

# Quick Response Control of PMSM Using Fast Current Loop

Ramesh T Ramamoorthy

## ABSTRACT

This application report helps to evaluate the fast current loop (FCL) software library for high-bandwidth inner loop current control of PM servo drives based on the TMS320F2837x MCU using TI's DesignDRIVE IDDK EVM kit and the C2000Ware MotorControl SDK. TMS320F2837x devices are a part of the C2000™ family of microcontrollers, which enable cost-effective design of intelligent, high-bandwidth controllers for 3-phase motors, by reducing the system components and increasing efficiency. The devices are appropriately supported by hardware accelerators to implement FCL algorithms to rival or surpass similar implementations using FPGA in terms of performance, cost, and development time.

The application report describes the following:

- Incremental build levels calling modular FCL functions spread across CPU and CLA
- Experimental results

## Contents

1	Introduction .....	3
2	Benefits of the TMS320F2837x MCU for High-Bandwidth Current Loop.....	3
3	Current Loops in Servo Drives .....	4
4	Outline of the Fast Current Loop Library .....	5
5	Fast Current Loop Library Evaluation .....	7
6	Incremental Build Level 1 .....	8
7	Incremental Build Level 2 .....	11
8	Incremental Build Level 3 .....	16
9	Incremental Build Level 4 .....	20
10	Incremental Build Level 5 .....	22
11	Incremental Build Level 6 .....	24

## List of Figures

1	Basic Scheme of FOC for AC Motor .....	4
2	Fast Current Loop Library Block Diagram .....	5
3	Level 1 Block Diagram .....	8
4	Expressions Window for Build Level 2.....	9
5	Voltage Angle and SVGEN Ta, Tb, and Tc .....	10
6	DAC Outputs Showing Ta, Tb Waveforms .....	11
7	Level 2 Block Diagram.....	12
8	Scope Plot of Reference Angle and Rotor Position .....	14
9	Expressions Window.....	15
10	Level 3 Block Diagram Showing Inner Most Loop - FCL .....	17
11	Expressions Window Snapshot For Latency .....	18
12	Scope Plot of ADCSoC and FCL Completion Events .....	19
13	Level 4 Block Diagram Showing Speed Loop With Inner FCL .....	20
14	Flux and Torque Components of the Stator Current in the Synchronous Reference Frame Under 0.33-pu	

	Step-Load and 0.3-pu Speed.....	21
15	Level 5 Block Diagram Showing Position Loop With Inner FCL .....	23
16	Scope Plot of Reference Position to Servo and Feedback Position.....	23
17	Level 6 Block Diagram.....	25
18	SFRA GUI .....	27
19	SFRA GUI MC .....	28
20	GUI Setup Diagram .....	29
21	SFRA GUI Connected to the C2000 MCU .....	30
22	SFRA Open Loop Bode Plots of the Current Loop Showing Magnitude and Phase Angle .....	31
23	SFRA Closed Loop Bode Plots of the Current Loop Showing Magnitude and Phase Angle .....	32
24	SFRA Open Loop Bode Plots of the Current Loop - Current Feedback With High SNR .....	33
25	SFRA Closed Loop Bode Plots of the Current Loop - Current Feedback With High SNR .....	33
26	Plot of Gain Cross over Frequency vs Phase Margin as Experimentally Obtained .....	35

### List of Tables

1	Summary of FCL Interface Functions .....	6
2	Functions Verified in Each Incremental System Build .....	7
3	Functional Modules Used in Each Incremental System Build .....	8

### Trademarks

C2000 is a trademark of Texas Instruments.

All other trademarks are the property of their respective owners.

## 1 Introduction

The concept of FOC of AC drives is well known and is already outlined in many earlier documents from TI. Modern AC servo drives, depending on the end application, need high-bandwidth current control and speed control to enable superior performance, such as in CNC machines or in fast and precision control applications. Because of the high computational burden and the need for flexible PWMs, a combination of FPGAs, fast external ADCs, and multiple MCUs are used by many designers.

With the TMS320F2837x MCU, due to its higher level of integration, it is possible to implement fast current loop algorithms with the same external hardware used in classical FOC methods. TI has developed the FCL algorithm on the F2837x MCU and implemented it on the DesignDRIVE IDDK platform. With a 10-kHz PWM carrier, the current loop gain crossover frequency is expected to exceed 3 kHz, and the closed loop bandwidth is expected at about 5KHz (per NEMA ICS 16 and Chinese GBT 16439-2009) and the maximum duty cycle is expected to be approximately 96%. Quantitative test results will be published when they are available. This document provides information that helps you evaluate this algorithm on the DesignDRIVE IDDK platform.

### 1.1 Acronyms Used in This Document

- DMC – Digital Motor Control
- IDDK – Industrial Drive Development Kit (from TI)
- MCU – Microcontroller Unit
- FOC – Field-Oriented Control
- TMU – Trigonometric Mathematical Unit (in C2000 MCU)
- CLA – Control Law Accelerator (in C2000 MCU)
- CLB – Configurable Logic Block (in C2000 MCU)
- PMSM – Permanent Magnet Synchronous Motor
- ACIM – AC Induction Motor
- FCL – Fast Current Loop
- HVDMC – High Voltage DMC
- CMPSS – Comparator Subsystem peripheral (in C2000 MCU)
- CNC – Computer Numerical Control
- PWM – Pulse Width Modulation
- FPGA – Field Programmable Gate Array
- ADC – Analog-to-Digital Converter
- ePWM – Enhanced Pulse Width Modulator
- eQEP – Enhanced Quadrature Encoder Pulse Module
- eCAP – Enhanced Capture Module

## 2 Benefits of the TMS320F2837x MCU for High-Bandwidth Current Loop

The C2000 MCU family of devices possesses the desired computation power to execute complex control algorithms and the correct combination of peripherals to interface with the various components of the DMC hardware, such as the ADC, ePWM, QEP, and eCAP. These peripherals have all the necessary hooks to provide flexible PWM protection, such as trip zones for PWMs and comparators.

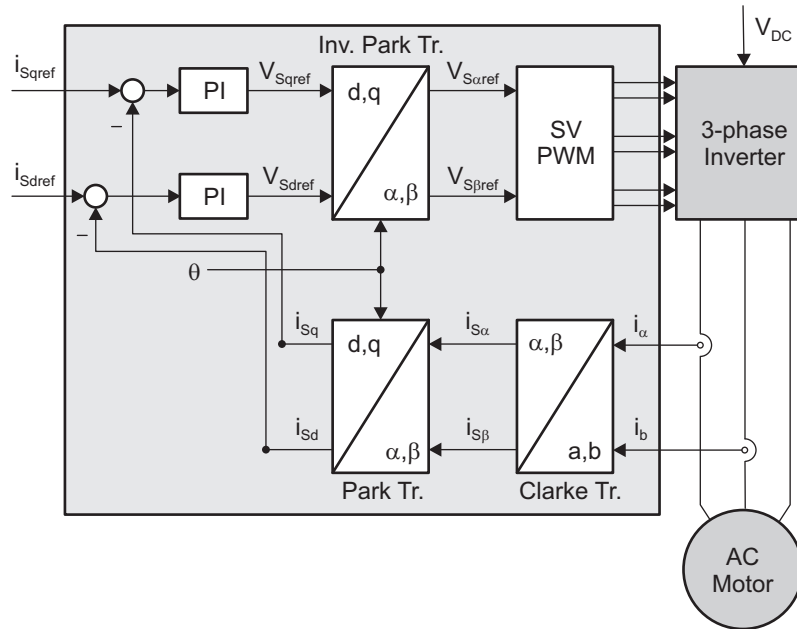
The F2837x MCU contains additional hardware features such as the following:

- Higher CPU and CLA clock frequency
- Four high-speed, 12- and 16-bit ADCs
- TMU
- Parallel processing block, such as the CLA
- CLBs

Together, these features provide enough hardware support to increase computational bandwidth compared to its predecessors and offer superior real-time control performance. In addition, the C2000 ecosystem of software (libraries and application software) and hardware (TMDXIDDK379D) help users reduce the time and effort needed to develop a high-end digital motor control solution.

### 3 Current Loops in Servo Drives

Figure 1 shows the basic current loop used in FOC servo drives.



**Figure 1. Basic Scheme of FOC for AC Motor**

Two motor phase currents are measured. These measurements feed the Clarke transformation module. The outputs of this projection are designated  $i_{sa}$  and  $i_{sb}$ . These two components of the current along with the rotor flux position are the inputs of the Park transformation, which transform them to currents ( $i_{sd}$  and  $i_{sq}$ ) in d and q rotating reference frame. The  $i_{sd}$  and  $i_{sq}$  components are compared to the references  $i_{sdref}$  (the flux reference) and  $i_{sqref}$  (the torque reference). At this point, the control structure shows an interesting advantage; it can be used to control either synchronous or asynchronous machines by simply changing the flux reference and obtaining the rotor flux position. In the synchronous permanent magnet motor, the rotor flux is fixed as determined by the magnets, so there is no need to create it. Therefore, when controlling a PMSM motor,  $i_{sdref}$  can be set to zero, except during field weakening.

Because ACIM motors need a rotor flux creation to operate, the flux reference must not be zero. This conveniently solves one of the major drawbacks of the classic control structures: the portability from asynchronous to synchronous drives. The torque command  $i_{sqref}$  can be connected to the output of the speed regulator. The outputs of the current regulators are  $V_{sqref}$  and  $V_{sdref}$ . These outputs are applied to the inverse Park transformation. Using the position of rotor flux, this projection generates  $V_{suref}$  and  $V_sprel$ , which are the components of the stator vector voltage in the stationary orthogonal reference frame. These components are the inputs of the Space Vector PWM. The outputs of this block are the signals that drive the inverter.

**NOTE:** Both Park and inverse Park transformations need the rotor flux position. Obtaining this rotor flux position depends on the AC machine type (synchronous or asynchronous).

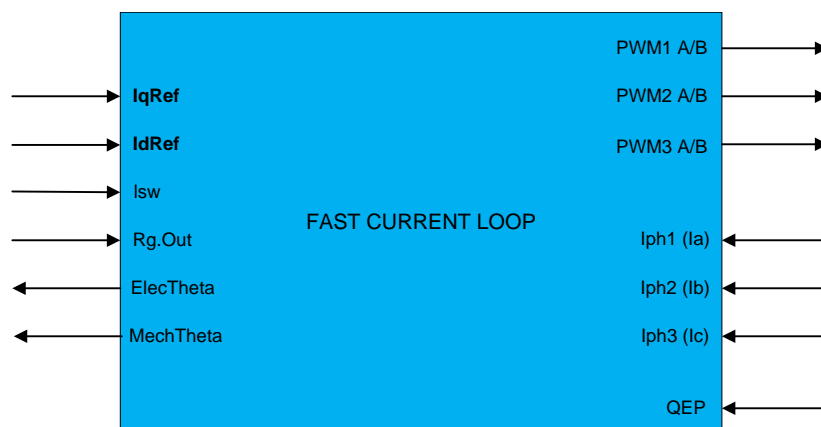
## 4 Outline of the Fast Current Loop Library

The major challenge in digital motor control systems is the influence of sample and hold, as well as transportation lag inside the loop that slows down the system, impacting its performance at higher frequencies and running speeds. To overcome this problem, improve the current loop bandwidth, and enable higher DC bus usage, the following are necessary:

- High computational power
- The correct set of control peripherals
- Superior control algorithm

While the TMS320F2837x provides the necessary hardware support for higher performance, TI's FCL library, which runs on the C2000 MCU, provides the necessary algorithmic support.

To improve the operational range of FCLs, the latency between feedback sampling and the PWM update must be as small as possible. Typically, a latency of 2  $\mu$ S or less is considered acceptable in many applications. Traditionally, this task is implemented using a combination of high-end FPGAs, external ADCs, and MCUs.



**Figure 2. Fast Current Loop Library Block Diagram**

The FCL library uses the following features in the F2837x MCU:

- TMU
- Four high-speed 12- and 16-bit ADCs
- Multiple parallel processing blocks such as CLA

**Figure 2** shows the block diagram of the FCL library with its inputs and outputs. The FCL library partitions the algorithm across the CPU, CLA, and TMU to bring down the latency to under 1.0  $\mu$ s compared to the acceptable 2  $\mu$ s. Further optimization is possible if the algorithm is written in assembly.

The FCL library supports two types of current regulators, a standard PI controller and a complex controller. The complex controller can provide additional bandwidth over the standard PI controller at higher speeds. Both current regulators are provided for user evaluation. In the example project, the current regulator can be selected by setting the FCL\_CNTLR macro appropriately and studying how they compare.

Table 1 lists the FCL API functions and their descriptions.

**Table 1. Summary of FCL Interface Functions**

API Function	Description
<code>uint32_t FCL_getSwVersion(void)</code>	Function that returns a 32-bit constant, and for this version the value returned is 0x00000006.
<code>void FCL_runComplexCtrl(void)</code>	Function that performs the complex control as part of the FCL
<code>void FCL_runPICtrl(void)</code>	Function that performs the PI control as part of the FCL
<code>void FCL_runPICtrlWrap(void)</code>	Wrap up function to be called by the user application at the completion of FCL in PI control mode before exiting ISR
<code>void FCL_runQEPWrap(void)</code>	Function to be called by the user application to wrap up the QEP feedback procedure. This function is used only in FCL_LEVE2.
<code>void FCL_runComplexCtrlWrap(void)</code>	Wrap up function to be called by the user application at the completion of FCL in complex control mode before exiting ISR
<code>void FCL_initPWM( uint32_t basePhaseU, uint32_t basePhaseV, uint32_t basePhaseW );</code>	Function to initialize PWMs for the FCL operation, this will be called by the user application during the initialization or setup process.
<code>void FCL_resetController(void)</code>	This function is called to reset the FCL variables and is useful when you want to stop the motor and restart the motor.
<code>void FCL_initQEP(uint32_t baseA);</code>	This function initializes the eQEP peripheral for connecting to the QEP
<code>void FCL_initADC(uint32_t resultBaseA, ADC_PPBNr baseA_PPBNr, uint32_t resultBaseB, ADC_PPBNr baseB_PPBNr, uint32_t adcBasePhaseW);</code>	This function initializes the ADCs that are used to sense the motor phase currents

For more information on the library, see the [Fast Current Loop MotorControl SDK Library User's Guide](#) available at:

`\\ti\c2000\C2000Ware_MotorControl_SDK_version\libraries\fc\docs.`

**NOTE:** The library is written in a modular format and is able to port over to user platforms using F2837x devices if the following conditions are met:

- Motor phase current feedbacks are read into variables internal to the library. However, D axis and Q axis current feedbacks are available.
- PWM modules controlling motor phase A, B, and C are linked to the library.
- A QEP module connecting to the QEP sensor is linked to the library.
- CLA tasks one through four are used by the library. This must be accommodated in the user application.

