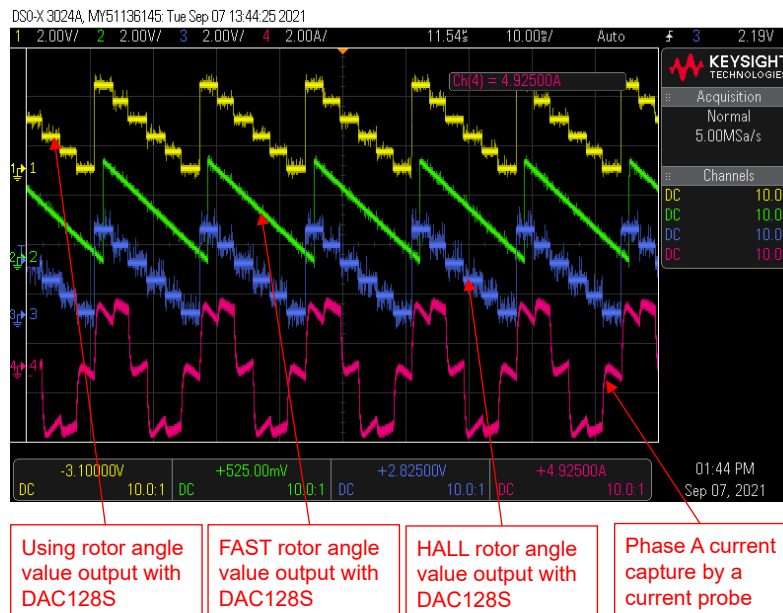


Figure 2-47. Build Level 4: Motor Rotor Angle with FAST and Hall Sensor, Phase Current Waveforms at Reversal Rotation

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 2.3.1](#). 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-FO, equals to ESTIMATOR_MODE_ENC that means the encoder estimator is using for sensed-FOC.
- The estimated rotor angles from eSMO and Encoder are shown in [Figure 2-48](#). 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 sensed-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 2-48](#) when the motor is running at forward rotation by setting *motorVars_M1.speedRef_Hz* to a positive reference value.

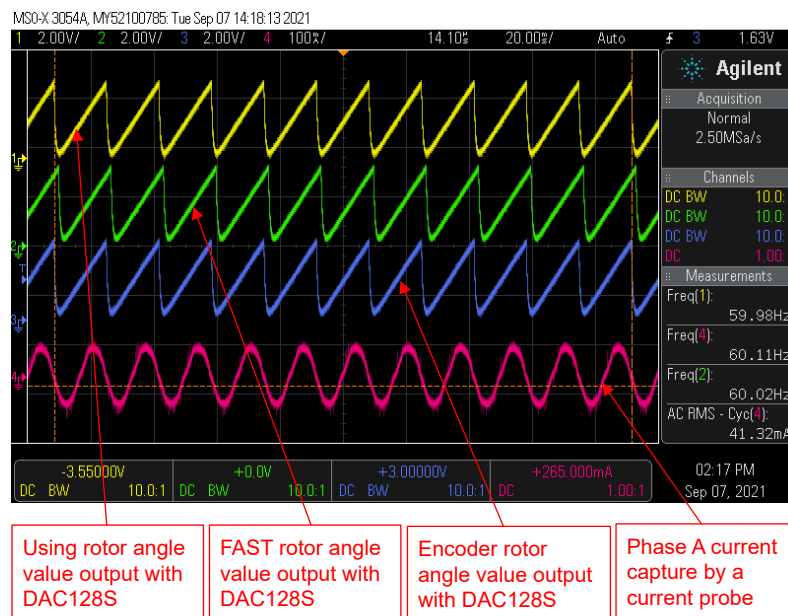


Figure 2-48. Build Level 4: Rotor Angle with eSMO and Encoder, Phase Current Waveforms at Forward Rotation

3 Building Custom Board

3.1 Building New Custom Board

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

3.1.1 Hardware Setup

If using a TI hardware kit, please follow the steps in [Section 2](#) to set up the inverter board and LaunchPad or controlCARD.

If using a customization board, please make sure that the power supply to C2000 controller and gate driver is correct, and the JTAG emulator can be connected successfully. Migrate the reference code to customization board as described in the following sections, and then run the code from build level 1 first to build level as illustrated in [Section 2.5](#).

3.1.2 Migrating Reference Code to Customization Board

To migrate the reference code to a new TI hardware kits or a customization board, the user needs to set the hardware and motor control parameters according to the hardware circuit in user_mtr1.h file, and configure the related peripherals in hal.h and hal.c files as described in the following sections.

