Servo Drive with CAN - Lab Projects User's guide

v0.3.0

Table of Contents

1.0 Abstract 2

2.0 Lab Description 2

2.1 Supporting Software 2

2.2 Supporting Kits 2

3.0 Software Setup 3

4.0 Hardware Setup 5

4.1 For testing DMC_LEVEL_1 and DMC_LEVEL_2 5

4.2 For testing DMC_LEVEL_3 8

4.3 For testing DMC_LEVEL_4 8

5.0 System Software Integration and Testing 10

5.1 DMC_LEVEL_1: Open loop Control to verify integrity of user hardware 10

5.2 DMC_LEVEL_2: Closing speed loop with step response Generation & Graphing for Controller Tuning 14

5.2.1 Step response generation for PI current controller 16

5.2.2 Step response generation of PI speed controller 17

5.3 DMC_LEVEL_3 : Closing speed loop using QEP encoder with CAN communication 18

5.4 DMC_LEVEL_4: Closing speed loop using QEP encoder with CAN communication for mobile robot system 21

6.0 Additional Debug information 23

IMPORTANT NOTICE AND DISCLAIMER 25

Document History

Version	Date	Notes
0.1	Sep 2021	First version
0.2	Sep. 2022	Second version, add F28003x support
0.3	Mar. 2023	Third version, add robot demo

1.0 Abstract

This document discusses how to set up a motor drive platform using Texas Instruments' motor EVM kits using C2000 to implement Sensored-FOC. This is based on motor drive using QEP encoder and integrates the CAN communication modules to control the motor status and speed using a host controller.

2.0 Lab Description

The system is incrementally built up in order for the final system can be confidently operated. Three phases of the incremental system build are designed to verify the major modules in the system. In each build level, a certain operation of the system, could be hardware or software, is verified and integrated incrementally. In the final build level, all operations are integrated to make a complete system. Table 1 summarizes the modules testing and using in each incremental system build.

Table 1 Functions verified in each incremental system build

Build level	Functional Integration
Level 1	Open loop control to verify integrity of user hardware and calibration of feedbacks
Level 2	Closing speed loop with step response Generation & Graphing for Controller Tuning
Level 3	Closing speed loop using QEP encoder with CAN communication
Level 4	Closing speed loop using QEP encoder with CAN communication for mobile robot system

2.1 Supporting Software

- Download and install the latest Code Composer Studio, version 12.0 or above is recommended, preferably in *C:/ti/<ccs_version>*
    - Reboot the system after installation!
- Download and install the Motor-Control SDK. Installation path is as follows –
  `C:\ti\c2000\C2000Ware_MotorControl_SDK_<version>\`
- servo_with_can lab is available in C2000Ware_MotorControl_SDK_<version> for F28004x, F28003x and F28002x
    - The folders **Solutions** and **Libraries** contains the required files, choose accordingly for your respective C2000 board.
    - Detect and import example projects of MotorControl SDK by using Resource Explore in CCS. View MotorControl SDK Resource Explorer on the web, or Resource Explorer can also be opened by going to the menu View-> Resource Explorer in CCS.
    - Read the documentation: C2000Ware Motor Control SDK Getting Started Guide.

```
C:\ti\c2000\C2000Ware_MotorControl_SDK_<version>\solutions\servo_drive_
with_can
```

2.2 Supporting Kits

The required hardware includes the following -

- C2000 LaunchPad™ development kits
    - LAUNCHXL-F280049C LaunchPad
    - C2000™ Piccolo™ F28004x Series LaunchPad™ Development Kit
- 3-phase Inverters
    - BOOSTXL-DRV8320RS

or

- C2000 LaunchPad™ development kits
    - LAUNCHXL-F280025C LaunchPad
    - F28002x Series LaunchPad™ Development Kit
- C2000 LaunchPad™ development kits

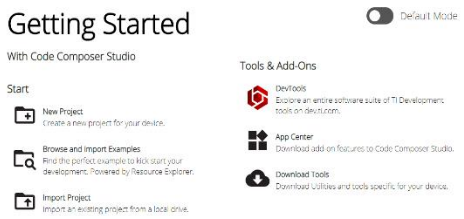
        - LAUNCHXL-F280039C LaunchPad
        - C2000™ F28003x Series LaunchPad™ Development Kit
    - 3-phase Inverters
        - BOOSTXL-DRV83**23**RS

If you are building for DMC_LEVEL_1 or 2, a single C2000 board should suffice. For DMC_LEVEL_3, to test the CAN interface an additional C2000 as a host controller is needed either F280049C, F280025 or F280039C. For DMC_LEVEL_4, the host controller is selected originally as TDA4VM, connecting with two F280039C boards. To only test the CAN interface, the host controller can be also either F280049C, F280025 or F280039C.

- Low voltage servo motor which includes an incremental quadrature encoder.
- ***2x*** USB to type-B cables for F280049C or F280025C board, or 1 USB to type-C cable for F280039C board.
- Jumper wires.

3.0 Software Setup

- Launch Code Composer Studio (ccs) and click on import new project. Note that this document assumes version 12.0.0 or later.
- Once Code Composer Studio opens, the workspace launcher to select a workspace location.
    a. Click the "Browse…" button. Create the path below by making new folders as necessary.
    b. Click "Launch" to open CCS



Getting Started
With Code Composer Studio

Start

New Project
Create a new project for your device.

Browse and Import Examples
Find the perfect example to kick start your development. Powered by Resource Explorer.

Import Project
Import an existing project from a local drive.

Tools & Add-Ons

DevTools
Explore an entire software suite of TI Development tools on dev.ti.com.

App Center
Download add-on features to Code Composer Studio.

Download Tools
Download Utilities and tools specific for your device.

- Import the servo drive with can project by clicking "Project->Import CCS Project"
- Navigate to the following path -
    - For f28004x board

```
        C:\ti\c2000\C2000Ware_MotorControl_SDK_<version>\solutions\
servo_drive_with_can\f28004x
```

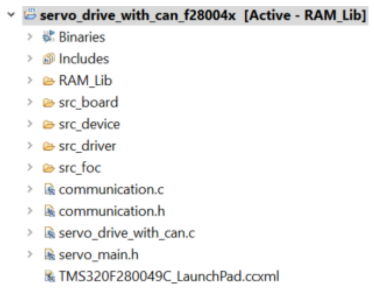
    - For f28002x board

```
        C:\ti\c2000\C2000Ware_MotorControl_SDK_<version>\solutions\
servo_drive_with_can\f28002x
```

    - For f28003x board

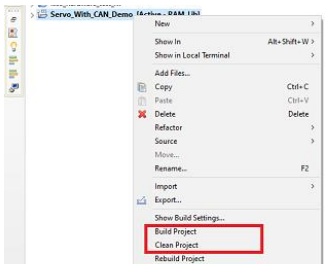
```
        C:\ti\c2000\C2000Ware_MotorControl_SDK_<version>\solutions\
servo_drive_with_can\f28003x
```

- Import and this may take a while to load. The following file is imported for f28004x, shown as an example below.



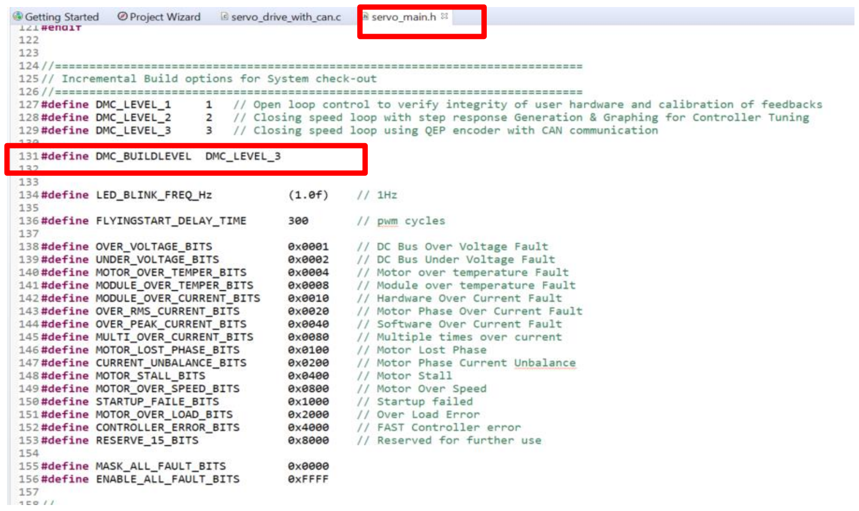
```
servo_drive_with_can_f28004x  [Active - RAM_Lib]
  Binaries
  Includes
  RAM_Lib
  src_board
  src_device
  src_driver
  src_foc
  communication.c
  communication.h
  servo_drive_with_can.c
  servo_main.h
  TMS320F280049C_LaunchPad.ccxml
```

- Make clean and build project successfully as below. If not, check the section [6.0 Debug.]() By default, the build level is set to DMC_LEVEL_1.



- The project can be configured to create code and run in either flash or RAM. You may select either of the two, however, for lab experiments use the RAM configuration most of the time and move to the FLASH configuration for production. Right-click on an individual project name and select "Build Configurations->Set Active->RAM_Lib" configuration.
- The program *servo_drive_with_can* is a combination of the following build levels:
    - **DMC_LEVEL_1**: Open loop Control to verify integrity of user hardware. This lab implements a scalar volts/frequency control to test the integrity of the hardware, namely to test the signal chain integrity, such as hardware current/voltage sensing and implement closed-loop control by introducing an incremental quadrature encoder to measure the motor angle.
    - **DMC_LEVEL_2**: Step response generation & graphing tuning the controller. This lab is to generate step responses for the current and the speed controller. With these step responses it is possible to configure the speed and current PI controller to fit the customer system requirements.
    - **DMC_LEVEL_3**: CAN communication from host motion controller. This lab incorporates a CAN interface to receive a speed reference from another host controller.
    - **DMC_LEVEL_4:** CAN communication to and from host motion controller. This lab is modified to work in an autonomous mobile robot (AMR) system.

