TI Spins Motors

InstaSPIN Projects and Labs User's Guide

InstaSPIN-FOC for F28004xC

Version 1.00.00.00 *Motor Control Solutions*

Product Overview

InstaSPIN-FOC is a sensorless FOC solution that identifies, tunes, and controls your motor in minutes. The solution features:

- The FAST unified software observer, which exploits the similarities between all motors that use magnetic flux for energy transduction. The FAST estimator measures rotor flux (magnitude and angle) in a sensorless FOC system.
- Automatic torque (current) loop tuning with an option for user adjustments
- Automatic or manual field weakening and field boosting
- Superior robustness and high torque output for low speed PMSM/BLDC motor drive

Additional information about the features and functionality of InstaSPIN-FOC can be found in the Technical Reference Manuals and User's Guide.

Lab Projects Overview

The example projects (labs) described in this section are intended for you to not only experiment with InstaSPIN but to also use as a reference for your design. InstaSPIN-FOC motor control solutions, as well as the lab projects, are delivered within the MotorControl SDK.

In the lab projects, you will learn how to modify “user.h,” the header file that stores all of the user parameters. Some of these parameters can be manipulated through the GUI or CCS during run-time, but the parameters must be updated in “user.h” to be saved permanently in your project.

The following table summarizes all the projects available and which projects apply to each and target device.

Solution	**Name**	**Hvkit w/o PGA**	**Boost8320 w/ PGA**	**Brief Description**
FOC	is01	✓	✓	HAL, Inverter setup and LED Blinking
FOC	is02	✓	✓	Offset/Gain calibration with CPU
FOC	is03	✓	✓	Scalar control for hardware integrity verification with CPU
FOC	is04	✓	✓	Closed current loop without position angle for signal chain verification
FOC	is05	✓	✓	Motor ID with CPU
FOC	is06	✓	✓	Torque mode and tuning Id/Iq PI
FOC	is07	✓	✓	Speed mode and tuning speed PI
FOC	is08	✓	✓	Space Vector Over-Modulation
FOC	is09	✓	✓	Flying Start
FOC	is10	✓	✓	Rs Online Recalibration
FOC	is11		✓	Dual Motor Control
FOC	is12	✓	✓	Online Variable Switching Frequencies
FOC	is13	✓	✓	Field-weakening and Maximum Torque Per Ampere Control for IPM motor

Version: 1.00.00.00

Revision History:

1.00.00.00	**March 2018**	**MotorControl SDK 1.00 release, support for CPU and PMSM only** **InstaSPIN-FOC Labs added in this release: 01-13**

Contents

Lab Descriptions

is01_intro_hal – CPU and Inverter Setup

Lab 01 covers how to use the HAL object to setup the F28004xC and inverter hardware. MotorControl SDK API function calls will be used to simplify the microprocessor setup.

is02_offset_gain_cal – Offset and Gain Calibration

Lab 02 demonstrates current and voltage offset calculation. Additionally, skipping auto-calibration to reduce start-up time is demonstrated. Lab 02 also introduces the option to bypass the Rs Fine Re-estimation.

is03_hardware_test – Open Loop Control for Hardware Integrity Verification

Lab 03 implements a scalar volts/frequency control to test the integrity of the hardware, namely the PWM and ADC modules for hardware abstraction layer (HAL) setup. While compatible with Texas Instruments' hardware, this lab is intended for custom hardware verification.

is04_signal_chain_test – Current Closed Loop Control for Signal Chain Integrity Verification

Lab 04 implements a volts/frequency closed current loop control to test the signal chain integrity, mainly the hardware current/voltage sensing circuitry and controller ADC module. While compatible with Texas Instruments' hardware, this lab is intended for custom hardware verification.

is05_motor_id – Motor Parameters Identification

InstaSPIN does not have to be executed completely out of ROM. Actually, most of the InstaSPIN code is provided as open source. The only closed source code is the FAST observer. This lab will show how to run the sensorless field oriented controller as open source in user RAM. The only function calls to ROM will be to update and to pull information from the FAST observer.

is06_torque_control – Torque Control Mode and Tuning Id/Iq PI Controller

For the current loop PI controllers, InstaSPIN calculates the starting Kp and Ki gains for both Id and Iq controllers. During start-up, InstaSPIN identifies the time constant of the motor to determine the Ki and Kp parameters. The Id and Iq controllers' Kp and Ki gains may need to be manually adjusted for an optimal setting. Lab 06 demonstrates manually adjusting the current PI controller provided in the InstaSPIN software.

is07_speed_control – Speed Control Mode and Tuning Speed PI Controller

InstaSPIN-FOC provides a standard PI speed controller. The InstaSPIN library will give a "rule of thumb" estimation of Kp and Ki for the speed controller based on the maximum current setting in user.h. The estimated PI controller gains are a good starting point, but to obtain better dynamic performance, the Kp and Ki terms need to be tuned based on the whole mechanical system that the motor is running. This lab will show how to adjust the Kp and Ki terms in the PI speed controller.

is08_overmodulation – Space Vector Over-Modulation

The SVM that is used by InstaSPIN is capable of saturating to a pre-specified duty cycle. When using a duty cycle over 100.0%, the SVM is considered to be in the over-modulation region. When in the over-modulation region, low-side inverter leg current shunt measurement windows become small or even disappear. This lab will show how to re-create the currents that cannot be measured due to high duty cycles during SVM over-modulation.

is09_flying_start – Using Flying Start (Motor rotor is already moving)

The lab uses the flying start function to track and control an already rotating motor and resume normal operation with a minimal impact on load or speed.

is10_rs_recalc – Using Rs Online Recalibration

With the motor under heavy load, Rs Online Recalibration is required to maintain the performance of FAST. This lab will explore using this feature.

is11_dual_motor – Dual Motor Sensorless Velocity Control

Sensorless InstaSPIN-FOC is implemented to control two inverters independently with one C2000 MCU. Currently only supports LaunchXL-F280049C + BoostXL-DRV8320RS.

is12_variable_pwm_frequency – Online Variable Switching Frequencies

The lab applies online variable switching frequency feature in InstaSPIN-FOC that allows the drive to change FETs switching frequency to optimize the motor drive efficiency without changing any more control parameters.

is13_fwc_mtpa – Field Weakening and Maximum Torque per Ampere Control

This lab implements the Field Weakening (FW) and Maximum Torque Per Ampere (MTPA) controlled three-phase Interior Permanent Magnet Synchronous Motor (IPMSM) over a wide speed range in an instaSPIN-FOC project. The operation mode will automatically change from constant torque region with MTPA control to constant power region with FW control based on the speed command and the input DC-link voltage.

is01_intro_hal – CPU and Inverter Setup

Abstract

This document covers how to use the HAL object to set up F28004xC hardware to control a motor drive inverter, in which MotorControl SDK API function calls will be used to simplify the microprocessor setup.

Introduction

The “Hardware Abstraction Layer” (HAL) objects are adopted and defined in the MotorControl SDK software package to configure the microcontroller peripherals to interface motor drive inverter. The first lab is an introduction to using HAL to set up F28004xC processor clock, GPIOs, watchdog, ePWM, ADC, and other peripherals. The definition of these HAL objects and the corresponding set of APIs can be found in the file hal_obj.h. All the following lab projects are built upon this lab, so it is highly recommended to perform this lab first before moving on.

Learning Objectives

- Use the HAL object to set up the F28004xC processor.
- Use the HAL object to setup and initialize the inverter.
- Use the enumerations to select settings for peripherals.

Background

The MotorControl SDK software package is used for this lab. The microcontroller peripheral setup is performed to interface the inverter by using HAL_obj associated APIs. Important commands to initialize the drive are listed in the flowchart of **Figure** , and a block diagram for “is01_intro_hal” lab is shown in **Figure 35**.

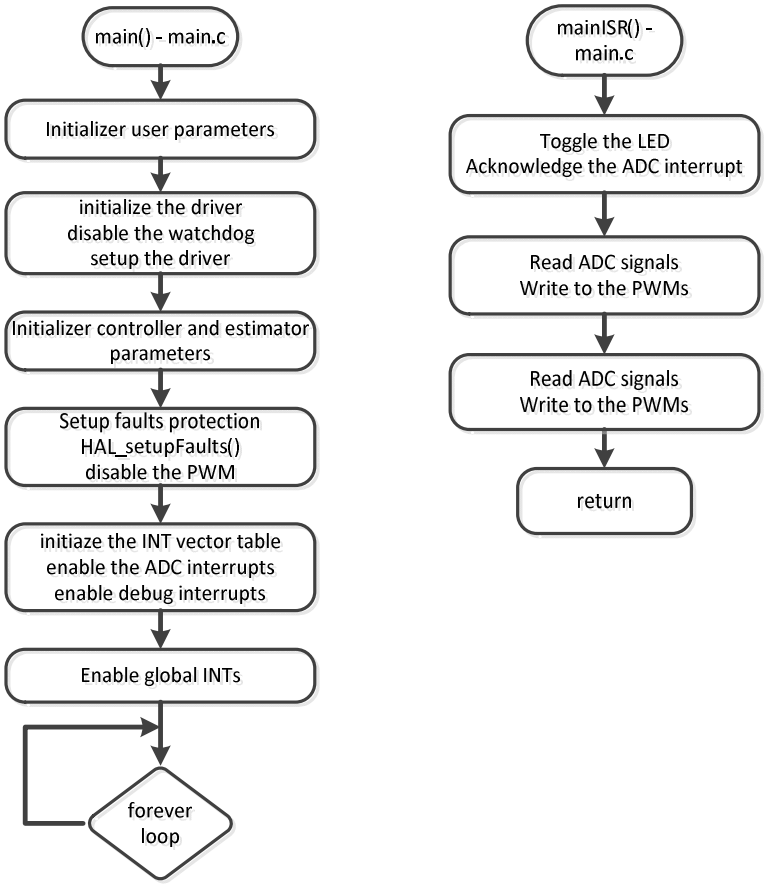


Figure 1: Project is01_intro_hal Software Flowchart

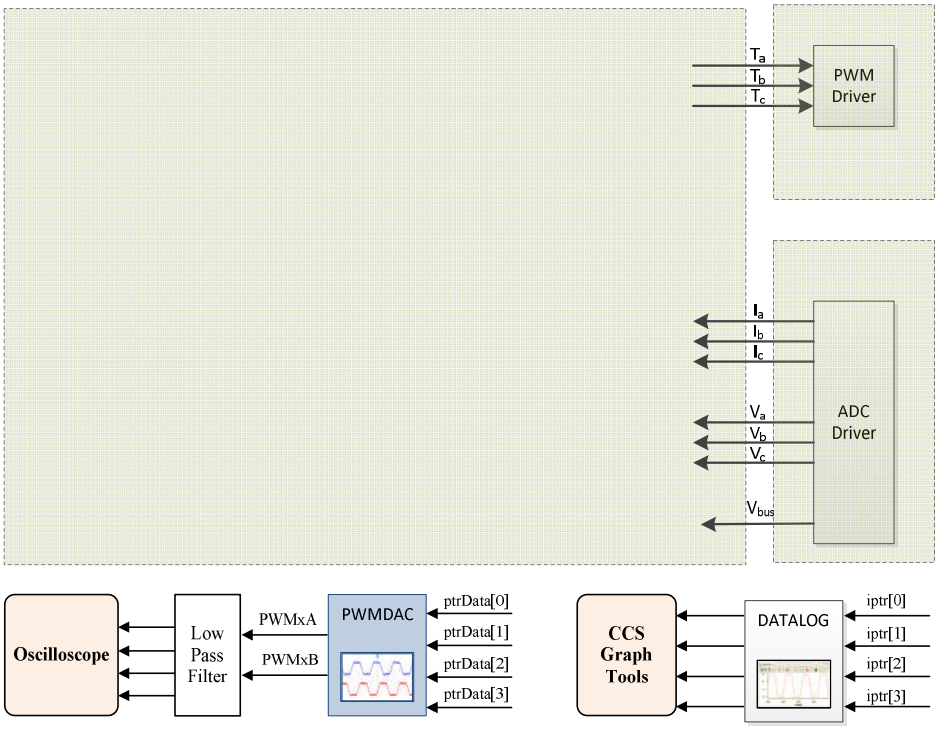


Figure 2: Block diagram of the "is01_intro_hal" lab

The files and project are located in the MotorControl SDK directory as shown in **Figure 3**, depending on which processor the user is working with.

For is01_intro_hal lab and all following InstaSPIN-FOC lab projects, the MotorControl SDK path refers to a directory referring to 28004xC devices, any of which include the appropriate ROM capability to run InstaSPIN-FOC projects. All projects are based on an inverter board (HVMTRINSPIN) paired with the TMDSCNCD280049C controlCARD, or an inverter board (BOOSTXL-DRV8320RS) paired with the LaunchPad-F280049C. The TMDSCNCD280049C controlCARD and LaunchPad-F280049C use the TMS320F280049C device.

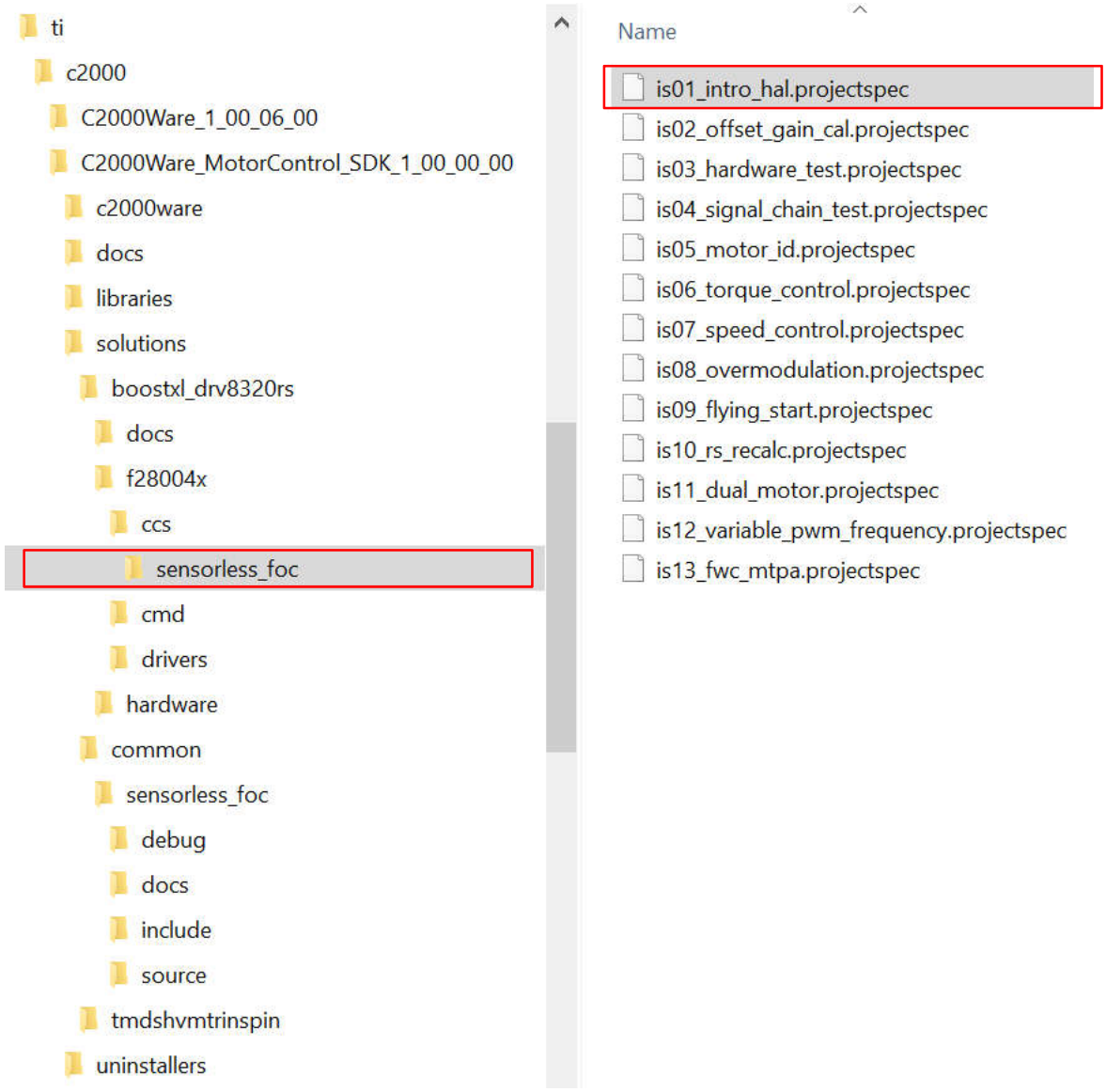


Figure 3: Project “is01_intro_hal” Location

The steps to open an InstaSPIN-FOC project from the MotorControl SDK directory are as follows:

1. From the CCS menu bar, select “Project → Import CCS projects.” The “Import CCS Eclipse Projects” window will open; click on “Browse” next to the “Select search-directory” box.
2. In the “Browse for Folder” window, navigate to the following folder for the project, is01_intro_hal: “\ti\c2000\C2000Ware_MotorControl_SDK_<version>\solutions\boostxl_drv8320rs\f28004x\ccs\sensorless_foc”, select “Ok.”
3. The resulting window displays a list of discovered projects within the f28004x\ folder. From this window, select the is01_intro_hal project as **Figure 4**, and then select “Finish”
4. The selected project(s) should now be viewable in the “CCS Project Explorer” window. You have now successfully imported an F28004x example project into CCS.

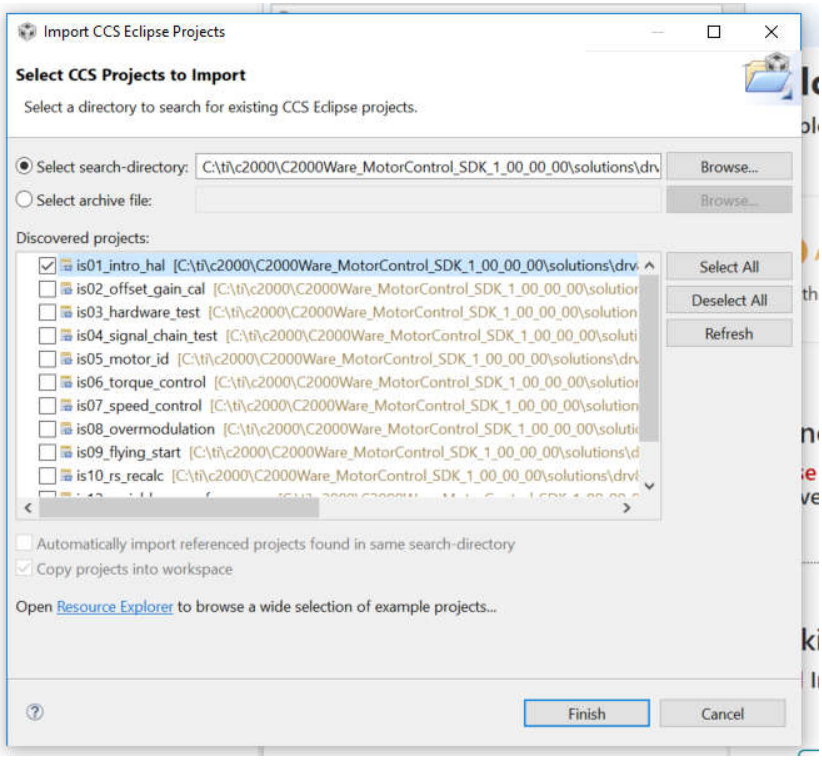


Figure 4: Import Project “is01_intro_hal” to CCS.

Package Structure

The MotorControl SDK software package is organized into the following directory structure as shown in **Table 1**:

Directory Name	Description
c2000ware	Contains the device-specific and core libraries.
docs	Contains the MotorControl SDK package user guides and a document HTML page.
libraries	Contains all application-specific support files.
solutions	Contains the MotorControl SDK application examples, board-specific hardware design guides and files, board-specific GUI resource files, and board-specific application support files.

Table 1: MotorControl SDK Package Structure

Project Files

CCS Project Explorer files for the is01_intro_hal lab are shown in **Figure 5**, and descriptions of these source files are shown in **Figure 6**. Additional source files for driver or FOC modules may be added into later projects.

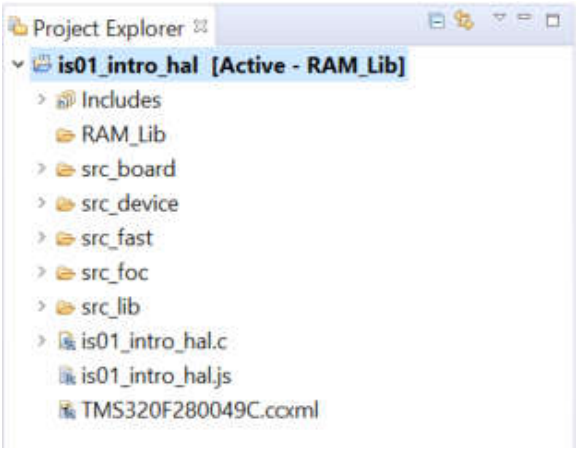


Figure 5: Project “is01_intro_hal” Files Explorer

RAM_Lib	The folder is created by CCS when active the related build configuration of the project.
src_board	
hal.c	Contains the various functions related to the HAL object
drv8320.c	Contains the various functions related to the DRV8320 object. (It's only applicable to the DRV8320RS BoosterPack)
src_device	
f28004x_globalvariabledefs.c	Define Global Peripheral Variables
f28004x_headers_nonbios.cmd	Linker Command File For F280049 Peripherals
f28004x_ram_cpu_is.cmd	Linker Command File For instaSPIN lab projects run in RAM
28004x_dcsm_lnk.cmd	Linker Command File For DCSM, only for Flash build configuration
f28004x_codestartbranch.asm	Branch for redirecting code execution after boot, only for Flash
f28004x_dcsm_z1otp.asm	Dual Code Security Module Zone 1 OTP, only for Flash
f28004x_dcsm_z2otp.asm	Dual Code Security Module Zone 2 OTP, only for Flash
f28004x_flash_cpu_is.cmd	Linker Command File For instaSPIN lab projects for Flash
src_fast	
ctrl.c	InstaSPIN control function
user.c	Function for setting initialization data for modules
src_foc	
clarke.c	Clarke transform library
datalog.c	Data logging module routines
filter_fo.c	first-order filter library
filter_so.c	second-order filter library
fwc.c	Field-weakening control library
ipark.c	inverse park transform library
mtpa.c	Maximum torque per ampere library
offset.c	InstaSPIN offset library
park.c	park transform library
pi.c	Proportional-Integral (PI) controller library
svgen.c	Space Vector Generator (SVGEN) library
traj.c	trajectory library
vsf.c	Online variable switching frequency library
src_lib	
f28004x_fast_rom_symbols_fpu32.lib	FAST estimator library
driverlib.lib	C2000Ware driver library for F28004x peripherals
is01_intro_hal.c	main control source files
is01_intro_hal.js	JavaScipt file for expressions in CCS
TMS320F280049C_LaunchPad.ccxml	Target configuration file for emulator on this hardware board

Figure 6: A description of source files in is01_intro_hal lab

Includes

A description of included header files for is01_intro_hal is shown in the table below. Note that "labs.h" is common across all labs; as a result, there are more header files #include directives that are needed for this lab as shown in **Table 2.**

labs.h	Header file containing all included files used in is01_intro_hal.c and following project main file	
	modules	
	math.h	Common math conversions, defines, and shifts
	est.h	Function definitions for the FAST ROM library
	platforms	
	hal.h	Device setup and peripheral drivers. Contains the HAL object.
	user.h	User file for configuration of the motor, drive, and system parameters

Table 2: Important header files needed for a lab setup

To view the contents of "labs.h," follow these steps:

1. Select the expand arrow for the file "is01_intro_hal.c" in the Project Explorer, allowing the header files included within this source file to be viewed
2. Right-mouse clicks on "labs.h;" selecting "Open" shows "is01_intro_hal.c" with the reference to "labs.h" highlighted, as shown in **Figure 7**.