The following state machine summarizes the related HAL configuration and motor control settings functions calls as shown in [Figure 3-1](#).

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 configuration of either a peripheral, or a driver IC.

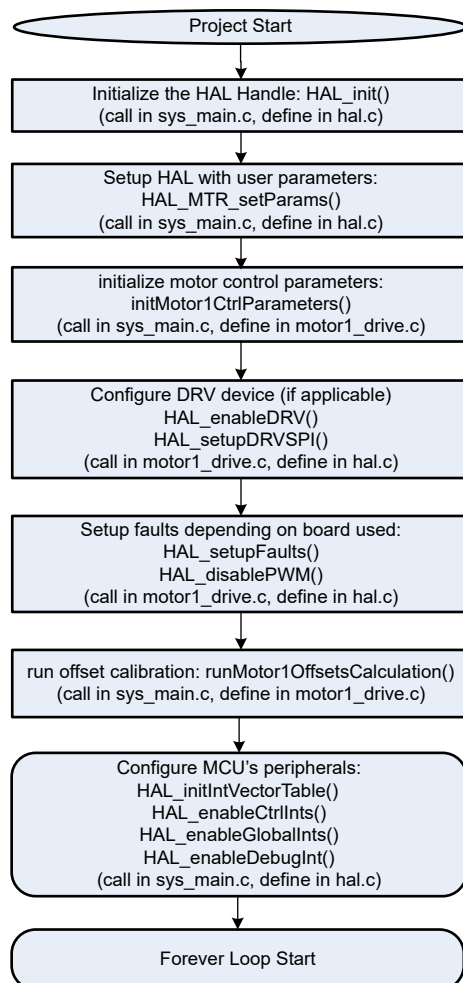


Figure 3-1. HAL Configuration and Motor Control Setting State Machines

3.1.2.1 Setting Hardware Board Parameters

The `user_mtr1.h` is where all user parameters are stored for motor control. The maximum phase current and phase voltage at the input to the AD converter, these values are hardware dependent and should be based on the current and voltage sensing and scaling to the ADC input. The number of phase current sensors and phase voltage sensors used are defined in `user_mtr1.h` that are hardware dependent.

All of the configurable parameters are defined in the `user_mtr1.h` file, which can be calculated using the Motor Control Parameters Calculation.xlsx Excel® spreadsheet. This file is included with the archive file at the folder: `\solutions\universal_motorcontrol_lab\doc` and copy these parameters marked **bold** to `user_mtr1.h` as shown in the following codes.

```

/// \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

```

3.1.2.2 Modifying Motor Control Parameters

The parameters provided in `user_mtr1.h` file for PMSM/BLDC motor are listed as shown in the following codes. The motor parameters can be identified if the FAST technique is implemented on this motor or getting from the motor data sheet.

```

#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)

```

3.1.2.3 Changing Pin Assignment

The `HAL_setupGPIOs()` function of `hal.c` configures the alternate function of the GPIO pins and sets the direction and mode of the specified pin according to the hardware board/kit used. Be careful to set the pad configuration for the specified pin, especially GPIOs will be used as PWM output function.

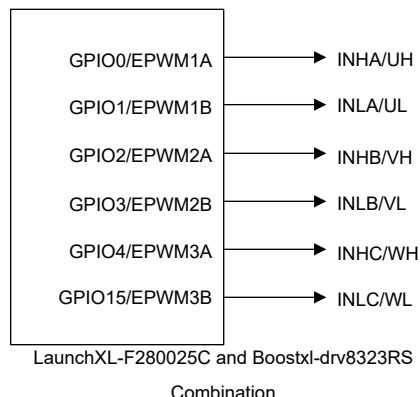
```

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

```

3.1.2.4 Configuring PWM Module

The HAL module configures the PWM channels. The application parameters to control the motors are written as `#define` configuring the PWM modules base address in "hal.h" according to the board, and the base address will be assigned to the PWM handles in `hal.c`. For example, the connection diagram on [LAUNCHXL-F280025C](#) and [BOOSTXL-DRV8323RS](#) combination as shown in [Figure 3-2](#).

**Figure 3-2. PWM Connection Diagram**

Which is configured as shown in following steps (Changes are **bold** for a customer's board), taken from hal.h and hal.c located in the following location: solutions\universal_motorcontrol_lab\28002x\drivers\include and \source:

1. Defines the PWM modules base address in "hal.h".

```

//! \ Motor 1
#define MTR1_PWM_U_BASE      EPWM1_BASE
#define MTR1_PWM_V_BASE      EPWM2_BASE
#define MTR1_PWM_W_BASE      EPWM3_BASE

```

2. Sets up the GPIOs as PWM outputs in HAL_setupGpios() in hal.c.

```

// 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. Assigns the address of the PWM modules used to the PWM handle in HAL_MTR1_init() in hal.c. Don't need to change the following code, it's just to show how to link the PWM module.