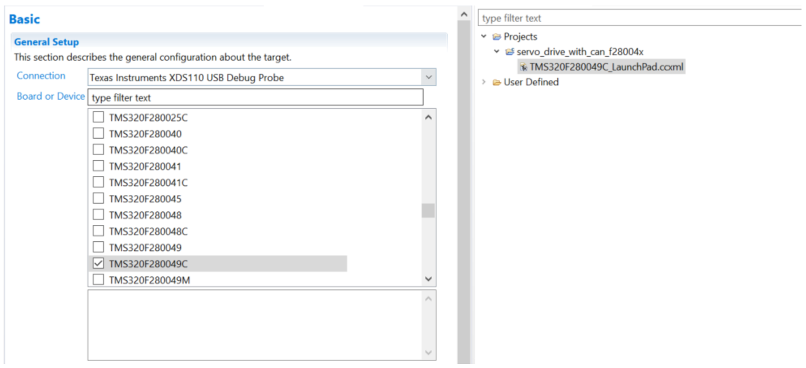
- The program can be debugged individually for each DMC_LEVEL_1/2/3/4. Edit the 'servo_main.h' file accordingly, for example to run all three build levels choose DMC_LEVEL_3.
    - For example, in the below code at line number #131, define the DMC_BUILDLEVEL to DMC_LEVEL_3 accordingly.



```
121#endif
122
123
124//==============================================================================
125// Incremental Build options for System check-out
126//==============================================================================
127#define DMC_LEVEL_1     1   // Open loop control to verify integrity of user hardware and calibration of feedbacks
128#define DMC_LEVEL_2     2   // Closing speed loop with step response Generation & Graphing for Controller Tuning
129#define DMC_LEVEL_3     3   // Closing speed loop using QEP encoder with CAN communication
130
131#define DMC_BUILDLEVEL  DMC_LEVEL_3
132
133
134#define LED_BLINK_FREQ_Hz           (1.0f)    // 1Hz
135
136#define FLYINGSTART_DELAY_TIME      300       // pwm cycles
137
138#define OVER_VOLTAGE_BITS           0x0001    // DC Bus Over Voltage Fault
139#define UNDER_VOLTAGE_BITS          0x0002    // DC Bus Under Voltage Fault
140#define MOTOR_OVER_TEMPER_BITS      0x0004    // Motor over temperature Fault
141#define MODULE_OVER_TEMPER_BITS     0x0008    // Module over temperature Fault
142#define MODULE_OVER_CURRENT_BITS    0x0010    // Hardware Over Current Fault
143#define OVER_RMS_CURRENT_BITS       0x0020    // Motor Phase Over Current Fault
144#define OVER_PEAK_CURRENT_BITS      0x0040    // Software Over Current Fault
145#define MULTI_OVER_CURRENT_BITS     0x0080    // Multiple times over current
146#define MOTOR_LOST_PHASE_BITS       0x0100    // Motor Lost Phase
147#define CURRENT_UNBALANCE_BITS      0x0200    // Motor Phase Current Unbalance
148#define MOTOR_STALL_BITS            0x0400    // Motor Stall
149#define MOTOR_OVER_SPEED_BITS       0x0800    // Motor Over Speed
150#define STARTUP_FAILE_BITS          0x1000    // Startup failed
151#define MOTOR_OVER_LOAD_BITS        0x2000    // Over Load Error
152#define CONTROLLER_ERROR_BITS       0x4000    // FAST Controller error
153#define RESERVE_15_BITS             0x8000    // Reserved for further use
154
155#define MASK_ALL_FAULT_BITS         0x0000
156#define ENABLE_ALL_FAULT_BITS       0xFFFF
157
158//
```

- Set the target configuration as follows. Example for the C2000™ Piccolo™ *F28004x* Series Launchpad is shown below, choose accordingly for *F28002x EVM or F28003x EVM*

```
View -> Target Configurations -> Projects ->
```



Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS110 USB Debug Probe

Board or Device: type filter text

- [ ] TMS320F280025C
- [ ] TMS320F280040
- [ ] TMS320F280040C
- [ ] TMS320F280041
- [ ] TMS320F280041C
- [ ] TMS320F280045
- [ ] TMS320F280048
- [ ] TMS320F280048C
- [ ] TMS320F280049
- [x] TMS320F280049C
- [ ] TMS320F280049M

type filter text

- Projects
  - servo_drive_with_can_f28004x
    - TMS320F280049C_LaunchPad.ccxml
- User Defined

4.0 Hardware Setup

The C2000 LAUNCHXL a low-cost development board for the Texas Instruments Piccolo F28002/4x/3x series of microcontrollers (MCUs). It is designed around the MCU and highlights the control, analog, and communications peripherals, as well as the integrated nonvolatile memory. The LaunchPad also features two independent BoosterPack XL expansion connectors, on-board Controller Area Network (CAN) transceiver, 5 V encoder connectors, FSI connector, and an on-board XDS110 debug probe.

4.1 For testing DMC_LEVEL_1 and DMC_LEVEL_2

- **Encoder interface:** Connect Ground, 5 V, Index (1I), 1A and 1B wires from the motor to the corresponding pins (J12) on the Launchpad (**F280049C/ F280025C/F280039C).**

Table 2 Encoder wire connections for reference kit motor

J12 on LaunchPad	J4 on Teknic M-2310P-LN-04K
J12-1/A	J4-1. A (Blue)
J12-2/B	J4-2. B (Pink)
J12-3/I	J4-3. I (Brown)
J12-4/+5V	J4-4. +5V (Red)
J12-5/GND	J4-5. GND (Black)

- **BoosterPack**: Mount BOOSTXL-DRV8320RS on LaunchPad LAUNCHXL-F280049C headers J1-J4 or BOOSTXL-DRV8323RS on LAUNCHXL-F280025C or LAUNCHXL-F280039C headers J1-J4.
- **Power up**: Disconnect the jumpers JP1, JP2 and JP3 and place a jumper on **JP9** to enable isolation between the Launchpad and the connected USB. JP9 isolates the output of the LMR62421 set-up voltage regulator from the 5V domain of the LaunchPad.
- Connect the LaunchPad to computer through an USB cable, this will light some LEDs on the emulator section of LaunchPad, indicating that the emulator is on. Power the BoosterPack with appropriate input dc voltage, 24V.
- On the BoostXL, connect the Vin and Gnd, this powers the set-up and connect the motor phase wires accordingly (A, B, C)

Table 3 Motor phase connections for reference kit motor

J5 on BoosterPack	Motor Phase of Teknic M-2310P-LN-04K
A/U	T (Black)
B/V	R (Red)
C/W	S (White)

- Overview of the switch states and set-up in the related LaunchPad User's Guide.
- **Idrive/VDS** pins need to be physically disconnected from the BoostXl-DRV8323RS which can be achieved by simply bending them as in Figure 1.

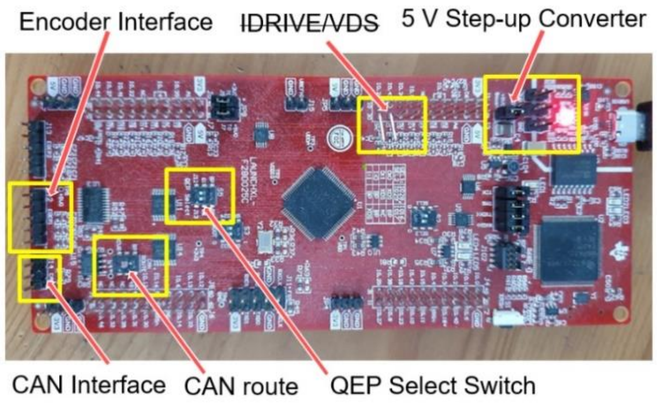


Figure 1 Switches on LAUNCHXL-F280025C

- Take a flying wire from GPIO8 to SPIA_STE/GPIO5 as in the picture below, this connects Chip select of the Launchpad to the Booster Pack

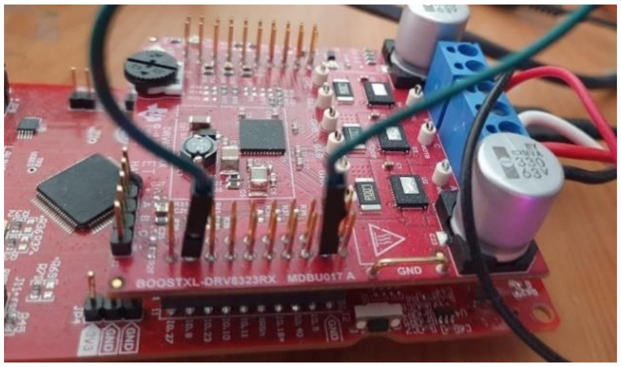
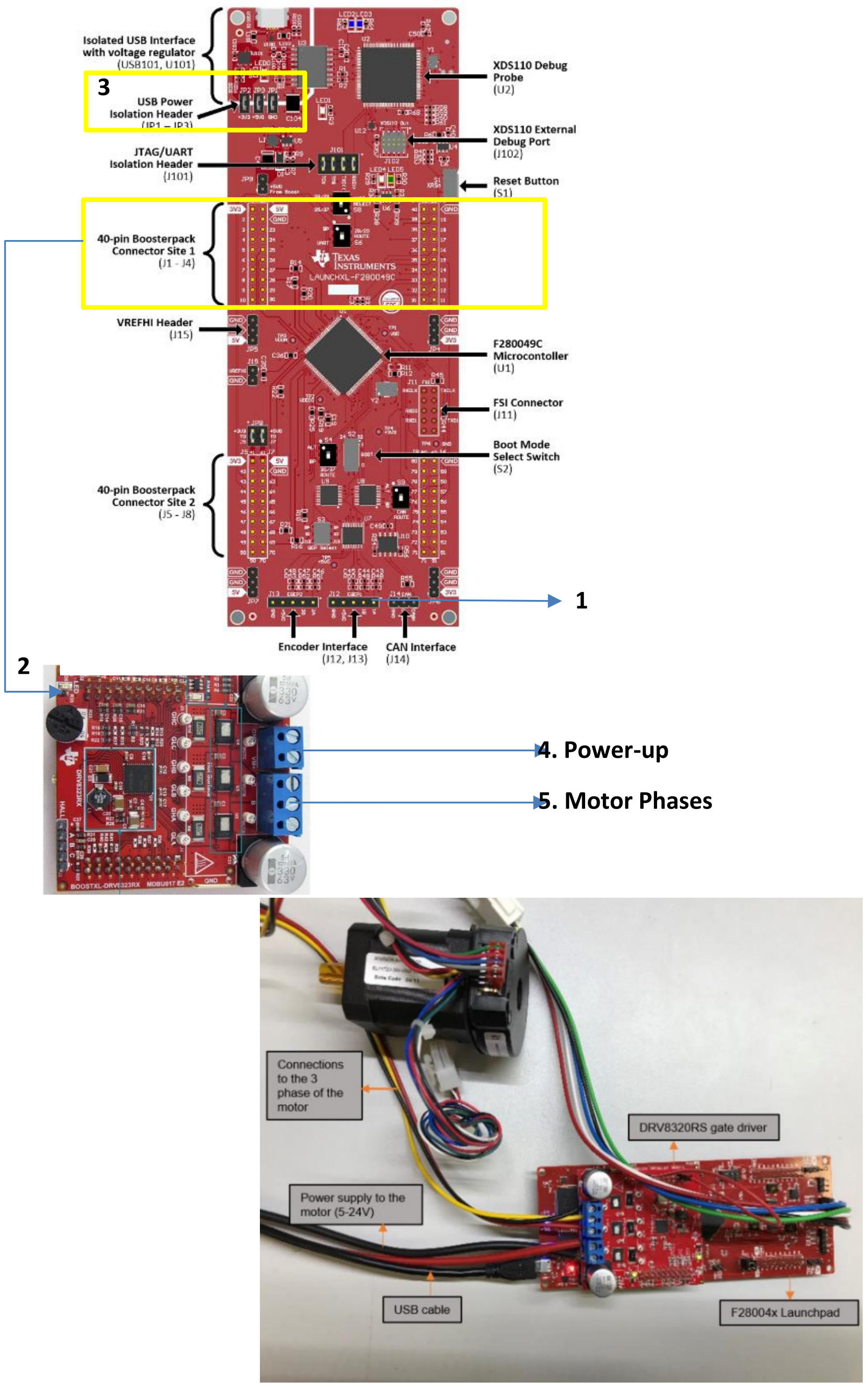


Figure 2 Jumper Wires on BOOSTXL-DRV8323RS for LAUNCHXL-F280025C

** Add a jumper to connect the GPIO22 to GPIO57 for using the SPIA_STE on LUANCHXL-F280039C and BOOSTXL-DRV8323RS. Set S5 switch on LUANCHXL-F280039C, S5-1 on J12 side for QEP, S5-2 on BP side. Set S2 switch on LAUNCHXL-F280039C, S2-SEL1 to UP and S2-SEL2 to DOWN.