## 5 Fast Current Loop Library Evaluation

TI provides the FCL algorithm software library for evaluation using the TMS320F2837x MCU on TI's DesignDRIVE IDDK platform. This section presents a step-by-step approach to evaluating the FCL software library to control a permanent magnet synchronous motor using an example project.

Example project features:

- Sensored FOC of the PMSM motor
- FCL using the FCL library
- Position, speed, and torque control loops
- Position sensor support: incremental encoder (QEP)
- Current sensing: analog feedback using the ADC (from LEM sensors (Fluxgate/HALL))

### 5.1 Evaluation Setup

#### 5.1.1 Hardware

The example project is evaluated on TI's [Design DRIVE Development Kit IDDK - TMDXIDDK379D](#). The [DesignDRIVE IDDK Hardware Reference Guide](#) and [DesignDRIVE IDDK User Guide](#) for the kit can be found at `C:\ti\c2000\C2000Ware_MotorControl_SDK_version\solutions\tmdxiddk379d\docs\`

The [DesignDRIVE IDDK Hardware Reference Guide](#) gives an overview of the various hardware functional blocks in the kit, discusses the various ground configurations it supports and the safety measures necessary to work with the kit. It is important to understand the hardware and the safety aspects before working with the kit.

The [DesignDRIVE IDDK User Guide](#) helps in setting up the hardware and in getting started with the software, connecting to the target platform and working with the debug environment using Code Composer Studio.

#### 5.1.2 Software

When the MCSDK software package is installed, the FCL software library can be found at `C:\ti\c2000\C2000Ware_MotorControl_SDK_version\libraries\observers\fc\lib`

The FCL example project can be found at:

`C:\ti\c2000\C2000Ware_MotorControl_SDK_version\solutions\tmdxiddk379d\2837x\ccs\sensored_foc`

Here, a few more build levels are added to step through with integrating the FCL library to run the motor in speed mode and position mode.

#### 5.1.3 Incremental System Build

The system is gradually built up through various build levels verifying specific functionality at each level so that at the final level, the system is fully verified and complete in all aspects. Five levels of incremental system build are designed to verify the various modules used in the system.

[Table 2](#) and [Table 3](#) summarize the core functions integrated and tested at each build level in the incremental build approach.

**Table 2. Functions Verified in Each Incremental System Build**

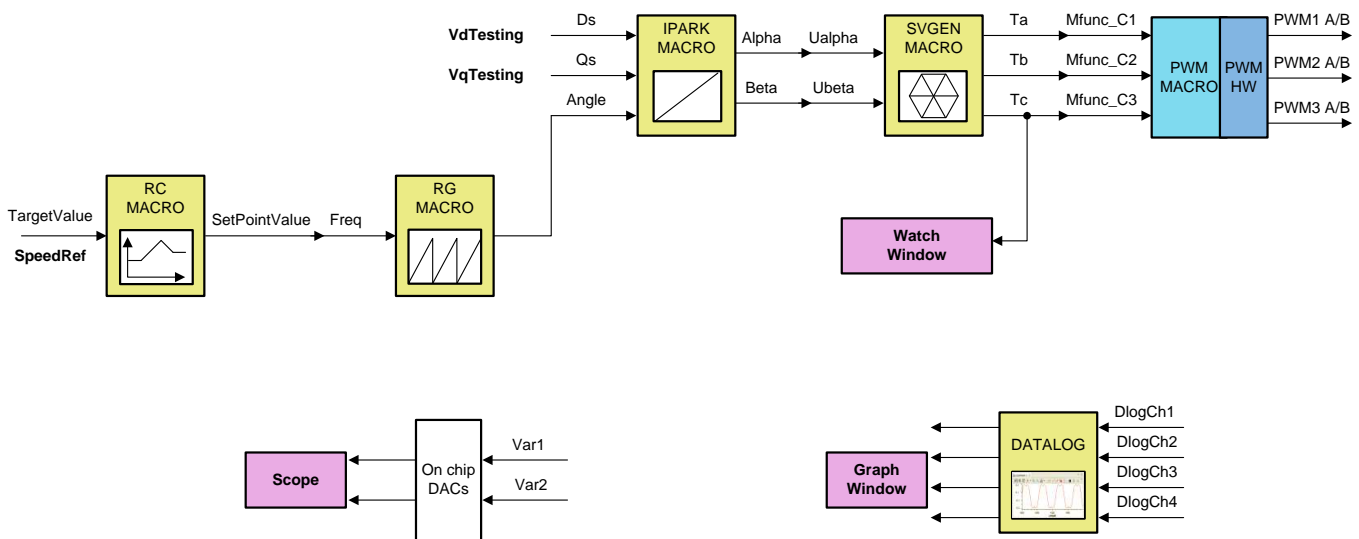
Build Level	Functional Integration/Verification
LEVEL 1	Basic PWM generation
LEVEL 2	Open loop control of motor / calibration of feedbacks
LEVEL 3	CURRENT MODE - Closing current loop using FCL library
LEVEL 4	SPEED MODE - Closing speed loop using inner FCL verified in LEVEL 3
LEVEL 5	POSITION MODE - Closing position loop using inner speed loop verified in LEVEL 4
LEVEL 6	SFRA ANALYSIS - Performing SFRA on current loop running motor in speed mode (LEVEL 4)

**Table 3. Functional Modules Used in Each Incremental System Build**

Software Module	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
PWM Generation	✓✓	✓	FCL lib	FCL lib	FCL lib	FCL lib
QEP Interface in CLA		✓✓	✓	✓	✓	✓
FOC functions			✓✓	✓	✓	✓
SFRA functions						✓✓

## 6 Incremental Build Level 1

The block diagram of the system built in BUILDLEVEL 1 is shown in Figure 3. During this step, keep the motor disconnected.

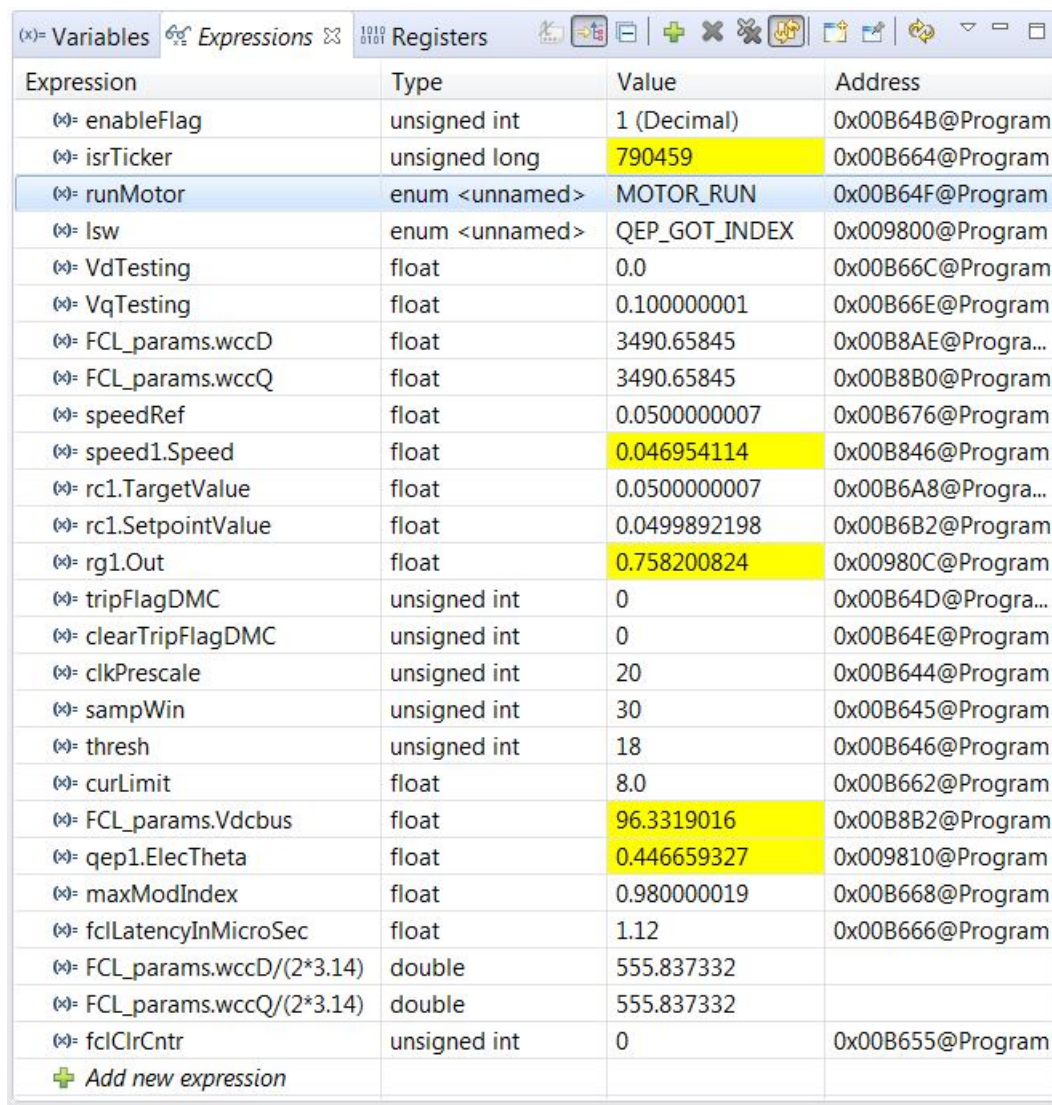

**Figure 3. Level 1 Block Diagram**

Assuming the load and build steps described in the [DesignDRIVE IDDK User Guide](#) completed successfully, this section describes the steps for a “minimum” system check-out which confirms operation of system interrupt, the peripheral and target independent inverse park transformation and space vector generator modules, and the PWM initializations and update modules.

1. Open `fcl_qep_f2837x_tmdxiddk_settings.h` and select the level 1 incremental build option by setting the BUILDLEVEL to FCL\_LEVEL1 (#define BUILDLEVEL FCL\_LEVEL1).
2. Right-click on the project name and click Rebuild Project.
3. When the build is complete, click on debug button, reset the CPU, restart, enable real time mode, and run.



4. Add variables to the expressions window by right-clicking within the Expressions Window and importing the *fcl\_f2837x\_tmdxiddk\_vars.txt* file from the debug directory. Figure 4 shows the variables imported into the Expressions Window from this file. Ignore the values shown against the variables for now.



Expression	Type	Value	Address
enableFlag	unsigned int	1 (Decimal)	0x00B64B@Program
isrTicker	unsigned long	790459	0x00B664@Program
runMotor	enum <unnamed>	MOTOR_RUN	0x00B64F@Program
lsw	enum <unnamed>	QEP_GOT_INDEX	0x009800@Program
VdTesting	float	0.0	0x00B66C@Program
VqTesting	float	0.100000001	0x00B66E@Program
FCL_params.wccD	float	3490.65845	0x00B8AE@Progra...
FCL_params.wccQ	float	3490.65845	0x00B8B0@Program
speedRef	float	0.0500000007	0x00B676@Program
speed1.Speed	float	0.046954114	0x00B846@Program
rc1.TargetValue	float	0.0500000007	0x00B6A8@Progra...
rc1.SetpointValue	float	0.0499892198	0x00B6B2@Program
rg1.Out	float	0.758200824	0x00980C@Program
tripFlagDMC	unsigned int	0	0x00B64D@Progra...
clearTripFlagDMC	unsigned int	0	0x00B64E@Program
clkPrescale	unsigned int	20	0x00B644@Program
sampWin	unsigned int	30	0x00B645@Program
thresh	unsigned int	18	0x00B646@Program
curlimit	float	8.0	0x00B662@Program
FCL_params.Vdcbus	float	96.3319016	0x00B8B2@Program
qep1.ElecTheta	float	0.446659327	0x009810@Program
maxModIndex	float	0.980000019	0x00B668@Program
fclLatencyInMicroSec	float	1.12	0x00B666@Program
FCL_params.wccD/(2*3.14)	double	555.837332	
FCL_params.wccQ/(2*3.14)	double	555.837332	
fclClrCntr	unsigned int	0	0x00B655@Program
+ Add new expression			

**Figure 4. Expressions Window for Build Level 2**

5. Set *EnableFlag* to 1 in the watch window. The variable named *IsrTicker* is incrementally increased, as seen in the watch windows, to confirm the interrupt working properly. In the software, the key variables to be adjusted are:
  - *speedRef*: for changing the rotor speed in per-unit.
  - *VdTesting*: for changing the d-qxis voltage in per-unit.
  - *VqTesting*: for changing the q-axis voltage in per-unit.

## 6.1 SVGEN Test

The *speedRef* value is fed into the ramp control module to ramp up the speed command. The output of the ramp module is fed into a ramp generator to generate the angle for sinewave generation. This angle as well as the variables *VdTesting* and *VqTesting* feeds into inverse park transformation block which then feeds into space vector modulation modules to generate three phase PWMs.