```

// 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. Configures the PWM in HAL_setupPWMS() in hal.c. Notice that the desired PWM period (USER_M1_PWM_TBPRD_NUM) for setting 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 hal.h, change these parameters according to the hardware board and control requirement. And also set up the PWM counter mode, PWM action qualifier outputs 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. Sets up the event trigger for ADC as above code snippet which must accommodate the definition in hal.h.

```
// Three-shunt
#define MTR1_ADC_TRIGGER_SOC      ADC_TRIGGER_EPWM1_SOC // EPWM1_SOC
```

3.1.2.5 Configuring ADC Module

Similar to the previous PWM section, the ADC can also be changed to accommodate another board compared to the TI motor control kits. The HAL module configures the ADC channels according to the using hardware board. For example, the connection diagram on Launchxl-f280025c and Boostxl-drv8323rs combination as shown in [Figure 3-3](#). The ADC modules configuration is described as following steps (Changes are highlighted for a customer's board). The step 1 and 2 are must-do for new hardware to run the motor.

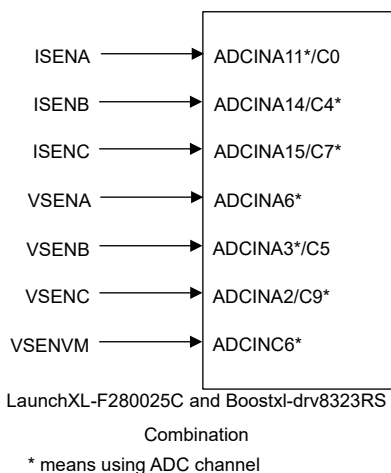


Figure 3-3. ADC Connection Diagram

Which is configured as shown in following steps (Changes are highlighted for a customer's board), taken from hal.h and hal.c located in the following location: solutions\universal_motorcontrol_lab\28002x\drivers\include and \source:

1. Defines the ADC modules base address, assigned channels and SOC in "hal.h"

```
#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    ADCARESLT_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    ADCARESLT_BASE     // ADCA-A6*
#define MTR1_VV_ADCRES_BASE    ADCARESLT_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   ADCARESLT_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_PP_B_NUM      ADC_PP_B_NUMBER1     // ADCA-A11*/C10-SOC1-PPB1
#define MTR1_IV_ADC_PP_B_NUM      ADC_PP_B_NUMBER1     // ADCC-A14/C4*  -SOC1-PPB1
#define MTR1_IW_ADC_PP_B_NUM      ADC_PP_B_NUMBER2     // ADCC-A15/C7*- SOC2-PPB2
```

2. Defines the interrupt sources for ISR in "hal.h".

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

#define MTR1_ADC_INT_BASE        ADCC_BASE            // ADCC-C6  -SOC6
#define MTR1_ADC_INT_NUM         ADC_INT_NUMBER1      // ADCC_INT1-SOC6
#define MTR1_ADC_INT_SOC         ADC_SOC_NUMBER6      // ADCC_INT1-SOC6

#define MTR1_PIE_INT_NUM         INT_ADCC1            // ADCC_INT1-SOC6
#define MTR1_CPU_INT_NUM         INTERRUPT_CPU_INT1   // ADCC_INT1-CPU_INT1
#define MTR1_INT_ACK_GROUP       INTERRUPT_ACK_GROUP1 // ADCC_INT1-CPU_INT1
```

3. Sets up ADC module in HAL_setupADCs() in hal.c if needed. If the users want to add any ADC channels to sample additional signals. This is done in function HAL_setupAdcs() of hal.c file as follows:

```
// 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. Read the ADC result in HAL_readMtr1ADCData() of hal.h file as follows if the users need to add additional signals as above.

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

3.1.2.6 Configuring CMPSS Module

Configure the connections between ADC pin and CMPSS modules in "hal.h" based on the hardware. For details, 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 using hardware board. For example, the connection diagram on Launchxl-f280025c and Boostxl-drv8323rs combination as shown in [Figure 3-4](#). The CMPSS modules configuration is described as following steps (Changes are **bold** for a customer's board).