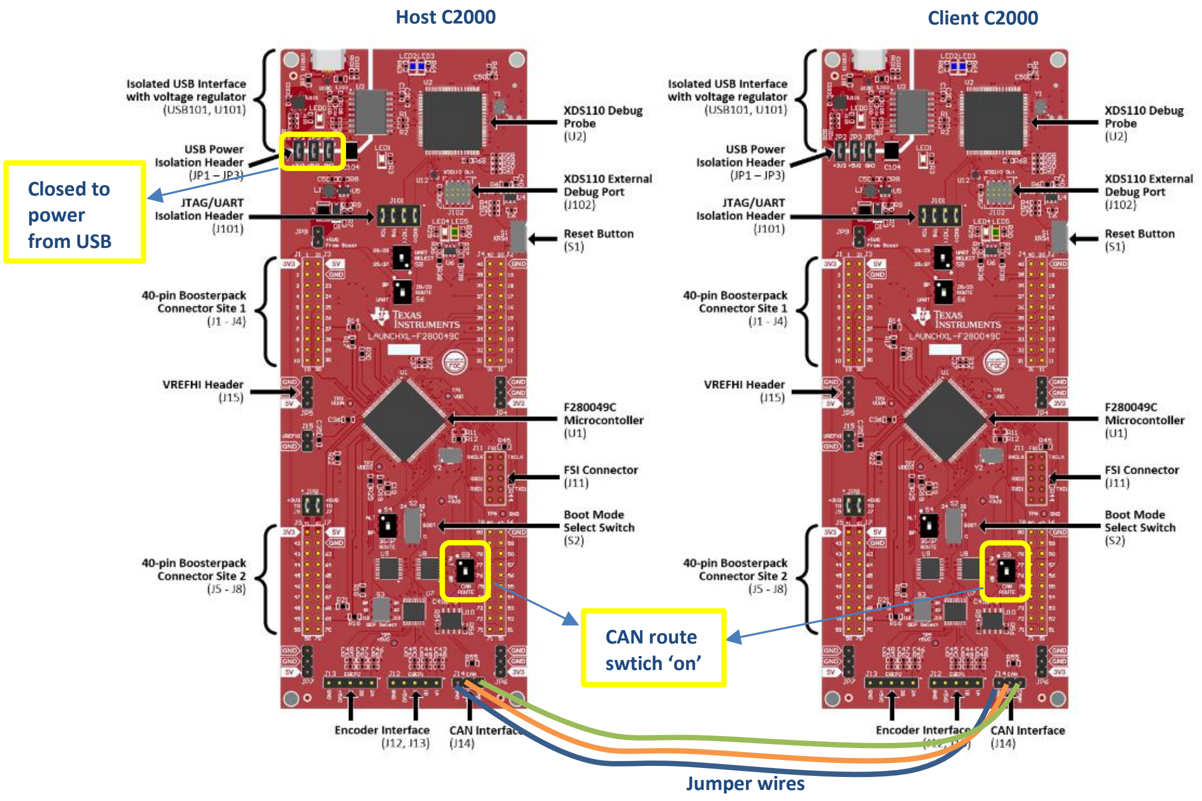
Isolated USB Interface with voltage regulator (USB101, U101)
3
USB Power Isolation Header (JP1 – JP3)
JTAG/UART Isolation Header (J101)
40-pin Boosterpack Connector Site 1 (J1 - J4)
VREFHI Header (J15)
40-pin Boosterpack Connector Site 2 (J5 - J8)
XDS110 Debug Probe (U2)
XDS110 External Debug Port (J102)
Reset Button (S1)
F280049C Microcontoller (U1)
FSI Connector (J11)
Boot Mode Select Switch (S2)
Encoder Interface (J12, J13)
CAN Interface (J14)
1
2
4. Power-up
5. Motor Phases
Connections to the 3 phase of the motor
DRV8320RS gate driver
Power supply to the motor (5-24V)
USB cable
F28004x Launchpad

** Add a jumper to connect GPIO23 to GPIO57 for using the SPIA_STE on LUANCHXL-F280049C and BOOSTXL-DRV8323RS.

Figure 3 Overview of the set-up with F280049C

4.2 For testing DMC_LEVEL_3

- Connect the CANH, GND and CANL pins on the host C2000 board (that sends the speed references to the client C2000 board which runs the motor control.
- Make sure that the CAN route switch (S9) is set 0 in both the boards.
- The host C2000 board is powered from the USB in our case, hence JP1-JP3 are closed where as the client C2000 is powered from the Boosterpack, hence follow the instructions here.


Host C2000
Client C2000
Closed to power from USB
Isolated USB Interface with voltage regulator (USB101, U101)
USB Power Isolation Header (JP1 – JP3)
JTAG/UART Isolation Header (J101)
40-pin Boosterpack Connector Site 1 (J1 - J4)
VREFHI Header (J15)
40-pin Boosterpack Connector Site 2 (J5 - J8)
XDS110 Debug Probe (U2)
XDS110 External Debug Port (J102)
Reset Button (S1)
F280049C Microcontoller (U1)
FSI Connector (J11)
Boot Mode Select Switch (S2)
Encoder Interface (J12, J13)
CAN Interface (J14)
CAN route swtich 'on'
Jumper wires
Figure 4 CAN interface hardware set-up

4.3 For testing DMC_LEVEL_4

- The basic configuration is the same as DMC_LEVEL_3.
- To only test the CAN interface, the same set-up as DMC_LEVEL_3 can be further used, where one of the LAUNCHXL-F280039C is regarded as host.
- For the test on the mobile robot, several changes need to be achieved:
    - **LAUNCHXL-F280039C:** This is be powered by a BOOSTXL-DRV8323RS and a TPS61379Q1EVM-082. To enable this, all jumpers of **JP1** and **JP8** on F280039C need to be removed firstly.
    - **TPS61379Q1EVM-082:** This board (boost converter) is used to convert 3.3V to 5V for powering encoders of motors. To set the Vout of the boost converter, move the zero-ohm resistor **R15** to **R14**; then add a jumper at **J5** to connect **R11** for 5V only; switch the jumper at **J4** from **EN-OFF** to **ON-EN**.

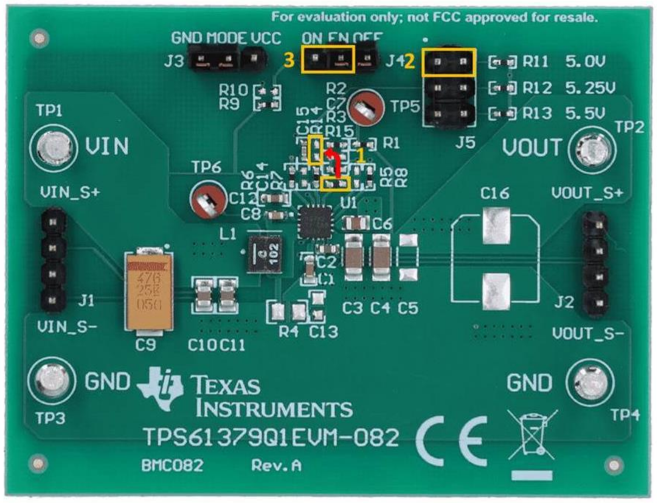


Figure 5 Boost converter set-up for DMC_LEVEL_4

After the configuration of the booster converter, it needs be mounted on a PCB adapter with pin connectors, which can be plugged in the connector J5 + J7, J8 + J6. While the long connector needs to plugged in JP8, J5 and J7, the short connector in J6 and J8. A single connector with wire from the adapter (IO.57) should be connected with a pin on BOOSTXL-DRV8323RS (booster pack), which also corresponds to IO.5 on LAUNCHXL-F280039C. In this way, the SPIA_STE is connected between LAUNCHXL-F280039C and BOOSTXL-DRV8323RS.

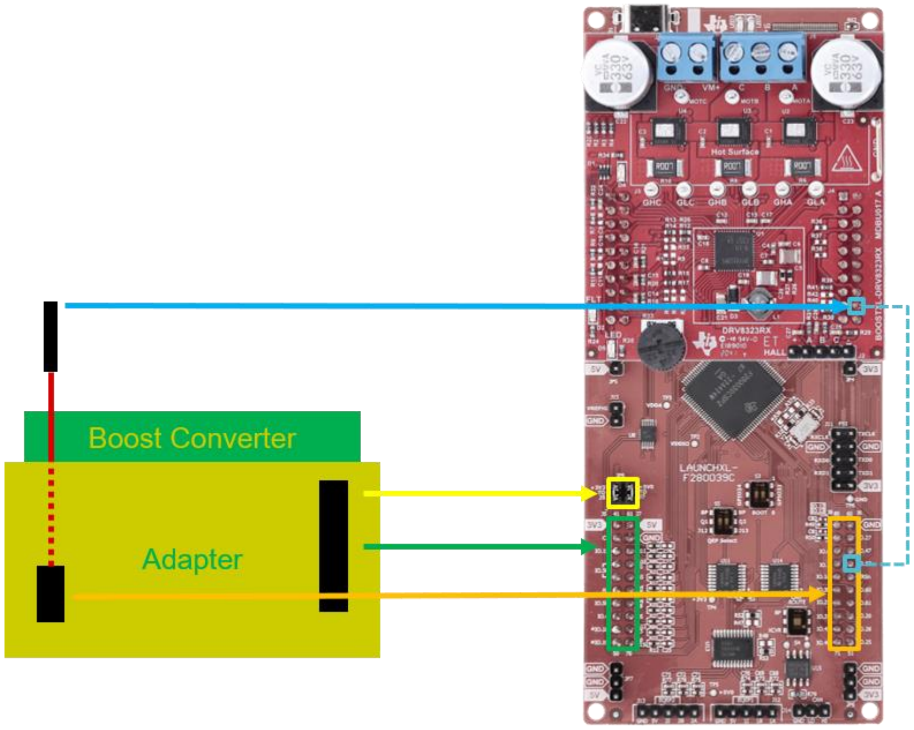


Figure 6 Boost converter adapter connections

- **System power up:** By connecting the power supply at least 12V, 2A to VM on BOOSTXL-DRV8323RS, the whole motor control set-up including BOOSTXL-DRV8323RS, LAUNCHXL-F280039C, TPS61379Q1EVM-082 as well as the motor (LVSERVOMTR) can be powered up. The experiments in this lab are tested by 24V, 2A and 12V, 2A.
- The connections of motors and encoders maintain the same as the former levels.
- **CAN communication:** The same connections can be used from DMC_LEVEL_3 to validate one motor control through another LAUNCHXL-F280039C. To assemble the robot system, the CAN interfaces of two LAUNCHXL-F280039C boards should be connected with any two of the four CAN connections (MCU_CAN0, CAN9, CAN0 or CAN5) on SK-TDA4VM board, the position of the four ports is shown as follows.