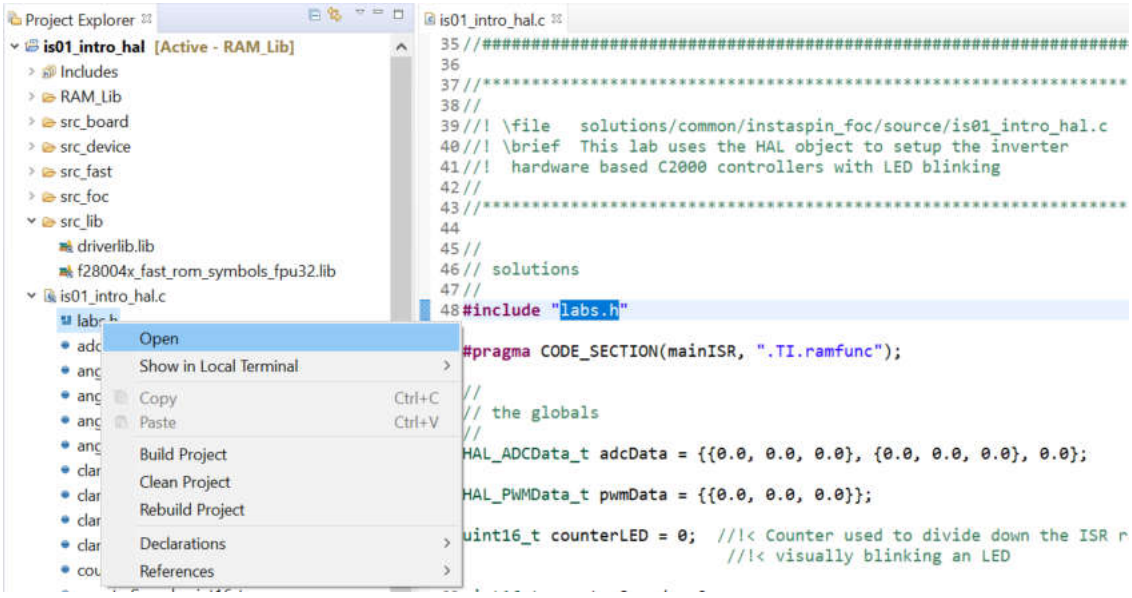


Figure 7: Opening a header file declaration from the Project Explorer tab

3. Right-mouse clicks on "labs.h" and select "Open Declaration" as shown in **Figure 8**.

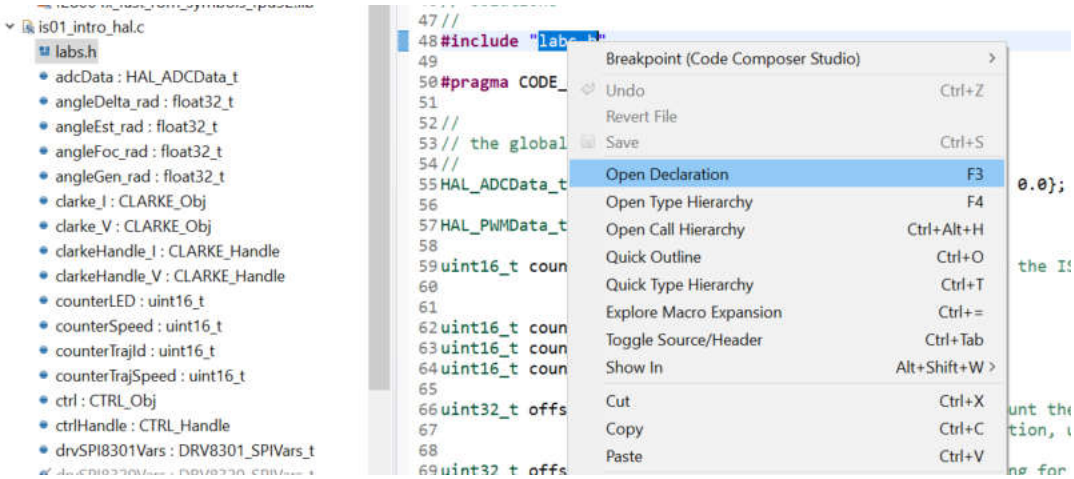


Figure 8: Opening a header file from its declaration

Notes: If the “labs.h” can be opened by this operation, right-mouse click “RAM_Lib” folder in the project, selecting “delete”, right-mouse click project “is01_intro_hal”, selecting “Clean Project”, and then right-mouse click project “is01_intro_hal” again, selecting “Rebuild Project”.

4. “labs.h” is now open for review. This header file includes additional header files that will be used in the project and global variable object definition for both this and future lab projects as shown in **Figure 9.**

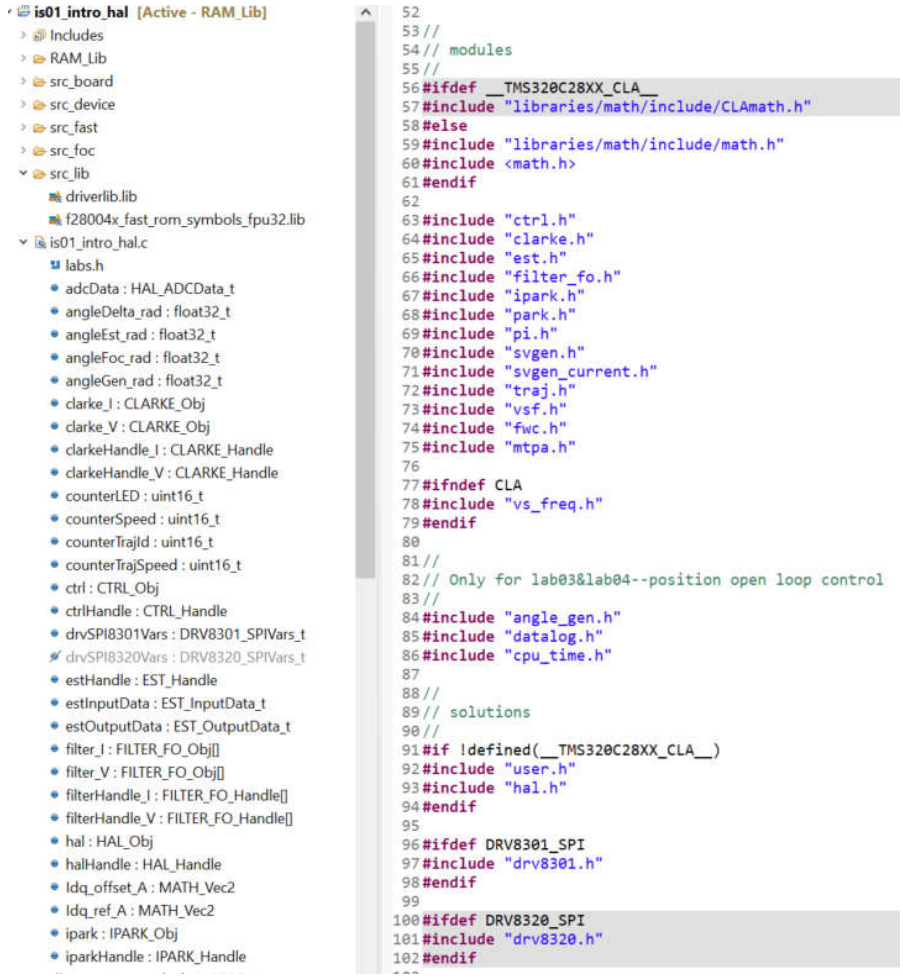


Figure 9: Additional header files required for project “is01_intro_hal” and later projects, found in labs.h

Global Object and Variable Declarations

The global objects and declarations listed in **Table 3** below are only the objects that are absolutely needed for the drive setup. Other objects and variable declarations are used for display or information for the purpose of this lab.

globals		
	HAL_Handle	The handle to the hardware abstraction layer object (HAL). The driver object contains handles to all microprocessor peripherals and is used when setting up and controlling the peripherals.
	USER_Params	Holds the scale factor information, motor information, and hardware information that is in user.h. Allows for scale factor updates in real-time, as well as informing the system about motor and hardware parameters.

Table 3: Global object and variable declarations necessary for project “is01_intro_hal”

To view the details of the objects HAL_Handle and USER_Params follow these steps:

1. In the file “is01_intro_hal.c” right-mouse click on HAL_Handle and select “Open Declaration” as shown in **Figure 10**.

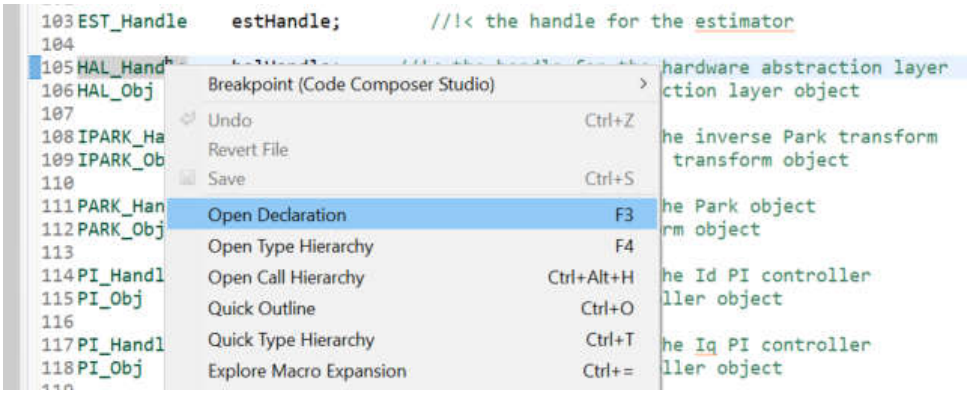


Figure 10: Opening the declaration for the HAL_Handle

Notes: If the “HAL_Handle” can be opened by this operation, right-mouse click “RAM_Lib” folder in the project, selecting “delete”, right-mouse click project “is01_intro_hal”, selecting “Clean Project”, and then right-mouse click project “is01_intro_hal” again, selecting “Rebuild Project”.

2. With the file “hal_obj.h” now open, right- mouse click on HAL_Handle and select “Show In Outline” as shown in **Figure 11**.

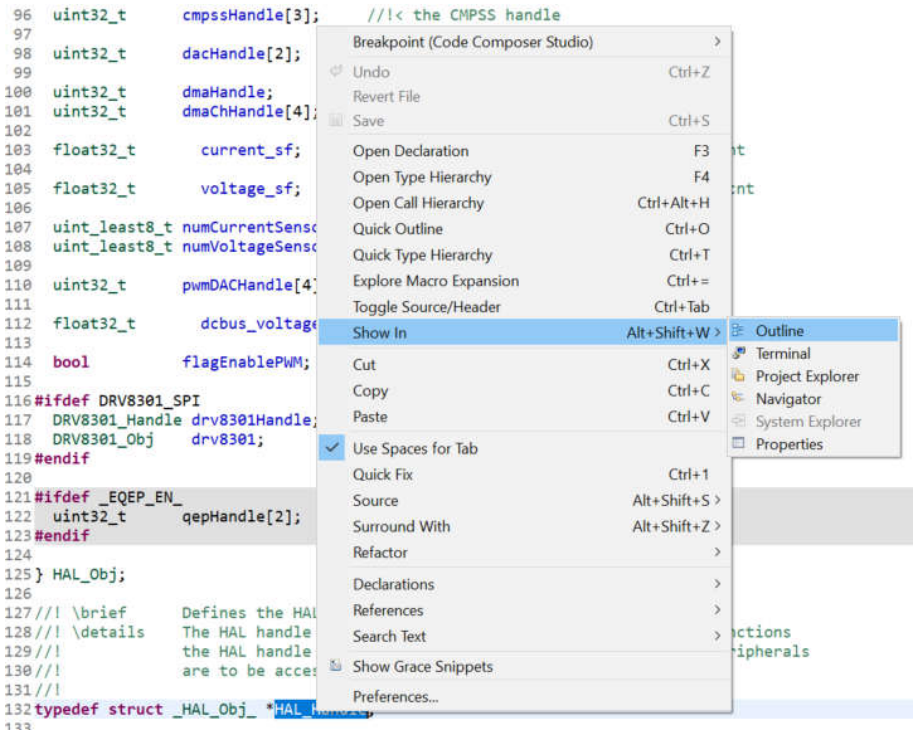


Figure 11: To show HAL_Handle and _HAL_Obj_ in Outline

3. With the Outline View open, expand "_HAL_Obj_" to see each member of the HAL object as shown in **Figure 12**.

Figure 12: HAL Object expand in Outline

4. In the file "is01_intro_hal.c" right-mouse clicks on USER_Params and select "Open Declaration". From the Outline view, expand _User_Params_ to display each member of the object as shown in **Figure 13**.

Figure 13: To show USER_Params object in Outline

Initialization and Setup

This section covers functions needed to set up the microcontroller and FOC software. Only functions mandatory for motor control are listed in **Table 4**; some functions present but not listed are for the enhanced capability of the lab and not fundamentally needed to set up the drive. For a more in-depth explanation of the function parameters and return values, please refer to the motor control section of the InstaSPIN-FOC User's Guide.

functions		
	HAL	
	HAL_init	Initializes all handles to the microcontroller peripherals. Returns a handle to the HAL object.
	USER_setParams	Copies all scale factors from the file user.h to the structure defined by USER_Params.
	HAL_setParams	Sets up the microcontroller peripherals. Creates all of the scale factors for the ADC voltage and current conversions. Sets the initial offset values for voltage and current measurements.
	HAL_initIntVectorTable	Points the ISR to the function mainISR.
	HAL_enableADCInts	Enables the ADC interrupt in the PIE, and CPU. Enables the interrupt to be sent from the ADC peripheral.
	HAL_enableGlobalInts	Enables the global interrupt.
	HAL_disablePWM	Set the inverter power switches to high impedance.

Table 4: Important setup functions needed for the motor control routines

mainISR

The methods called inside of mainISR() are time critical. When integrating this ISR into your code, it is important to verify that this ISR runs in real-time. The code in this lab will blink an LED and read ADC values which will eventually be three motor currents, three motor voltages, and one DC bus value. PWM values are also written to the inverter by calling HAL_writePWMData(); initializing pwmData{} values to zero results in a 50% duty cycle when written to the PWM registers. **Table 5** explains the functions used in the mainISR.

mainISR	
HAL_toggleLED	Toggles the LED on the motor inverter.
HAL_ackADCInt	Acknowledges the ADC interrupt so that another ADC interrupt can happen again.
HAL_readADCDataWithOffsets	Reads in the Adc result registers, adjusts for offsets, and scales the values according to the settings in user.h. The structure "adcData" holds three phase voltages, three line currents, and one DC bus voltage.
`	Converts the pwm values in "pwmData" to uint16 format and writes these values to the EPWM compare registers.

Table 5: The mainISR API functions that are used for Lab 1

Lab Procedure

Project “is01_intro_hal” is an introductory lab, similar to other “Hello World” programs. The corresponding program blinks an LED on the controlCARD or LaunchPad. The goal is to review the MCU and inverter setup functions, specifically the HAL object, and make sure the LED blinks.

1. Insert the C2000 controlCARD or LaunchPad onto one of the supported hardware kits, connect the USB cable to the controlCARD or LaunchPad, and apply power to the kit.
2. Project “is01_intro_hal” can be configured to run from FLASH or RAM. To change the build configuration, right-mouse click on is01_intro_hal” in the Project Explorer pane and select “Project->Build Configuration->Set Active”. From this window, select the configuration needed for the current build: “2 RAM_Lib” to run code from RAM, or “1 Flash_Lib” to run from FLASH. Alternatively, select “Project->Build Configuration->Manage…” to create a new build configuration. **Figure 14**Error! Reference source not found. illustrates selecting a RAM build configuration.

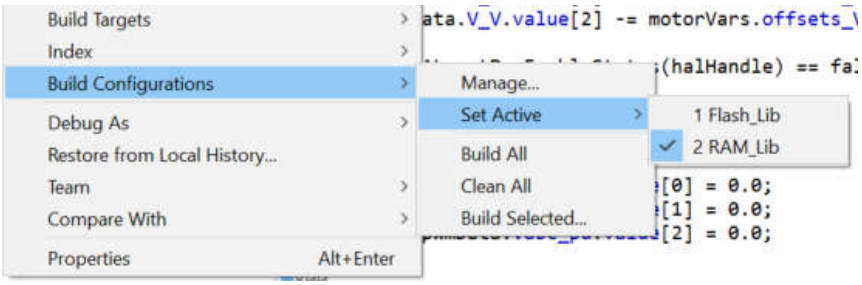


Figure 14: Selecting a RAM Build Configuration for is01_intro_hal

Once the build settings are configured, select the project(s) to be built by selecting them in the Project Explorer pane. The project is recognized as selected if the name is bolded with the word ‘Active’ next to it.

Notice the different files included for the two configurations: “Flash_Lib” uses the “f28004x_flash_cpu_is.cmd” linker command file, while the “RAM_Lib” configuration uses “f28004x_ram_cpu_is.cmd.”CCS will handle excluding the un-used configuration depending on whether RAM or FLASH build has been chosen. The “28004x_dcsm_lnk.cmd”, “f28004x_codestartbranch.asm”, “f28004x_dcsm_z1otp.asm” and “f28004x_dcsm_z2otp.asm” files are included to support the “Flash_Lib” configuration. Select the FLASH configuration to enable the program to boot from FLASH without the JTAG emulator.

3. In the “CCS Edit” view, select the “Build” hammer to build the project as shown in **Figure 15**; alternatively, right-mouse click the project in the Project Explorer pane and select “Project ->Build Project.” To build all the projects in the workspace, select “Project -> Build All.”

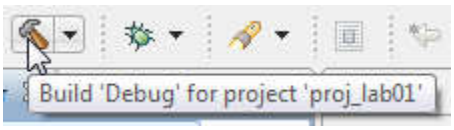


Figure 15: The CCS "Build" hammer icon

After the build is complete, a message will appear in the “Console” window indicating the build status for the selected project. If the build failed, check the “Problems” window for errors. Resolve

the errors and repeat the build process. Verify that the project has built successfully. If no errors exist in the “Problems” window, the project is built and ready to be loaded and run.

4. Select the green bug icon that symbolizes “Debug” as shown in **Figure 16**. This button should automatically change the CCS perspective to “CCS Debug” as well as load the .out file to the target.

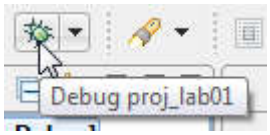


Figure 16: The CCS "Debug" bug icon

Notice the “CCS Debug” icon in the upper right-hand corner indicates that the perspective has changed to the “CCS Debug” view. After the program is loaded, the console window will indicate that the Memory Map Initialization is complete. At this point, the program ran through the C-environment initialization routine and stopped at **main()**. The source file containing main() will open with a blue arrow pointing to the first line of code to be executed. The program has now been successfully loaded on to the target board and it is ready to run.

5. Select “Real-Time Silicon Mode,” as shown in **Figure 17** which looks like a clock on the icon bar, and choose “Yes” if a small window pops up as shown in **Figure 18**.

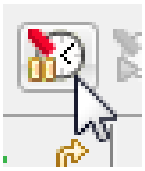


Figure 17: The CCS "Real-Time Mode" clock icon

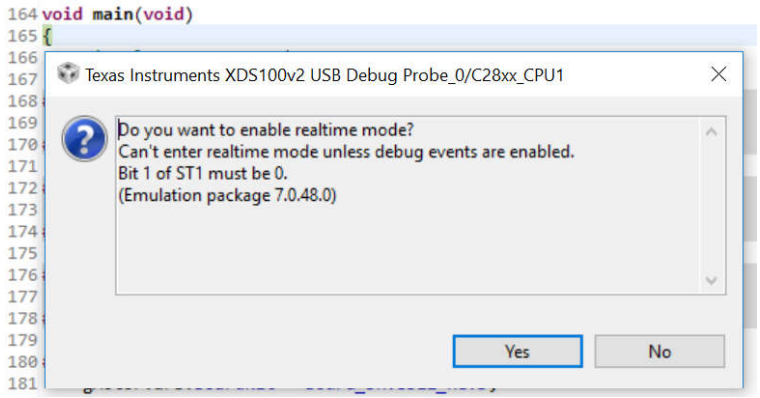


Figure 18: The pop-up window querying the option to enable real-time mode

6. Select “Resume,” as shown in **Figure 19** which looks like a yellow vertical line with a green triangle beside it, to begin executing program code.

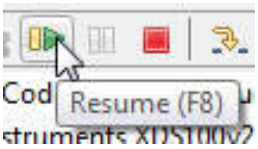


Figure 19: The "Resume" play icon

7. Open the command file “\solutions\common\sensorless_foc\debug\is01_intro_hal.js” via “View->Scripting Console”, this will add the variables that will be using for this project into the watch window.

8. At this point, one of the LEDs on the controlCARD or LaunchPad board will blink. The blinking LED verifies that code in the mainISR() is being executed.

9. Lab project “is01_intro_hal” is complete.

Conclusion

The HAL object was created to set up the MCU and inverter hardware. Project “is01_intro_hal” demonstrated how to use the HAL object to setup and initialize the MCU and inverter. Future labs will build on this functionality to demonstrate and enable motor control using InstaSPIN.

is02_offset_gain_cal – Current and Voltage Offsets Calibration

Abstract

Lab project “is02_offset_gain_cal” introduces the concept of current and voltage offsets for motor control, which are important to ensure the sampling quality of current and voltage feedback signals. Also introduced in this lab is the option to bypass offset calibration to reduce the startup time.

Introduction

For typical usage, an offset calibration takes place after InstaSPIN is first enabled. During calibration, offsets for the current and voltage feedback circuits are measured and recorded. After initial board calibration, these offset values should be updated for your specific hardware in the according to the user header file, as to make them available to the controller after project compilation. During motor operation, these offsets are subtracted from the ADC measurements to provide accurate voltage and current feedback to the InstaSPIN estimator and vector controller.

Prerequisites

Lab project “is02_offset_gain_cal” assumes knowledge of the project “is01_intro_hal”.

Objectives Learned

- Implement offset recalibration as needed depending on the quality of the hardware components used in a particular board, and use offset recalibration to verify your specific hardware.
- Write current and voltage offsets to the HAL object to reduce calibration time before motor startup.
- Bypass offset calibration, reducing calibration time before motor startup.

Detailed Description

In this lab, when the InstaSPIN controller is enabled and identification or running begins, the first task performed by the controller state machine is offset calculation. The estimator (EST) state stays in the idle state (EST_State_Idle) during the offset calculation state, and the motor is at standstill.