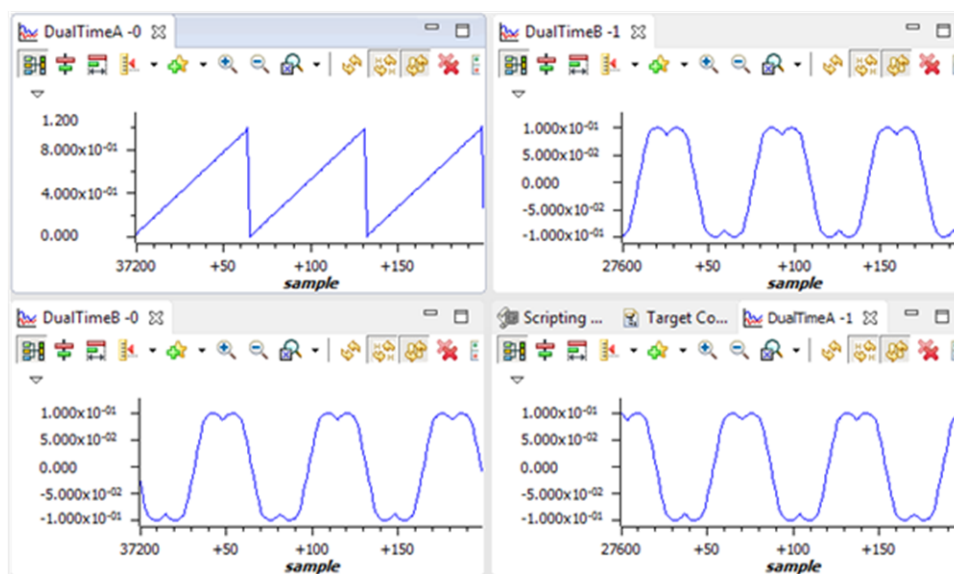
The outputs from space vector generation module can be viewed using the graph tool from the debug environment by clicking **Tools --> Graph --> Dual Time**. Then, from the graph window, click Import and browse to and select:

```
lsolutions\txmdxiddk379d\2837x\debug\fc1_f2837x_tmdxiddk_graph1.graphProp
```

This plots two graphs representing variables pointed by dlogCh1 and dlogCh2. Likewise, another graph can be opened by selecting:

```
solutions\txmdxiddk379d\2837x\debug\fc1_f2837x_tmdxiddk_graph2.graphProp
```

This plots two graphs representing variables pointed by dlogCh3 and dlogCh4.

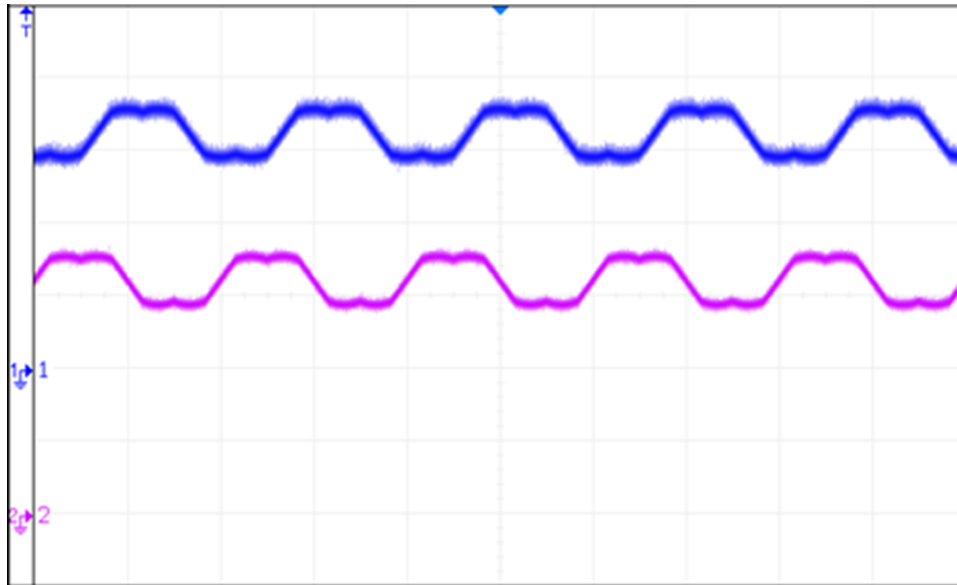


**Figure 5. Voltage Angle and SVGEN Ta, Tb, and Tc**

These are shown in [Figure 5](#). These are the voltage vector angle, and the pulse width values for the phases A, B, and C and are denoted as Ta, Tb, and Tc, where Ta, Tb, and Tc waveforms are 120 degrees apart from each other. Specifically, Tb lags Ta by 120 degrees and Tc leads Ta by 120 degrees. These are generated based on the values of *speedRef*, *VdTesting* and *VqTesting*. These values can be changed to see the impact on these waveforms. Check the PWM test points on the board to observe PWM pulses (PWM-1H to 3H and PWM-1L to 3L) and ensure that the PWM module is running properly.

## 6.2 Testing SVGEN With DACs

To monitor internal signal values in real time, onchip DACs are used. DACs are part of the analog module. DACs B and C are available for this purpose. This is shown in [Figure 6](#).



**Figure 6. DAC Outputs Showing Ta, Tb Waveforms**

## 6.3 Inverter Functionality Verification

After verifying the space vector generation and PWM modules, the 3-phase inverter hardware can be tested by looking at inverter outputs U, V, and W on a scope. This can be compared against the PWM pulses (PWM-1H to 3H) fed into the inverter. It is advisable to gradually increase the DC bus voltage during this test. Check inverter outputs U, V, and W using an oscilloscope with respect to the inverter GND, while taking care of scope isolation requirements. This ensures that the inverter is working properly.

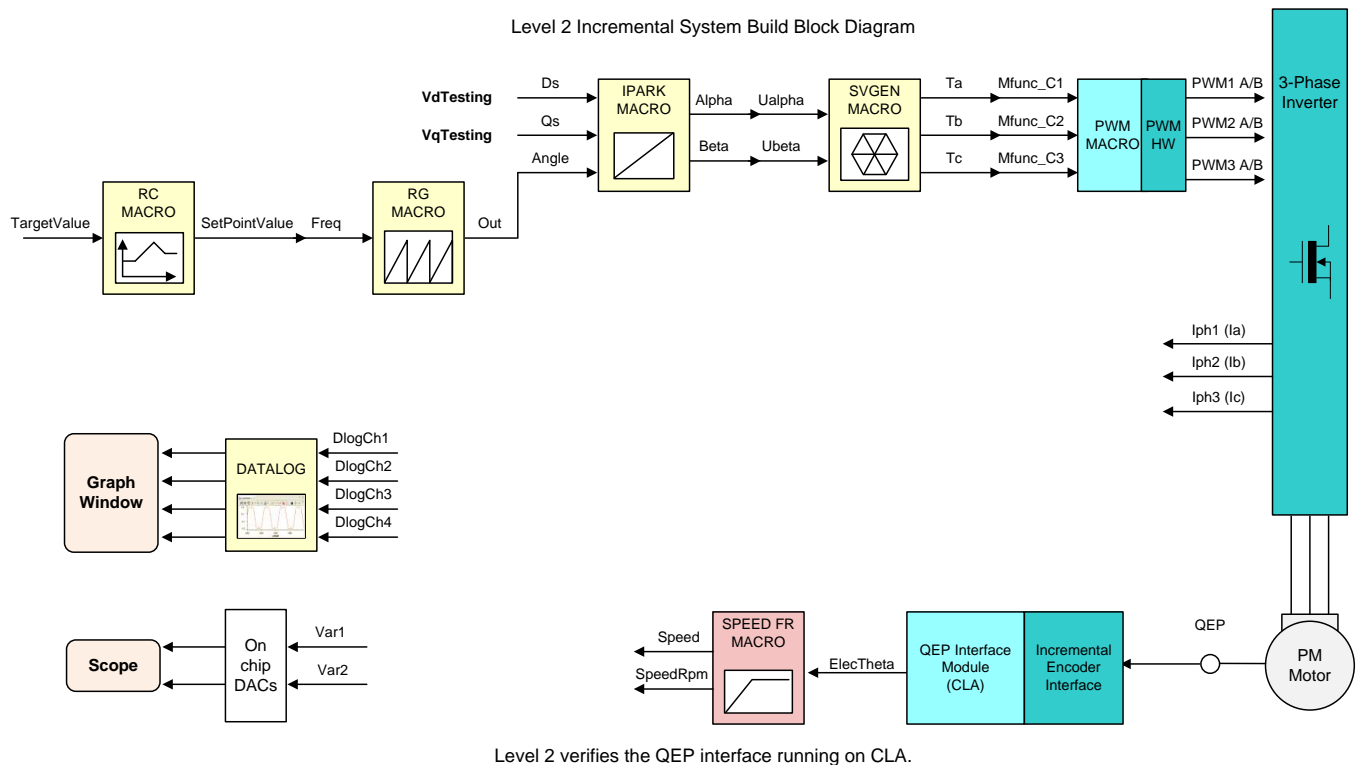
## 7 Incremental Build Level 2

Assuming build level 1 is completed successfully, this section verifies the overcurrent protection limits of the inverter and QEP interface running out of the CLA. In this build, the motor is run in open loop.

The motor can be connected to the HVDMC board because the PWM signals are successfully proven through the incremental build level 1.

1. Open `fcl_qep_f2837x_tmdxiddk_settings.h` and select the level 2 incremental build option by setting the BUILDLEVEL to FCL\_LEVEL2 (`#define BUILDLEVEL FCL_LEVEL2`).
2. Select CURRENT\_SENSE to LEM\_CURRENT\_SENSE and POSITION\_ENCODER to QEP\_POS\_ENCODER.
3. Right-click on the project name and click *Rebuild Project*.
4. When the build is complete, click *Debug*, reset the CPU, restart, enable real-time mode, and run.

Figure 7 shows the level 2 block diagram.



**Figure 7. Level 2 Block Diagram**

Set *enableFlag* to 1 in the watch window. The variable named *isrTicker* is incrementally increased as shown in the Expressions window to confirm the interrupt is working properly. Now set the variable named *runMotor* to *MOTOR\_RUN*; the motor starts spinning after a few seconds if enough voltage is applied to the DC-Bus.

In the software, the key variables to be adjusted are same as in previous level and are given again below for reference.

- *speedRef*: for changing the rotor speed in per-unit
- *VdTesting*: for changing the d-axis voltage in per-unit
- *VqTesting*: for changing the q-axis voltage in per-unit

During the open loop tests, *VqTesting*, *speedRef*, and DC bus voltages must be adjusted carefully for PM motors so that the generated *Bemf* is lower than the average voltage applied to motor winding. This adjustment prevents the motor from stalling or vibrating.

## 7.1 Setting the Overcurrent Limit in the Software

The board has various current sense methods, such as shunt, LEM, and SDFM. Overcurrent monitoring is provided for signals generated from shunt and LEM using an on-chip comparator subsystem (CMPSS) module. The module has a programmable comparator and a programmable digital filter. The comparator generates the protection signal. The reference to the comparator is user programmable for both positive and negative currents. The digital filter module qualifies the comparator output signal, verifying its sanity by periodically sampling and validating the signal for a certain count time within a certain count window, where the periodicity, count, and count window are user programmable.

In the Expressions window, users can see the following variables:

- *clkPrescale* – sets the sampling frequency of the digital filter
- *sampWin* – sets the count window
- *thresh* – sets the minimum count to qualify the signal within *sampWin*
- *curLimit* – sets the permitted current maximum through both shunt and LEM current sensors

*tripFlagDMC* is a flag variable that represents the overcurrent trip status of the inverter. If this flag is set, then you can adjust the previous settings and try to rerun the inverter by setting *clearTripFlagDMC* to 1. This clears *tripFlagDMC* and restarts the PWMs.

The default current limit setting is to shut down at 8 A. Any of these settings can be fine-tuned to suit your system. When satisfactory values are identified, write them down, modify the code with these new values, and rebuild and reload for further tests.

It is possible to shut down the inverter using a digital signal from an external source through H9. No code is provided currently, but it can be used as an exercise to experiment and learn.

## 7.2 Current Sense Method

The CURRENT\_SENSE method chosen for this library is LEM\_CURRENT\_SENSE. It is possible to use the SHUNT\_CURRENT\_SENSE method in certain configurations, but it is not included in the library.

## 7.3 Voltage Sense Method

Voltage is sensed using the sigma delta filter module 3. Look out for the variable *FCL\_params.Vdcbus* in the Expressions window. Vary the DC bus voltage slowly and verify whether this variable tracks this change properly. For example, a 100-V DC voltage should be shown as 100.0 by this variable.

## 7.4 Setting Current Regulator Limits

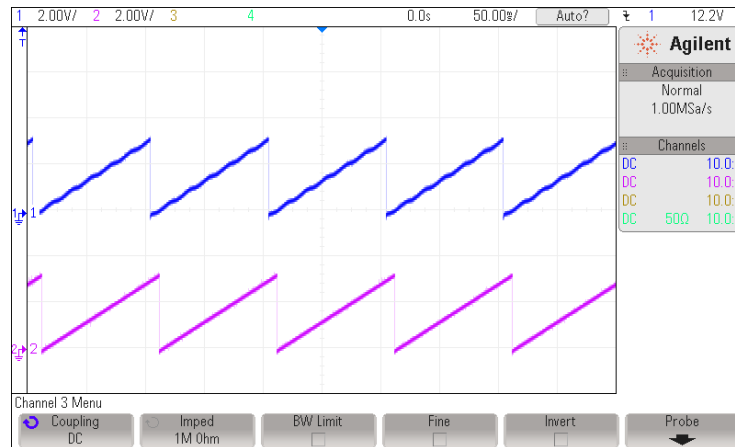
The outputs of the current regulators control the voltages applied on both the d-axis and q-axis. The vector sum of the d and q outputs must be less than 1.0, which refers to the maximum duty cycle for the SVGEN macro. In this particular application, the maximum allowed duty cycle is set to 0.96. Higher computational speeds allow higher duty cycle operation and better use of the DC bus voltage.