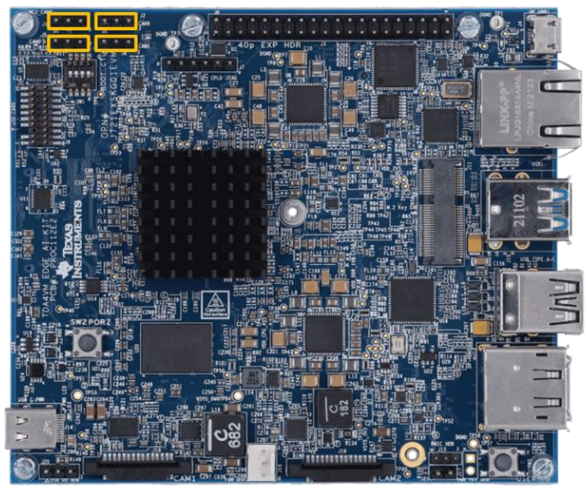


Figure 7 SK-TDA4VM CAN connections

5.0 System Software Integration and Testing

This section deals with various incremental build levels starting from first level up to the final level where the system is fully integrated.

5.1 DMC_LEVEL_1: Open loop Control to verify integrity of user hardware

Assuming the load and build steps described in the Section **Software Setup** is completed successfully. This section implements a scalar volts/frequency control to test the integrity of the hardware, namely to test the signal chain integrity, such as hardware current/voltage sensing and implement closed-loop control by introducing an incremental quadrature encoder to measure the motor angle.

This build level introduces an incremental quadrature encoder to measure shaft position. However, to begin with, it utilizes an angle generator module to generate the angle based on the motor target frequency. This uses a Volt/Hertz profile to generate an output command voltage to drive the motor. During initial rotation of the motor the QEP is calibrated. When an index pulse is detected, the angle determined by the QEP module is fed into the FOC, hence closed loop operation. MotorControl SDK API function calls will be used to simplify the microprocessor setup.

The setup of peripherals and even the inverter will be taken care of by MotorControl SDK APIs, specifically the HAL object and APIs. Important commands to initialize the drive are listed in Figure 8.

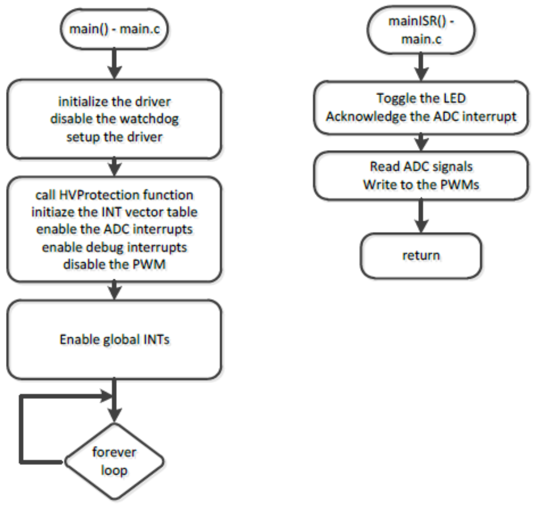
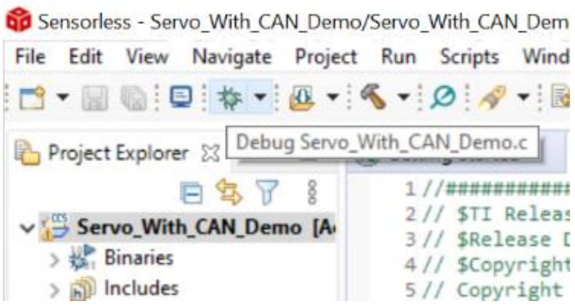


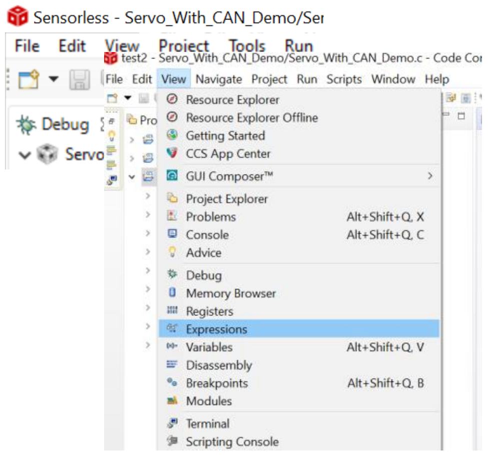
Figure 8 Software Flow Chart in Build Level 1

- Connect the LaunchPad/BoostXL USB cable and switch on the power. Open project within CCS and set DMC_BUILDLEVEL to DMC_LEVEL_1 in "servo_main.h", build and load project as described in the Software setup, to debug, click the green bug that symbolizes “Debug”. This button should automatically change the CCS perspective to “CCS Debug” as well as load the .out file to the target. (Learn more about ccs debugging)


Sensorless - Servo_With_CAN_Demo/Servo_With_CAN_Dem
File Edit View Navigate Project Run Scripts Wind
Debug Servo_With_CAN_Demo.c
Project Explorer
Servo_With_CAN_Demo [A
Binaries
Includes

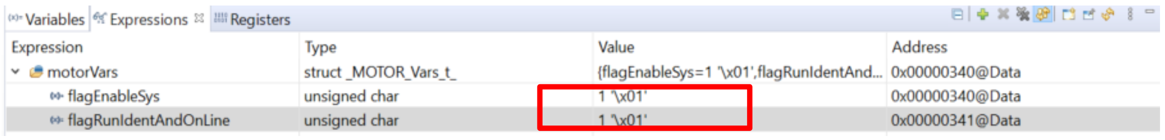
```
//############
// $TI Releas
// $Release I
// $Copyright
// Copyright
```

- Click “Resume” which looks like a yellow vertical line with a green triangle beside it.



- To run the motor, set the following variables in the expression window as follows

```
View -> Expressions
'Add motorVars'
'Add gGraphVars'
'Add gStepVars'
```



Variables | Expressions | Registers

Expression	Type	Value	Address
motorVars	struct _MOTOR_Vars_t_	{flagEnableSys=1 '\x01',flagRunIdentAnd...	0x00000340@Data
flagEnableSys	unsigned char	1 '\x01'	0x00000340@Data
flagRunIdentAndOnLine	unsigned char	1 '\x01'	0x00000341@Data

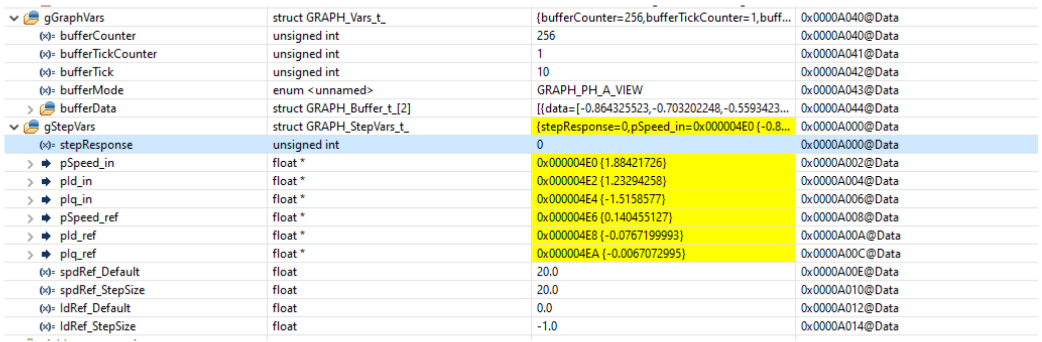
```
"flagEnableSys" = 1
"flagRunIdentAndOnLine" = 1
```

- The motor should be spinning after setting the above variables by clicking on resume button.
- If the motor does not spin in open loop control, please refer to the Debug section below. (Reset Fault condition).
- Import the predefined graph settings (Learn more about ccs graph) . The tools option is available in the debug window. This passes the electrical angle of the angleGen function and that of the QEP module to the graph function

```
Tools-> Graph-> Dual Time-> Import

C:\ti\c2000\C2000Ware_MotorControl_SDK_3_01_00_00\solutions\servo_drive_with_can\common\debug\servo_drive_with_can_view.graphProp
```

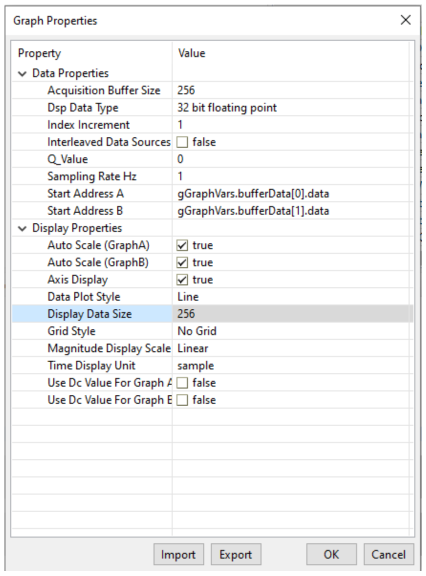
- Change bufferMode to Graph_PH_A_VIEW by setting gGraphVars.bufferMode, and set gStepVars.stepResponse to "1" in Expression Window as shown in Figure 9, this will graph the electrical angle of the angleGen module and that of the QEP module as shown in Figure 10.

```
Expressions-> gGraphVars>bufferMode -> Graph_PH_A_View
Expressions-> gStepVars>stepResponse->1
```



gGraphVars	struct GRAPH_Vars_t_	{bufferCounter=256,bufferTickCounter=1,buff...	0x0000A040@Data
bufferCounter	unsigned int	256	0x0000A040@Data
bufferTickCounter	unsigned int	1	0x0000A041@Data
bufferTick	unsigned int	10	0x0000A042@Data
bufferMode	enum <unnamed>	GRAPH_PH_A_VIEW	0x0000A043@Data
bufferData	struct GRAPH_Buffer_t_[2]	[{data=[-0.864325523,-0.703202248,-0.5593423...	0x0000A044@Data
gStepVars	struct GRAPH_StepVars_t_	{stepResponse=0,pSpeed_in=0x000004E0 {-0.8...	0x0000A000@Data
stepResponse	unsigned int	0	0x0000A000@Data
pSpeed_in	float *	0x000004E0 {1.88421726}	0x0000A002@Data
pId_in	float *	0x000004E2 {1.23294258}	0x0000A004@Data
pIq_in	float *	0x000004E4 {-1.5158577}	0x0000A006@Data
pSpeed_ref	float *	0x000004E6 {0.140455127}	0x0000A008@Data
pId_ref	float *	0x000004E8 {-0.0767199993}	0x0000A00A@Data
pIq_ref	float *	0x000004EA {-0.0067072995}	0x0000A00C@Data
spdRef_Default	float	20.0	0x0000A00E@Data
spdRef_StepSize	float	20.0	0x0000A010@Data
IdRef_Default	float	0.0	0x0000A012@Data
IdRef_StepSize	float	-1.0	0x0000A014@Data

Figure 9 Setup Variables for Graphing Angle



Graph Properties

Property	Value
Data Properties	
Acquisition Buffer Size	256
Dsp Data Type	32 bit floating point
Index Increment	1
Interleaved Data Sources	☐ false
Q_Value	0
Sampling Rate Hz	1
Start Address A	gGraphVars.bufferData[0].data
Start Address B	gGraphVars.bufferData[1].data
Display Properties	
Auto Scale (GraphA)	✓ true
Auto Scale (GraphB)	✓ true
Axis Display	✓ true
Data Plot Style	Line
Display Data Size	256
Grid Style	No Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
Use Dc Value For Graph A	☐ false
Use Dc Value For Graph B	☐ false

Figure 10 Graph Properties Settings for Build Leve 1

- As seen in the graph, the electrical angle of the motor (from QEP module) and the generated angle (Angle_gen) are aligned as shown in Figure 11.