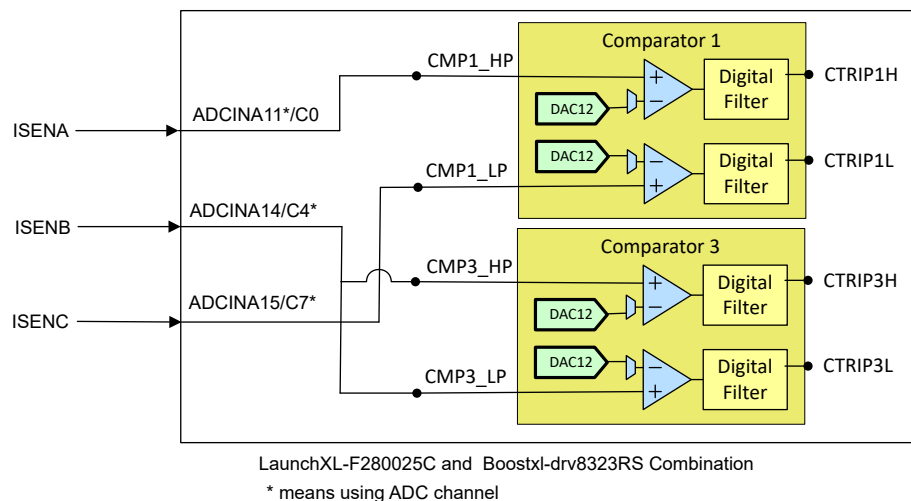


Figure 3-4. CMPSS Connection Diagram

1. Defines the CMPSS modules base address, assigned channels and SOC's in "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. Defines the selection value for CMPSS in hal.h file as following.

```
// CMPSS
#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*

#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. Sets up ADC module in HAL_setupCMPSSs() of hal.c file according to the using hardware. In this example, the CMPSS1_HP is linked to phase A current, and CMPSS1_LP is linked to phase C current, so the CMPSS1 is configured twice to simplify the codes. The user may refer to the codes for High Voltage kit to configure the CMPSS modules, respectively.

```
void HAL_setupCMPSSs(HAL_MTR_Handle handle)
{
    HAL_MTR_Obj *obj = (HAL_MTR_Obj *)handle;
    ...
    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 maximum 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 COMPHYCTL 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;
}
```

3.1.2.7 Configuring Fault Protection Function

Configure the trip signals from CMPSS and trip input to be passed to EPWM in "*hal.h*" based on the hardware, the details refer to ePWM X-BAR Mux Configuration Table and OUTPUT X-BAR Mux Configuration Table in [TMS320F28002x Real-Time Microcontrollers Technical Reference Manual](#).

1. Defines the GPIOs linked to fault signal for trip input in "*hal.h*" file as following:

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

2. Defines ePWM X-BAR to link the signals from CMPSS and GPIO to trip zone sub module of ePWM in "*hal.h*" file as following:

```
#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
#define MTR1_IDC_XBAR_MUX XBAR_MUX05 // CMPSS3-LP

#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. Configures ePWM X-BAR and trip mechanism for motor control in HAL_setupMtrFaults() of "*hal.c*" file as following:

```
void HAL_setupMtrFaults(HAL_MTR_Handle handle)
{
    ...
    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], MTR1_TZ_OSHT1);

    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

```

3.1.3 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 communication bus like SCI or CAN.

3.1.3.1 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 GPIO23 is connected to a button, enables the pre-define symbol `CMD_SWITCH_EN` in project build properties as shown in [Figure 2-14](#). The reading GPIO state will be assigned to `motorVars_M1.flagEnableRunAndIdentify` to start/stop the motor.

Follow the following steps to connect a button to the other GPIO.

1. Defines GPIO number in `hal.h` file

```
#define MTR1_CMD_SWITCH_GPIO    23
```

2. Configure the GPIO in `HAL_setupGpios()` function of `hal.c` file to allow this pin to be inputs.

```
// 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. Reads a GPIO with a digital filter as follows in `updateCmdSwitch()` in `motor_common.c` file.

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

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

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

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

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

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

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

3.1.3.2 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 ADCA12 is connected to a potentiometer, enables the pre-define symbol CMD_POT_EN in project build properties as shown in [Figure 2-14](#). The reading ADC result will be converted to a setting speed 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. Defines ADC channel for connecting to potentiometer in 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 hal.c file as follows:

```
// 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);
```

3. Reads the ADC result register and scales the value in HAL_readMtr1ADCData() of 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 reading value to the setting speed in updateExtCmdPotFreq() of motor_common.c file.

3.1.3.3 Adding CAN Functionality

The CAN functionality is added into the lab project as communication bus for setting the start/stop command and getting the feedback running states. Enables the pre-define symbol CMD_CAN_EN in project build properties as shown in [Figure 2-14](#). 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".

<Installation\c2000ware\driverlib\f28002x\driverlib\can.c

2. Edit the HAL_Obj to add canHandle in "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 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 "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 "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 hal.c file.