The offsets calculation is done in order to set the zeros for current measurements and voltage measurements. In order to calculate the offsets, a 50% duty cycle is applied to the ePWM pins for a preconfigured period of time. The time in which these offsets are calculated can be changed by the user, and it is configured in the “is02_offset_gain_cal.c” source file as shown below.

```
uint32_t gOffsetCalcWaitTime = 50000;
```

After initial board calibration, the voltage and current offset values should be updated in user.h for your specific hardware. These initial offset parameters can then be used to bypass offset calibration in future projects. The user.h current and voltage offset definitions for are shown below:

```
//! \brief ADC current offsets for A, B, and C phases
#define IA_OFFSET_A    (-21.428)        // ~=0.5*USER_ADC_FULL_SCALE_CURRENT_A
#define IB_OFFSET_A    (-21.428)        // ~=0.5*USER_ADC_FULL_SCALE_CURRENT_A
#define IC_OFFSET_A    (-21.428)        // ~=0.5*USER_ADC_FULL_SCALE_CURRENT_A

//! \brief ADC voltage offsets for A, B, and C phases
#define VA_OFFSET_V    (0.990514159)    // ~=1.0
#define VB_OFFSET_V    (0.986255884)    // ~=1.0
#define VC_OFFSET_V    (0.983381569)    // ~=1.0
```

Refer to “Section 5.2: Hardware Prerequisites” of the InstaSPIN User’s Guide, the section in question outlines the need to set the correct sign, positive or negative, of the current offset. The sign is to be based on the current feedback polarity of hardware board being used. The correct polarity of the current feedback is necessary to ensure the microcontroller has an accurate current measurement. The sign is negative for a positive feedback signal input; conversely, the sign is positive for a negative feedback signal input. Once the offset calibration is complete, the result will be stored in the motorVars.offsets_I_A and motorVars.offsets_V_V struct members.

Project Files

Compared to project “is01_intro_hal”, project “is02_offset_gain_cal” needs additional source files for offset calibration. These are shown in **Table 6**:

is02_offset_gain_cal		
	offset.c	Contains the offset code used to determine the voltage and current feedback offsets.
	filter_fo.c	Contains code for a first-order filter used for offset calibration.

Table 6: New files included for offset calibration

Includes

A description of the included files for project “is02_offset_gain_cal” is shown in the below tables. Note that labs.h is common across all InstaSPIN projects, meaning there will be more included header files than needed for the project “is02_offset_gain_cal” as shown in **Table 7**.

labs.h		
	modules	
	math.h	Common math conversions, defines, and shifts
	platforms	
	user.h	Contains the motor control initialization data for the CTRL, HAL, and EST modules

Table 7: Required header files for offset calibration routines

Global Object and Variable Declarations

Global objects and declarations listed in the table below are necessary to drive initialization. Some additional object and variable declarations are used for display or information purposes in this lab.

globals			
	MOTOR_Vars_t	motorVars	Contains flags and variables necessary to turn on and implement InstaSPIN.
	FILTER_FO_Handle	filterHandle[6]	The handles for the 3 current and 3 voltage filters used for offset calculation
	FILTER_FO_Obj	filter[6]	The 3 current and 3 voltage filter objects needed for offset calculation
	uint32_t	offsetCalcCount	Counter used to count the wait time for offset calibration, unit: ISR cycles
	uint32_t	offsetCalcWaitTime	Wait time setting for current/voltage offset calibration, unit: ISR cycles
	bool	flagEnableOffsetCalibration	The enable flag for offset calibration always set for debug purposes.

Table 8: Global object and variable declarations necessary for lab “is02_offset_gain_cal”

The offset calibration feature allows recalculating voltage and current offsets as desired while the motor is at standstill. The approach taken to calculate the offsets uses 6 first order filters, which are declared at the top of is02_offset_gain_cal.c program code.

```
FILTER_FO_Handle  filterHandle[6];  //!< the handles for the 3-current and 3-voltage filters
for offset calculation


FILTER_FO_Obj     filter[6];        //!< the 3-current and 3-voltage filters for offset
calculation
```

Next, the filters are initialized using the cutoff frequency specified in user.h via **USER_OFFSET_POLE_rps**.

```
//! \brief Defines the pole location for the voltage and current offset estimation, rad/s
#define USER_OFFSET_POLE_rps            ((float_t)(20.0))
```

The offsets are calculated only when the enable flag, motorVars.flagEnableOffsetCalc is set; this logic is checked in the ISR as shown in runOffsetsCalculation() in “is02_offset_gain_cal.c”. Offset calculation will be bypassed, and the initial offsets instead read from “user.h” when “motorVars.flagEnableOffsetCalc” is equal to “false.”

```
else if(motorVars.flagEnableOffsetCalc == true)
{
    runOffsetsCalculation();

    // Below two lines code only used in this lab for hardware verification
    // these two lines code will be removed for later lab projects
    if(flagEnableOffsetCalibration == true)
        motorVars.flagEnableOffsetCalc = true;
 }
```

The runOffsetsCalculation() function is described in “is02_offset_gain_cal.c”. The PWM duty cycles are set to 50% by writing 0 to the period (Tabc) values, and the filters are run. Once the preset time has elapsed, the calculated offsets are stored in global variables motorVars.offsets_I_A.value and motorVars.offsets_V_V.value. A block diagram of the lab “is02_offset_gain_cal” is shown in **Figure 20**.

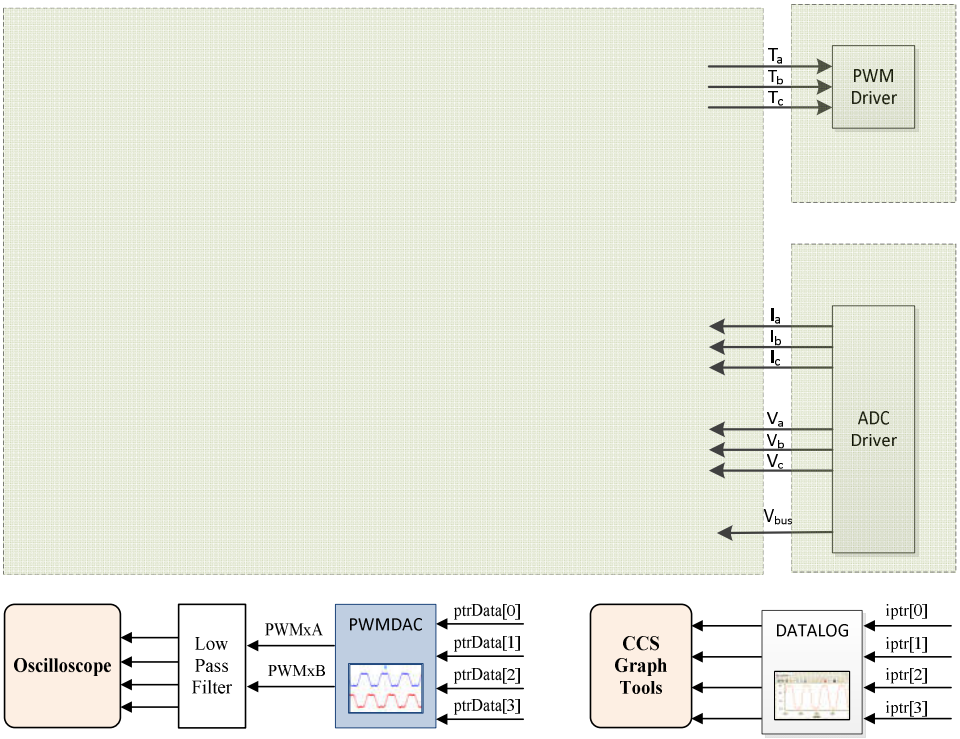


Figure 20: Block diagram of the “is02_offset_gain_cal” lab

Lab Procedure

1. Connect an InstaSPIN-enabled C2000 microcontroller controlCARD or LaunchPad to a supported EVM or Booster Pack, connect a supported motor type, and supply power to the kit. For this write-up, the BOOSTXL-DRV8320RS RevA kit and LAUNCHXL_F280049C LaunchPad were used.

2. Import project “is02_offset_gain_cal” by choosing “Project->Import CCS Projects…,” as was done for previous lab projects. Select “is02_offset_gain_cal” at the top of the “Project Explorer” tab, and use “Project->Build Configuration->Set Active->2 Ram_Lib” to run this code from RAM.

3. In the “CCS Edit” view, click the “Build” hammer, or select “Project ->Build Project” to build project “is02_offset_gain_cal”. To build all the projects in the workspace, select “Project -> Build All.” Connect to the target and load the .out file.
   - Open the Javascript file “\solutions\common\sensorless_foc\debug\is02_offset_gain_cal.js” via the “View->Scripting Console.” This will add the variables used for this project into the “Expressions” watch window
   - Click “Real-Time Silicon Mode” to enable the real-time debugger
   - Click the run button.
   - Enable “Continuous Refresh” on the “Expressions” watch window.

4. Check the initial offset settings as shown in **Figure 21**; these values were loaded from the voltage and current definitions found in user.h. The current offset value is equal to half of the USER_ADC_FULL_SCALE_CURRENT_A value, as defined in the user header file.

motorVars.offsets_I_A.value[0]	float	-21.4279995
motorVars.offsets_I_A.value[1]	float	-21.4279995
motorVars.offsets_I_A.value[2]	float	-21.4279995
motorVars.offsets_V_V.value[0]	float	0.990514159
motorVars.offsets_V_V.value[1]	float	0.986255884
motorVars.offsets_V_V.value[2]	float	0.983381569

Figure 21: Initial values for current and voltage offsets

View the current and voltage sampling ADC results in the Code Composer “Registers” tab of the “CCS Debug” pane, as shown in **Figure 22**. For this write-up, AdcaResultRegs.ADCRESULT0, AdcbResultRegs.ADCRESULT0, and AdccResultRegs.ADCRESULT0 are the motor phase current sampling results; these three values are equal to half of the maximum ADC converter value since the reference voltage of amplification circuits is 1.65V in 3.3V systems. AdcaResultRegs.ADCRESULT1, AdcbResultRegs.ADCRESULT1, AdccResultRegs.ADCRESULT1 are the motor phase voltage sampling results, and these values are near 0 since there is no voltage output on the motor phases. AdcbResultRegs.ADCRESULT2 is for the DC bus voltage sampling result, which is converted to real voltage and stored in adcData.dcBus_V. adcData.dcBus_V represents the actual DC bus input voltage.

Please note that the ADC modules and result registers may be different than this guide depending on which hardware board is being used.

AdcaResultRegs.ADCRESULT0	unsigned int	2040
AdcbResultRegs.ADCRESULT0	unsigned int	2039
AdccResultRegs.ADCRESULT0	unsigned int	2026
AdcaResultRegs.ADCRESULT1	unsigned int	852
AdcbResultRegs.ADCRESULT1	unsigned int	857
AdccResultRegs.ADCRESULT1	unsigned int	860
AdcbResultRegs.ADCRESULT2	unsigned int	1714

Figure 22: ADC Result registers tab in CCS "Registers" window

5. To start the project, set the variable “motorVars.flagEnableSys” equal to 1 in the “Expressions” watch window. The PWM duty cycles are set to 50% and the PWM compare register (CMPA) value is equal to half of the full-scale period (TBPRD) as shown in **Figure 23**.

pwmData.Vabc_pu.value[0]	float	0.0
pwmData.Vabc_pu.value[1]	float	0.0
pwmData.Vabc_pu.value[2]	float	0.0
EPwm6Regs.TBPRD	unsigned int	2500
EPwm6Regs.CMPA.bit.CMPA	unsigned int	1250
EPwm5Regs.CMPA.bit.CMPA	unsigned int	1250
EPwm3Regs.CMPA.bit.CMPA	unsigned int	1250

Figure 23: PWM and period values as shown in the "Expressions" watch window

Observe the ADC Result registers and the converted feedback values for current and voltage in the adcData data structure. As seen in each adcData.I_A.value[] value is approximately half of the USER_ADC_FULL_SCALE_CURRENT_A value; similarly, the adcData.V_V.value[] values are approximately half of the adcData.dcBus_V value as shown in **Figure 24**.

adcData.dcBus_V	float	24.0589504
adcData.I_A	struct _MATH_Vec...	{value=[-21.3587418,-...
value	float[3]	[-21.3587418,-21.118...
[0]	float	-21.1191998
[1]	float	-21.0245476
[2]	float	-21.2959824
adcData.V_V	struct _MATH_Vec...	{value=[11.9459085,1...
value	float[3]	[11.9803181,12.05054...
[0]	float	11.9803181
[1]	float	12.0364981
[2]	float	12.0645876

Figure 24: Current and Voltage without Offset as shown in the "Expressions" watch window

Note: There may be hardware issues if the observed values are not similar to the above. If the observed values differ greatly, it may be necessary to check the current and/or voltage sampling circuits, or PWM drive circuit of the inverter, to determine if any layout issues are present.

6. Once finished, change the “flagEnableOffsetCalibration” value from 1 to 0 to complete offset calibration. The final offset value will be updated in motorVars.offsets_I_A and motorVars.offsets_V_V, shown in **Figure 25**. For reference, the “flagEnableOffsetCalibration” variable is only used in this lab project for offset calibration debug. In later projects, offset calibration will be completed automatically.

motorVars.offsets_I_A.value[0]	float	-21.3030872
motorVars.offsets_I_A.value[1]	float	-21.092535
motorVars.offsets_I_A.value[2]	float	-21.2666092
motorVars.offsets_V_V.value[0]	float	0.994864643
motorVars.offsets_V_V.value[1]	float	1.00112736
motorVars.offsets_V_V.value[2]	float	1.00301814

Figure 25: Final offset values

7. Copy and paste the motorVars.offsets_I_A. and motorVars.offsets_V_V from the Watch Window into the corresponding #define statements in the file user.h as mentioned above. Storing the voltage and current offsets for future bypassing the offset calibration process if the variable “motorVars.flagEnableOffsetCalc” is set to “0”.

8. Stop the motor when finished experimenting by:
   - Set the variable “motorVars.flagEnableSys” to 0, disabling the PWM output.
   - Disable “Real-Time Control” and stop the debugger.
   - Turn off the power supplied to the drive kit

Conclusion

Lab project “is02_offset_gain_cal” demonstrated how to implement offset calibration, and use the offset calibration process to verify the current and voltage sampling circuits of the hardware board. Additionally, this write-up discussed how to set an inverter’s current and voltage offset values in a “user.h” file in order to bypass offset calibration.

is03_hardware_test – Open Loop Control for Hardware Integrity Verification

Abstract

The SVM that is used by InstaSPIN is capable of saturating to a pre-specified duty cycle. When using a duty cycle over 100.0%, the SVM is considered to be in the over-modulation region. When in the over-modulation region, current shunt measurement windows become small or even disappear. This lab will show how to re-create the currents that cannot be measured due to high duty cycles during SVM over-modulation.

Introduction

Lab “is03_hardware_test” demonstrates using an angle generator module to simulate the flux angle based on the motor target frequency, and utilizes a volts/Hertz profile to command the resultant three-phase motor voltage. Although the FAST estimator is not used to generate the output angle here, lab “is03_hardware_test” will test several InstaSPIN-FOC modules through the scalar volts/Hertz control.

Prerequisites

Lab “is03_hardware_test” assumes knowledge of up to the lab “is02_offset_gain_cal”.

Objectives Learned

- How to implement a scalar volts/Frequency motor control scheme.
- How to test the following InstaSPIN-FOC modules using a volts/Hertz control technique
  - Pulse-Width Modulation (PWM) module
  - Analog-to-Digital Converter (ADC) module
  - Clarke transform (CLARK) module
  - Park transform (PARK) module
  - Space Vector Generator (SVGEN) module

Background

In order to achieve better dynamic performance, a more complex control scheme needs to be applied for control of ACI or PM motors. Scalar control refers to a simpler form of motor control using non-vector controlled drive schemes. An AC motor can be led to steady state by either voltage fed, current controlled, or speed controlled schemes

In V/Hz control, the speed of the AC motor is controlled by an adjustable magnitude of stator voltages and frequencies. The V/Hz control is applied in such a way that the flux is always maintained at the desired value during steady-state, and the torque becomes independent of the supply frequency. The stator voltage to frequency ratio is usually based on the rated values of these variables. The typical V/Hz profile, as this ratio is known, is shown in **Figure 26**. Basically, there are three-speed ranges in the V/Hz profile as follows:

- At 0-f_c Hz, a voltage is required, so the voltage drop across the stator resistance cannot be neglected and must be compensated for by increasing Vs. As a result, the V/Hz profile is not linear. The cutoff frequency (f_c) and the suitable stator voltages may be analytically computed from the steady-state equivalent circuit with Rs ≠ 0.
- At f_c-f_{rated} Hz, the profile follows a constant V/Hz relationship.
- At frequencies are greater than f_{rated} Hz, the constant Vs/f ratio cannot be satisfied due to the limit of DC bus input voltage.. In this region, the resulting air gap flux would be reduced, unavoidably causing the decrease of developed torque. This region is usually called the “constant power region” and a constant (“flat”) V/Hz curve is implemented to the output voltage corresponding to the motor frequencies here.

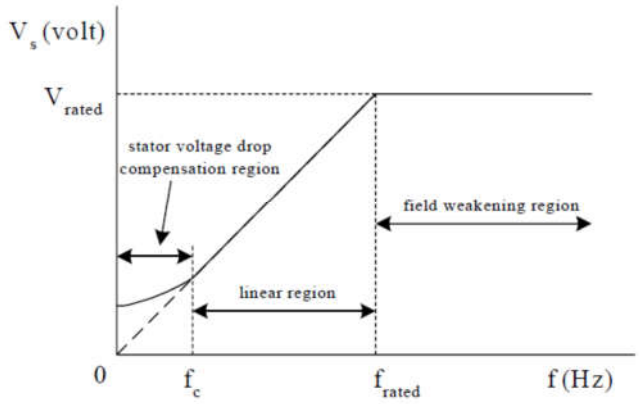


Figure 26: Stator voltage versus frequency profile under V/Hz control

In this lab, the profile is modified, as shown in **Figure 27**, by imposing a lower limit on frequency. This approach is acceptable in applications such as fan and blower drives where the speed response at low frequencies is not critical. Since the rated voltage, which is also the maximum voltage, is applied to the motor at rated frequency, only the rated minimum and maximum frequency information are needed to implement the v/f profile.

The command frequency is allowed to go below the minimum frequency, f_{min}, with the output voltage saturating at a minimum value, V_{min}. Also, when the command frequency is higher than the maximum frequency, f_{max}, the output voltage is saturated at a maximum value, V_{max}.

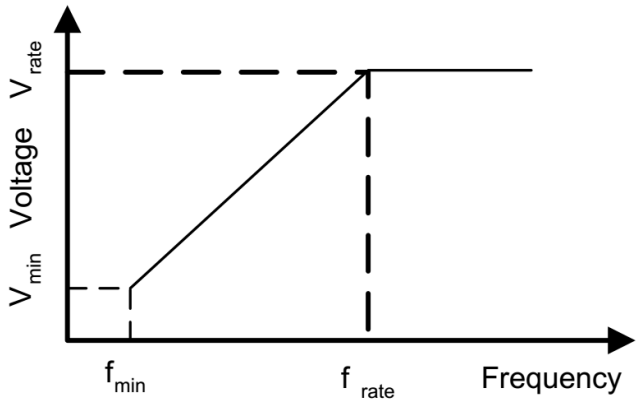


Figure 27: Modified v/f profile

Lab “is03_hardware_test” uses a ramp generator module to generate the angle based on the motor target frequency as **Equation 1**. For this lab, the variable *StepAngleMax* is used to determine the minimum period (1/ sampling loop frequency) of the ramp signal. Adding a fixed step value to the *Angle* variable causes the value in *Angle* to cycle at a constant rate.

$$Angle(k+1) = Angle(k) + StepAngleMax * f \quad \text{Equation 1}$$

For this implementation, the maximum step size per sampling period is calculated as **Equation 2**:

$$StepAngleMax = 2\pi \times T_s \quad \text{Equation 2}$$

where T_s is the sampling period (sec).

At the end limit, the value in *Angle* simply wraps around and continues at the next modulo value given by the step size. For a given step size, the frequency of the ramp output (in Hz) is given by **Equation 3**:

$$f = StepAngle * f_s \quad \text{Equation 3}$$

where f_s is the sampling loop frequency in Hz..

PWMDAC

The PWMDAC module can be used to monitor variables as a virtual oscilloscope that is converted into PWM signals output by the EPWMxA/B channels. The output represents the variable as an analog signal after the channel outputs of the PWMxA/B pins are passed through external RC low-pass filters. Instantiate the PWMDAC module using the following steps:

Step 1: Declare a PWMDAC object in “is03_hardware_test.c”.

```
// the PWMDAC variable
HAL_PWMDACData_t pwmDACData;
```

Step 2a: Set the correct offset and gain for each PWMDAC channel.

Note: Follow step 2a to monitor the ANGLE_GEN for rotor angle and SVGEN for PWM output. Steps 2b and 2c, shown later in this guide, will allow you to monitor a different set of variables using the PWMDAC module.

```
// set DAC parameters
pwmDACData.periodMax = PWMDAC_getPeriod(halHandle->pwmDACHandle[PWMDAC_Number_1]);

pwmDACData.ptrData[0] = &angleFoc_rad;
pwmDACData.ptrData[1] = &pwmData.Vabc_pu.value[0];
pwmDACData.ptrData[2] = &pwmData.Vabc_pu.value[1];
pwmDACData.ptrData[3] = &pwmData.Vabc_pu.value[2];

pwmDACData.offset[0] = 1.0;
pwmDACData.offset[1] = 0.5;
pwmDACData.offset[2] = 0.5;
pwmDACData.offset[3] = 0.5;

pwmDACData.gain[0] = -MATH_ONE_OVER_TWO_PI;
pwmDACData.gain[1] = 1.0;
pwmDACData.gain[2] = 1.0;
pwmDACData.gain[3] = 1.0;
```

Step 3a: Call the PWMDAC module to compute the compare value of PWM from the previous step

```
// connect inputs of the PWMDAC module.
HAL_writePWMDDACData(halHandle,&pwmDacData);
```

Following steps 2a and 3a, the output waveforms of the PWMDACs are as shown in **Figure 28**.

Ch1 (Yellow) -> PWMDAC1: Rotor angle, the output of angle generator module
Ch2 (Gree) -> PWMDAC2: SVGEN output for phase A
Ch3 (Blue) -> PWMDAC3: SVGEN output for phase B
Ch4 (Pink) -> PWMDAC4: SVGEN output for phase C

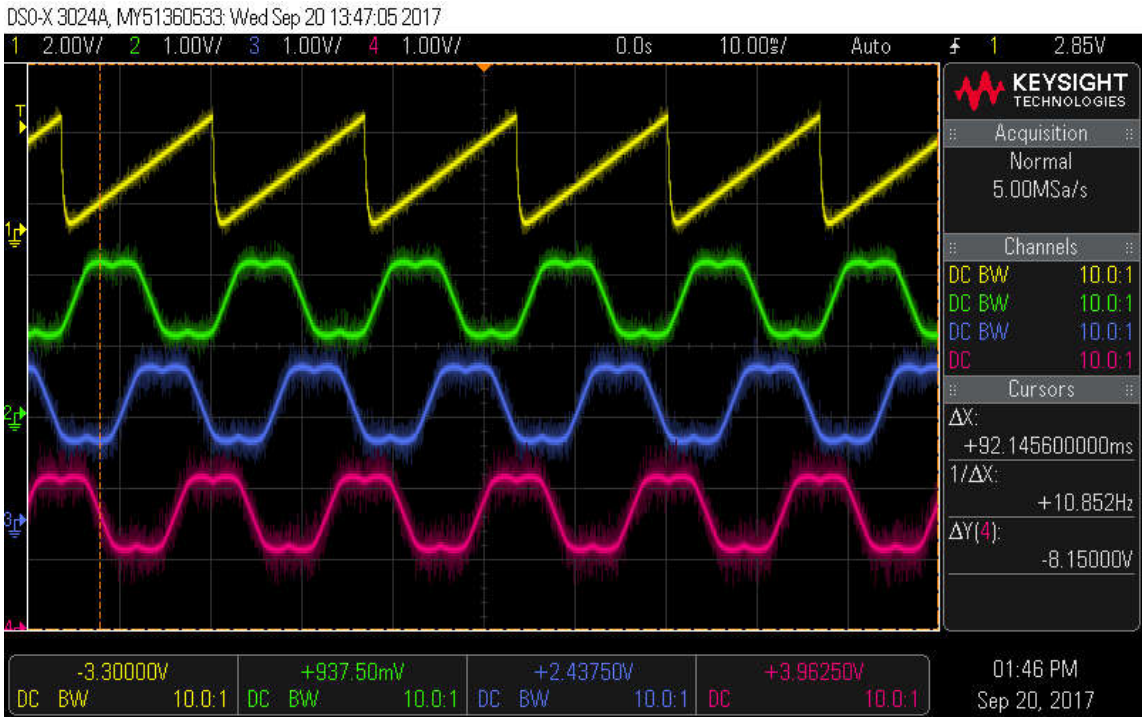


Figure 28: Rotor Angle and SVGEN output waveform

Note: Follow Step 2b to monitor the 3 phase motor current samples.