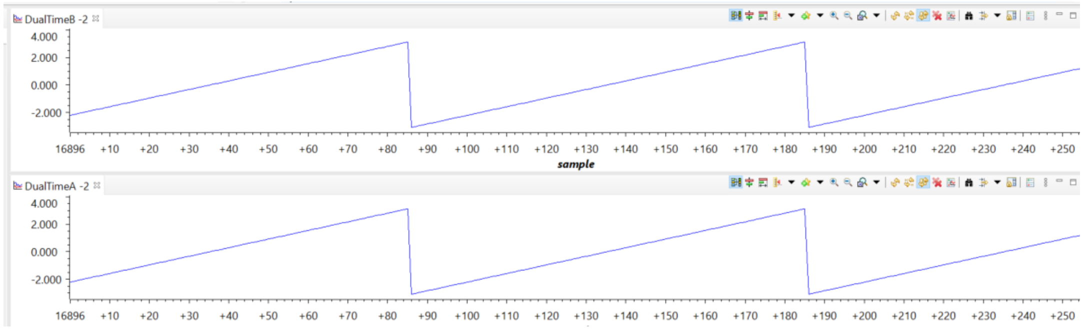


Figure 11 Current Angles from Angle Generator and QEP module

- Change bufferMode to Graph_PH_B_VIEW by setting gGraphVars.bufferMode, and set gStepVars.stepResponse to “1” in Expression Window, this will graph the using angle for motor drive and phase A sampling current as shown in Figure 12.

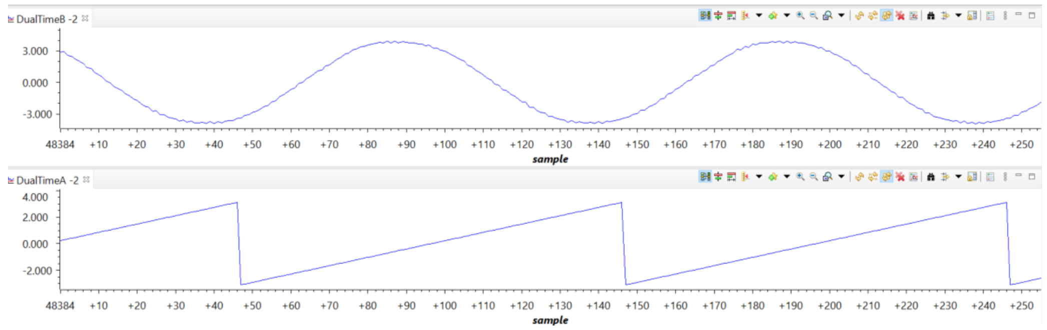


Figure 12 Using Angle and Phase A Current

- Change bufferMode to Graph_PH_C_VIEW by setting gGraphVars.bufferMode, and set gStepVars.stepResponse to “1” in Expression Window, this will graph phase B and C sampling current as shown in Figure 13.

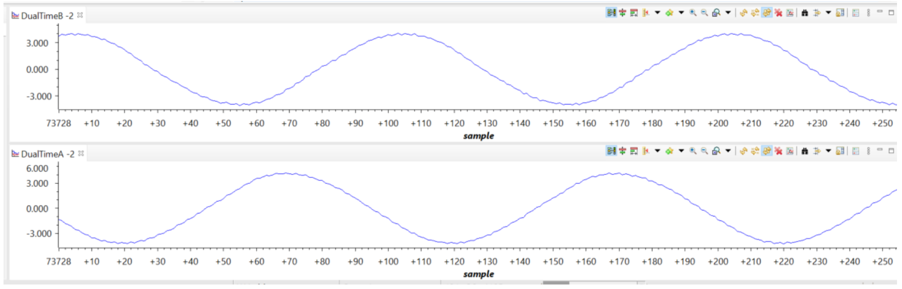


Figure 13 Phase B and C Current

- To stop the motor by setting the motorVars.flagRunIdentAndOnLine to “0” in Expression Window.

5.2 DMC_LEVEL_2: Closing speed loop with step response Generation & Graphing for Controller Tuning

In this build level, learn how to create a real-time step response for both PI current and PI speed controllers and display the step response in CCS graph tool. Use the tool to adjust the corresponding PI-controller parameters in MotorWare using the C2000 Real Time Debug capability with Code Composer Studio. This can be used to adjust the motor dynamic performance according to your system specification.

Step response generation will be performed in real-time with the motor running in closed loop using the Real Time Debug features. This project supports step response generation for either Id current or speed.

Build level 2 demonstrates a technique of tuning the proportional gain (Kp) and integral gain (Ki) for the current and speed controllers, this lab utilizes the graphing function of CCS to do step response testing. The default PI controller is just a starting point and needs tuning to meet the dynamic performance of the customer requirements.

Testing and adjusting the controller gains allows you to optimize the system for your specific requirements. Using a step response, it is possible to test the effects of the controller gains in your application and choose the values which meet your system response, stability, and efficiency targets. An overview of the used variables is shown in Figure 14.

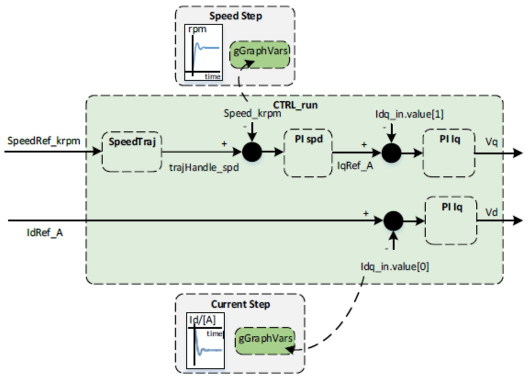


Figure 14 Cascaded speed and current controller of the InstaSPIN-FOC

The purpose is to generate a step response in the Id current controller to be almost independent of load torque changes. This removes the need for a load emulator like a dynamometer when trying to generate the step response in Iq.

- Set DMC_BUILDLEVEL to DMC_LEVEL_2 in "servo_main.h", build and reload project as described in the Section 2.0
- To run the motor by setting motorVars.flagEnableSys and motorVars.flagRunIdentAndOnLine to "1" in Expression Window. The motor will start running at the speed defined in the variable "motorVars.speedRef_Hz", and the actual motor speed can be seen in the variable "motorVars.speed_Hz".

```
        Flag_enableSys= 1
        flagRunIdentAndOnLine = 1
```

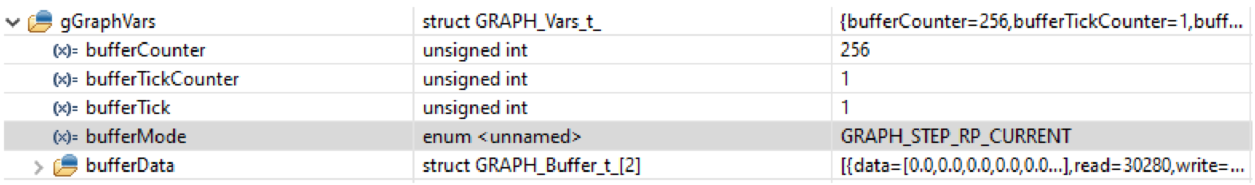
  As usual, the motor will start running at the speed defined in the variable:

```
        SpeedRef_krpm
```

  The actual motor speed can be seen in the variable:

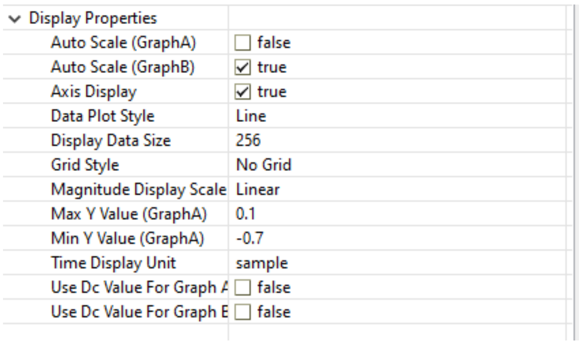
```
        Speed_krpm
```

- The variable "*gGraphVars.BufferMode*" selects between current step response generation or speed step response generation.
  a. Current step response generation is done setting the variable to *GRAPH_STEP_RP_CURRENT* (default)
  b. Speed step response generation is done setting the variable to *GRAPH_STEP_RP_SPEED*



gGraphVars	struct GRAPH_Vars_t_	{bufferCounter=256,bufferTickCounter=1,buff...
(x)= bufferCounter	unsigned int	256
(x)= bufferTickCounter	unsigned int	1
(x)= bufferTick	unsigned int	1
(x)= bufferMode	enum <unnamed>	GRAPH_STEP_RP_CURRENT
bufferData	struct GRAPH_Buffer_t_[2]	[{data=[0.0,0.0,0.0,0.0,0.0...],read=30280,write=...

- The variables are used in the following way.
  "bufferCounter" used as an index to write into the buffer away structure
  "bufferTickCounter" used to count the interrupts.
  "buffer Tick" defines how many interrupts happen per graph write
  "bufferMode" can be used to define different values to record for each mode during run time
  "bufferData" is the array where the data is stored
- Load "servo_drive_with_can_current_step.graphProp" as in previous steps to make sure that the graph tool Y-scale to view the 0.5 A step response.



Display Properties	
Auto Scale (GraphA)	☐ false
Auto Scale (GraphB)	✓ true
Axis Display	✓ true
Data Plot Style	Line
Display Data Size	256
Grid Style	No Grid
Magnitude Display Scale	Linear
Max Y Value (GraphA)	0.1
Min Y Value (GraphA)	-0.7
Time Display Unit	sample
Use Dc Value For Graph A	☐ false
Use Dc Value For Graph B	☐ false

- When using the graph there are several ways of updating the window. These must be selected for each Graph A and B.

Press this button to refresh the current graph window with the current values written into the data array. This button needs to be pressed every time you have received new data, to show the new graph.

Press this button to enable automatic graph update, every time the data array receives new data.

Reset the current graph window to auto fit the new scale of the data array.

5.2.1 Step response generation for PI current controller

For Sensored-FOC in this lab, the speed and current PI controller are cascaded, so to start generating the step response the user has to begin with the current controller.

```
      gGraphVars.BufferMode = GRAPH_STEP_RP_CURRENT (default)
```

For your real application it will be important to test the step response generation while running with loads similar to those in the real system, but you can start testing the system with a lightly loaded or without load system. For this example, the motor is running without load and the speed is set to relatively low around 10% of rated value to ensure the motor is running stable. The graphs measured are done on the BOOSTXL-DRV8323RS.

```
      motorVars.speedRef_Hz = 40.0 (Hz)
```

To generate the current step the Id current is used. Because both Id and Iq controllers will use the same gains, we can keep test the effects of the step response on Id more simply without having to disconnect the output of the speed controller from Idq_ref_A.value[1]. The Id current is set using the variable Idq_ref_A.value[0], which is the reference current that is put on the direct axis of the magnetic field. Normally this is set to 0.0 A to orient the stator flux to the rotor. A negative "Idq_ref_A.value[0]" results in field weakening control.