The current regulator output is represented by the same variable *pi\_id.out* and *pi\_iq.out* in both PI and complex controller modes. The regulator limits are set by *pi\_id.Umax/min* and *pi\_iq.Umax/min*.

Bring the system to a safe stop by reducing the bus voltage to zero, taking the controller out of real-time mode, and resetting.

## 7.5 Position Encoder Feedback and runSpeedFR() Test

During all the previous tests, the position encoder interface was continuously estimating position information. Therefore, no new code is needed to verify the position encoder interface. When the motor is commanded to run, it is subjected to an initial alignment stage where the electrical angle and the QEP angle count are set to zero. If a resolver or absolute encoder (EnDat or BiSS-C) is used, its initial position at electrical angle zero is identified for run-time corrections. Estimated position information is made available on DAC-C, while the reference position (*rg1.Out*) used to perform open-loop motor control is displayed on DAC-B. [Figure 8](#) shows these signals brought out on H10 on the IDDK, and their scope plots.



**Figure 8. Scope Plot of Reference Angle and Rotor Position**

The waveform of channel 2 represents the reference position, while channel 1 represents the estimated position. The ripple in position estimate is indicative of the fact that the motor runs with some minor speed oscillation. Because of open-loop control, the rotor position and reference position may not align. However, it is important to ensure that the sense of change of the estimated angle is the same as that of the reference; otherwise, it indicates that the motor has a reverse sense of rotation. This can be fixed by either swapping any two wires connecting to the motor, or reversing the angle estimate as in the pseudocode in the software (see [Equation 1](#)).

$$\text{angle} = 1.0 - \text{angle} \quad (1)$$



To ensure that the function *runSpeedFR()* works, change the *speedRef* variable in the Expressions window, as shown in [Figure 9](#), and check whether the estimated speed variable, *speed1.Speed*, follows the commanded speed. Because the motor is a PM motor, where there is no slip, the running speed follows the commanded speed regardless of the control being open loop.

Expression	Type	Value	Address
enableFlag	unsigned int	1 (Decimal)	0x00B64B@Program
isrTicker	unsigned long	886275	0x00B664@Program
runMotor	enum <unnamed>	MOTOR_RUN	0x00B64F@Program
lsw	enum <unnamed>	QEP_GOT_INDEX	0x009800@Program
VdTesting	float	0.0	0x00B66C@Program
VqTesting	float	0.100000001	0x00B66E@Program
speedRef	float	0.0500000007	0x00B676@Program
speed1.Speed	float	0.0521390103	0x00B846@Program
rc1.TargetValue	float	0.0500000007	0x00B6A8@Program
rc1.SetpointValue	float	0.0499892198	0x00B6B2@Program
rg1.Out	float	0.160170674	0x00980C@Program
tripFlagDMC	unsigned int	0	0x00B64D@Program
clearTripFlagDMC	unsigned int	0	0x00B64E@Program
clkPrescale	unsigned int	20	0x00B644@Program
sampWin	unsigned int	30	0x00B645@Program
thresh	unsigned int	18	0x00B646@Program
curlimit	float	8.0	0x00B662@Program
FCL_params.Vdcbus	float	96.4182739	0x00B8B2@Program
qep1.ElecTheta	float	0.719906092	0x009810@Program
Add new expression			

**Figure 9. Expressions Window**

When the tests are complete, bring the system to a safe stop by reducing the bus voltage, taking the controller out of real-time mode, and resetting it. Now the motor stops.

## 8 Incremental Build Level 3

Assuming the previous section is completed successfully, this section verifies the dq-axis current regulation performed by the FCL. One of the two current controllers can be chosen: PI or complex. The bandwidth of the controllers can be set in the debug window.

---

**NOTE:** In this build, control is done based on the actual rotor position; therefore, the motor can run at higher speeds if the commanded  $I_qRef$  is higher and there is no load on the motor. TI advises to either add some mechanical load on the motor before the test or apply lower values of  $I_qRef$ . When the motor is commanded to run, it is subjected to an initial alignment stage where the electrical angle and the QEP angle count are set to zero. They are then forced to run based on an enforced angle until the QEP index pulse is received. Then the motor runs in full self-control mode based on its own angular position.

---

Open `fcl_qep_f2837x_tmdxiddk_settings.h` and select the level 3 incremental build option by setting `BUILDLEVEL` to `FCL_LEVEL3` (`#define BUILDLEVEL FCL_LEVEL3`). The current loop regulator can be selected to be the PI controller or the complex controller by setting `FCL_CNTLR` to `PI_CNTLR` or `CMPLX_CNTLR`. The `CURRENT_SENSE` method chosen for this library is `LEM_CURRENT_SENSE`. The current and position feedbacks can be sampled once or twice per PWM period, depending on the sampling method. The sampling is synchronized to the carrier maximum in the single sampling method, and to the carrier maximum and carrier zero in the double sampling method. This sample method selection is done in the example by selecting `SAMPLING_METHOD` to `SINGLE_SAMPLING` or `SAMPLING_METHOD` to `DOUBLE_SAMPLING`. The maximum modulation index changes from 0.98 in the `SINGLE_SAMPLING` method to 0.96 in the `DOUBLE_SAMPLING` method. If the `PWM_FREQUENCY` is changed from 10 kHz, the maximum modulation index also changes.

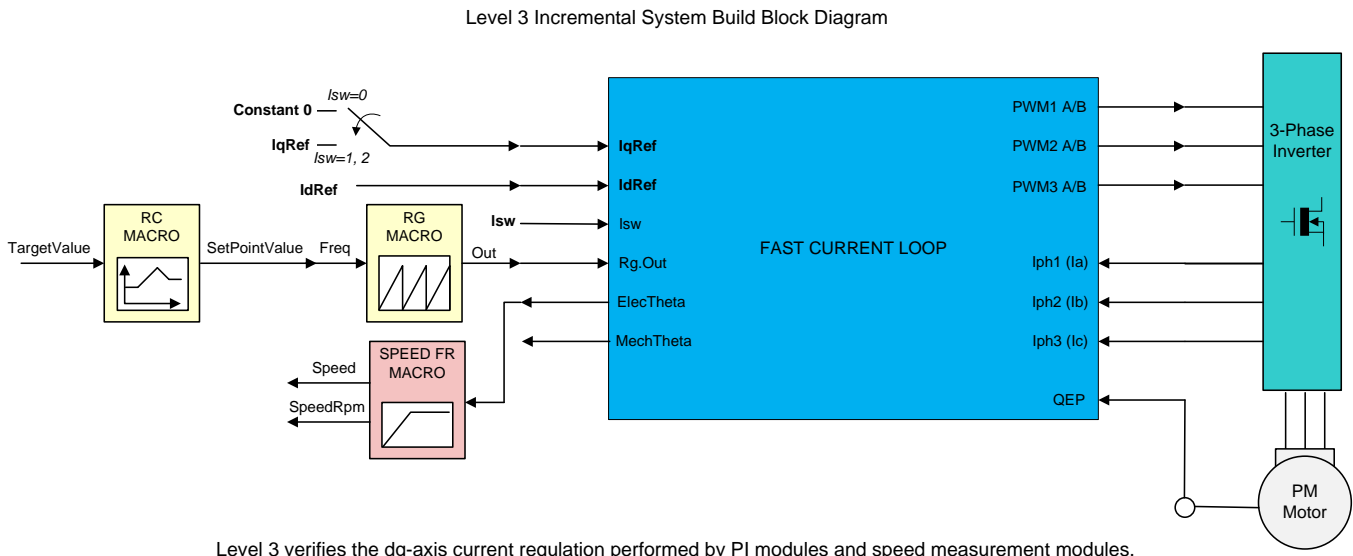
Right-click on the project name, and then click *Rebuild Project*. When the build is complete, click the *Debug* button, reset the CPU, restart, enable real-time mode, and run.

In the software, the key variables to add, adjust, or monitor are summarized as follows:

- `maxModIndex` : maximum modulation index
- `IdRef` : changes the d-axis voltage in per-unit
- `IqRef` : changes the q-axis voltage in per-unit
- `FCL_params.WccD` : preferred bandwidth of d-axis current loop
- `FCL_params.WccQ` : preferred bandwidth of q-axis current loop
- `fclLatencyInMicroSec` : shows latency between ADC and QEP sampling, and PWM in  $\mu$ s
- `fclClrCntr` : flag to clear the variable `fclLatencyInMicroSec` and let it refresh
- `runMotor` : flag to RUN or STOP the motor



Figure 10 shows the level 3 block diagram.



**Figure 10. Level 3 Block Diagram Showing Inner Most Loop - FCL**

The key steps are explained as follows:

1. Set *enableFlag* to 1 in the watch window. The *isrTicker* variable is incrementally increased, as seen in watch windows to confirm the interrupt is working properly.
2. Verify if the *maxModIndex* value is either 0.96 in double-sampling method or 0.98 in single-sampling method.
3. Set *speedRef* to 0.3 pu (or another suitable value if the base speed is different), *IdRef* to zero, and *IqRef* to 0.03 pu (or another suitable value). *speedRef* helps only until the QEP index pulse is received. Thereafter, the motor is controlled based on its rotor position. The soft-switch variable (*Isw*) is autopromoted in a sequence inside the FCL library. Here, *Isw* manages the loop setting as follows:
  - a. *Isw* = QEP\_ALIGNMENT --> lock the rotor of the motor
  - b. *Isw* = QEP\_WAIT\_FOR\_INDEX --> motor in run mode and waiting for the first instance of QEP index pulse
  - c. *Isw* = QEP\_GOT\_INDEX --> motor in run mode - QEP index pulse occurred
4. Gradually increase voltage at variac / DC power supply to, for example, 20% of the rated voltage.
5. Set *runMotor* flag to MOTOR\_RUN to run the motor.
6. Check *pi\_id.fbk* in the watch windows with the continuous refresh feature and see if it can track *IdRef*.
7. Check *pi\_iq.fbk* in the watch windows with the continuous refresh feature and see if it can track *IqRef*.
8. To confirm these two current regulator modules, try different values of *pi\_id.ref* and *pi\_iq.ref*, by changing the values of *IdRef* and *IqRef*, respectively.
9. Try different bandwidths for the current loop by tweaking the values of *FCL\_params.wccD* and *FCL\_params.wccQ*. The default setting for the bandwidth is 1/18 of the sampling frequency.
10. If the motor shaft can be held tight, then the *IqRef* value can be changed back and forth from 0.5 to -0.5, to study the effect of loop bandwidth.
11. Bring the system to a safe stop by reducing the bus voltage, taking the controller out of real-time mode, and resetting. Now the motor stops.

## 8.1 Observation One – Latency

### 8.1.1 From the Expressions Window

While running the motor in this build level and subsequent build levels, observe the variable *fclLatencyInMicroSec* in the Expressions window.

Figure 11 shows a snapshot of the Expressions window.

fclLatencyInMicroSec	float	0.959999979	0x0000B032@Data
FCL_params.wccD/(2*3.14)	double	555.837371	
FCL_params.wccQ/(2*3.14)	double	555.837371	

**Figure 11. Expressions Window Snapshot For Latency**

This variable indicates the amount of time elapsed between the feedback sampling and PWM updating. The elapsed time, or latency, is computed based on the count of the EPWM timer right after the PWM update. The value shown here is more than the actual update time by a few clock cycles. Immediately after setting the *runMotor* flag to *MOTOR\_RUN* and the motor begins to run, the latency time shows up as nearly 1.25  $\mu$ S due to initial setup in the code. This amount of latency occurs at a time when the duty cycle is moderate and is therefore acceptable. After this period, you can refresh the latency time by setting *fclClrCntr* to 1. Regardless of *SAMPLING\_METHOD*, latency remains the same for a given *FCL\_CNTLRLR*. When *FCL\_CNTLRLR* is a *PI\_CNTLRLR*, the latency is about 0.96  $\mu$ s compared to 0.98  $\mu$ s with a *CMPLX\_CNTLRLR* (see the following note).

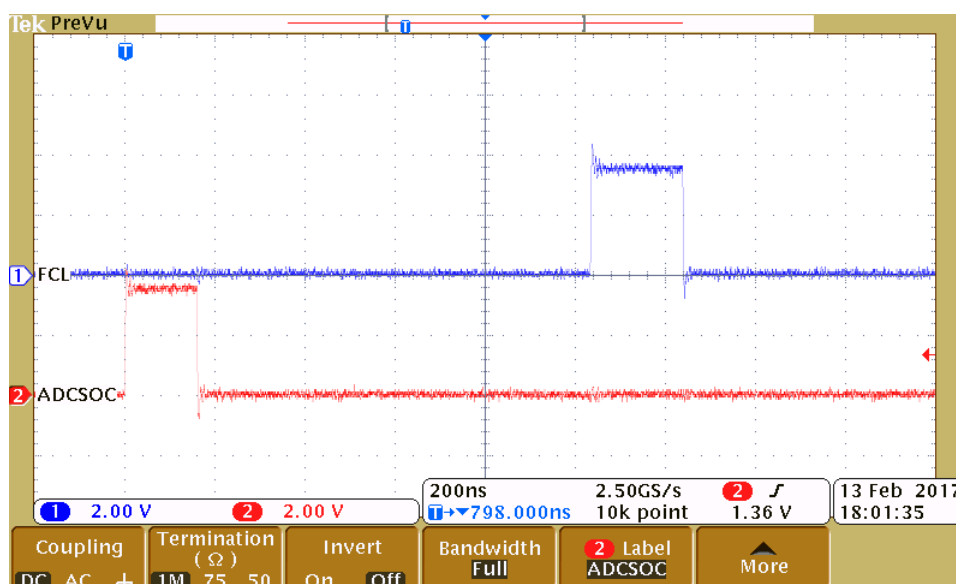
#### NOTE:

- These times can be reduced further by around 0.1- $\mu$ s range using **code inline** and other optimization techniques. Because the evaluation code is in library format, it has certain overheads.
- The **sampling window** for ADC is kept wide enough to ensure a cleaner signal acquisition. Depending on board layout and circuits feeding in to ADC channels, it may be possible to reduce this time window by nearly 60%.