Step 2b: Set the correct offset and gain for each PWMDAC channel as shown below. Please follow step 3a above to again connect the PWMDAC inputs to the variables to be monitored. Following these steps, the PWMDAC output waveform is as shown in **Figure 29**.

```
    pwmDACData.ptrData[0] = &angleFoc_rad;
    pwmDACData.ptrData[1] = &adcData.I_A.value[0];
    pwmDACData.ptrData[2] = &adcData.I_A.value[1];
    pwmDACData.ptrData[3] = &adcData.I_A.value[2];

    pwmDACData.gain[0] = -MATH_ONE_OVER_TWO_PI;
    pwmDACData.gain[1] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[2] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[3] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
```

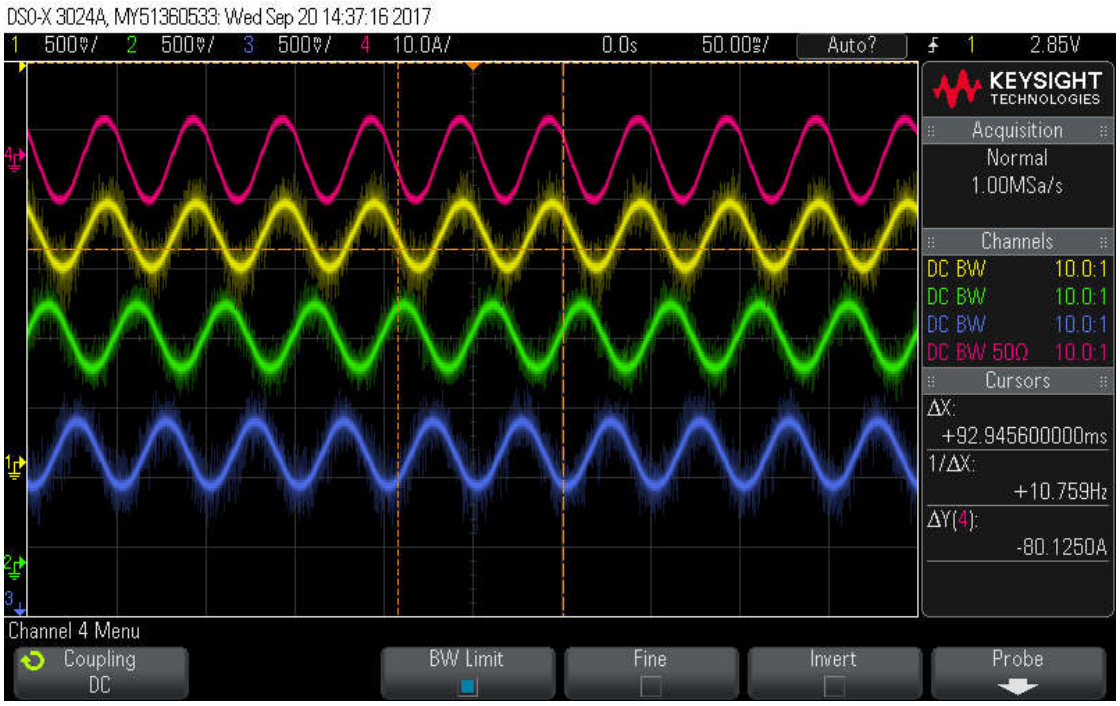


Figure 29: Three phase current samples and measured current via external current probe

Where

Ch1 (Yellow) -> PWMDAC2: Phase A current sampling from ADC
Ch2 (Gree) -> PWMDAC3: Phase B current sampling from ADC
Ch3 (Blue) -> PWMDAC4: Phase C current sampling from ADC
Ch4 (Pink) ->Oscilloscope: Phase A current waveform with current probe

Note: Follow Step 2c to monitor the 3 phase motor voltage samples.

Step 2c: Set the correct offset and gain for each PWMDAC channel. Assign the PWMDAC inputs as shown in step 3a above; following, the output waveform is as shown in **Figure 30**.

```
    pwmDACData.ptrData[0] = &angleFoc_rad;
    pwmDACData.ptrData[1] = &adcData.I_A.value[0];
    pwmDACData.ptrData[2] = &adcData.I_A.value[1];
    pwmDACData.ptrData[3] = &adcData.I_A.value[2];

    pwmDACData.gain[0] = -MATH_ONE_OVER_TWO_PI;
    pwmDACData.gain[1] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[2] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[3] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
```

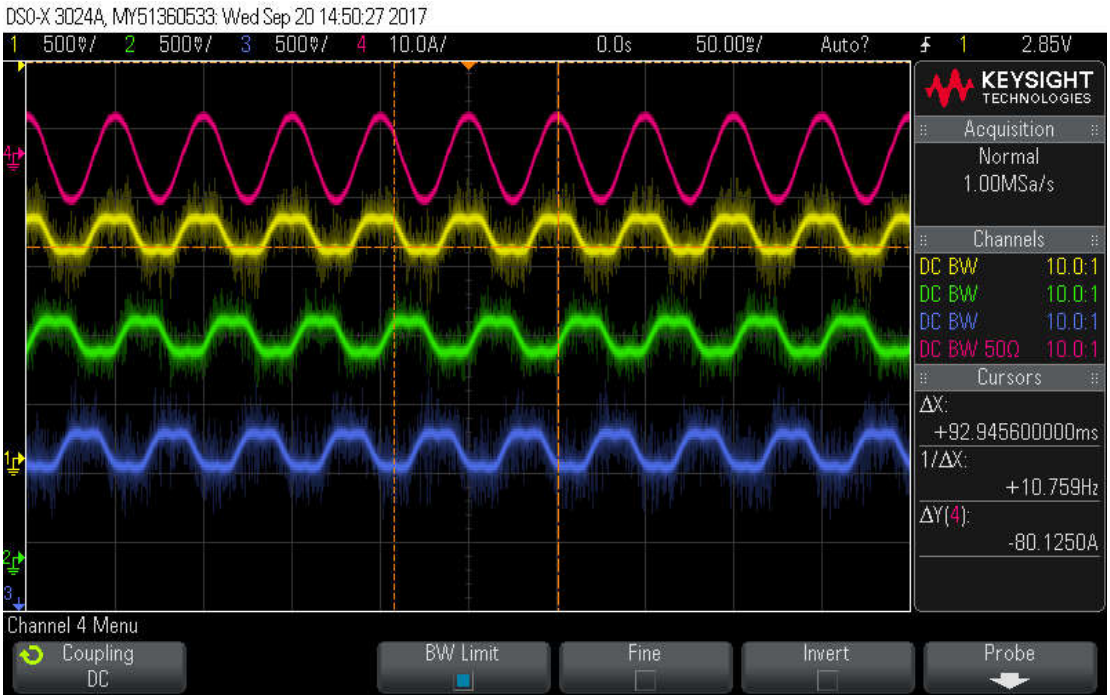


Figure 30: Three-phase voltage sample and measured current via external current probe

Where

Ch1 (Yellow) -> PWMDAC2: Phase A voltage sampling from ADC
Ch2 (Gree) -> PWMDAC3: Phase B voltage sampling from ADC
Ch3 (Blue) -> PWMDAC4: Phase C voltage sampling from ADC
Ch4 (Pink) ->Oscilloscope: Phase A current waveform with current probe

DATALOG_DMA

Lab “is03_hardware_test” uses the datalog module to store variable data and implements a virtual oscilloscope using the CCS graphing tool to monitor the input/output waveforms.

Setup and use the graph window

The following steps describe how to initialize the CCS graphing tool:

1. Select “Tools->Graph->Dual time” from the CCS menu bar. Select “Import” and choose the appropriate graph configuration file from “..\solutions\common\sensorless_foc\debug -> is03_hardware_test_d12.graphProp”.

2. Select “Tools->Graph->Single time” from the CCS menu bar. Select “Import” and choose the appropriate graph configuration file from “..\solutions \ common \ sensorless_foc \debug -> is03_hardware_test_d34.graphProp”.

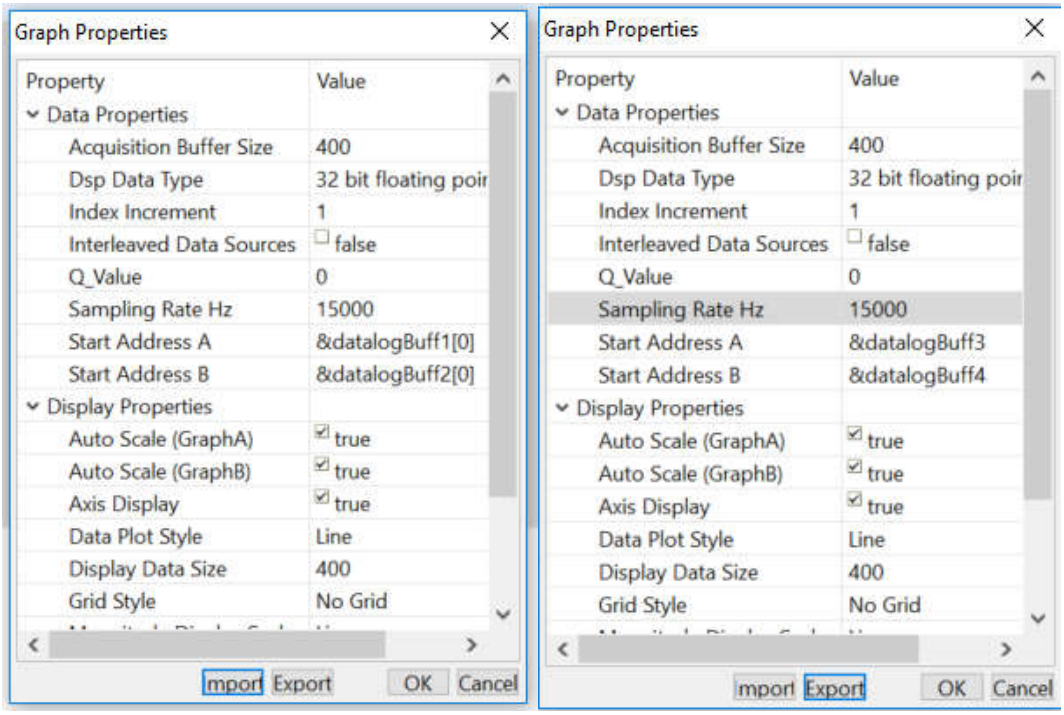


Figure 31: Graph Tool properties configuration menu

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

- Press the  button icon to reset the current graph window to auto fit the new scale of the data array.
- Press the  button icon 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 the  button icon to enable automatic graph update, every time the data array receives new data.
- Press the  button icon to refresh the graph every time you halt the debug session.

3. For this step, follow part (a) to view the SVGEN graph data, part (b) for phase current sampling data, and part (c) for the phase voltage sampling data

   a. The inputs necessary to connect the SVGEN module to the datalog are shown below, and the SVGEN output is shown in

   b. **Figure** 32.

```
// set datalog parameters
datalogObj->iptr[0] = &angleFoc_rad;
datalogObj->iptr[1] = &pwmData.Vabc_pu.value[0];
datalogObj->iptr[2] = &pwmData.Vabc_pu.value[1];
datalogObj->iptr[3] = &pwmData.Vabc_pu.value[2];
```

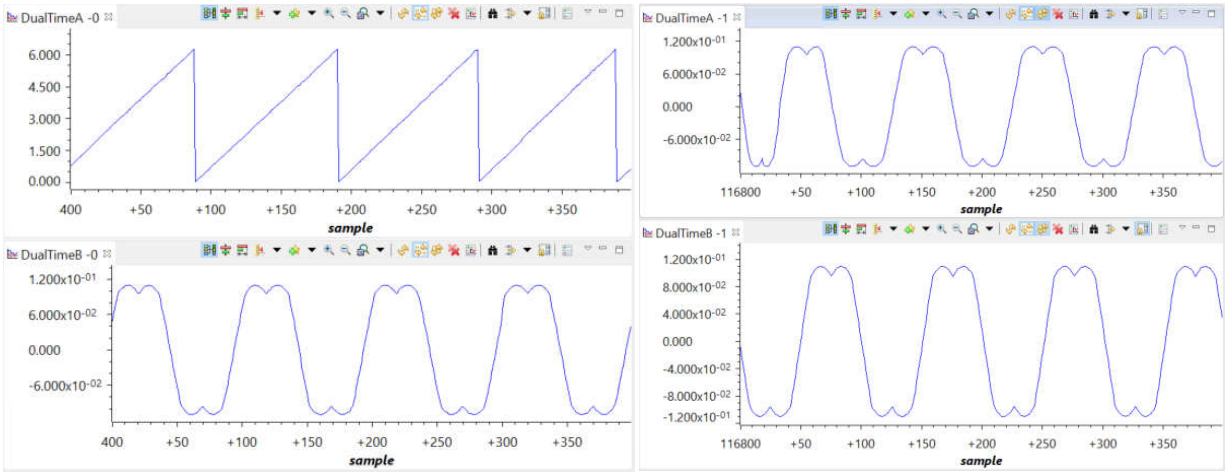


Figure 32: datalog graphs for SVGEN output

c. The inputs necessary to connect the sampled phase currents to the datalog are shown below, and the rotor angle and sampled currents are shown in **Figure 33**.

```
// set datalog parameters
datalogObj->iptr[0] = &angleFoc_rad;
datalogObj->iptr[1] = &adcData.I_A.value[0];
datalogObj->iptr[2] = &adcData.I_A.value[1];
datalogObj->iptr[3] = &adcData.I_A.value[2];
```

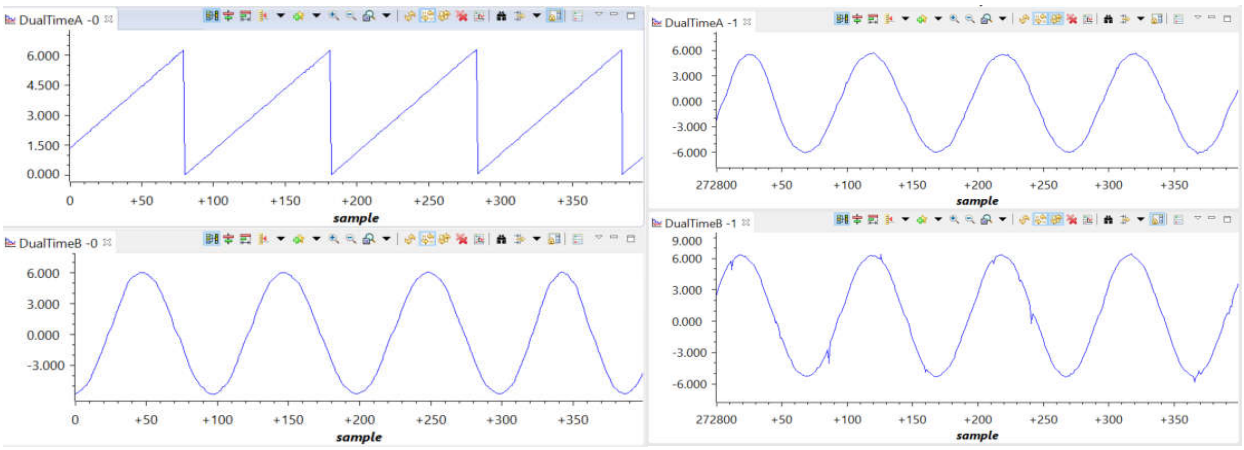


Figure 33: datalog graphs for rotor angle and three phase current ADC values

d. The necessary inputs for displaying the phase voltage samples via the datalog module are shown below, and the angle and ADC voltage samples are shown in **Figure 34**.

```
// set datalog parameters
datalogObj->iptr[0] = &angleFoc_rad;
datalogObj->iptr[1] = &adcData.V_V.value[0];
datalogObj->iptr[2] = &adcData.V_V.value[1];
datalogObj->iptr[3] = &adcData.V_V.value[2];
```

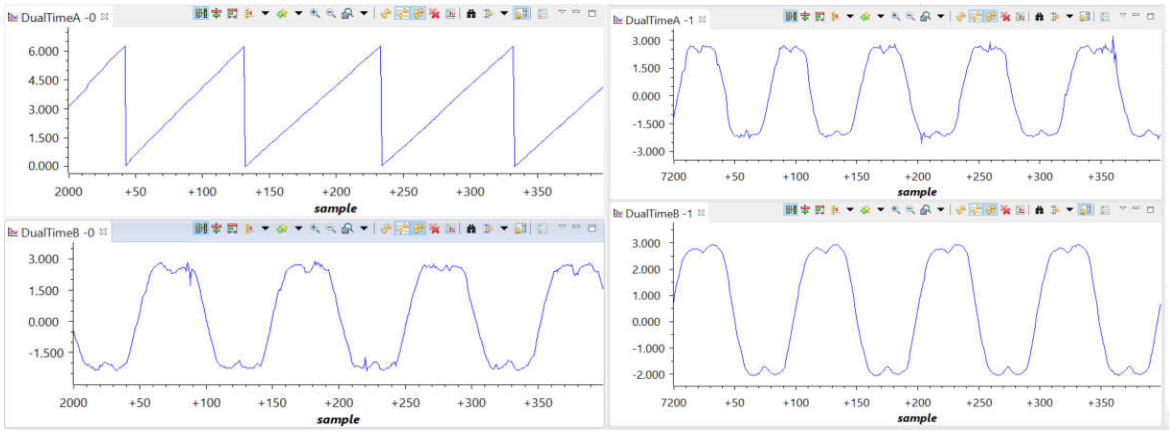


Figure 34: Datalog graphs of rotor angle and three-phase voltage ADC samples

Project Files

Compared to lab “is02_offset_gain_cal”, the lab “is03_hardware_test” adds new source files, as shown in **Table 9**.

is03_hardware_test		
	angle_gen.c	Defines the angle generator module routines
	vs_freq.c	Defines the volts/Hertz profile module routines

Table 9: New files that must be included in lab “is03_hardware_test”

Includes

A description of the included files for lab “is03_hardware_test” is shown in the below **Table 10**. Note that “labs.h” is common across the project; as a result, there will be more included files than needed for this lab.

labs.h	The header file containing all included files used in “is03_hardware_test.c”. Defines the structures, global initialization, and functions used in this lab.	
	modules	
	angle_gen.h	Contains the public interface to the angle generator module routines
	vs_freq.h	Contains the public interface to the volts/Hertz module routines
	platforms	
	hal.h	Device setup and peripheral drivers. Contains the HAL object.
	user.h	Contains the motor control initialization data for the CTRL, HAL, and EST modules

Table 10: Important header files needed for lab “is03_hardware_test”

Global Object and Variable Declarations

Global objects and declarations listed in

globals			
	MOTOR_Vars_t	motorVars	Contains all of the flags and variables to turn on and adjust InstaSPIN.
	ANGLE_GEN_Handle ANGLE_GEN_Obj	angle_genHandle angle_gen	The object and handle of an Angle Generator struct.
	VS_FREQ_Handle VS_FREQ_Obj	vs_freqHandle vs_freq	The object and handle of a Volts/Hertz Profile struct.
	HAL_DacData_t	dacData	The object of a PWMDAC.

Table 11
below are those necessary for the lab setup. Other object and variable declarations are used for display or information purposes for this lab.

globals			
	MOTOR_Vars_t	motorVars	Contains all of the flags and variables to turn on and adjust InstaSPIN.
	ANGLE_GEN_Handle ANGLE_GEN_Obj	angle_genHandle angle_gen	The object and handle of an Angle Generator struct.
	VS_FREQ_Handle VS_FREQ_Obj	vs_freqHandle vs_freq	The object and handle of a Volts/Hertz Profile struct.
	HAL_DacData_t	dacData	The object of a PWMDAC.

Table 11: Global object and variable declarations

Initialization and Setup

This section covers functions needed to set up the microcontroller and the FOC software. For a more in-depth explanation of the parameters and return values, go to the motor control section of the InstaSPIN-FOC User's Guide. The below snippets of code demonstrate setting up the angle generator module and volts/frequency profile module.

```
//
// initialize the angle generate module
//
angleGenHandle = ANGLE_GEN_init(&angleGen, sizeof(angleGen));
ANGLE_GEN_setParams(angleGenHandle, userParams.ctrlPeriod_sec);

//
// initialize the Vs per Freq module
//
VsFreqHandle = VS_FREQ_init(&VsFreq, sizeof(VsFreq));
VS_FREQ_setVsMagPu(VsFreqHandle, userParams.maxVsMag_pu);
VS_FREQ_setMaxFreq(VsFreqHandle, USER_MOTOR_FREQ_MAX_HZ);
VS_FREQ_setProfile(VsFreqHandle,
                   USER_MOTOR_FREQ_LOW_HZ, USER_MOTOR_FREQ_HIGH_HZ,
                   USER_MOTOR_VOLT_MIN_V, USER_MOTOR_VOLT_MAX_V);
```

Background

mainISR

The main ISR function is a time critical function that performs the FOC loop. A block diagram of lab "is03_hardware_test" is shown in **Figure 35**. The main ISR function calls the Clarke module, reads the ADC, performs ANGLE_GEN_run, VS_FREQ_run, and inverse Park transform, runs SVGEN and finally writes to the PWM for scalar control.

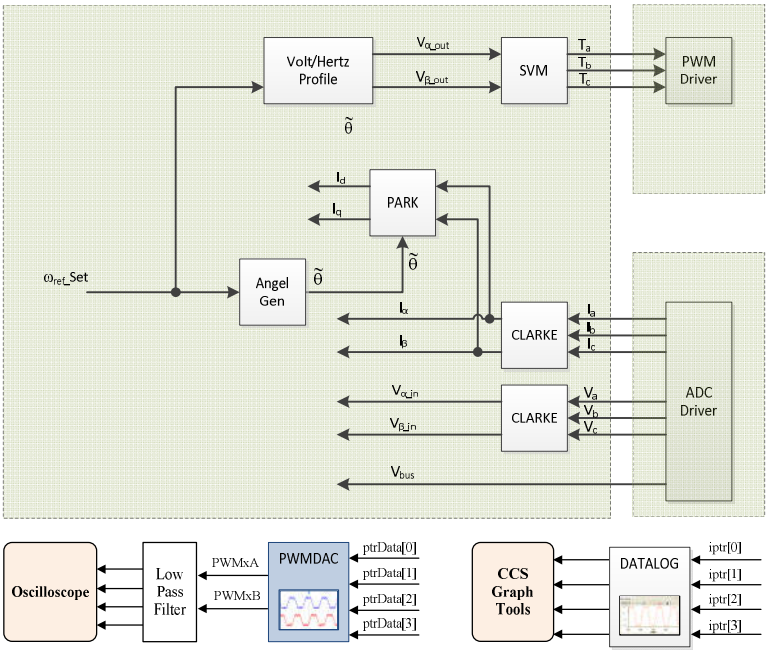


Figure 35: Block diagram of the volt/Hz motor control technique used in this lab

Lab Procedure

**Please note: It is possible to experience motor vibration while running this lab; as this lab is intended for hardware verification, it is not an issue to see this effect in this lab.