The step chosen with the Teknic M2310PLN04K motor was be -0.5A, which was around 10% of the maximum rated Id current for that motor. For your specific motor you may choose, for example, a step of 10% of the rated Id current as an initial value.

Note that when the step response is executed and data has been logged the value of d-axis current, returns back to the default value to "gStepVars.IdRef_Value[0]". For the current step response the following setting needs to be defined, default values are as below.

```
gStepVars.IdRef_Default = 0.0 // Default starting Id reference value
gStepVars.IdRef_StepSize = -0.5 // Step size of Id reference value
gGraphVars. bufferTick = 1 //
```

Now set the following variable to:

```
gStepVars.StepResponse = 1
```

To tune your controller, watch the impact on the step response of the current controller changing the following variables in Expression Window.

```
motorVars.Kp_Id
motorVars.Ki_Id
```

Now you change either Kp or Ki one at a time. When one of the values has been changed, generate a new step response by setting the variable "gStepVars.StepResponse" to 1 to see the effect as shown in Figure 15. Every time a new step response has been generated refresh the graph window to see the new step response (or turn on continuous refresh). A

As an example, increase the value of Kp to get to the target IdRef_A value with a quick (stiff) response. Increasing it too much results in overshoot. Reducing it too much results in a very slow (soft) response. The Ki value is calculated based on the ratio of Rs to Ls, and the controller period, so it will be as accurate as those values were identified. In most cases the Ki value does not need to be changed, but you can try out values (half the default, twice the default) to see the effect on the settling time or ringing of the step response.

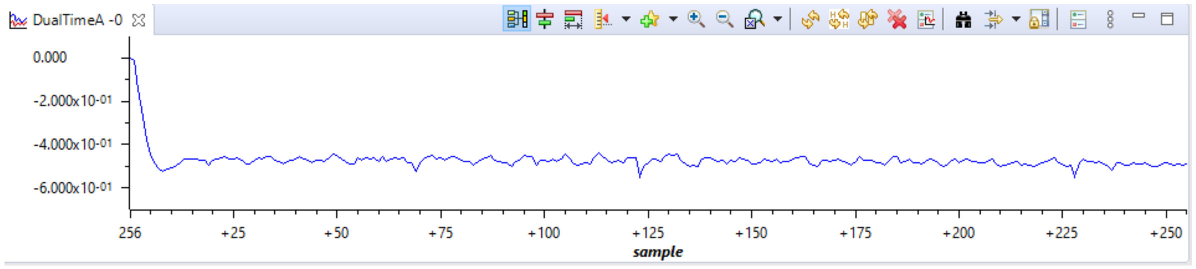


Figure 15 Current step response showing no over shoot at high bandwidth

Note: For more information on the theory of control loop considerations, it is highly recommended to either see the InstaSPIN user guide (spruhj1) or read the book "Control Theory Fundamentals" ISBN-13 978-1496040732.

5.2.2 Step response generation of PI speed controller

- Load "servo_drive_with_can_speed_step.graphProp" as in previous steps to make sure that the graph tool Y-scale to view the 60Hz step response.
- To change the configuration of measuring the PI controller from current step response to a speed step response change the variable:

```
      gGraphVars.BufferMode = GRAPH_STEP_RP_SPEED
      gGraphVars. bufferTick = 20 // Change accordingly to the graph
view
```

- To create a step speed response set:

```
      motorVars.accelerationMax_Hzps = 100
```

- For the speed step response, the following setting needs to be defined, default values are as below.

```
      gStepVars.spdRef_Default = 40.0 // Default starting value of
reference speed (Hz)
      gStepVars.spdRef_Stepsize = 60.0 // Step size of reference
speed (Hz)
```

- Depending on your motor max speed, define a step from around 1/8 to 2/8 of the max speed of the motor with the given Vbus voltage. When the step response is executed and the date has been logged the value of the "motorVars.speed_Hz" returns back to the default value to "motorVars.speedRef_Hz
- As done with the current step response generation, set the following variable to:

```
      gStepVars.StepResponse = 1
```

- Now refresh the graph window, to see the speed step response as shown in Figure 16.
- To change the step response of the speed controller the following variables from the motorVars are used.

```
      motorVars.Kp_spd
      motorVars.Ki_spd
```

- It can be seen as expected at with the chosen PI configuration and ramp, no overshoot of the speed is happening when changing speed.

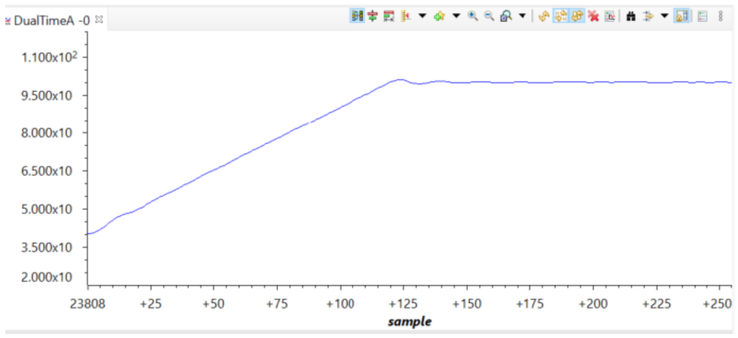


Figure 16 Speed step response showing no overshoot at high bandwidth

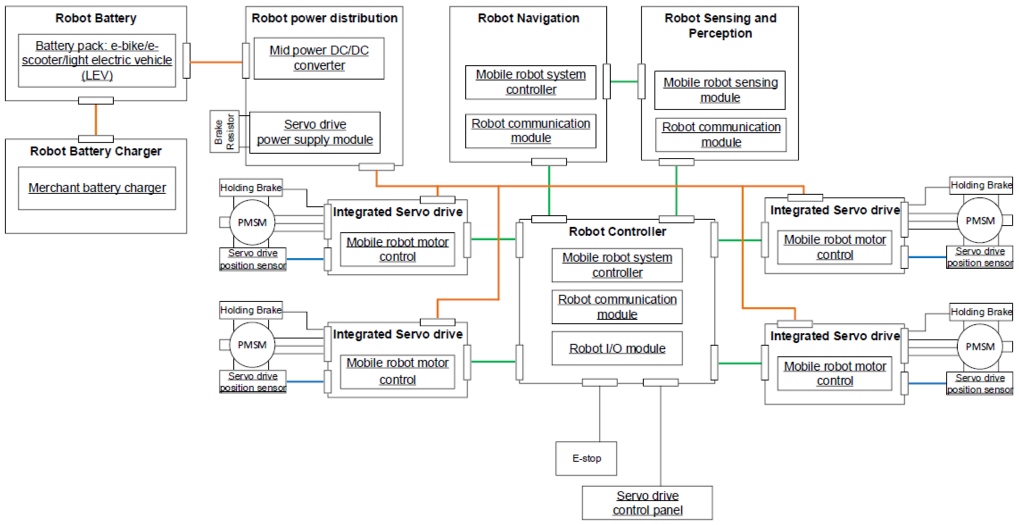
- Using this project and its options it is possible to generate step responses for the current and the speed controller. With these step responses it is possible to configure the PI speed and current controller to fit the customer system requirements. This adjusting of the PI controller can improve the overall system performance, efficiency and reliability, ensuring that the system is not regulating the current and speed too fast or too slow.

5.3 DMC_LEVEL_3 : Closing speed loop using QEP encoder with CAN communication

This lab incorporates a CAN interface to receive a speed reference from a host motion controller. In this lab, communicate with the LAUNCHXL-F280025C using the onboard CAN transceiver. To achieve this another LaunchPad (F280049C, F280025C or F280039C) is used to send a speed reference and start/stop command to the motor controller. The DCAN peripheral is a CAN implementation on a subset of devices within the C2000™ family. Some devices that have a DCAN peripheral include the TMS320F2837xD, TMS320F2837xS, TMS320F2807x, TMS320F28004x, TMS320F28002x and TMS320F28003x devices. The examples are meant to be run in any C2000 MCU with a DCAN module. For a complete list of devices that contain the DCAN module, see the C2000 Real-Time Control Peripherals Reference Guide. This application report describes several programming examples to illustrate how the DCAN module is set up for different modes of operation as below.

- Use the tuned control parameters from the previous lab. The code examples were tested on a TMS320F280049/25/39 device. However, the examples can be easily adapted to run on any C2000 device that features the DCAN module.
- Receive speed reference updates over the CAN interface.

CAN is a serial protocol that was originally developed for automotive applications. Due to its robustness and reliability, it now finds applications in diverse areas such as Industrial equipment, appliances, medical electronics, trains, air crafts, and so forth. CAN protocol features sophisticated error detection (and isolation) mechanisms and lends itself to simple wiring at the physical level

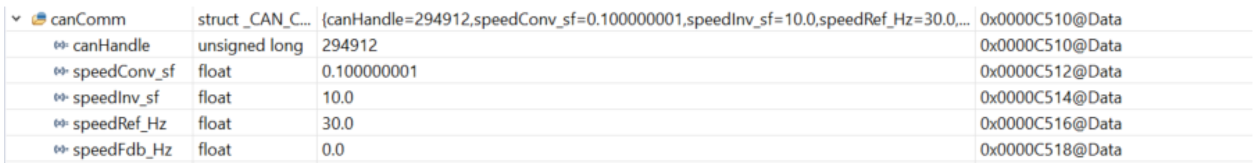


Robot Battery
Battery pack: e-bike/e-scooter/light electric vehicle (LEV)
Robot Battery Charger
Merchant battery charger
Robot power distribution
Mid power DC/DC converter
Brake Resistor
Servo drive power supply module
Robot Navigation
Mobile robot system controller
Robot communication module
Robot Sensing and Perception
Mobile robot sensing module
Robot communication module
Holding Brake
PMSM
Servo drive position sensor
Integrated Servo drive
Mobile robot motor control
Robot Controller
Mobile robot system controller
Robot communication module
Robot I/O module
E-stop
Servo drive control panel

Figure 17 Block diagram of typical 3-phase motor drive system

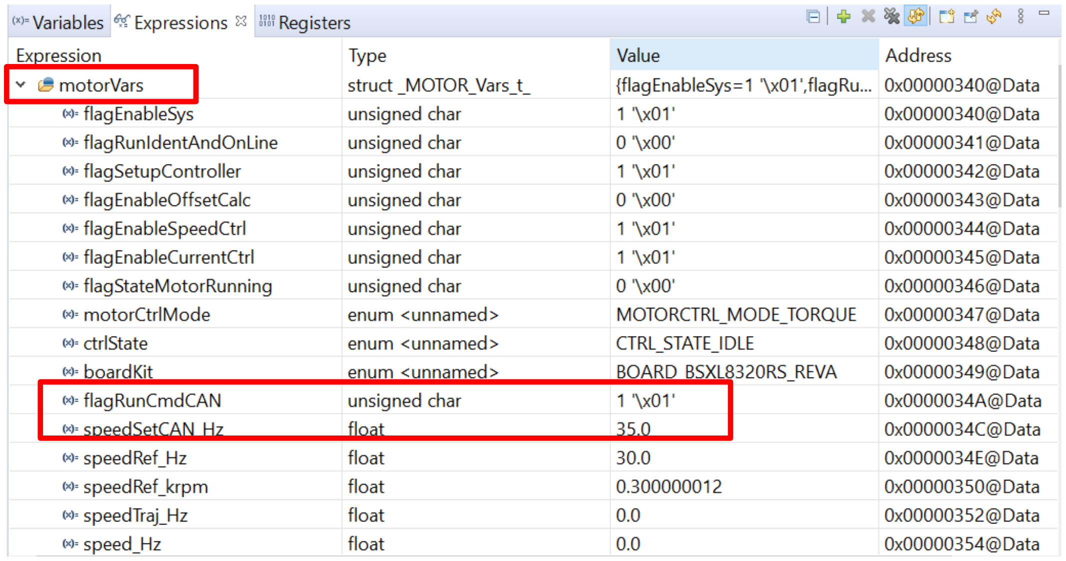
Figure 17 shows a typical setup of a mobile robot motor control system, in this lab the CAN interface is used to define a new speed reference from a host controller. This feature can used to update any other parameter.