## 8.1.2 From the Scope Plot

**NOTE:** Because H7 is not populated, GPIO16 and GPIO18 are used for timing purposes instead of for designed functional assignment. If H7 is to become populated, comment out the associated code and restore the functional assignment.

Figure 12 shows the latency discussed previously in the form of a scope plot, where the rising edge of the channel 2 and channel 1 waveforms signify the instances of the ADC SoC event and completion of all PWM updates, respectively. These events are brought out over GPIO16 and GPIO18, and may be probed over [Main]-R31 and [Main]-R33, respectively. The time seen in the scope plot may be slightly more than *fclLatencyInMicroSec*. This additional time is needed for the CPU to return from FCL library and to set GPIO .



**Figure 12. Scope Plot of ADCSoC and FCL Completion Events**

## 9 Incremental Build Level 4

Assuming the previous section is completed successfully, this section verifies the speed PI module and speed loop. When the motor is commanded to run, it is subjected to an initial alignment stage where the electrical angle and the QEP angle count are set to zero. After ensuring a stable alignment, the motor starts running.

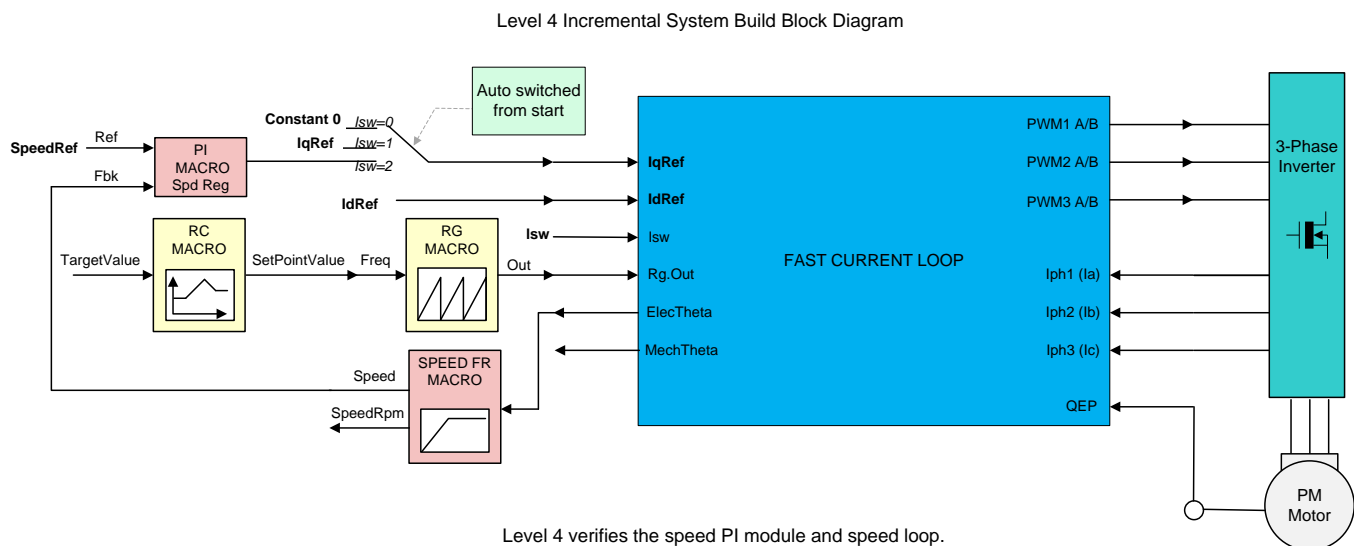
Open `fcl_qep_f2837x_tmdxiddk_settings.h` and select the level 4 incremental build option by setting the BUILDLEVEL to FCL\_LEVEL4 (`#define BUILDLEVEL FCL_LEVEL4`). The current loop regulator can be selected to be PI controller or complex controller by setting FCL\_CNTLR to PI\_CNTLR or CMLX\_CNTLR. The CURRENT\_SENSE method chosen for this library is LEM\_CURRENT\_SENSE.

Right-click on the project name, and then click *Rebuild Project*. When the build is complete, click the *Debug* button, reset the CPU, restart, enable real-time mode, and run.

In the software, the key variables to be adjusted are summarized as follows:

- *speedRef*: for changing the rotor speed in per-unit.
- *IdRef*: for changing the d-axis voltage in per-unit.
- *IqRef*: for changing the q-axis voltage in per-unit.

Figure 13 shows the implementation block diagram.

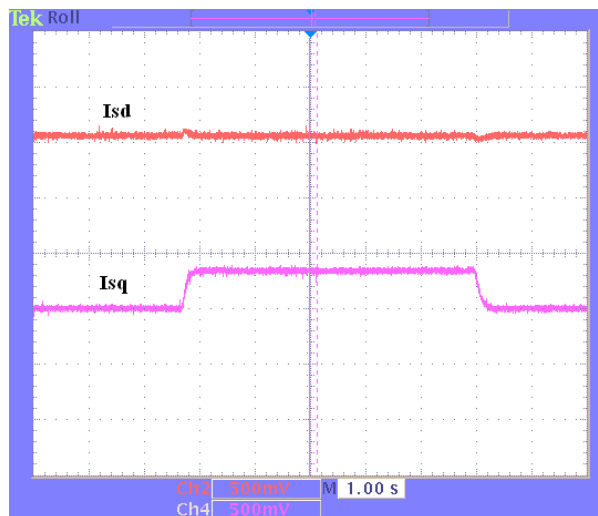


**Figure 13. Level 4 Block Diagram Showing Speed Loop With Inner FCL**

The key steps are explained as follows:

1. Set *enableFlag* to 1 in the watch window. The *isrTicker* variable is incrementally increased as seen in watch windows to confirm the interrupt is working properly.
2. Set *speedRef* to 0.3 pu (or another suitable value if the base speed is different).
3. Add *pid\_spd* variable to the Expressions window
4. Gradually increase voltage at variac to get an appropriate DC-bus voltage.
5. Add the switch variable *runMotor* to the watch window to start the motor. The soft-switch variable (*lsw*) is autopromoted in a sequence. In the code, *lsw* manages the loop setting as follows:
  - a. *lsw* = QEP\_ALIGNMENT --> lock the rotor of the motor
  - b. *lsw* = QEP\_WAIT\_FOR\_INDEX --> motor in run mode and waiting for the first instance of QEP index pulse
  - c. *lsw* = QEP\_GOT\_INDEX --> motor in run mode - QEP index pulse occurred
6. Set *runMotor* to MOTOR\_RUN; now the motor runs with this reference speed (0.3 pu). Compare the speed with *speedRef* in the watch windows with the continuous refresh feature to see whether or not it is nearly the same.
7. To confirm this speed PID module, try different values of *speedRef* (positive or negative). The P, I, and D gains may be tweaked to get a satisfactory response.
8. At a very low speed range, the performance of the speed response relies heavily on the good rotor position angle provided by the QEP encoder.
9. Bring the system to a safe stop by reducing the bus voltage, taking the controller out of real-time mode, and resetting. Now the motor stops.

Figure 14 shows flux and torque components of the stator current in the synchronous reference frame.



**Figure 14. Flux and Torque Components of the Stator Current in the Synchronous Reference Frame Under 0.33-pu Step-Load and 0.3-pu Speed**

## 9.1 Observation

In the default set up, the current loop bandwidth is set up as 1/18 of the SAMPLING FREQUENCY for both CMPLX\_CNTLR and PI\_CNTLR. As the bandwidth is increased, control becomes stiff and the motor operation becomes noisier as the controller reacts to tiny perturbations in the feedback trying to correct them, see the note below. The gain cross over frequency (or open loop bandwidth) can be taken up to 1/6 of the SAMPLING\_FREQUENCY and still get good transient response over the entire speed range including higher speeds. If the motor rotation direction is reversed occasionally due to any malfunction, try restarting it by setting *runMotor* to MOTOR\_STOP and then MOTOR\_RUN again. It may need some fine tuning in transitioning from *IsW* = QEP\_WAIT\_FOR\_INDEX to *IsW* = GOT\_INDEX. This can be used as an exercise to fix it.

---

**NOTE:** The fast current loop is a high bandwidth enabler. When the designed bandwidth is high, the loop gains can also be high. This pretty much ties the loop performance to the quality of current feedback. If the SNR of current feedback signal into the digital domain is poor, then the loop can be very audibly noisy as the controller tries to minimise the error. If the noise is bothersome, the user may be required to reduce the bandwidth to avoid the audible noise.

---

## 10 Incremental Build Level 5

This section verifies the position PI module and position loop with a QEP. For this loop to work properly, the speed loop must have been completed successfully. When the motor is commanded to run, it is subjected to an initial alignment stage where the electrical angle and the QEP angle count are set to zero. After ensuring a stable alignment, the motor starts to run.

Open *fcl\_qep\_f2837x\_tmdxiddk\_settings.h* and select the level 5 incremental build option by setting the BUILDLEVEL to FCL\_LEVEL5 (#define BUILDLEVEL FCL\_LEVEL5). The current loop regulator can be selected to be PI controller or complex controller by setting FCL\_CNTLR to PI\_CNTLR or CMPLX\_CNTLR. The CURRENT\_SENSE method chosen for this library is LEM\_CURRENT\_SENSE.

Right-click on the project name, and then click *Rebuild Project*. When the build is complete, click the *Debug* button, reset the CPU, restart, enable real-time mode, and run. Set *runMotor* to MOTOR\_RUN in the Expressions window. Setting this flag runs the motor through predefined motion profiles and position settings as set by the refPosGen() module. This module basically cycles the position reference through a set of values as defined in an array posArray. These values represent the number of the rotations and turns with respect to the initial alignment position. Once a certain position value as defined in the array is reached, it pauses for a while before slewing toward the next position in the array. Therefore, these array values can be referred to as parking positions. During transition from one parking position to the next, the rate of transition (or speed) is set by posSlewRate. The number of positions in posArray through which it passes before restarting from the first value is decided by ptrMax. Hence, add the variables posArray, ptrMax, and posSlewRate to the Expressions window.

The key steps are explained as follows:

1. Set *enableFlag* to 1 in the watch window. The variable named *isrTicker* is incrementally increased as seen in the watch windows to confirm the interrupt is working properly.
2. Add variables *pi\_pos*, *posArray*, *ptrMax*, and *posSlewRate* to the Expressions window.
3. Gradually increase voltage at variac to get an appropriate DC-bus voltage.
4. Set *runMotor* to MOTOR\_RUN to run the motor. The motor must be turning to follow the commanded position (see the following note if the motor does not turn properly).
5. The parking positions in *posArray* can be changed to different values to determine if the motor turns as many rotations as set.
6. The number of parking positions *ptrMax* can also be changed to set a rotation pattern.
7. The position slew rate can be changed using *posSlewRate*. This rate represents the angle (in pu) per sampling instant.
8. The proportional and integral gains of the speed and position PI controllers may be returned to get satisfactory responses. TI advises to first tune the speed loop and then the position loop.
9. Bring the system to a safe stop by reducing the bus voltage, taking the controller out of real-time mode and reset. Now the motor stops.

Figure 15 shows the implementation block diagram.

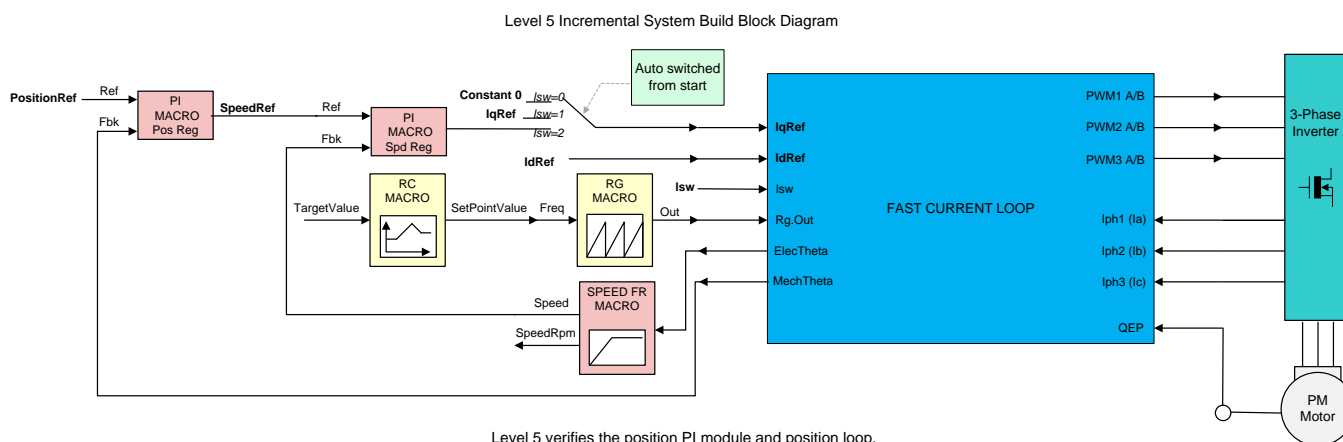


Figure 15. Level 5 Block Diagram Showing Position Loop With Inner FCL

In the scope plot shown in Figure 16, the position reference, and position feedback are plotted. They are aligned with negligible lag, which may be attributed to software. If the Kp and Ki gains of the position loop controller are not chosen properly, this may lead to oscillations in the feedback or a lagged response.