Before running a motor with lab “is03_hardware_test”, changes are needed for the “user.h” header file.

Open user.h following these steps:

1. Expand user.c from the CCS “Project Explorer” window
2. Right-mouse click on user.c and select “Open”, this opens the file user.c
3. Right-mouse click on the highlighted “user.h” and select “Open Declaration”, this opens user.h
4. Opening the CCS “Outline View” will provide an outline of the user.h contents

For a new motor, it may be necessary to change the below parameters to ensure proper project execution.

```
#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
```

The structure ‘motorVars’ contains the variables necessary to run lab “is03_hardware_test” project and has been defined in labs.h. A script has been written to easily add these variables to the CCS “Real-Time Watch Window.”

- Select the scripting tool from the debugger menu via “View->Scripting Console”.
- The scripting console window will appear somewhere in the debugger.
- Open the script by clicking the  icon that is in the upper right corner of the scripting tool.
- Select the file “\solutions\common\sensorless_foc\debug\is03_hardware_test.js”.
- The appropriate motor variables are now automatically populated into the watch window as shown in the following figure.
- The variables should look like **Figure 36**
    - Note the number format.
    - For example, if “motorVars.flagEnableSys” is displayed as a character, right- click on it and select “Number Format -> Decimal”

Expression	Type	Value	Address
motorVars.flagEnableSys	unsigned char	1 '\x01' (Decimal)	0x00000440@Data
motorVars.flagRunIdentAndOnLine	unsigned char	1 '\x01' (Decimal)	0x00000442@Data
motorVars.flagMotorIdentified	unsigned char	1 '\x01' (Decimal)	0x00000443@Data
motorVars.flagEnableRsRecalc	unsigned char	0 '\x00' (Decimal)	0x00000446@Data
motorVars.flagEnableUserParams	unsigned char	1 '\x01' (Decimal)	0x00000448@Data
motorVars.flagEnableOffsetCalc	unsigned char	0 '\x00' (Decimal)	0x00000449@Data
motorVars.flagEnableForceAngle	unsigned char	1 '\x01' (Decimal)	0x00000445@Data
motorVars.flagEnablePowerWarp	unsigned char	0 '\x00' (Decimal)	0x0000044A@Data
motorVars.flagBypassLockRotor	unsigned char	0 '\x00' (Decimal)	0x0000044B@Data
motorVars.flagEnableSpeedCtrl	unsigned char	1 '\x01' (Decimal)	0x0000044C@Data
motorVars.estState	enum <unnamed>	EST_STATE_ONLINE (Decimal)	0x0000045C@Data
motorVars.trajState	enum <unnamed>	EST_TRAJ_STATE_ONLINE (D...	0x0000045D@Data
motorVars.RoverL_rps	float	2714.96411	0x0000047C@Data
motorVars.Rr_Ohm	float	0.0	0x00000472@Data
motorVars.Rs_Ohm	float	0.380397081	0x00000474@Data
motorVars.Ls_d_H	float	0.000140111268	0x00000478@Data
motorVars.Ls_q_H	float	0.000140111268	0x0000047A@Data
motorVars.RsOnLine_Ohm	float	0.380397081	0x00000476@Data
motorVars.RoverL_rps	float	2714.96411	0x0000047C@Data
motorVars.flux_VpHz	float	0.0413461663	0x0000047E@Data
motorVars.speedRef_Hz	float	20.0	0x00000460@Data
motorVars.accelerationMax_Hzps	float	10.0	0x0000046A@Data
motorVars.speed_Hz	float	0.0	0x00000466@Data
motorVars.speed_krpm	float	0.0	0x00000468@Data
motorVars.torque_Nm	float	0.0	0x00000482@Data
motorVars.Vs_V	float	2.0	0x0000049A@Data
motorVars.VsRef_V	float	19.2000008	0x00000498@Data
motorVars.VdcBus_V	float	23.8337498	0x0000049C@Data
motorVars.Is_A	float	0.0	0x000004A2@Data
VsFreq	struct _VS_FREQ_Obj_	{maxVsMag_pu=0.5,Freq=2...	0x0000D250@Data
angleGen	struct _ANGLE_GEN_...	{fs_pu=20.0,angleDeltaFact...	0x0000D1AC@Data
pwmDACData	struct _HAL_PwmDa...	{periodMax=0,cmpValue=[0...	0x00000418@Data

Figure 36: Expressions watch window in project “is03_hardware_test”

The previous discussion has been an investigation and initialization of the volts/Hz control scheme employed in this lab. The steps to run the motor using the volts/Hz are now as follows:

1. With the motor disconnected, power on the EVM kit and import the lab “is03_hardware_test” project. From the previous write-up sections, follow the PWMDAC and DATALOG module instruction to monitor signals from either ANGEL_GEN or SVGEN modules.
    - Enable real-time debugger.  .
        - A dialog box will appear, select “Yes”.
    - Click the “Run” button  .
    - Enable “Continuous Refresh” on the watch window.  .
    - Set the variable “motorVars.flagEnableSys” equal to 1.
    - Set the variable “motorVars.flagRunIdentAndOnLine” equal to 1.

2. Power off the EVM kit, and follow the PWMDAC and DATALOG module instructions to now monitor the discussed ADC signals.
    - Set the variable “motorVars.flagRunIdentAndOnLine” to 0 to disable the PWM outputs.
    - Turn off “Real-Time Control” and stop the debugger.
    - Power off the EVM kit
3. Connect the motor and power on the EVM kit to now view the ADC signals
    - Enable real-time debugger again.
    - Set the variable “motorVars.flagEnableSys” and “motorVars.flagRunIdentAndOnLine” equal to 1 again.

- Set the variable “motorVars.speedRef_Hz” to a different value to ensure the motor runs smoothly.

4. Set “motorVars.flagRunIdentAndOnLine” and “motorVars.flagEnableSys” equal to 0 and power off the EVM kit when finished the experimenting.

Conclusion

Lab “is03_hardware_test” demonstrated how to implement a scalar volts/frequency control to test the integrity of the PWM and ADC hardware modules. Additionally, the PWMDAC and Datalog functions were used to verify the correct operation of the SVGEN, Park, and PWM modules.

is04_signal_chain_test – Current Closed Loop Control for Signal Chain Integrity Verification

Abstract

This lab implements a scalar current/frequency motor control technique to test the signal chain integrity—mainly the hardware current/voltage sensing and controller ADC module. Additionally, lab "is04_signal_chain_test" employs a closed current loop, open speed loop PI control topology. While compatible with Texas Instruments' hardware, this lab is intended for custom hardware verification.

Introduction

Lab "is04_signal_chain_test" shows an example without using the FAST estimator. This lab uses an angle generator module to generate the angle based on target frequency of the motor, and a closed current loop to control Id and Iq to run a motor. The objective is to test the InstaSPIN-FOC modules through current closed loop control without rotor position information. Tested software modules include PI, PWM, ADC, CLARK, PARK I-PARK and SVGEN.

Prerequisites

Assumes knowledge of up to the lab "is03_hardware_test".

Objectives Learned

- How to implement a current closed loop control of motor without estimator angle.
- How to test ADC sampling and some InstaSPIN-FOC modules

PWMDAC

As in lab "is03_hardware_test", this lab "is04_signal_chain_test" also uses the PWMDAC module to view critical FOC signals from the ADC or PI modules.

Step 1: Follow the same instructions for step 1 as shown in the lab project "is04_signal_chain_test" guide.
Step 2: Set the correct offset and gain for each PWMDAC channel as shown below.

```
    pwmDACData.ptrData[0] = &pi_Iq.refValue;
    pwmDACData.ptrData[1] = &pi_Iq.fbackValue;
    pwmDACData.ptrData[2] = &pi_Id.refValue;
    pwmDACData.ptrData[3] = &pi_Id.fbackValue;

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

    pwmDACData.gain[0] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[1] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[2] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
    pwmDACData.gain[3] = 1.0/USER_ADC_FULL_SCALE_CURRENT_A;
```

Step 3: Follow the same instructions for step 2 as shown in the lab project “is04_signal_chain_test” guide.

In lab “is04_signal_chain_test”, we connect current reference and feedback variables to the PWMDAC module and monitor the output waveforms as shown in **Figure 37**.

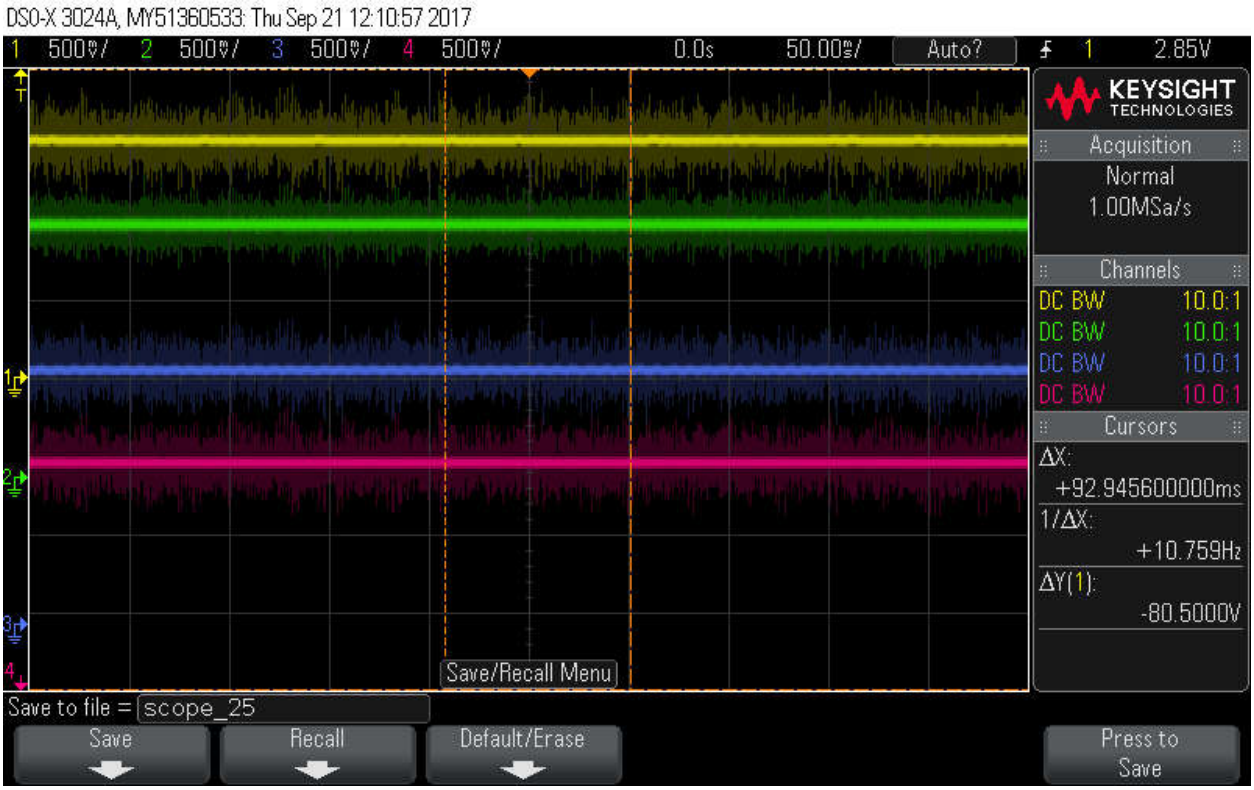


Figure 37: Id and Iq reference and feedback current waveforms

Where

Ch1-> PWMDAC1: Iq reference
Ch2-> PWMDAC2: Iq feedback
Ch3-> PWMDAC3: Id reference
Ch4-> PWMDAC4: Id feedback

DATALOG_DMA

As in lab project “is03_hardware_test”, the lab project “is04_signal_chain_test” uses the datalog and graph tools to monitor the input/output waveforms of this project.

Step 1: Select “Tools->Graph->Dual time” in the CCS menu. Choose “Import” and select “..\solutions\common\ sensorless_foc \debug->is04_signal_chain_test_d12.graphProp”

Step 2: Select “Tools->Graph->Dual time” in the CCS menu. Choose “Import” and select “..\solutions\common\ sensorless_foc \debug->is04_signal_chain_test_d34.graphProp”

Step 3: Connect the inputs of the datalog module as shown in the code snippet below. The datalog graphs for Id and Iq PI reference and feedback signals are shown in **Figure 38**.

```
// set datalog parameters
   datalogObj->iptr[0] = &pi_Iq.refValue;
   datalogObj->iptr[1] = &pi_Iq.fbackValue;
   datalogObj->iptr[2] = &pi_Id.refValue;
   datalogObj->iptr[3] = &pi_Id.fbackValue;
```

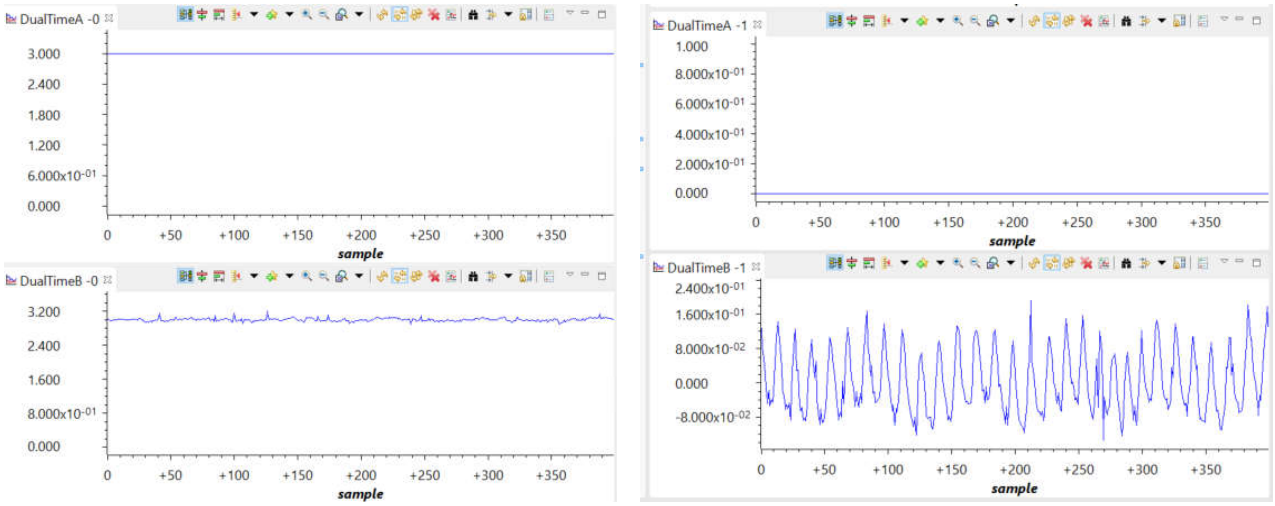


Figure 38: Id and Iq reference and feedback current datalog waveforms

Project Files

No new project files in the lab project “is04_signal_chain_test” compared to the lab project “is03_hardware_test”.

Includes

No new includes in files in the lab project “is04_signal_chain_test” compared to the lab project “is03_hardware_test”.

Global Object and Variable Declarations

There are no new global object and variable declarations.

Initialization and Setup

Nothing has changed in initialization and setup from the previous project “is04_signal_chain_test”.

Background

The background loop makes use of functions that allow user interaction with the FOC software.

mainISR

A block diagram of the mainISR used in lab project “is04_signal_chain_test” is shown in **Figure 39.**

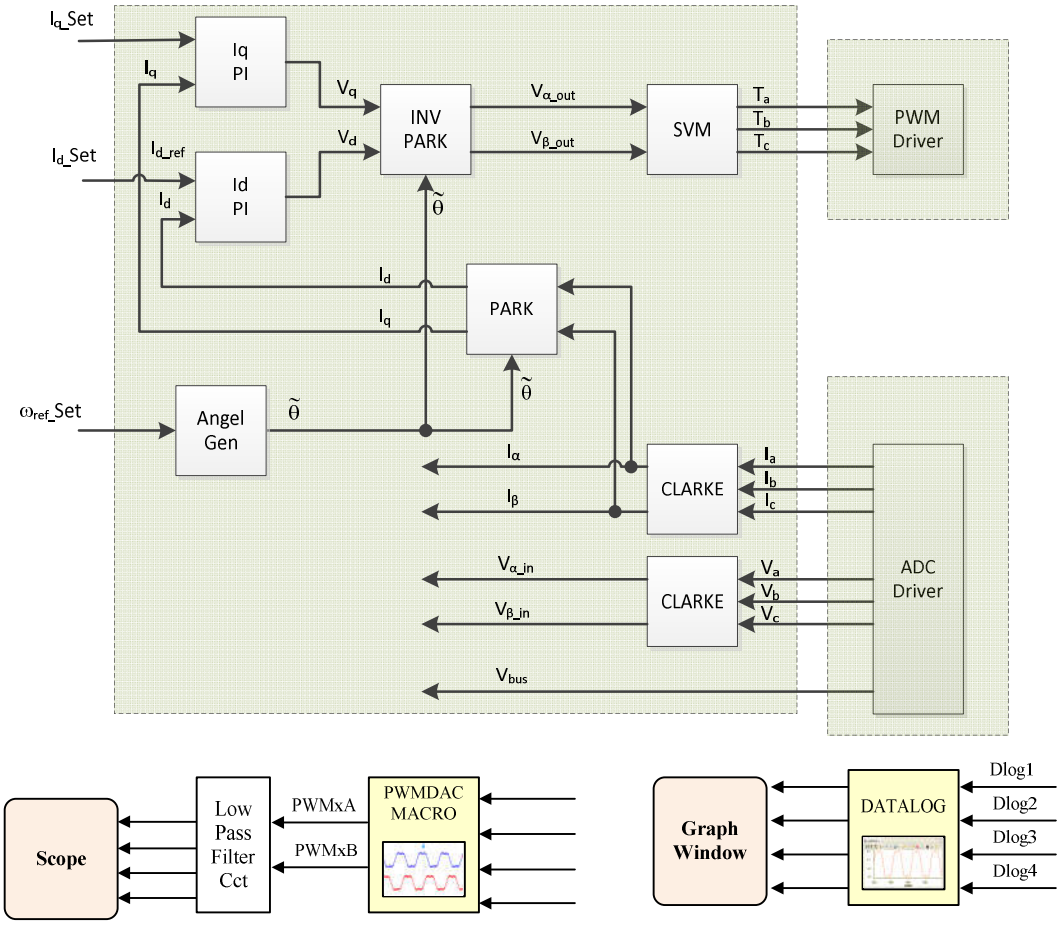


Figure 39: Block diagram of scalar motor control technique with current closed loop

Lab Procedure

**Please note: It is possible to experience motor vibration while running this lab; as this lab is intended for hardware verification, it is not an issue to see this effect.

Step 1: Connect the motor and power on the EVM kit. Start the lab as shown in previous write-ups, and follow the PWMDAC and DATALOG instructions to monitor signals from the SVGEN module.

- Enable the real-time debugger.  .
  - A dialog box will appear, select “Yes”.
- Click the run button.  .
- Enable continuous refresh on the watch window.  .

- Set the variable “motorVars.flagEnableSys” equal to 1.
- Set the variable “motorVars.flagRunIdentAndOnLine” equal to 1.
- Set the variables “motorVars.speedRef_krpm”, “motorVars.IdSet_A” and “motorVars.IqSet_A” to different values to ensure the motor runs smoothly.

Step 2: Power off the EVM kit, change the PWMDAC and DATALOG configurations to monitor signals from ANGEL_GEN and ADC modules instead of SVGEN.

Step 3: Once done experimenting with the scalar control and are satisfied with the performance of the signal chain, disable the debugger to end the lab.

- Set the variable “motorVars.flagRunIdentAndOnLine” equal to 0 to disable the PWM output
- Turn off real-time control and stop the debugger
- Power off EVM kit

Conclusion

Lab “is04_signal_chain_test” demonstrated how to implement a current closed loop scalar control to test the integrity of the signal chain hardware. The intended result is verification of the signal chain hardware abstraction layer (HAL) setup, specifically the ADC module for current sensing. While functional for existing Texas Instruments’ EVMs, this lab should mainly be used to verify ADC functionality on custom hardware not provided in the MotorControl SDK. Additionally, the PWMDAC and Datalog functions were used to verify the PI, PWM, ADC, CLARK, PARK, I-PARK and SVGEN modules.

is05_motor_id – Motor Parameters Identification

Abstract

Motor parameter identification is an InstaSPIN-FOC feature that allows the identification of the parameters needed by the estimator to sensorless control the motor during closed loop operation. The motor identification feature of InstaSPIN enables users to run their motor to its highest performance, even when motor parameters are unknown.

Introduction

InstaSPIN-FOC utilizes Texas Instruments' FAST technology to create a self-sensored, field oriented motor controller, offering sensorless estimation and cascaded FOC and speed control loops. MotorControl SDK InstaSPIN labs have been developed to showcase the various features of the InstaSPIN-FOC estimator from on-chip ROM. All other FOC modules, besides the FAST estimator, are executable from RAM/FLASH, and are open source available. The library is very robust and can be customized for many different applications. Lab "is05_motor_id" will demonstrate what enables the motor parameter identification as well as how to start the motor.

Objectives Learned

- Include FOC open source code and call the API functions to set up the sensorless FOC system.
- Setup the user.h file for the motor and inverter.
- Start the automatic motor parameter estimation.
- Update user.h for your motor using the identified parameters.
- How the FAST observer is initialized and set up.
- How to run the FAST observer.

Background

Lab "is05_motor_id" adds the function calls necessary for identifying and running a motor. The block diagram of **Figure 40** shows both the FAST estimator functionality of the ROM library, and the InstaSPIN-FOC modules executed in user memory. Lab "is05_motor_id" follows this block diagram to implement a full sensorless FOC drive with FAST estimator and InstaSPIN-FOC function calls.