```
SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CANA);
```

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

```
// 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. Prototypes the CAN interrupt setup function in "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 sys_main.c.

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

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

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

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

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

14. Define the CAN interrupt routine canaISR() in "communication.c" file. Define the "canaISR" 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 "communication.h" file and "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 forever 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;
    }
}
```

3.2 Supporting New BLDC Motor Driver Board

C2000 MCUs can be used with BLDC motor driver for driving three-phase BLDC or PMSM motor applications. This universal lab project can support various 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 the newer or other BLDC motor drivers. In this section, uses DRV8323RS with SPI device as an example.

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

If the BLDC motor driver is an SPI supporting device, you might refer to the existing BLDC motor driver file, drv8323s.h and drv8323s.c to change the registers and API functions definitions in drv8xxx.h and drv8xxx.c. 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 folder to locate the driver file as DRV8323 ("libraries\drv8323\include" and "libraries\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.projectspeg" projectspec file, add the files to the project as the following

```
<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" />
```

Or 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 include header file in 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"

Or add the directory to the header files by adding the following content in the "universal_motorcontrol_lab.projectspeg" 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 DRV8323 to add the supporting codes as the following steps.

Add the defines in 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 DRVIC_VARS_Handle DRV8323_VARS_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 and communicate with the BLDC motor driver device properly.

Configure the related GPIOs to SPI function in HAL_setupGPIOs() of "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 "hal.c":

```
// 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_POLOPHA0,
              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);
```

- 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 "hal.c" and the defines in "hal.h" files as the following:

```
//! \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
```

- Call in HAL_setupSPI() and HAL_setupGate() functions in HAL_MTR_setParams() of "hal.c":

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

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

- Call the drivers functions in "motor1_drive.c" 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.OCF_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);
```

3.3 Porting Reference Code to New C2000 MCU

The Motor Control Universal Lab project can be easily ported to other FPU and TMU enabled C2000 MCU controllers. For example, port the lab from F28002x to F28004x. The detailed steps are described in the following.

1. Browse to "<installation>\solutions\universal_motorcontrol_lab", copy the whole "f28002x" folder and paste it in this directory, change the folder name to "f28004x".
2. Browse to "<installation>\solutions\universal_motorcontrol_lab\f28004x\cmd", change the two ".cmd" files names in this folder to "f28004x_dcsn_lnk_eabi.cmd" and "f28004x_flash_lib_is_eabi.cmd".
3. Browse to "<installation>\solutions\universal_motorcontrol_lab\f28004x\ccs\motor_control", use an editor to open the projectspec file.

Change the text marked **bold** below for F28004x based project.

```
name="universal_motorcontrol_lab_f28004x"
device="TMS320F280049C" cgtVersion="20.2.5.LTS" outputFormat="ELF" launchWizard="False"
linkerCommandFile="" enableSysConfigTool="true" sysConfigBuildOptions="--product $
{C2000WARE_ROOT}\.metadata\sdk.json --device F28004x --package 100PZ --part F28004x_100PZ"
```

Find "--idiv_support=idiv0" and replace it with a space since the F28004x doesn't support "idiv" function.

Open the "replace" in the editor menu, to find and replace all of the "28002x" with "28004x", and the "280025C" with "280049C" in "universal_motorcontrol_lab.projectspec" file.

4. Import the "universal_motorcontrol_lab_f28004x" project into CCS, open the "f28004x_flash_lib_is_eabi.cmd" file to change the memory map according to the used F28004x device. The generic C2000Ware cmd files can be used as a reference.
5. Modify the GPIO, PWM, ADC, and CMPSS modules and channels defines in "hal.h" file according to F28004x based hardware kit as described in [Section 3.1.2](#).
6. Modify the GPIO, PWM, ADC, and CMPSS assigned modules configurations in "hal.h" file according to F28004x based hardware kit.
7. Rebuild the lab project, any errors in the project will be displayed in the CCS Console window, follow the prompting messages to fixed the errors and warnings. There are a few difference on the APIs of driverlib between F28002x and F28004x.
8. Refer to the example functions in C2000Ware or MotorControlSDK to add the function for PGA peripheral, if needed.

4 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](#)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (<https://www.ti.com/legal/termsofsale.html>) or other applicable terms available either on [ti.com](https://www.ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2021, Texas Instruments Incorporated