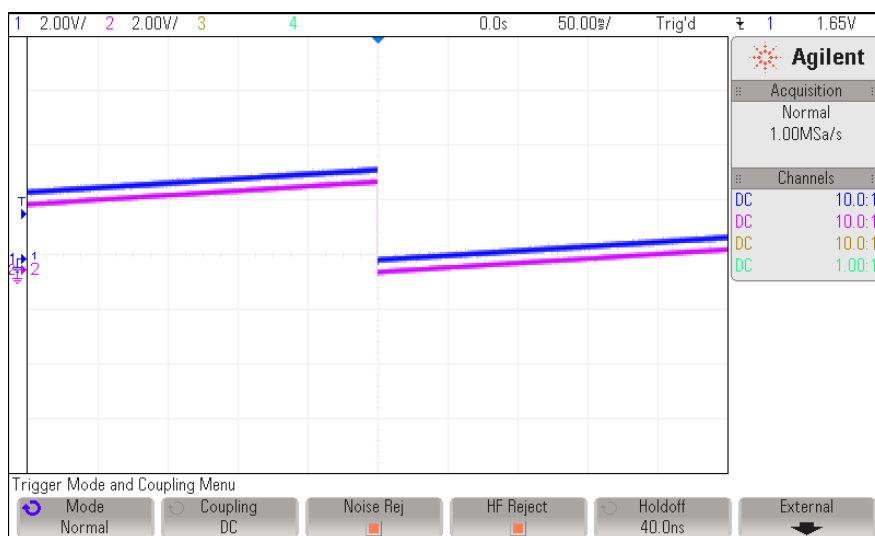


Figure 16. Scope Plot of Reference Position to Servo and Feedback Position

**NOTE:**

- If the motor response is erratic, then the sense of turn of the motor shaft and the encoder may be opposite. Swap any two phase connections to the motor and repeat the test.
- The position control implemented here is based on an initial aligned electrical position (= 0). If the motor has multiple pole pairs, then this alignment can leave the shaft in different mechanical positions depending on the prestart mechanical position of the rotor. If the mechanical position repeatability or consistency is needed, then the QEP index pulse must be used to set a reference point. This may be taken as an exercise.

## 11 Incremental Build Level 6

Assuming the previous build level is successfully completed, this build level attempts to study the frequency response analysis of the Fast Current Loop using C2000's Software Frequency Response Analyzer (SFRA) tool, available as a library in the DigitalPower SDK.

### 11.1 Integrating SFRA Library

The user guide ([C2000™ Software Frequency Response Analyzer \(SFRA\) Library and Compensation Designer in SDK Framework](#)) that describes the SFRA tool and guides the user in integrating it into the C2000 platform can be found at:

`C:\ti\c2000\C2000Ware_DigitalPower_SDK_<version>\libraries\sfra\Doc`

The embedded firmware is available as a library in the DigitalPower SDK at:

`C:\ti\c2000\C2000Ware_DigitalPower_SDK_<version>\libraries\sfra`

The SFRA GUIs are available as executable applications in the DigitalPower SDK at:

`C:\ti\c2000\C2000Ware_DigitalPower_SDK_<version>\libraries\sfra\gui`

Some example projects to understand SFRA are available at:

`c:\ti\c2000\C2000Ware_DigitalPower_SDK_<version>\libraries\sfra\examples`

In the ISR code, there are two functions that inject noise for SFRA and then collect the feedback data from the loop, they are:

- `injectSFRA()`
- `collectSFRA()`

The roles of these functions are self-explanatory from their names. They should be used in the sequence they are used in the code to collect the data in right sequence.

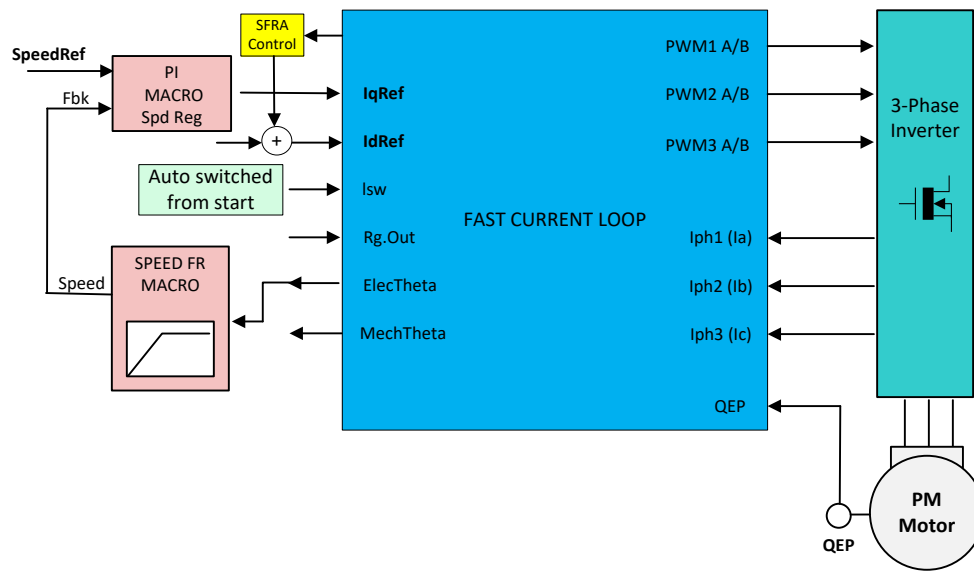
---

**NOTE:**

- The disturbances due to analog signal path and quantization will impact the loop performance and hinder high bandwidth selections that can be verified using SFRA results. Therefore it is important to provide a current feedback with a higher SNR.
  - When evaluating current loops, if it is possible to hold the motor speed constant, it will help to minimize the impact of speed jitter related errors in the SFRA results. This is particularly useful when studying the Iq loop. If the voltage decoupling of current loop is good, then this requirement may not matter.
- 

This tool provides the ability to study the D-axis or Q-axis current loops or the speed loop. The motor can be run at different speed / load conditions and at different bandwidths and the performance can be evaluated at each of these conditions. It can be seen that the controller can provide the designed bandwidth under all these conditions with a certain tolerance.





**Figure 17. Level 6 Block Diagram**

The implementation block diagram is given in [Figure 17](#). The SFRA tool injects noise signal into the system at various frequencies and analyzes the system response and provides a Bode Plot of the actual physical system as seen during the test.

## 11.2 Initial Setup Before Starting SFRA

The setting up involves co-ordination between the debug environment and the SFRA GUI. Until getting familiar with connecting the SFRA GUIs to the target platform, it is a good idea to turn off the high voltage power input to the target platform. Open '*fcl\_qep\_f2837x\_tmdxiddk\_settings.h*' and select level 6 incremental build option by setting the BUILDLEVEL to FCL\_LEVEL6 (#define BUILDLEVEL FCL\_LEVEL6).

Open '*fcl\_f2837x\_sfra\_settings.h*' and watch out for the definitions:

- SFRA\_FREQ\_START
- SFRA\_FREQ\_LENGTH
- FREQ\_STEP\_MULTIPLY

These definitions inform the GUI about the starting value of noise frequency, number of different noise frequencies to sweep and the ratio between successive sweep frequencies respectively. More information is available in the [C2000™ Software Frequency Response Analyzer \(SFRA\) Library and Compensation Designer User's Guide](#) associated with SFRA. In the context of this evaluation project, it is important to know and appreciate these parameters to tweak them for further repeat tests.

In this motor control project, SFRA can be performed on any of the three control loops such as the speed loop, D axis current loop and Q axis current loop. Technically, this could be performed on position loop as well, but is not included in this project scope and the user can take it as an experiment if desired.

Right click on the project name and click Rebuild Project. Once the build is complete click on debug button, reset CPU, restart, enable real time mode and run. From the debug environment, the steps to be followed are shown below:

Add the following variable in the 'Expressions Window':

- sfraTestLoop : for selecting the control loop on which to evaluate SFRA, letting you choose between:
  - SFRA\_TEST\_D\_AXIS - D axis current loop
  - SFRA\_TEST\_Q\_AXIS - Q axis current loop
  - SFRA\_TEST\_SPEEDLOOP - speed loop

The key steps can be explained as follows:

1. Set *enableFlag* to 1 in the watch window. The variable named *IsrTicker* will be incrementally increased as seen in watch windows to confirm the interrupt working properly.
2. The SCI initialisations needed to communicate with the GUIs should be all done by now.
3. Further steps with the debug window will follow after setting up the GUIs to connect to target platform.

### 11.3 SFRA GUIs

There are two GUIs available to perform the frequency response analysis, one (SFRA\_GUI) to plot open loop and plant Bode diagram, and another (SFRA\_GUI\_MC) to plot open loop and closed loop Bode diagram. They can be invoked and connected to the target platform to study the control loops. The GUI executables are available in the location as mentioned in [Section 11.1](#).

Double click on the choice of GUI executable and the GUI screen will appear as shown in [Figure 18](#) for SFRA\_GUI or in [Figure 19](#) for SFRA\_GUI\_MC. They look almost identical, the difference is in the pull down menu under 'FRA Settings' starting with the label 'Open Loop'.

- In the SFRA\_GUI, this pull down menu helps to select between Open Loop Model and Plant Model
- In the SFRA\_GUI\_MC, this pull down menu helps to select between Open Loop Model and Closed Loop Model

---

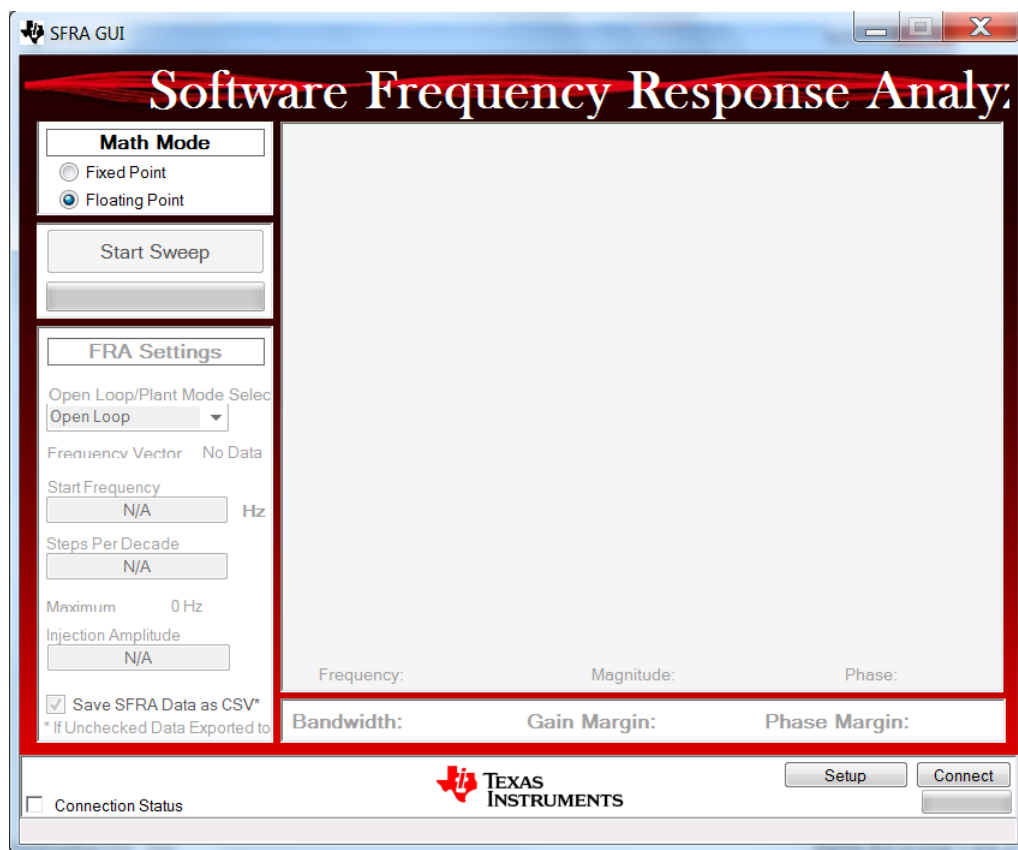
**NOTE:** The GUIs interpretation of bandwidth is different.

- In the SFRA\_GUI, bandwidth is defined as the open loop gain cross over frequency
  - In SFRA\_GUI\_MC, bandwidth is defined according to Chinese standard **GBT 16439-2009** and **NEMA ICS 16 (Speed Loop)** for servo drives. It defines bandwidth as the frequency where the closed loop output magnitude drops by 3dB or the phase shift lag exceed 90 degrees for the speed loop. It is suggestive that bandwidth is the frequency where the first of these two instances would occur. This approach is used to analyze current loops as well in this demo.
- 

This demo uses the GUI 'SFRA\_GUI\_MC'. However, the user is encouraged to experiment with the other GUI as well to study the plant. With the SFRA\_GUI, the user can plot the same graph as in SFRA\_GUI\_MC by passing the argument 'SFRA\_GUI\_PLOT\_GH\_CL' in the function *configureSFRA()* as shown in the code snippet below.

```
configureSFRA(SFRA_GUI_PLOT_GH_CL, SAMPLING_FREQ); // to plot GH and CL plots using SFRA_GUI
```

But the inferences from the plots are not according to that in SFRA\_GUI\_MC. Therefore, we advice to configure SFRA for 'SFRA\_GUI\_PLOT\_GH\_H' so that the user can see open loop, closed loop and plant model plots using these two GUIs, by using one at a time. For digital power applications, SFRA\_GUI\_MC inferences may not apply and so SFRA\_GUI can be used to display closed loop plots as well.