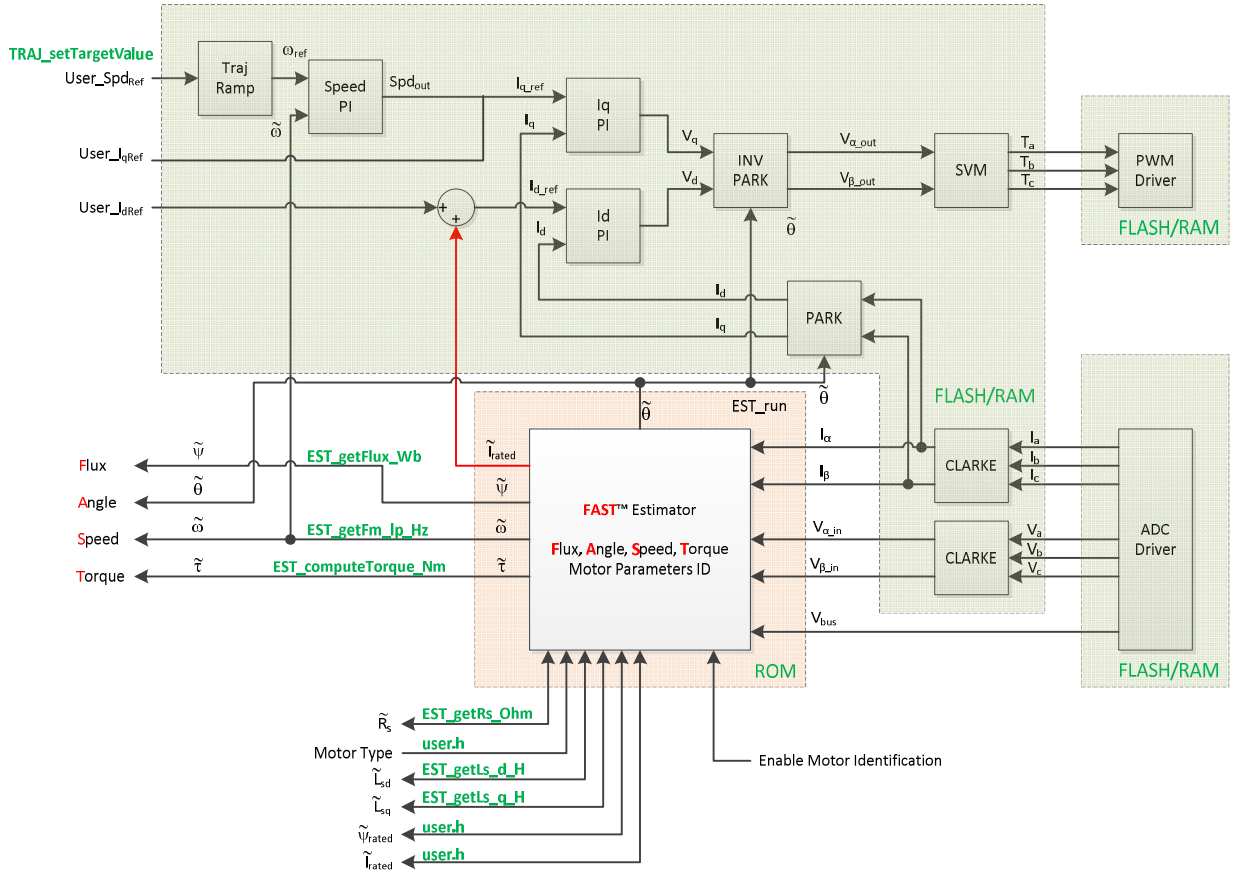


Figure 40: Block diagram of FAST in ROM with the rest of InstaSPIN-FOC in user memory

Project Files

Lab project “is05_motor_id” adds new InstaSPIN-FOC source files when compared to the project “is04_signal_chain_test”, which are shown in **Table 12**:

is05_motor_id		
	ctrl.c	Contains code for CTRL_run and CTRL_setup, the module that runs the FOC.
	traj.c	Contains code for creating ramp functions.

Table 12: New files included in project “is05_motor_id”

Includes

A description of the new included files critical for InstaSPIN setup is shown in the figure below. Note that labs.h is common across all labs so there will be more includes in labs.h than are needed for this lab as shown in **Table 13**.

labs.h	Header file containing all included files used in main.c	
	modules	
	math.h	Common math conversions, defines, and shifts
	est.h	Function definitions for the FAST ROM library
	platforms	
	ctrl.h	Function definitions for the CTRL ROM library. Contains the CTRL object declaration.
	hal.h	Device setup and peripheral drivers. Contains the HAL object declaration.
	user.h	User file for configuration of the motor, drive, and system parameters.

Table 13: Important header files needed for motor parameter identification

Global Object and Variable Declarations

Global object and declarations that are listed in the table below are only the objects that are absolutely needed for the motor controller. Other object and variable declarations are used for display or information for the purpose of this lab as shown in **Table 14**.

globals		
	CTRL	
	CTRL_Handle	The handle to a controller object (CTRL). The controller object implements all of the FOC algorithms and calls the FAST observer functions.
	OTHER	
	MOTOR_Vars_t	Not needed for the implementation of InstaSPIN but in the project this structure contains all of the flags and variables to turn on and adjust InstaSPIN. The structure defined by this declaration will be put into the CCS watch window.

Table 14: Global object and variable declarations necessary for motor parameter identification

Initialization and Setup

This section covers functions needed to set up the microcontroller and the FOC software. Only the functions that are mandatory will be listed in the table below. Functions that are not listed in **Table 15** are in the project for enhanced capability of the laboratory and not fundamentally needed to set up the motor control. For a more in-depth explanation for definitions of the parameters and return values go to the document motor control section of this document (InstaSPIN-FOC User's Guide).

setup		
	CTRL	
	CTRL_initCtrl	Initializes all handles required for field oriented control and FAST observer interface. Returns a handle to the CTRL object.
	CTRL_setParams	Copies all scale factors that are defined in the file user.h and used by CTRL into the CTRL object.
	EST_initEst	Initialize the handle for estimator
	EST_setParams	Set the default estimator parameters

Table 15: Important setup functions needed for the motor control estimator and controller

Main Run-Time loop (forever loop)

The background loop makes use of functions that allow user interaction with the FAST observer and FOC software. **Table 16** lists the important functions used. The flowchart of **Figure 41** shows the logic of the background (forever) loop. The flowchart block labeled "Update Global Variables" is used to update variables such as speed, stator resistance, inductance, etc.

	TRAJ_setMaxDelta	Sets the maximum acceleration rate of the speed reference.
	HAL	
	HAL_enablePWM	Turns on the outputs of the EPWM peripherals which will allow the power switches to be controlled.
	HAL_disablePWM	Turns off the outputs of the EPWM peripherals which will put the power switches into a high impedance state.
	EST	
	EST_enable	Enables the FAST estimator
	EST_enableTraj	Enables the trajectory generator for estimator
	EST_updateState	Updates the estimator state
	EST_configureCtrl	Configures the controller for each of the estimator states

Table 16: Functions used for error checking and setup of the InstaSPIN controller and FAST observer

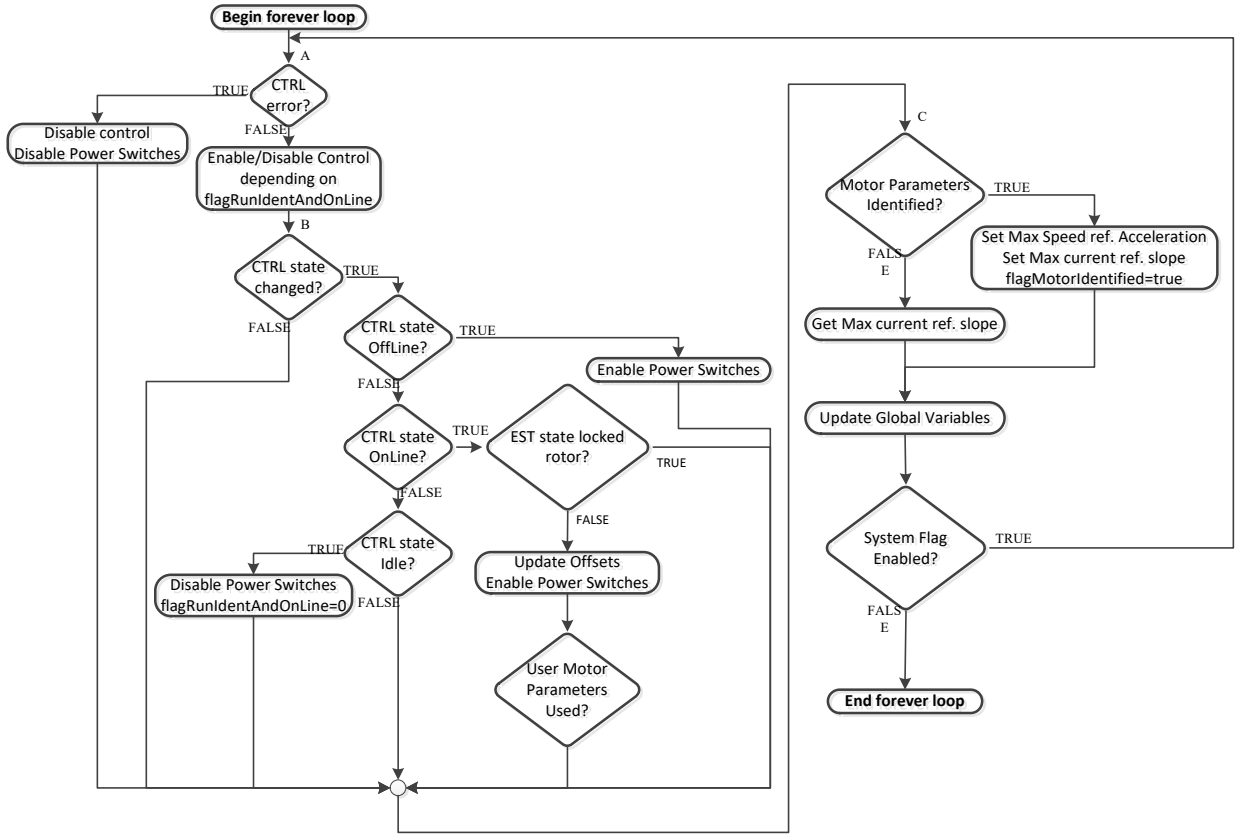


Figure 41 : Motor Identification forever loop flowchart

Main ISR

The main ISR calls time critical functions that run the FOC and FAST observer. The new functions required for project "is05_motor_id" are listed in **Table 17** below.

mainISR		
	CTRL	
	EST_setupTraj	Sets up the trajectory generator
	EST_runTraj	Runs the trajectory generator
	EST_run	Runs the estimator
	CTRL_run	Runs the controller
	CTRL_setup	Is responsible for updating the CTRL state machine and must be called in the same timing sequence as CTRL_run().

Table 17: InstaSPIN functions called in the main ISR

Lab Procedure

The code for lab "is05_motor_id" is set up according to the flowchart shown in **Figure 41**. The first step when running a motor with InstaSPIN is to fill the library with nameplate data from the motor. As such, the first topic that needs to be covered before running any motor with InstaSPIN is the "user.h" header file.

Open “user.h” following these steps:

5. Expand “src_fast->user.c” from the “Project Explorer” window
6. Right-click on “user.c” and select “Open”, this opens the file “user.c”
7. Right-click on the highlighted “user.h” and select “Open Declaration,” this opens the file “user.h”.
8. Opening the “Outline View” will provide an outline of the “user.h” contents as shown in **Figure 42**.

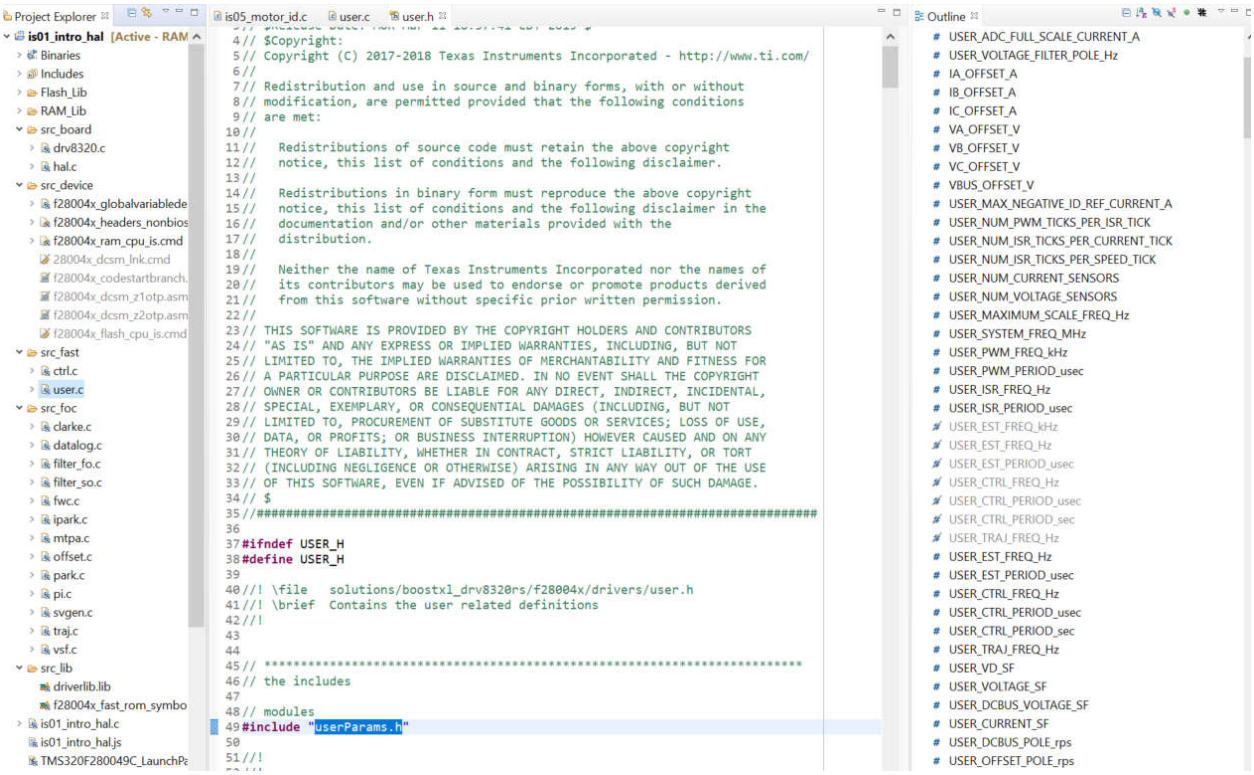


Figure 42: the content of “user.h” header file in Outline

Alternatively, open “user.h” from the directory in MotorControl SDK as shown in **Figure 43**.

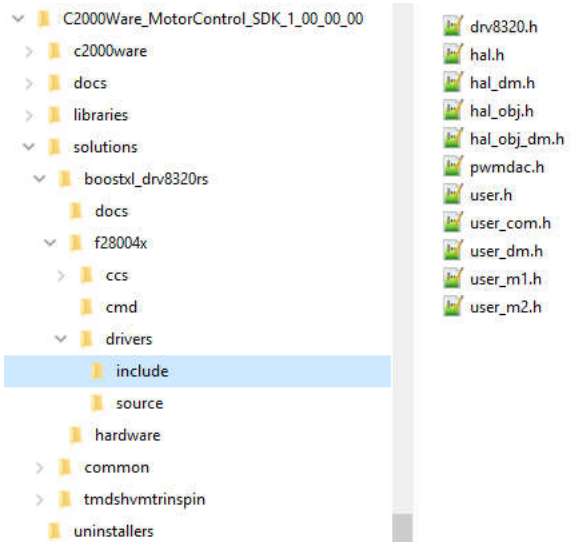


Figure 43: Directory structure for the “user.h” header file

Halfway through the “user.h” file, there is a section containing definitions of motor parameters. The section of code starts with the name “*USER MOTOR & ID SETTINGS*.” To customize this file, a new motor definition must be created; for now, name the new definition “My_Motor.”

To define a new motor, add a line with a unique number:

```
#define my_motor_1                  301
```

Comment out the current motor definition by placing a double slash “//” before the code, which will look like the following:

```
#define USER_MOTOR teknic_2310P
```

Then add the following line instead:

```
#define USER_MOTOR my_motor_1
```

For the actual motor parameters, copy and paste an empty set of motor parameter definitions (see “my_motor_1” for PMSM placeholder) and convert them as below if it is a PMSM, IPM or BLDC motor:

```
#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               (NULL)
#define USER_MOTOR_Ls_d_H               (NULL)
#define USER_MOTOR_Ls_q_H               (NULL)
#define USER_MOTOR_RATED_FLUX_VpHz      (NULL)
#define USER_MOTOR_MAGNETIZING_CURRENT_A (NULL)
#define USER_MOTOR_RES_EST_CURRENT_A    (2.0)
#define USER_MOTOR_IND_EST_CURRENT_A    (-2.0)
#define USER_MOTOR_MAX_CURRENT_A        (5.0)
#define USER_MOTOR_FLUX_EXC_FREQ_Hz     (40.0)
```

And as below if it is an ACIM motor (see “my_motor_2” for ACIM placeholder):

```
#define USER_MOTOR_TYPE                 MOTOR_TYPE_INDUCTION
#define USER_MOTOR_NUM_POLE_PAIRS       (2)
#define USER_MOTOR_Rr_Ohm               (NULL)
#define USER_MOTOR_Rs_Ohm               (NULL)
#define USER_MOTOR_Ls_d_H               (NULL)
#define USER_MOTOR_Ls_q_H               (NULL)
#define USER_MOTOR_RATED_FLUX_VpHz      (0.8165*230.0/60.0)
#define USER_MOTOR_MAGNETIZING_CURRENT_A (NULL)
#define USER_MOTOR_RES_EST_CURRENT_A    (0.5)
#define USER_MOTOR_IND_EST_CURRENT_A    (NULL)
#define USER_MOTOR_MAX_CURRENT_A        (5.0)
#define USER_MOTOR_FLUX_EXC_FREQ_Hz     (5.0)
```

A few values can already be populated in “user.h” before motor parameter identification.

- `USER_MOTOR_TYPE = MOTOR_TYPE_PM` or `MOTOR_TYPE_INDUCTION` → Motor type must be known and entered in this parameter.
- `USER_MOTOR_NUM_POLE_PAIRS` → Number of pole pairs of the motor
- `USER_MOTOR_MAX_CURRENT` → Maximum nameplate current of the motor
- `USER_MOTOR_RES_EST_CURRENT` → The motor will have to initially be started in open loop during identification. This value sets the peak of the current used during initial startup of the motor. If the motor has high cogging torque or some kind of load, increase this current value until the motor will start spinning. After motor identification, this value is never used.
- `USER_MOTOR_IND_EST_CURRENT` → Must be zero for ACIM motors. For PMSM motors this value can be set to the negative of the current used for `USER_MOTOR_RES_EST_CURRENT`. For example, if `USER_MOTOR_RES_EST_CURRENT` is 1.0, then `USER_MOTOR_IND_EST_CURRENT` can be -1.0.
- `USER_MOTOR_NUM_POLE_PAIRS` → Number of pole pairs of the motor
- `USER_MOTOR_RATED_FLUX` → Must be zero for PMSM motors. For ACIM motors the rated flux should be set to nameplate values calculated as follows:
  `USER_MOTOR_RATED_FLUX = SQRT(2)/SQRT(3)*Rated_VAC/Rated_F`
  So for a 220VAC motor with a rated frequency of 60 Hz, then the rated flux would be:
  `USER_MOTOR_RATED_FLUX = SQRT(2)/SQRT(3)*220.0/60.0 = 2.9938`
- `USER_MOTOR_FLUX_EST_FREQ_Hz` → A starting point for this frequency if the motor is a PMSM motor is 20.0 Hz, and if it is an ACIM motor, a good starting point is 5.0 Hz.

Later in the lab, after the motor parameters are identified, the appropriate NULL values will be updated with the identified values. One thing to note is that if the motor is defined to be a permanent magnet motor, the terms “Magnetizing Current” and “Rr” are not needed and therefore will always be left NULL. Also, note that the inverter has already been defined. In the top half of the “user.h” file, there are definitions for currents and voltages, clocks and timers, and poles. These definitions are used to set up current, voltage scaling, and filter parameters for the library.
Now, connect the motor that will be run with InstaSPIN to the kit. Connect the USB cable to the controlCARD or LaunchPad. Finally, apply power to the kit. In Code Composer Studio, build lab project “is05_motor_id”. Start a Debug session and download the “is05_motor_id.out” file to the MCU.

A structure containing the variables to run this lab from the Code Composer “Real-Time Watch Window” has been created by the name of “motorVars” and is defined in “labs.h.” A script has been written to easily add these variables to the watch window as shown in **Figure 44**.

- Select the scripting tool, from the debugger menu “View->Scripting Console”.
- The scripting console window will appear somewhere in the debugger.
- Open the script by clicking the icon that is in the upper right corner of the scripting tool.
- Select the file “\solutions\common\sensorless_foc\debug\is05_motor_id.js”.
- The appropriate motor variables are now automatically populated into the watch window as shown in the following figure.
- The variables should look like below
  - Note the number format.
  - For example, if “motorVars.flagEnableSys” is displayed as a character, right-mouse click on it and select “Number Format -> Decimal”
- Enable the real-time debugger. .
  - A dialog box will appear, select “Yes”.
- Click the run button .
- Enable continuous refresh on the watch window. .

To start the motor identification,

- Set the variable “motorVars.flagEnableSys” equal to 1.
- Set the variable “motorVars.flagRunIdentAndOnLine” equal to 1.

Expression	Type	Value	Address
motorVars.flagEnableSys	unsigned char	1 '\x01' (Decimal)	0x00000440@Data
motorVars.flagRunIdentAndOnLine	unsigned char	0 '\x00' (Decimal)	0x00000442@Data
motorVars.flagMotorIdentified	unsigned char	1 '\x01' (Decimal)	0x00000443@Data
motorVars.flagEnableRsRecalc	unsigned char	0 '\x00' (Decimal)	0x00000446@Data
motorVars.flagEnableUserParams	unsigned char	0 '\x00' (Decimal)	0x00000448@Data
motorVars.flagEnableOffsetCalc	unsigned char	0 '\x00' (Decimal)	0x00000449@Data
motorVars.flagEnableForceAngle	unsigned char	1 '\x01' (Decimal)	0x00000445@Data
motorVars.flagEnablePowerWarp	unsigned char	0 '\x00' (Decimal)	0x0000044A@Data
motorVars.flagBypassLockRotor	unsigned char	0 '\x00' (Decimal)	0x0000044B@Data
motorVars.flagEnableSpeedCtrl	unsigned char	1 '\x01' (Decimal)	0x0000044C@Data
motorVars.estState	enum <unnamed>	EST_STATE_IDLE (De...	0x0000045C@Data
motorVars.trajState	enum <unnamed>	EST_TRAJ_STATE_ID...	0x0000045D@Data
motorVars.RoverL_rps	float	2142.17798	0x0000047C@Data
motorVars.Rr_Ohm	float	0.0	0x00000472@Data
motorVars.Rs_Ohm	float	0.404352903	0x00000474@Data
motorVars.Ls_d_H	float	0.000201336734	0x00000478@Data
motorVars.Ls_q_H	float	0.000201336734	0x0000047A@Data
motorVars.RsOnLine_Ohm	float	0.404352903	0x00000476@Data
motorVars.RoverL_rps	float	2142.17798	0x0000047C@Data
motorVars.flux_VpHz	float	0.040826235	0x0000047E@Data
motorVars.magneticCurrent_A	float	0.0	0x00000470@Data
motorVars.speedRef_Hz	float	20.0	0x00000460@Data
motorVars.accelerationMax_Hzps	float	10.0	0x0000046A@Data
motorVars.speed_Hz	float	40.0754433	0x00000466@Data
motorVars.speed_krpm	float	0.601131618	0x00000468@Data
motorVars.torque_Nm	float	0.0245898385	0x00000482@Data
motorVars.Vs_V	float	0.658133805	0x0000049A@Data
motorVars.VsRef_V	float	19.2000008	0x00000498@Data
motorVars.VdcBus_V	float	23.7851753	0x0000049C@Data
motorVars.Is_A	float	0.0	0x000004A2@Data

Figure 44: Expressions Watch Window after running the lab project “is05_motor_id” variable script

The controller will now start identifying the motor. Be sure not to try to stop the shaft of the motor while identification is running or else there will be inaccurate identification results. Once the

"motorVars.flagRunIdentAndOnLine" is equal to 0, and we are identifying a PMSM motor, the motor parameters have been identified. If we are identifying an ACIM motor, the controller and estimator states will show the following state:

If this is the case, then lock the rotor, and enable the controller again by setting "motorVars. flagEnableRunAndIdentify " to 1. Once "motorVars. flagEnableRunAndIdentify" is equal to 0, and then the ACIM is done identifying.