- Follow the hardware set-up as explained in For testing DMC_LEVEL_3
- Set DMC_BUILDLEVEL to DMC_LEVEL_3 in "servo_main.h", build and reload project for the host and client controllers as described in the Section 2. The project on host and client controllers **uses the same project, preferably in different ccs projects loaded one after the other.**
- To view the reference speed by adding “canComm”, “motorVars.flagRunCmdCAN”, and “motorVars.speedSetCAN_Hz” to the Expressions Window.



canComm	struct _CAN_C...	{canHandle=294912,speedConv_sf=0.100000001,speedInv_sf=10.0,speedRef_Hz=30.0,...	0x0000C510@Data
canHandle	unsigned long	294912	0x0000C510@Data
speedConv_sf	float	0.100000001	0x0000C512@Data
speedInv_sf	float	10.0	0x0000C514@Data
speedRef_Hz	float	30.0	0x0000C516@Data
speedFdb_Hz	float	0.0	0x0000C518@Data

- To start the motor on another LaunchPad and BoosterPack by setting the following variables in host controller side. Set the speed accordingly, here for reference it is set to 35Hz as shown in Figure 18.



Variables | Expressions | Registers

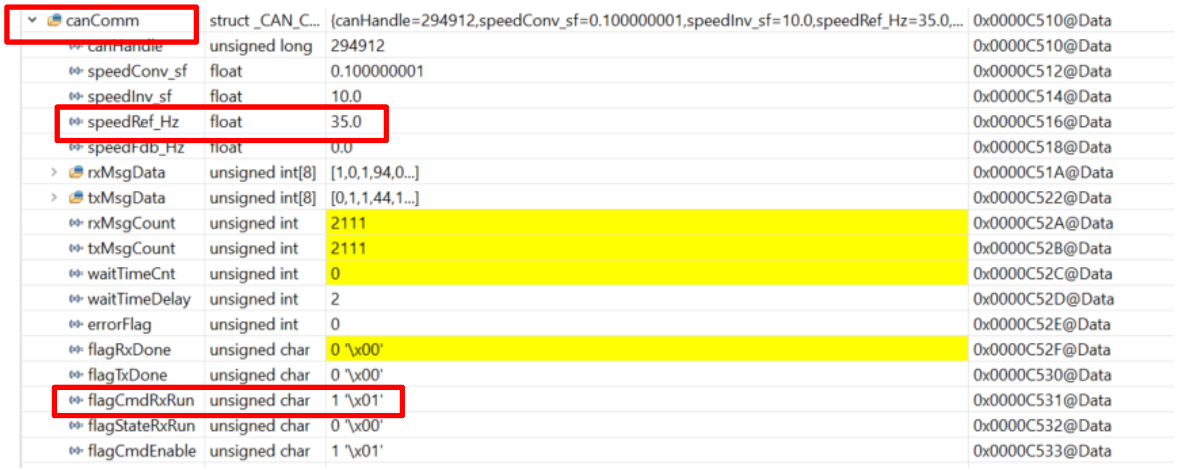
Expression	Type	Value	Address
motorVars	struct _MOTOR_Vars_t_	{flagEnableSys=1 '\x01',flagRu...	0x00000340@Data
flagEnableSys	unsigned char	1 '\x01'	0x00000340@Data
flagRunIdentAndOnLine	unsigned char	0 '\x00'	0x00000341@Data
flagSetupController	unsigned char	1 '\x01'	0x00000342@Data
flagEnableOffsetCalc	unsigned char	0 '\x00'	0x00000343@Data
flagEnableSpeedCtrl	unsigned char	1 '\x01'	0x00000344@Data
flagEnableCurrentCtrl	unsigned char	1 '\x01'	0x00000345@Data
flagStateMotorRunning	unsigned char	0 '\x00'	0x00000346@Data
motorCtrlMode	enum <unnamed>	MOTORCTRL_MODE_TORQUE	0x00000347@Data
ctrlState	enum <unnamed>	CTRL_STATE_IDLE	0x00000348@Data
boardKit	enum <unnamed>	BOARD_BSXL8320RS_REVA	0x00000349@Data
flagRunCmdCAN	unsigned char	1 '\x01'	0x0000034A@Data
speedSetCAN_Hz	float	35.0	0x0000034C@Data
speedRef_Hz	float	30.0	0x0000034E@Data
speedRef_krpm	float	0.300000012	0x00000350@Data
speedTraj_Hz	float	0.0	0x00000352@Data
speed_Hz	float	0.0	0x00000354@Data

Figure 18 Set speed in host controller

```
motorVars.flagRunCmdCAN = 1 // 0-Stop the motor, 1-Start the motor
motorVars.speedSetCAN_Hz = 35.0 // Reference speed
```

- The client controller will receive the command and reference speed in the following variables to start run the motor as shown in Figure 19.

```
canComm.flagCmdRxRun is equal to motorVars.flagRunCmdCAN
canComm.speedRef_Hz is equal to motorVars.speedSetCAN_Hz
```

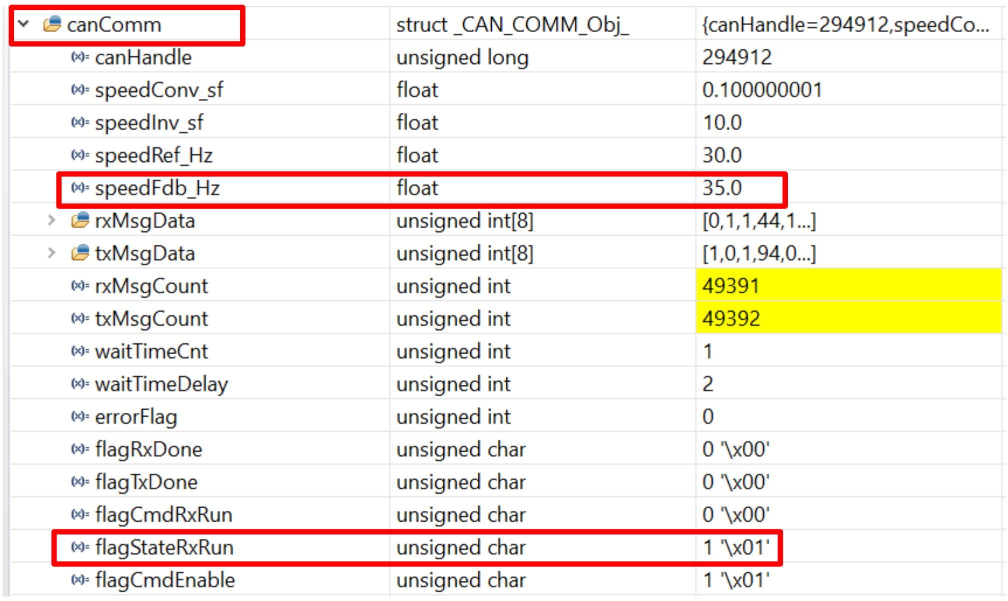


canComm	struct _CAN_C...	{canHandle=294912,speedConv_sf=0.100000001,speedInv_sf=10.0,speedRef_Hz=35.0,...	0x0000C510@Data
canHandle	unsigned long	294912	0x0000C510@Data
speedConv_sf	float	0.100000001	0x0000C512@Data
speedInv_sf	float	10.0	0x0000C514@Data
speedRef_Hz	float	35.0	0x0000C516@Data
speedFdb_Hz	float	0.0	0x0000C518@Data
rxMsgData	unsigned int[8]	[1,0,1,94,0...]	0x0000C51A@Data
txMsgData	unsigned int[8]	[0,1,1,44,1...]	0x0000C522@Data
rxMsgCount	unsigned int	2111	0x0000C52A@Data
txMsgCount	unsigned int	2111	0x0000C52B@Data
waitTimeCnt	unsigned int	0	0x0000C52C@Data
waitTimeDelay	unsigned int	2	0x0000C52D@Data
errorFlag	unsigned int	0	0x0000C52E@Data
flagRxDone	unsigned char	0 '\x00'	0x0000C52F@Data
flagTxDone	unsigned char	0 '\x00'	0x0000C530@Data
flagCmdRxRun	unsigned char	1 '\x01'	0x0000C531@Data
flagStateRxRun	unsigned char	0 '\x00'	0x0000C532@Data
flagCmdEnable	unsigned char	1 '\x01'	0x0000C533@Data

Figure 19 Speed reference and motor state as received by the client

- The host controller will receive the running state and speed from the client controller as shown in Figure 20.

```
canComm.flagStateRxRun
canComm.speedFdb_Hz
```



canComm	struct _CAN_COMM_Obj_	{canHandle=294912,speedCo...
canHandle	unsigned long	294912
speedConv_sf	float	0.100000001
speedInv_sf	float	10.0
speedRef_Hz	float	30.0
speedFdb_Hz	float	35.0
rxMsgData	unsigned int[8]	[0,1,1,44,1...]
txMsgData	unsigned int[8]	[1,0,1,94,0...]
rxMsgCount	unsigned int	49391
txMsgCount	unsigned int	49392
waitTimeCnt	unsigned int	1
waitTimeDelay	unsigned int	2
errorFlag	unsigned int	0
flagRxDone	unsigned char	0 '\x00'
flagTxDone	unsigned char	0 '\x00'
flagCmdRxRun	unsigned char	0 '\x00'
flagStateRxRun	unsigned char	1 '\x01'
flagCmdEnable	unsigned char	1 '\x01'

Figure 20 Speed received by host controller

5.4 DMC_LEVEL_4: Closing speed loop using QEP encoder with CAN communication for mobile robot system

Based on the level 3, this lab is modified to control the speed of two motors for a mobile robot system. The system structure is shown as follows:

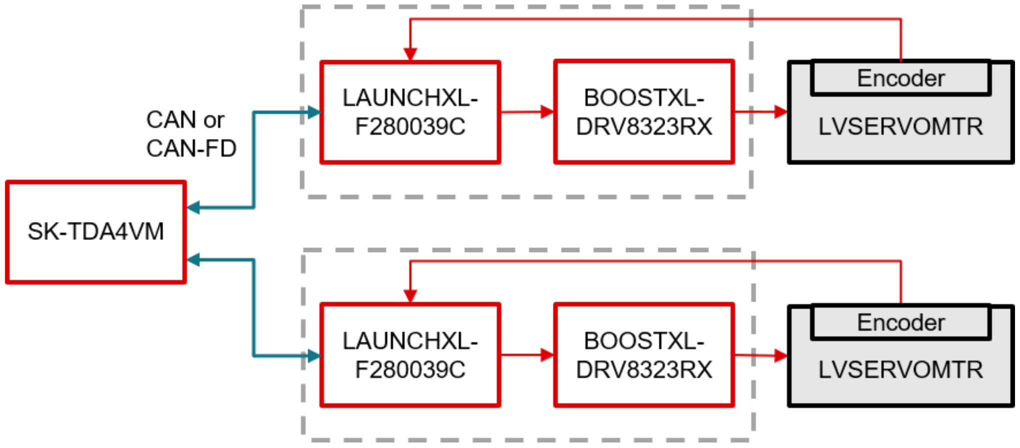


Figure 21 System block diagram of motor control

Each servo motor (LVSERVOMTR) is controlled by a LAUNCHXL-F280039C and BOOSTXL-DRV8323RS board, and the sensored FOC algorithm is implemented here. During the control loop, the frequency of the electrical field in Hz is given by SK-TDA4VM board through CAN or CAN-FD to both LAUNCHXL-F280039C boards. Once the frequency command is received, the LAUNCHXL-F280039C will update the applied frequency for the motor. The position information of the motor rotor is determined by the built-in encoder on the motor, which helps to calculate the applied FOC commutation signals. The current in motor phases is measured to implement the FOC motor control and prevent overcurrent.

The electrical field frequency is used to control the motor, it is also related to the rotor speed. To determine it, the frequency of the electrical field f_e can be first converted into rpm. The calculation is shown as follows. According to the parameter of motor (Teknic M-2310P-LN-04K), the pole pair of the rotor PP is 4 (8 poles). The mechanical frequency of the rotor can be given by:

$$f_m = f_e/PP = 0.25 \cdot f_e$$

So, the rotor speed in rpm:

$$n_m = f_m \cdot 60 = 15 f_e$$

The speed control is based on the asynchronous communication between SK-TDA4VM and LAUNCHXL-F280039C, which is shown in the figure below:

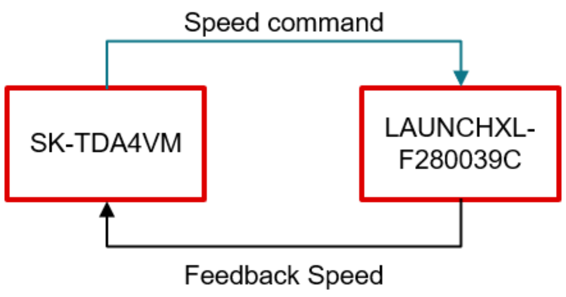


Figure 22 Speed control loop via CAN

The measured speed, also feedback speed is sent continuously from LAUNCHXL-F280039C to SK-TDA4VM with a certain frequency, which can be configured in "servo_main.h" under the name "CAN_FEEDBACK_FREQ_kHz". The speed command from SK-TDA4VM to LAUNCHXL-F280039C is sent independently with another frequency. It is recommended to set the frequency of speed command slower than the feedback frequency.

Similar to DMC_LEVEL_3, the format of a single CAN dataframe is shown in the following table:

Data bit	Data type	Description
[0]	unsigned int	Set motor flag: 1 run, 0 stop
[1]	unsigned int	Motor state flag: 1 run, 0 stop
[2]	unsigned int	MSB and LSB of set speed
[3]	unsigned int	
[4]	unsigned int	MSB and LSB of feedback speed
[5]	unsigned int	
[6]	unsigned int	Unused
[7]	unsigned int	Unused

Table 4 CAN data frame

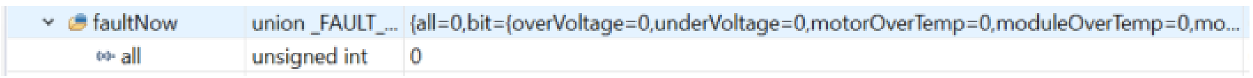
Each data frame includes 8 unsigned int type elements. Both the data element sending and receiving share the same data format, where data bit [0], [2], [3] are used to save the set speed command from SK-TDA4VM, data bit [1], [4], [5] the feedback data. The bit [0] and [1] indicates respectively the enable motor command and the current motor states. Before the receiving and sending the speed value into the data frame, both of them will be multiplied by factor 10 to carry one decimal place. Then, the first 8 bits of the speed will be saved as MSB, the last 8 bits as LSB. Negative values of speed are also available for speed setting and receiving.

To start this lab, please finish the hardware set-up in “4.3 For testing DMC_LEVEL_4” first. Then, download the software into LAUNCHXL-F280039C and SK-TDA4VM.

For more information on SK-TDA4VM side, please refer [Edge AI Robotics Academy]

6.0 Additional Debug information

- Motor does not spin after setting the flags “flagEnableSys” =1 and “flagRunIdentAndOnLine” = 1
    - Check if the “fault_Now.all” flag is set to 16, if yes, there a fault. Clear the flag by resetting the “fault_Now.all” flag to 0.
    - The expression is found in motorVars→ faultNow→ all



faultNow	union _FAULT_...	{all=0,bit={overVoltage=0,underVoltage=0,motorOverTemp=0,moduleOverTemp=0,mo...
all	unsigned int	0

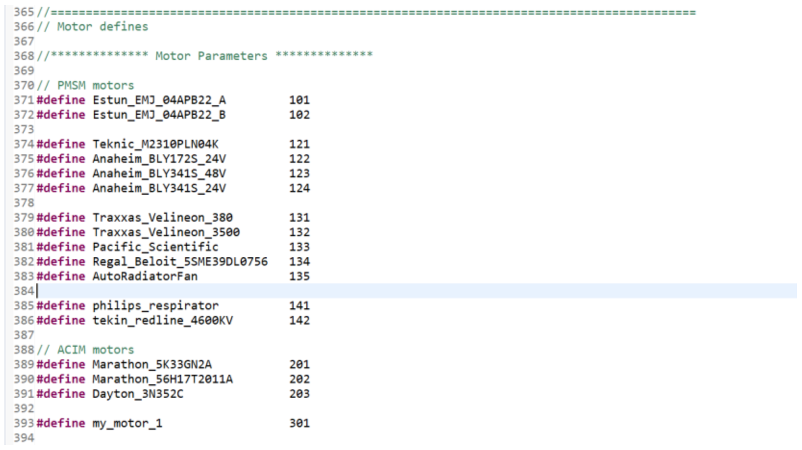
- Where to modify the CAN settings?
    - To modify the data length, check “communication.h” file

```
// the typedefs
#define MSG_DATA_LENGTH         8
#define TX_MSG_OBJ_ID           1
#define RX_MSG_OBJ_ID           2
```

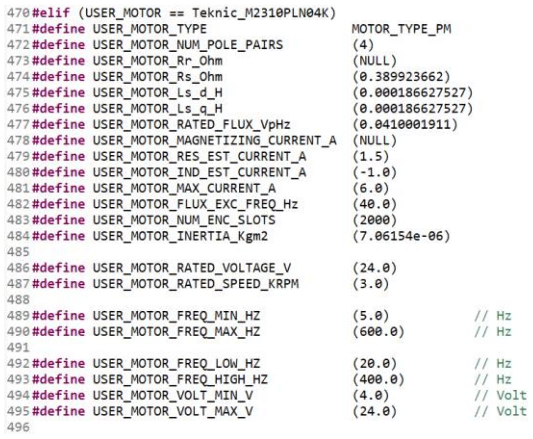
    - To modify the data length, check “communication.c” file

```
//assign CANA base
obj->canHandle = CANA_BASE;

obj->speedConv_sf = 0.1f;          // uint16->float(unit=0.1Hz)
obj->speedInv_sf = 10.0f;          // float->uint16(unit=0.1Hz)

obj->txMsgCount = 0;                // for debug
obj->rxMsgCount = 0;                // for debug

obj->waitTimeCnt = 1000;            // 1s/1000ms
obj->waitTimeDelay = 2;             // 2ms
```

- Encoder Calibration
    - In the file user.h , motor parameters and encoder parameter are defined. Suggest to tune it accordingly for your design and motor.



```
//============================================================================================
// Motor defines

//************** Motor Parameters **************

// PMSM motors
#define Estun_EMJ_04APB22_A        101
#define Estun_EMJ_04APB22_B        102

#define Teknic_M2310PLN04K         121
#define Anaheim_BLY172S_24V        122
#define Anaheim_BLY341S_48V        123
#define Anaheim_BLY341S_24V        124

#define Traxxas_Velineon_380       131
#define Traxxas_Velineon_3500      132
#define Pacific_Scientific         133
#define Regal_Beloit_5SME39DL0756  134
#define AutoRadiatorFan            135

#define philips_respirator         141
#define tekin_redline_4600KV       142

// ACIM motors
#define Marathon_5K33GN2A          201
#define Marathon_56H17T2011A       202
#define Dayton_3N352C              203

#define my_motor_1                 301
```



```
#elif (USER_MOTOR == Teknic_M2310PLN04K)
#define USER_MOTOR_TYPE                   MOTOR_TYPE_PM
#define USER_MOTOR_NUM_POLE_PAIRS         (4)
#define USER_MOTOR_Rr_Ohm                 (NULL)
#define USER_MOTOR_Rs_Ohm                 (0.389923662)
#define USER_MOTOR_Ls_d_H                 (0.000186627527)
#define USER_MOTOR_Ls_q_H                 (0.000186627527)
#define USER_MOTOR_RATED_FLUX_VpHz        (0.0410001911)
#define USER_MOTOR_MAGNETIZING_CURRENT_A  (NULL)
#define USER_MOTOR_RES_EST_CURRENT_A      (1.5)
#define USER_MOTOR_IND_EST_CURRENT_A      (-1.0)
#define USER_MOTOR_MAX_CURRENT_A          (6.0)
#define USER_MOTOR_FLUX_EXC_FREQ_Hz       (40.0)
#define USER_MOTOR_NUM_ENC_SLOTS          (2000)
#define USER_MOTOR_INERTIA_Kgm2           (7.06154e-06)

#define USER_MOTOR_RATED_VOLTAGE_V        (24.0)
#define USER_MOTOR_RATED_SPEED_KRPM       (3.0)

#define USER_MOTOR_FREQ_MIN_HZ            (5.0)          // Hz
#define USER_MOTOR_FREQ_MAX_HZ            (600.0)        // Hz

#define USER_MOTOR_FREQ_LOW_HZ            (20.0)         // Hz
#define USER_MOTOR_FREQ_HIGH_HZ           (400.0)        // Hz
#define USER_MOTOR_VOLT_MIN_V             (4.0)          // Volt
#define USER_MOTOR_VOLT_MAX_V             (24.0)         // Volt
```