**Figure 18. SFRA GUI**

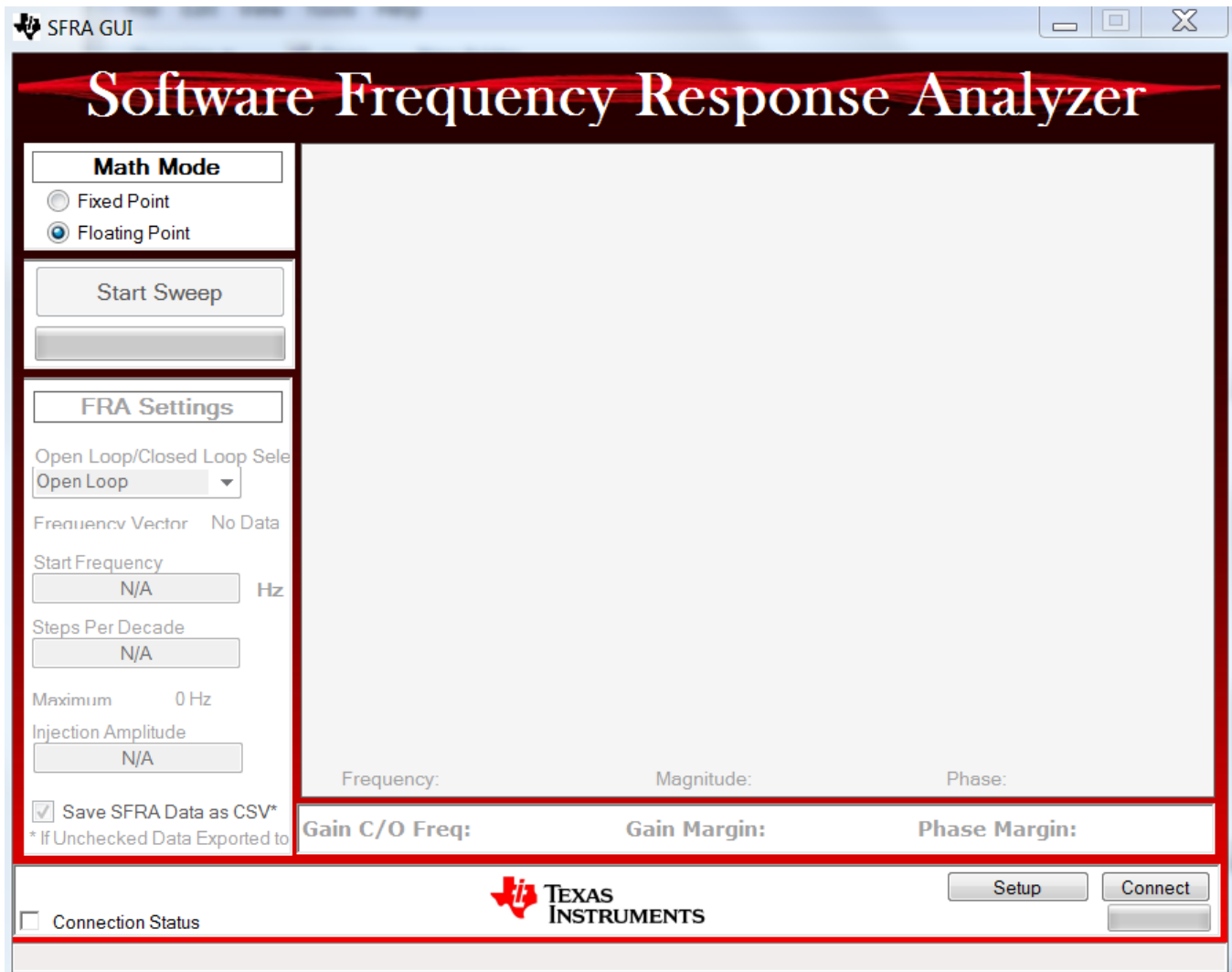


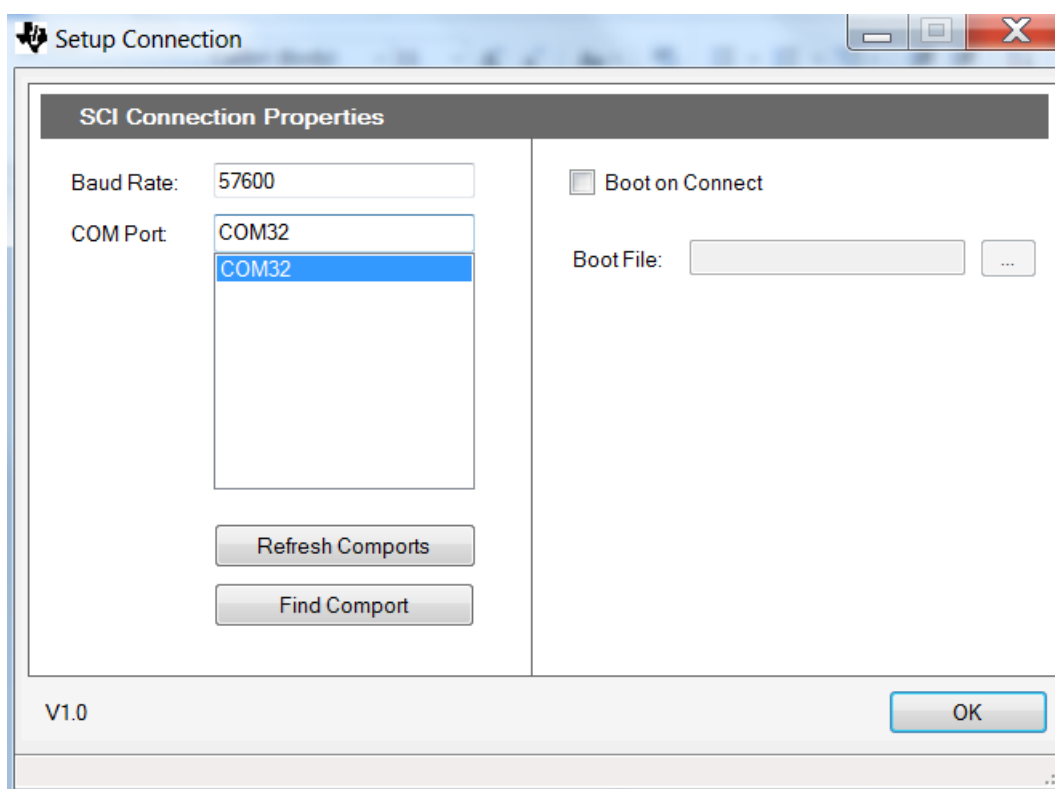
Figure 19. SFRA GUI MC

## 11.4 Setting Up the GUIs to Connect to Target Platform

Both GUIs have identical procedures to connect to the target platform. The GUI lets the user select appropriate settings based on the target platform and development computer.

The following is a list of things to do on the GUI before starting the analysis.

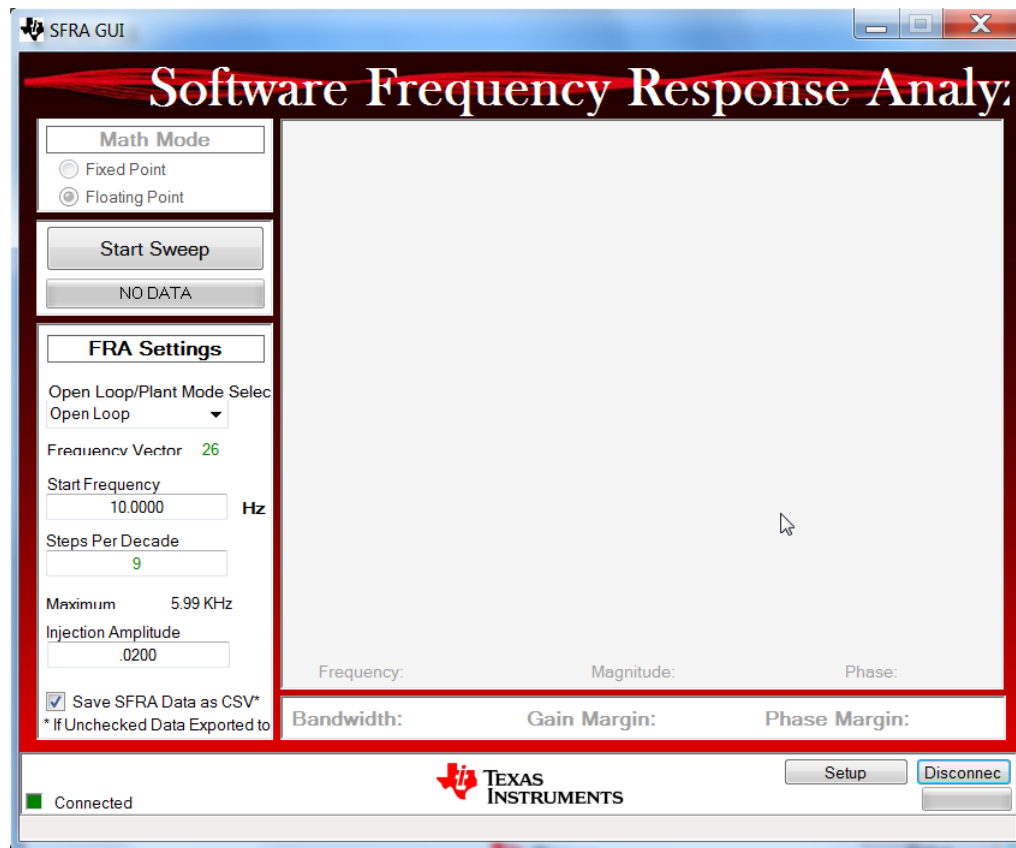
1. Math Mode: Depending on the target C2000 development platform, either the fixed point or floating point option is chosen. For F2837x or later C2000 MCUs, select the 'Floating Point' option
2. Since the USB port on the control card / launch pad is already connected to the computer for JTAG purposes, no additional connection is required. However, for a standalone operation, an USB connector needs to be connected to the target board. In the XDS100 emulator present on the control card / launch pad, in addition to a JTAG link, an SCI port link is also provided and the GUI uses this link to connect to the SCI port of the target platform. While the debug environment of CCS is using JTAG, the GUI can also use SCI at the same time.
3. Click on the Setup button at the bottom right corner. This will pop open a Setup Connection window as shown in [Figure 20](#).



**Figure 20. GUI Setup Diagram**

4. Click 'Refresh Comports' button to get the Comport number show up in the window.
5. Select the Comport representing the connection to the target C2000 board.
6. Uncheck 'Boot on Connect'
7. Click OK button

8. This should establish the connection to the MCU control card/launchpad and the GUI will appear as shown in [Figure 21](#) indicating the connection status at the bottom left corner.



**Figure 21. SFRA GUI Connected to the C2000 MCU**

9. The frequency sweep related settings are shown in 'FRA Settings' Panel. These values are already pre-filled from the C2000 device and they can be left as is.
10. As mentioned earlier, the visual difference between the two GUIs is in the pull down menu under 'FRA Settings' starting with Open Loop.
  - a. In the SFRA\_GUI, this pull down menu helps to select between Open Loop Model and Plant Model
  - b. In the SFRA\_GUI\_MC, this pull down menu helps to select between Open Loop Model and Closed Loop Model
  - c. This menu becomes relevant after a complete noise injection sweep of the system at various frequencies. Then the user can pick and view the plot of choice using this menu.
  - d. Bandwidth reporting is different as mentioned earlier. Gain cross over frequency is reported in the open loop plot of the SFRA\_GUI\_MC instead of bandwidth as in SFRA\_GUI.

This completes the initial setup of GUI environment.

## 11.5 Running the SFRA GUIs

If the high voltage (HV) power input to the target platform is turned off before, restore it back now. From the debug environment, the steps to be followed are as shown below:

1. Verify that *sfraTestLoop* is set to *SFRA\_TEST\_D\_AXIS* so as to test Id loop.
2. Set *FCL\_params.wccD* to the desired value, within limits, (when test is performed for Q axis, adjust this parameter for Q axis - *FCL\_params.wccQ*)
3. Set *speedRef* = 0.05 (in pu, 1 pu = 250Hz) and then set *runMotor* = *MOTOR\_RUN* to run the motor. Now motor shaft should start spinning and settle at the commanded speed.
4. The state machine variable (*lsw*) is autopromoted in a sequence, its states are as follows:
  - a. *lsw* = *QEP\_ALIGNMENT* --> lock the rotor of the motor
  - b. *lsw* = *QEP\_WAIT\_FOR\_INDEX* --> motor in run mode and waiting for the first instance of QEP index pulse
  - c. *lsw* = *QEP\_GOT\_INDEX* --> motor in run mode - QEP index pulse occurred
5. Now the GUI can be called into perform a frequency sweep of the D axis current loop by clicking on the 'Start Sweep' button in the GUI. The sweep progress will be indicated by a green bar in the location marked as 'NO DATA'.
6. When the frequency sweep is fully done, it will compute the Bode plot and display the results as shown in Figure 22 and \*Figure 23.
7. The GUI also computes and displays the loop bandwidth, gain margin and phase margin.
8. Repeat the test, if desired, by changing *FCL\_params.wccD*, and at different speed and load conditions.
9. To disconnect the GUI, click the 'Disconnect' button on the GUI.
10. To stop the motor, reduce the HV dc input voltage and set *runMotor* to *MOTOR\_STOP*.
11. After the motor stops, take the controller out of real-time mode and reset.