- Record the watch window values with the newly defined motor parameters in user.h as follows:
  - `USER_MOTOR_Rr` = motorVars.Rr_Ohm's value (ACIM motors only)
  - `USER_MOTOR_Rs` = motorVars.Rs_Ohm's value
  - `USER_MOTOR_Ls_d` = motorVars.Ls_d_H's value
  - `USER_MOTOR_Ls_q` = motorVars.Ls_q_H's value
  - `USER_MOTOR_RATED_FLUX` = motorVars.flux_VpHz's value
  - `USER_MOTOR_MAGNETIZING_CURRENT` = motorVars.magneticCurrent_A's value (ACIM motors only)

The motor is not energized anymore. If an ACIM was identified, remove whatever instrument was used to lock the rotor at this time. To run the motor,

- Set the variable "motorVars.flagRunIdentAndOnLine" equal to 1 again.

The control will re-calibrate the feedback offsets and then re-measure Rs_Ohm. After the measurements are done, the motor shaft will accelerate to the default target speed. The speed feedback is shown (in electrical frequency with Hertz) by the variable "motorVars.speed_Hz". The target speed reference is (in electrical frequency with Hertz) set by the variable "motorVars.speedRef_Hz".

- Set the variable "motorVars.speedRef_Hz" to a different value and watch how the motor shaft speed will follow.

Notice that when changing between speeds, the motor shaft speed does not change instantaneously. An acceleration trajectory is set up between the input speed reference and the actual speed reference commanding the input of the speed PI controller.
To change the acceleration,

- Enter a different acceleration value for the variable "motorVars.accelerationMax_Hzps".

When done experimenting with the motor,

- Set the variable "motorVars.flagRunIdentAndOnLine" to 0 to disable the PWM output
- Turn off real-time control and stop the debugger.

API functions used to interface with the variables in the watch window during this lab are shown in **Table** .

updateGlobalVariables		
	TRAJ	
	TRAJ_setTargetValue	Sets the output speed reference value in the controller in Hz.
	TRAJ_setMaxDelta	Sets the maximum acceleration rate of the speed reference in Hzps.
	EST	
	EST_getFm_lp_Hz	Gets the speed value in Hz.
	EST_computeTorque_Nm	Gets the torque value in N.m.
	EST_getRs_Ohm	Gets the stator resistance value in Ohms.
	EST_getLs_d_H	Gets the direct stator inductance value in Henries (H)
	EST_getLs_q_H	Gets the stator inductance value in the quadrature coordinate direction in Henries (H)
	EST_getFlux_Wb	The estimator continuously calculates the flux linkage between the rotor and stator, which is the portion of the flux that produces torque. This function returns the flux linkage, ignoring the number of turns, between the rotor and stator coils, in Volts per Hertz, or V/Hz.
	EST_getState	Gets the state of the estimator.

Table 18: Functions used to interface with the watch window variables during lab project “is05_motor_id”

Conclusion

Lab “is05_motor_id” has demonstrated the basics of using the FAST estimator and InstaSPIN-FOC library. A new motor has been identified and the values were entered into the file user.h. The recorded motor parameters will be used in the following labs to bypass motor commissioning and speed up the initial startup of the motor.

is06_torque_control – Torque Control Mode and Tuning Id/Iq PI Controller

Abstract

A technique of setting the proportional gain (Kp) and integral gain (Ki) for the current controllers of the InstaSPIN-FOC system is explored in this lab. After the Kp and Ki gains are calculated, we will then learn how to program InstaSPIN with these values.

Introduction

This lab explains how to tune PI gains for current control when controlling an electric motor using InstaSPIN-FOC

Objectives Learned

- How to calculate the PI gains for the current controller.
- Program the Kp and Ki gains into InstaSPIN.

Background

A popular form of the proportional-integral (PI) controller (and the one used for this lab) is the "series" topology which is shown in **Figure** .

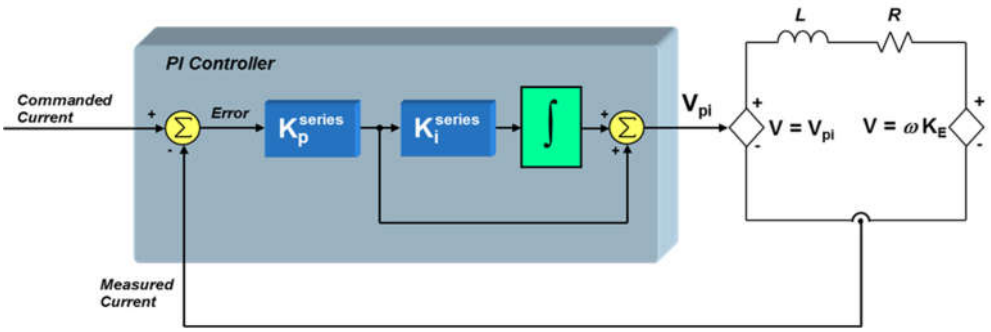


Figure 45: Series PI current controlled motor system including the stator.

PI and PID loops are a fundamental system explored in classic control theory, and many publications exist that expand on the topic. Additionally, PI controllers are integral parts of FOC-based motion control. InstaSPIN-FOC depends on a nested combination of a PI current controller and a PI speed controller. The PI speed controller will be discussed in lab 06. With respect to this lab, K_i^{series} sets the zero of the PI controller, and K_P^{series} sets the bandwidth of the closed-loop system response, with respective equations shown in **Equation 4** and **Equation 19**.

$$K_i^{series} = \frac{R}{L} \qquad \text{Equation 4}$$

$$G(s) = \frac{1}{\frac{L}{K_p^{series}}s + 1} \rightarrow K_p^{series} = L * Bandwidth$$

Equation 5

Project Files

There are no new project files.

Includes

There are no new includes.

Global Object and Variable Declarations

There are no new global object and variable declarations.

Initialization and Setup

There are no new initialization and setup operations. The block diagram of this lab is shown in **Figure 46.**

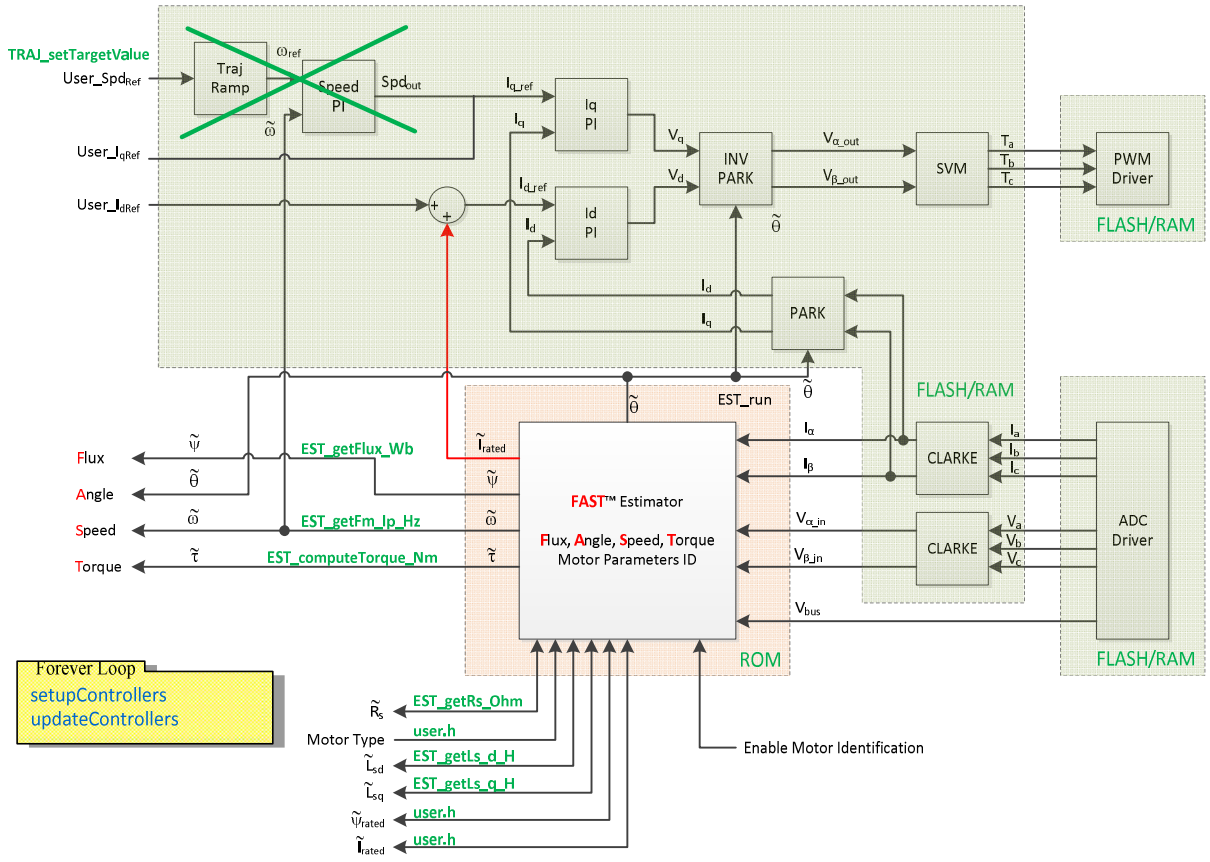


Figure 46: Block diagram of an opened source InstaSPIN-FOC implementation

Main Run-Time loop (forever loop)

During motor identification or when motor parameters from "user.h" are used, the K_i^{series} gain is calculated based on the motor R/L pole. The bandwidth of the controller, or K_p^{series}, is set to not be too high and cause instability in the current control loop. Immediately after motor identification has finished, the K_p^{series} and K_i^{series} gains for the current controller are calculated in the function setupControllers(). After setupControllers() is called, the global variables motorVars.Kp_Id, motorVars.Kp_Iq, motorVars.Ki_Id, and motorVars.Ki_Iq are initialized with the newly calculated K_p^{series} and K_i^{series} gains. The following **Figure 47** shows the logic flowchart needed to implement the current controller gain initialization.

Forever Loop		
	setupControllers	Calculates the speed and current PI controller gains based on motor parameters
	updateControllers	Sets the custom speed and current PI controller gains

Table 19: New API function calls during the main run-time loop

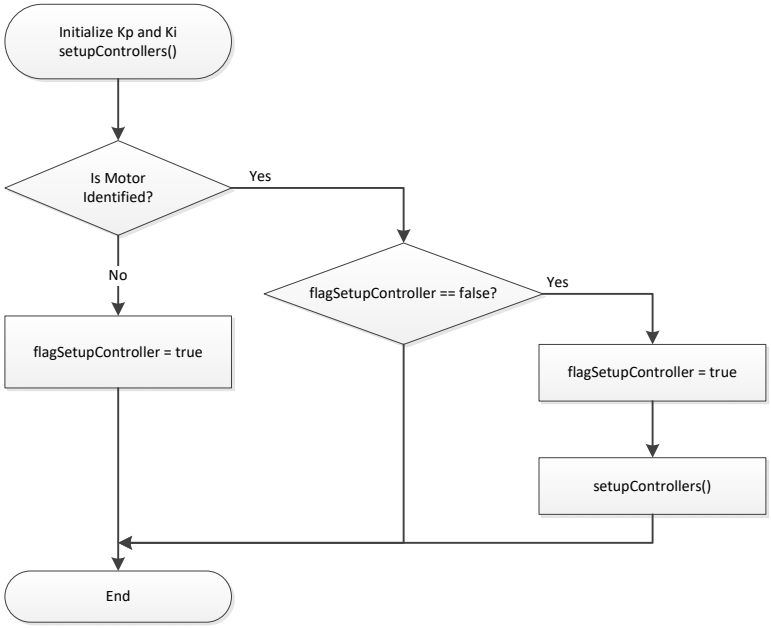


Figure 47: Flowchart showing how the watch window K_p^{series} and K_i^{series} variables are initialized.

Main ISR

Nothing has changed in this section of the code from the previous lab.

Lab Procedure

Build lab project "is06_torque_control", connect to the target and load the .out file.

- Add the appropriate watch window variables by calling the script "is06_torque_control.js".
- Enable real-time debugger.

- Click the run button.
- Enable continuous refresh on the watch window.

Calculate the K_i^{series} gain using the relationship: $K_i^{series} = R/L$.

- Record the Rs and Ls values that are stored in "user.h" for the motor being tested.
- Record the sampling frequency from "user.h," listed as USER_PWM_FREQ_Khz.
- K_i^{series} has to be per unitized to $K_i^{series}(PU)$

Calculate current controller period:

$$T_i = \frac{1}{PWM\_Freq\_kHz \cdot 1000} \cdot PWMvsISRtick \cdot ISRvsCTRLtick \cdot CTRLvsCURRENTtick$$

Where:

- T_i is the current controller period
- PWM_Freq_kHz can be taken from USER_PWM_FREQ_kHz parameter in user.h
- $PWMvsISR\ tick$ is the tick rate between PWM and interrupts, USER_NUM_PWM_TICKS_PER_ISR_TICK
- $ISRvsCURRENT\ tick$ is the tick rate between interrupts and current control, USER_NUM_ISR_TICKS_PER_CURRENT_TICK

Calculate the $K_i^{series}(PU) = (Rs/Ls)Ti$.

Start the control

- Set motorVars.flagEnableSys = true
- Set motorVars.flagRunIdentAndOnLine = true

Compare K_i^{series} values

- Compare the calculated K_i^{series} against the K_i^{series} terms that are initially stored in "motorVars.Ki_Id" and "motorVars.Ki_Iq." The two values should be the same or both Iq and Id.

Compare and adjust K_p^{series} values

- Since the K_p^{series} gain controls bandwidth of the controller, adjustment of this parameter is optimized when knowing the mechanics of the whole motor system and the required system time response. For this experiment, we will show effective ranges to adjust the K_p^{series} gain.
- Calculate the K_p^{series} gain based on the ISR frequency.
  - Use a bandwidth that is 1/20 of the current controller frequency. Keep in mind that the bandwidth needed to calculate the controller gains is in radians per second, and the controller frequency is in Hz, so the following conversion is used:
    $$Bandwidth(\frac{rad}{s}) = 2\pi \cdot CurrentControllerFreq(Hz) \cdot \frac{1}{20}$$
  - Example: $\frac{1}{T_i} = 10\text{kHz}$, $L_s = 620\mu\text{H}$
  - $K_p^{series} = 0.00062 \cdot 2\pi \cdot (10000/20) = 1.95$
  - K_p^{series} has to be per unitized to $K_p^{series}(PU)$
  - $K_p^{series}(PU) = K_p^{series} \cdot I_{fs}/V_{fs}$
    - $I_{fs} = \text{USER\_IQ\_FULL\_SCALE\_CURRENT\_A}$ → found in "user.h"

- V_{fs} = USER_IQ_FULL_SCALE_VOLTAGE_V → found in "user.h"

- Put your new calculated K_p^{series} into motorVars.Kp_Id and motorVars.Kp_Iq.

As the bandwidth increases, the sampling delay more negatively affects the phase margin of the controller, causing the control loop to become unstable. **Figure 48** is a plot of the motor current waveform with a stable K_p^{series} setting. As K_p^{series} increases, the phase margin of the control loop becomes smaller. After a while, the control loop becomes unstable and starts to oscillate as shown in **Figure 49**. The current controller gain should not be set to this high of a value. When current loop instability occurs, lower the K_p^{series} gain until the current waveform is like the one in the following figure.

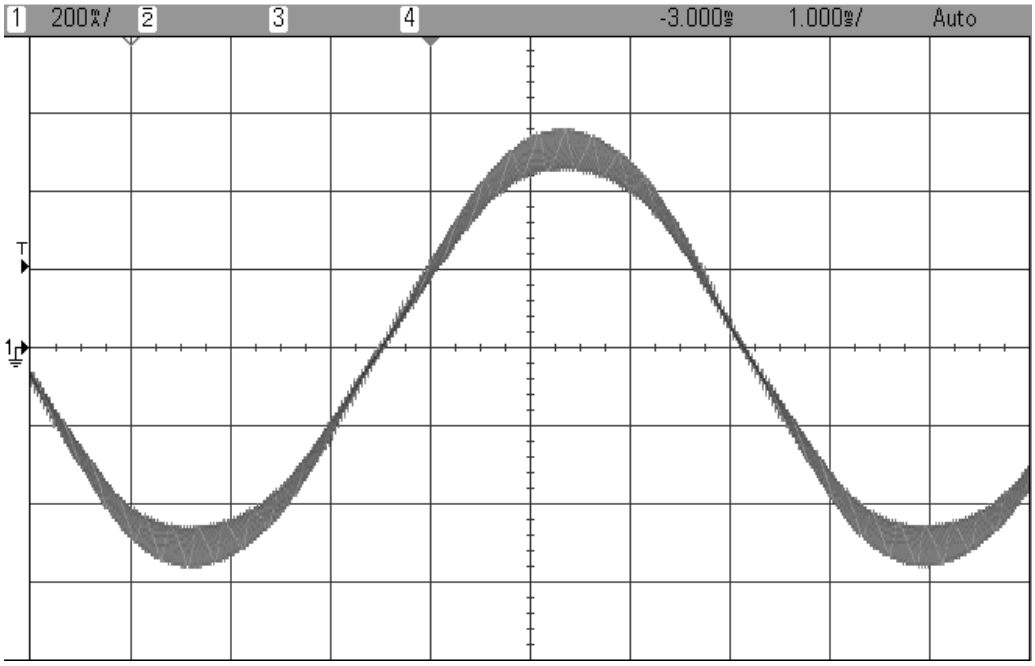


Figure 48: K_p^{series} setting that has been calculated at 1/20th of the bandwidth.

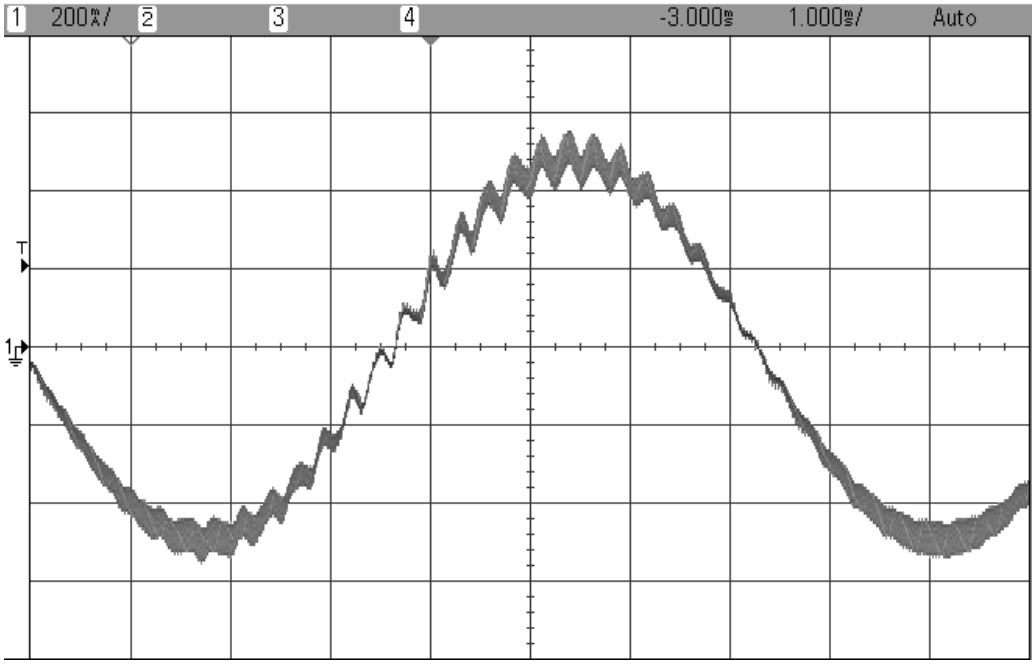


Figure 49: K_p^{series} setting that is too high, resulting in the controller becoming unstable.

Increasing the K_p^{series} gain and bandwidth

- Start with the K_p^{series} gain set to 1/20 of the bandwidth and gradually increase K_p^{series} until the motor starts making a higher pitch noise.
- When the motor makes the high pitch noise, the current waveform looks like that in **Figure 49**.
- Reset K_p^{series} back to the value that was calculated before.
    - The above is a passable way of tuning the current control bandwidth when no current measurement is available.

When done experimenting with the motor:

- Set the variable “motorVars.flagRunIdentAndOnLine” to 0 to disable the PWM output to the motor.
- Turn off real-time control and stop the debugger.

It is important to notice that by default in the ROM code the current controller gains are set to the following values:

$$K_p^{series} = 0.25 \cdot L_s \cdot \frac{1}{T_i}$$

$$K_i^{series} = \frac{R_s}{L_s} \cdot T_i$$

Although it is considered full bandwidth when it is equal to the same frequency of the current controller as follows:

$$K_p^{series} = L_s \cdot \frac{2\pi}{T_i}$$

Conclusion

In the lab project “is06_torque_control”, the K_p^{series} and K_i^{series} gains of the current controller were adjusted. The K_i^{series} gain creates a zero that cancels the pole of the motor’s stator and can easily be calculated. The K_p^{series} gain adjusts the bandwidth of the current controller-motor system. When a speed controlled system is needed for a certain damping, the K_p^{series} gain of the current controller will be related to the time constant of the speed controlled system; as such, it is advised to obtain more knowledge of the mechanical system before calculating the current controller’s K_p^{series}.

is07_speed_control – Speed Control Mode and Tuning Speed PI Controller

Abstract

InstaSPIN-FOC provides a standard PI speed controller. The InstaSPIN library will give a "rule of thumb" estimation of Kp and Ki for the speed controller based on the maximum current and motor inertia setting in "user.h." The estimated PI controller gains are a good starting point, but to obtain better dynamic performance the Kp and Ki terms need to be tuned based on the whole mechanical system that the motor is running. This lab will show how to adjust the Kp and Ki terms in the PI speed controller.

Introduction

Tuning the speed controller is much more difficult than tuning the current controller. The speed controller resides in the mechanical domain which has much slower time constants where phase delays can be tighter, having more of an effect on the stability of the system. The most important parameter needed for accurately tuning a speed controlled system is inertia. That being said, two different approaches for tuning the speed loop are covered here. The first technique uses trial and error and can be used if no parameters of the mechanical system are known. The second technique assumes that inertia and mechanical bandwidth are already known and then designs the current control and speed control gains.

Objectives Learned

- Tune the speed controller quickly using a trial and error technique.
- Tune the speed controller with the knowledge of inertia and mechanical bandwidth.
- Program the Kp and Ki gains into InstaSPIN.

Background

Trial and Error Tuning:

Many times when trying to tune an electric motor, the inertia is not immediately available. InstaSPIN provides the capability to use a very simple but effective technique to quickly tune the PI speed control loop without knowledge of any mechanical parameters. For this next discussion, InstaSPIN uses the "parallel" PI controller for the speed control loop which is illustrated in **Figure 50**.

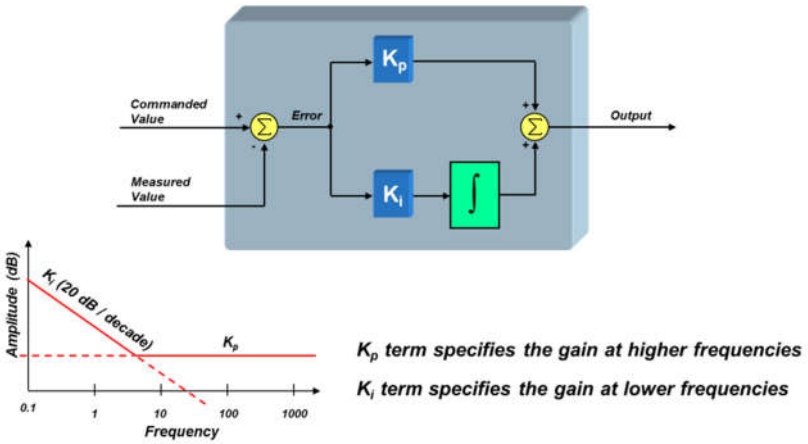


Figure 50: Parallel PI control.