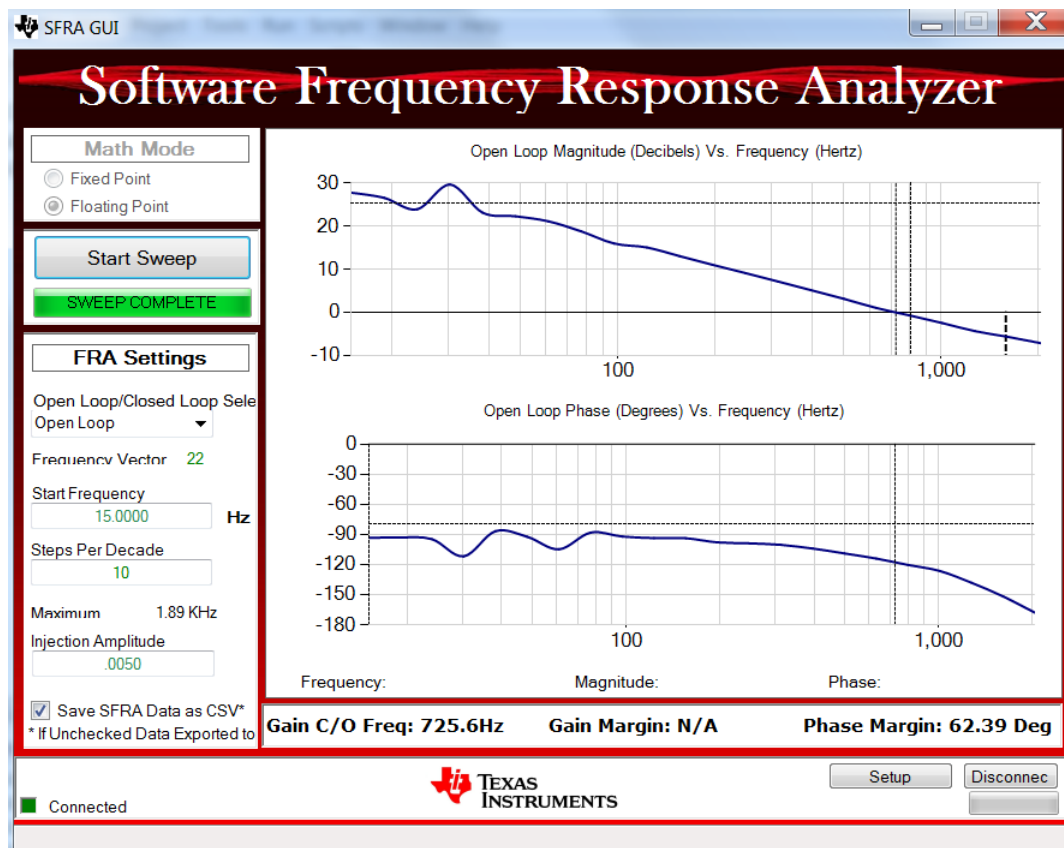


Figure 22. SFRA Open Loop Bode Plots of the Current Loop Showing Magnitude and Phase Angle

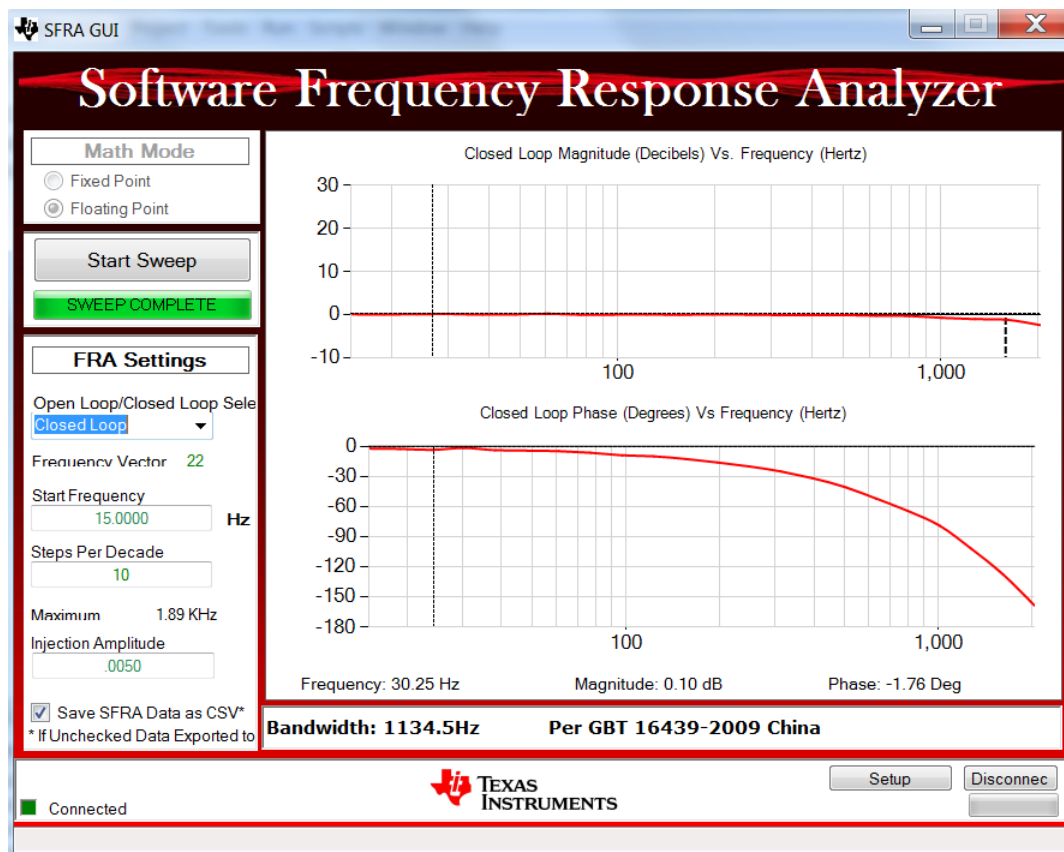


Figure 23. SFRA Closed Loop Bode Plots of the Current Loop Showing Magnitude and Phase Angle



## 11.6 Influence of Current Feedback SNR

The fast current loop is a high bandwidth enabler. When the designed bandwidth is high, the loop gains can also be high. This pretty much ties the loop performance to the quality of current feedback. If the SNR of current feedback signal into the digital domain is poor, then the loop can be very audibly noisy as the controller tries to minimise the error. If the noise is bothersome, the user may be required to reduce the bandwidth to avoid the audible noise. Therefore, for a higher bandwidth and higher performance, the feedbacks should be of high SNR to get the frequency responses as shown in Figure 24 and Figure 25.

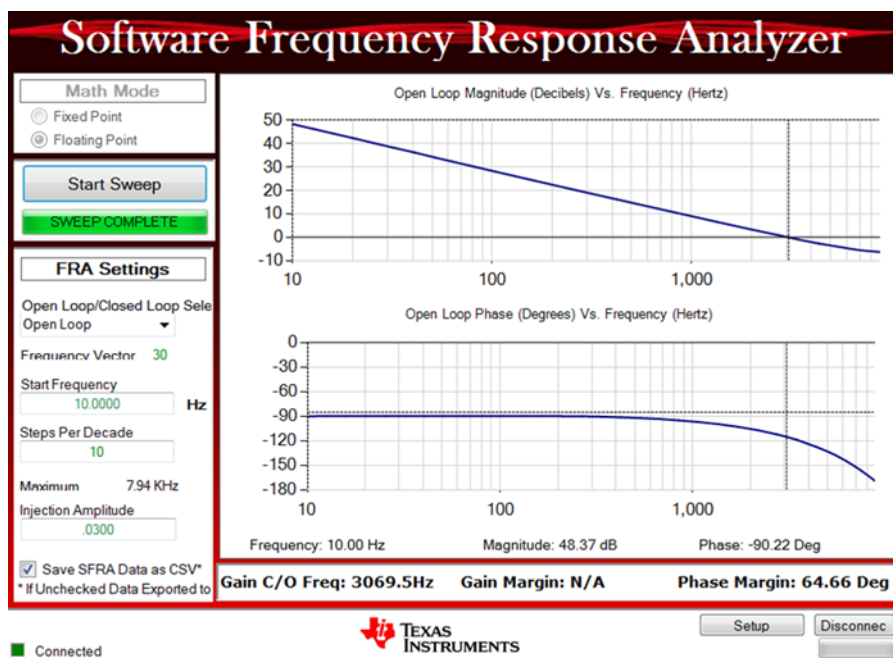


Figure 24. SFRA Open Loop Bode Plots of the Current Loop - Current Feedback With High SNR

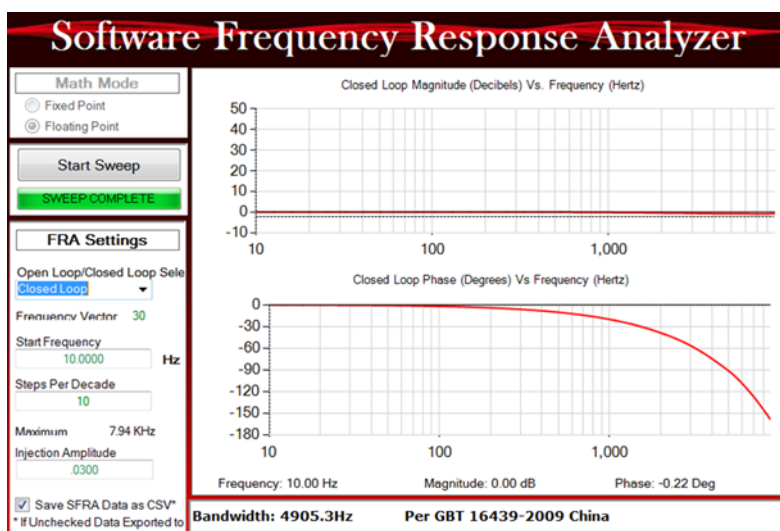


Figure 25. SFRA Closed Loop Bode Plots of the Current Loop - Current Feedback With High SNR

**NOTE:** With the hardware platform IDDK, there is scope for improving the SNR of the current signal feeding into the ADCs of the MCU. Therefore, higher bandwidth tests can be more noisier (chattering in nature) on this platform.

## 11.7 Inferences

### 11.7.1 Bandwidth Determination From Closed Loop Plot

The controller implemented for the open loop and closed loop plots shown in [Figure 24](#) and [Figure 25](#) is a dead beat controller where the output catches up to the input in just one sample cycle without any overshoots or requiring multiple cycles. From the closed loop plot, it is clear that the closed loop gain is always 0dB (unity gain) at all frequencies and therefore, magnitude based bandwidth determination is not practical. Hence, the phase plot is chosen as reference, and the frequency at which the phase lag goes beyond 90 deg is taken as bandwidth per the Chinese standard GBT 16439-2009 or NEMA ICS 16 (speed loop). In this test case, the PWM frequency is chosen as 10 KHz and the sampling frequency is 20 KHz and the current loop bandwidth obtained from the closed loop plot is about 5000 Hz per these guidelines.

### 11.7.2 Phase Margin Determination From Open Loop Plot

From the open loop plots, the phase margin obtained is about 65 degrees. Such a high margin should give a very robust performance across the frequency ranges within the bandwidth obtained.

### 11.7.3 Maximum Modulation Index Determination From PWM Update Time

From [Section 8.1](#), the time lapse between feedback sampling instantiation to PWM update is about 0.9  $\mu$ s. In this system where the PWM frequency is 10 KHz, the maximum modulation index is limited by the sampling method as follows

- Double sampling - just above 96%
- Single sampling - just about 98%

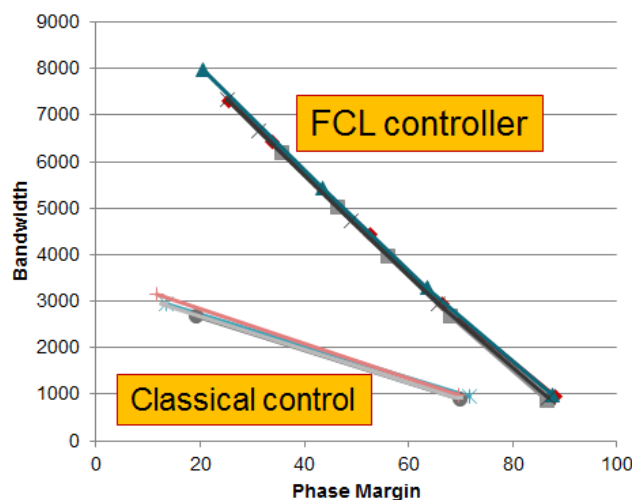
This is quite comparable to FPGA based systems where the entire algorithm is implemented in hardware.

### 11.7.4 Voltage Decoupling in Current Loop

The SFRA test can be performed at zero speed (by rigidly locking the shaft if needed) to get one plot as reference. The gain crossover frequency and phase margin at zero speed may be noted down. Then at different speeds and load conditions, this test can be repeated to verify if there is any change in bandwidth or phase margin. Any variation in the plot at different speed is indicative of the quality of decoupling in current loops.

## 11.8 Phase Margin vs Gain Crossover Frequency

By varying the control bandwidth and repeating these tests and noting down the resulting gain cross over frequency and phase margin, a set of plots are obtained for Id loop as shown in Figure 26. Two sets of tests are performed, one based on classical current control method, and the other based on FCL. Both these tests were performed using different current regulators. They all gave converging results.



**Figure 26. Plot of Gain Cross over Frequency vs Phase Margin as Experimentally Obtained**

The group of plots at the bottom is obtained for conventional control, and it is obvious that the gain cross over frequency is too low and that as the gain cross over frequency is increased, the phase margin drops a lot faster.

The group of plots at the top is obtained with FCL. The gain cross over frequency is nearly thrice that of the classical method for a given phase margin. And also, as the gain cross over frequency is increased, the relative drop in phase margin is very low compared to the classical method. This effectively means that FCL can provide a higher bandwidth or gain cross over frequency at a higher phase margin.

## 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 ([www.ti.com/legal/termsofsale.html](http://www.ti.com/legal/termsofsale.html)) or other applicable terms available either on [ti.com](http://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 © 2019, Texas Instruments Incorporated