Generally, increasing K_i gain stiffens the systems as strengthening the spring. The dampening of the system is controlled by the K_p gain. For example, if the K_p gain is set very low, K_i will take over and the motor control system will act like a spring. When a step load is applied to the system, it will oscillate. Increasing the damping (K_p) will reduce the oscillations.

Calculated Speed PI Tuning

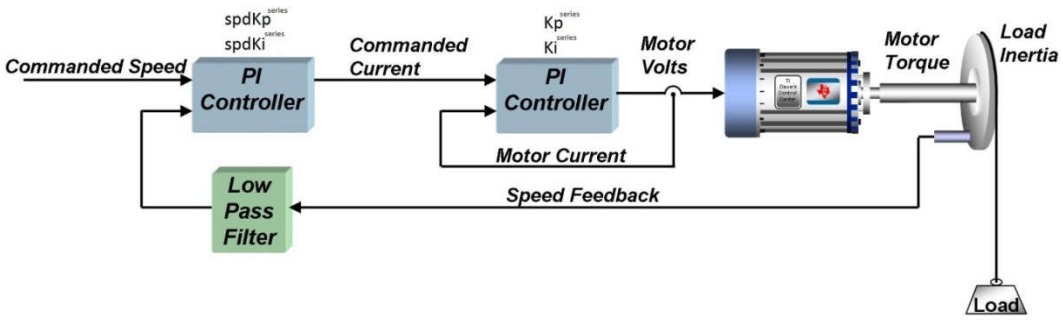


Figure 51: Speed controller cascaded with a current controller and speed filter.

The speed signal often needs to be filtered before it is usable by the control system. For our purposes, let's assume that we are using a single-pole low pass filter of the form

$$Vel_{filter}(s) = \frac{1}{\tau s + 1}$$ Equation 6

Where τ is the time constant of the velocity filter low pass filter (green block from above diagram).

Then, the current control, the closed-loop transfer function is:

$$G_{current}(s) = \frac{1}{\frac{L}{K_p^{series}}s + 1} \cong 1$$ Equation 7

K_p^{series} is the error multiplier term in the current regulator's PI structure. K_i^{series} is not visible to the outside world since it is set to cause pole/zero cancellation within the current controller's transfer function. The K can be set to torque constant Kt over inertia J as shown in the following **Equation 8**, and $G_{current}$ can be approximated to 1 if the current control bandwidth is sufficiently higher than the speed control bandwidth, like **Equation 19**. Therefore, if we eliminate the effect of the current controller pole, the open loop transfer function becomes **Equation 9**.

$$K = \frac{K_T}{J} \qquad \text{Equation 8}$$

$$GH(s) = \frac{K \cdot spdK_p^{series} \cdot spdK_i^{series}\left(1 + \frac{s}{spdK_i^{series}}\right)}{s^2(1 + \tau s)} \qquad \text{Equation 9}$$

Assuming that the zero dB frequency occurs somewhere between the zero at $s = spdK_i^{series}$ and the two nonzero poles in the denominator of the expression, we should end up with a Bode plot that looks something like **Figure 52**:

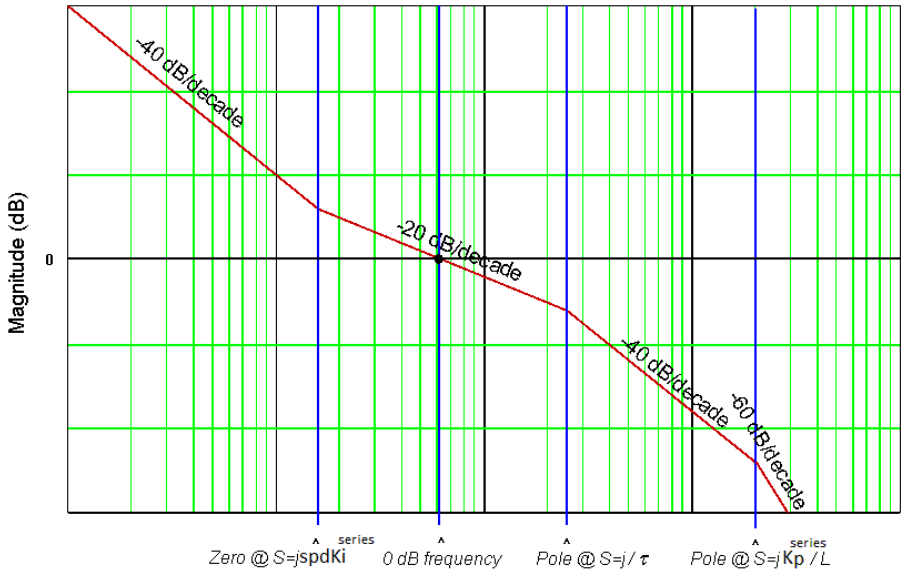


Figure 52: Bode plot of how the system should be tuned.

The reason the shape of this curve is so important is because the phase shift at the 0 dB frequency determines the stability of the system. In general, in order to get a phase shift at 0 dB that leads to good stability, the magnitude response should cross 0 dB at a rate no steeper than -20 dB per decade.

For now, let's assume that the delta in frequency between the pole $1/\tau$ and the zero $spdK_i^{series}$ is fixed. In order to achieve maximum phase margin (phase shift +180°), the unity gain frequency should occur exactly half way in between these two frequencies on a logarithmic scale. Translating from dB to a normal gain scale, this means the following is true:

$$\omega_{unity\_gain} = \delta \cdot spdK_i^{series}$$

And $\frac{1}{\tau} = \delta \cdot \omega_{unity\_gain}$

Combining the last two equations, we establish that:

$\frac{1}{\tau} = \delta^2 \cdot spdK_i^{series}$

Solving for $spdK_i^{series}$:

$spdK_i^{series} = \frac{1}{\delta^2 \tau}$

Where we will use δ as the "damping factor." If δ is increased, it forces the zero corner frequency ($spdK_i^{series}$) and the velocity filter pole ($1/\tau$) to be further apart. Theoretically, any value of δ > 1 is stable since phase margin > 0. However, values of δ close to 1 are usually not practical as they result in severely underdamped performance.

From **Figure 53**, you can see how decreasing the damping factor increases the bandwidth of the velocity loop. The values below two (δ = 1.5, 2.5) are usually unacceptable due to the large amount of overshoot they produce. At the other end of the scale, values much above 35 produce extremely long rise and settling times. Your design target window will usually be somewhere in between these two values.

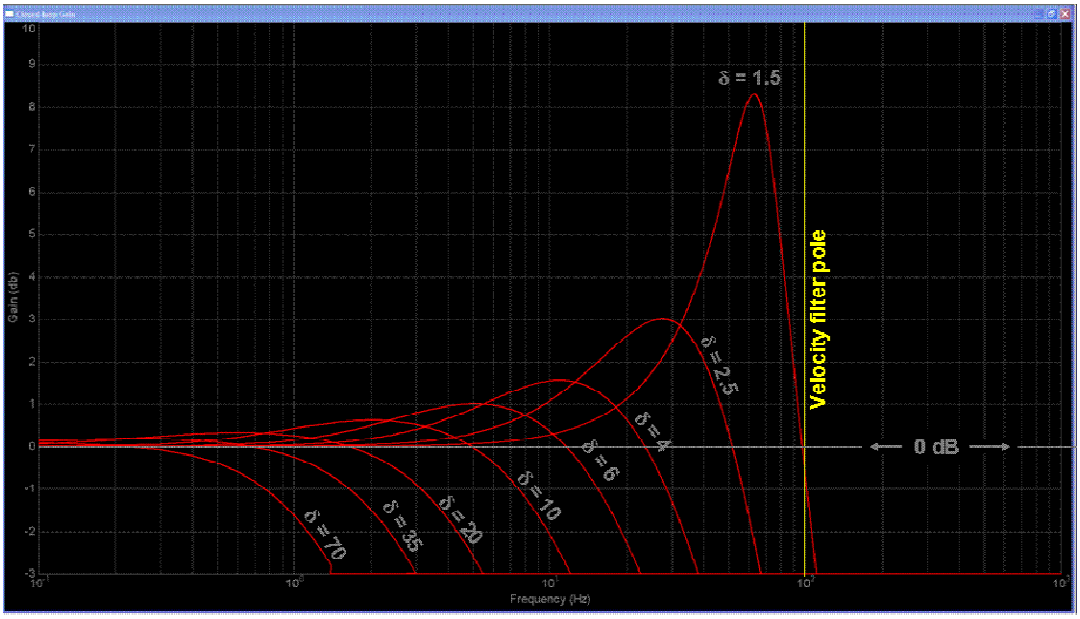


Figure 53: Closed loop magnitude response of the speed loop for various δ

We select a value for the damping factor (δ) which allows us to precisely quantify the tradeoff between velocity loop stability and bandwidth. Through some additional algebra calculations and approximation, the $spdK_i^{series}$ and $spdK_p^{series}$ become as follows.

$$spdK_i^{series} = \frac{1}{\delta^2 \cdot \tau}$$ Equation 10

$$spdK_p^{series} = \frac{\delta \cdot spdK_i^{series}}{K} = \frac{1}{\delta \cdot K \cdot \tau}$$ Equation 11

The benefit of this approach is that instead of trying to empirically tune four PI coefficients which have seemingly little correlation to system performance, you just need to define two meaningful system parameters: the bandwidth of the current controller and the damping coefficient of the speed loop. Once these are selected, the four PI coefficients are calculated automatically.

The current controller bandwidth is a meaningful system parameter, but in speed controlled systems, it is usually the bandwidth of the speed controller that we would like to specify first and then set the current controller bandwidth based on that.

EXAMPLE

An Anaheim Automation 24V permanent magnet synchronous motor has the following characteristics:

Rs = 0.4 ohms

Ls = 0.65 mH

Back-EMF = 0.0054 v-sec/radians (peak voltage phase to neutral, which also equals flux in Webers in the SI system)

Inertia = 2E-4 kg-m^2

Rotor poles = 8

Speed filter pole = 100 rad/sec

Sample frequency, Fs = 10 kHz (or sampling period, Ts = 100 µs)

The desired current controller bandwidth is 20 times lower than the sampling frequency and we would like a damping factor (δ) of 4. Find all the current and speed PI coefficients:

SOLUTION

Since we are trying to set the current bandwidth 20 times lower than the sampling frequency, we solve this equation:

$$BW_c = \frac{2\pi F_s}{20} = \frac{2\pi \cdot 10\text{kHz}}{20} = 3141.59$$

And now we calculate the controller gain based on this bandwidth:

$$K_p^{series} = BW_c \cdot L = 2.042$$

Now, the integral gain of the current controller is using the following equation:

$$K_i^{series} = \frac{R}{L} = \frac{0.4\Omega}{0.65\text{mH}} = 615.3846$$

For the speed controller, we take into account the speed filter and using the following equation:

$$spdK_i^{series} = \frac{1}{\delta^2 \tau} = \frac{1}{4^2\left(\frac{1}{200}\right)} = 6.25$$

Finally, recall that

$$K = \frac{3P\lambda_r}{4J} = \frac{3 \cdot 8 \cdot 0.0054}{4 \cdot 0.0002} = 162$$

And

$$spdK_p^{series} = \frac{1}{\delta \cdot K \cdot \tau} = \frac{1}{4 \cdot 162 \cdot \left(\frac{1}{100}\right)} = 0.1543$$

The simulated speed transient step response for this example is shown in **Figure 54** where the time axis is now scaled appropriately for this design example.

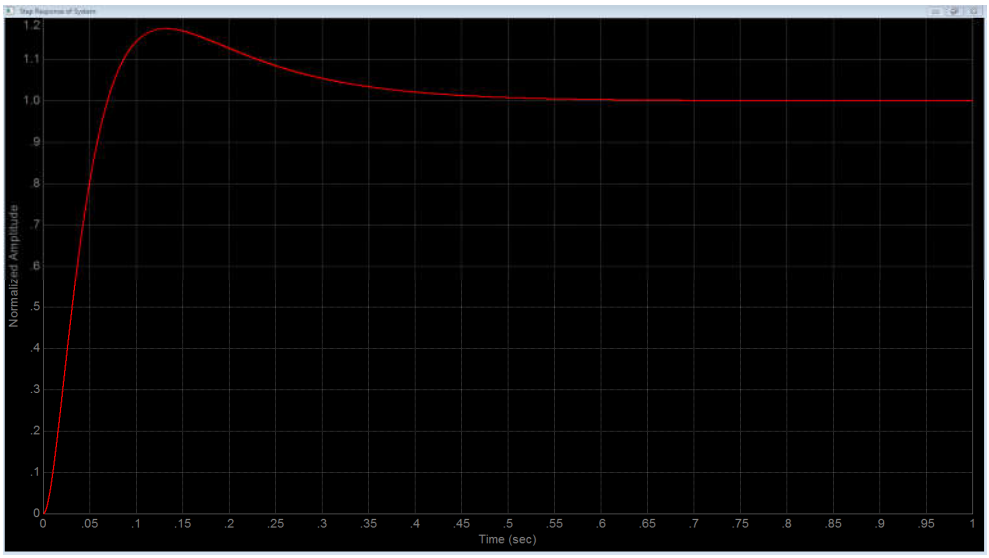


Figure 54: speed transient step response

Project Files

There are no new project files.

Includes

There are no new includes.

Global Object and Variable Declarations

There are no new global object and variable declarations.

Initialization and Setup

There are no new initialization and setup operations. The block diagram of this lab is shown in **Figure 55.**

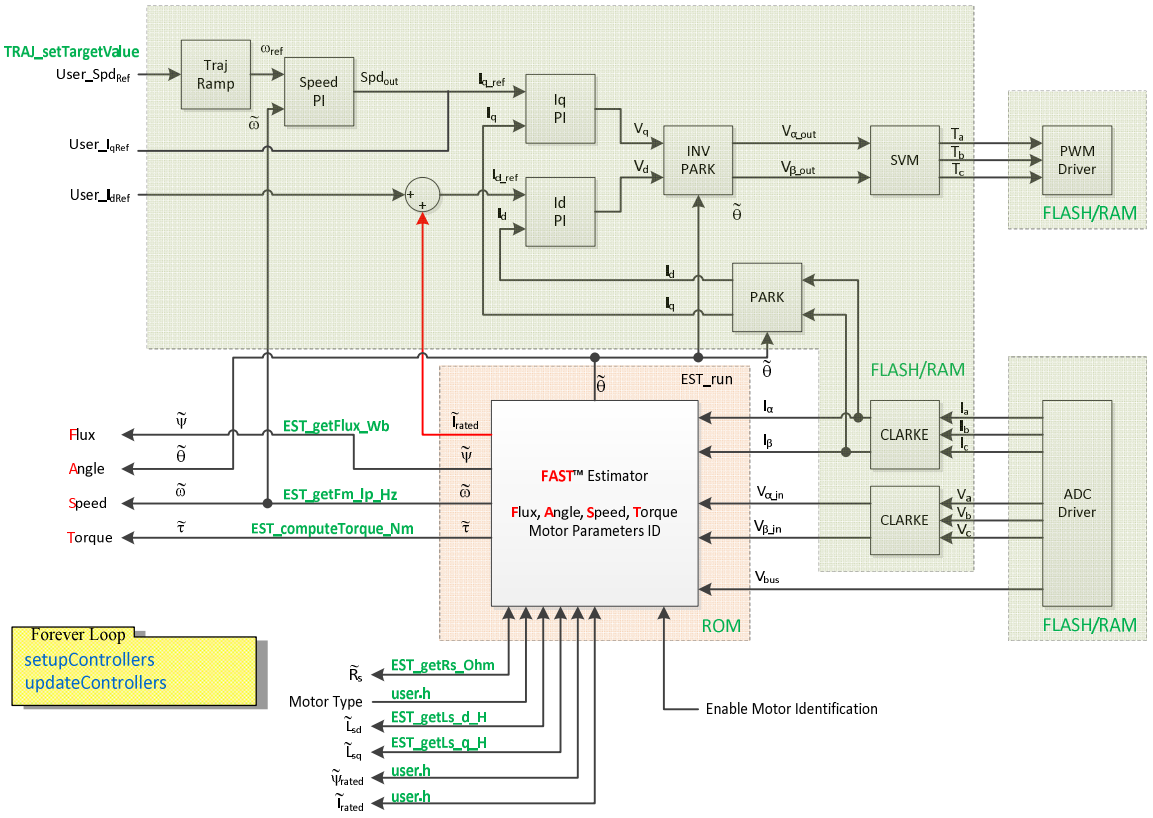


Figure 55: Block diagram of an opened source InstaSPIN implementation.

Main Run-Time loop (forever loop)

The get and set functions for the Kp and Ki speed controller have been added. Immediately after identification, the speed PI gains are updated to pre-calculated versions that were used during motor identification. After the gains are updated, they can be changed in real time by using the "motorVars" structure.

Main ISR

Nothing has changed in this section of the code from the previous lab.

Lab Procedure

Build lab project "is07_speed_control", connect to the target and load the target .out file.

- Add the appropriate watch window variables by calling the script "is07_speed_control.js".
- Enable the real-time debugger.
- Click the run button.
- Enable continuous refresh on the watch window.

Trial and Error Tuning of the Motor

First, we will not worry about finding any data for the motor that is being tuned. The motor control will be set to reference speed of 0 rpm. Then by hand, one can feel how the motor is performing.

Turn on the motor control

- Set “motorVars.flagEnableOffsetCalc” = 1
- Set “motorVars.flagEnableSys” = 1
- Set “motorVars.flagRunIdentAndOnLine” = 1
- Set “motorVars.flagEnableForceAngle” = 0

Turn the motor into a spring

- Set “motorVars.speedRef_Hz” = 0.0
- While quickly turning the motor shaft by about 90 degrees and then letting go, decrease the Kp gain of the speed control with “Kp_spd” until the motor shaft has a dampened oscillation. Note that Kp_spd can be reduced by as much as 100 times from its original calculated value.
    - As the Kp_spd gain is reduced, notice how the motor shaft behaves more like a spring.
    - If the Ki_spd setting is too large, it will be harder to turn the motor shaft. Reduce the Ki_spd value so that the motor behaves like a weak spring.
    - Example values for the BOOSTXL-DRV8320RS RevA kit + LAUNCHXL_F280049C LaunchPadand a Anaheim BLY172S motor:
        - Kp_spd = 0.1
        - Ki_spd = 0.018

Dampen the Motor

- Increase the Kp_spd gain until the spring feeling is gone. Notice how increasing the Kp_spd gain causes the motor to be more dampened.
- Because Kp_spd causes dampening, it can be increased to a large value and example for the BOOSTXL-DRV8320RS RevA kit + LAUNCHXL_F280049C LaunchPad with the Anaheim BLY172S motor is:
    - Kp_spd = 8.0

Increase the stiffness of the system

- Now increase the Ki_spd gain to increase the stiffness.
- A typical value for the BOOSTXL-DRV8320RS RevA kit + LAUNCHXL_F280049C LaunchPad with the Anaheim BLY172S motor is:
    - Ki_spd = 0.1

By knowing that the Ki_spd value increases the spring constant of the system if a speed controlled system is unstable, reduce the Ki_spd value to stabilize the system. Knowing that the Kp_spd gain dampens the speed controlled system can help stabilize the system by increasing Kp_spd.

Calculated Speed Loop Tuning

Obtain the motor parameters

- Rs – Motor resistance
- Ls – Motor total inductance
- P – Number of poles for the motor
- Ke – Motor flux constant in V/Hz
- J - Inertia of the whole mechanical system

Obtain the controller scale values from “user.h:”

$$T_i = \frac{1}{PWM\_Freq\_kHz \cdot 1000} \cdot PWMvsISRtick \cdot ISRvsCURRENTtick$$

Where:

- T_i is the current controller period
- PWM_Freq_kHz can be taken from USER_PWM_FREQ_kHz parameter in user.h

- $PWMvsISRtick$ is the tick rate between PWM and interrupts, USER_NUM_PWM_TICKS_PER_ISR_TICK
- $ISRvsCURRENTtick$ is the tick rate between interrupts and current controllers, USER_NUM_ISR_TICKS_PER_CURRENT_TICK

$$T_v = \frac{1}{PWM\_Freq\_kHz \cdot 1000} \cdot PWMvsISRtick \cdot ISRvsSPEEDtick$$

Where:

- T_v is the speed controller period
- $ISRvsSPEEDtick$ is the tick rate between interrupts and speed controllers, USER_NUM_ISR_TICKS_PER_SPEED_TICK
- $Vel_{fs} = USER\_FULL\_SCALE\_FREQ\_Hz$ – *Full scale frequency in Hz*
- $I_{fs} = USER\_ADC\_FULL\_SCALE\_CURRENT\_A$ – *Full scale current in A*
- $V_{fs} = USER\_ADC\_FULL\_SCALE\_VOLTAGE\_V$ – *Full scale voltage in V*

Choose a speed loop damping factor

- For this lab $\delta = 4$

Calculate K_p^{series} from the current controller bandwidth, keeping these limits in mind:

$$\frac{10L}{\delta\tau} < K_p^{series} < \frac{2\pi L}{10T_s}$$

- $K_p^{series} = BW_c \cdot L_s$

CalculateK_i^{series}

- $K_i^{series} = \frac{Rs}{Ls}$

Calculate $spdK_i^{series}$

- $spdK_i^{series} = \frac{1}{\delta^2 \cdot \tau}$

Calculate the constant K t from the motor parameters

- $K = \frac{3P\lambda_r}{4J}$

Calculate $spdK_p^{series}$

- $spdK_p^{series} = \frac{1}{\delta \cdot K \cdot \tau}$

As a reminder, the PI analysis that came up with these calculations is based on the series PI loop. InstaSPIN uses a series PI loop for the current controllers and a parallel PI loop for the speed controller. The speed PI gains have to be converted from the series form to the parallel form. **Equation 12** and shows the conversion.

$$spdK_p^{parallel} = spdK_p^{series} \qquad \text{Equation 12}$$

$$spdK_i^{parallel} = spdK_i^{series} \cdot spdK_p^{parallel} \qquad \text{Equation 13}$$

The calculations that have been done so far have not been converted to be used in the digital PI regulator. All of the Ki gains precede a digital integrator. The digital integrator is multiplied by the sampling time. To

reduce the number of multiplies that are needed in the code, the sampling time must be multiplied by the Ki gains before importing the values into the code.

Convert the integral gains to the suitable value for use in the digital PI control

- $spdK_i = spdK_i^{parallel} \cdot T_v \cdot \frac{4\pi \cdot Vel_{fs}}{I_{fs} \cdot P}$
- $curK_i = K_i^{series} \cdot T_i$

The proportional gains must be per-unitized before being entered into the digital PI control

- $spdK_p = spdK_p^{parallel} \cdot \frac{4\pi \cdot Vel_{fs}}{I_{fs} \cdot P}$
- $curK_p = K_p^{series} \cdot \frac{I_{fs}}{V_{fs}}$

Enter the per-unit gain values into the appropriate gain values

- motorVars.Kp_spd =$spdK_p$
- motorVars.Ki_spd =$spdK_i$
- motorVars.Kp_Id = $curK_p$
- motorVars.Ki_Id = $curK_i$
- motorVars.Kp_Iq = $curK_p$
- motorVars.Ki_Iq = $curK_i$

Run the motor and load the shaft to see the performance

Compare the gain values between the trial and error and calculated tuning techniques

When done experimenting with the motor:

- Set the variable “motorVars.flagRunIdentAndOnLine” to disable PWM outputs to the motor.
- Turn off real-time control and stop the debugger.

The resulting plot of this speed controller, compared to a simulation using the exact same gains looks like **Figure 56**:

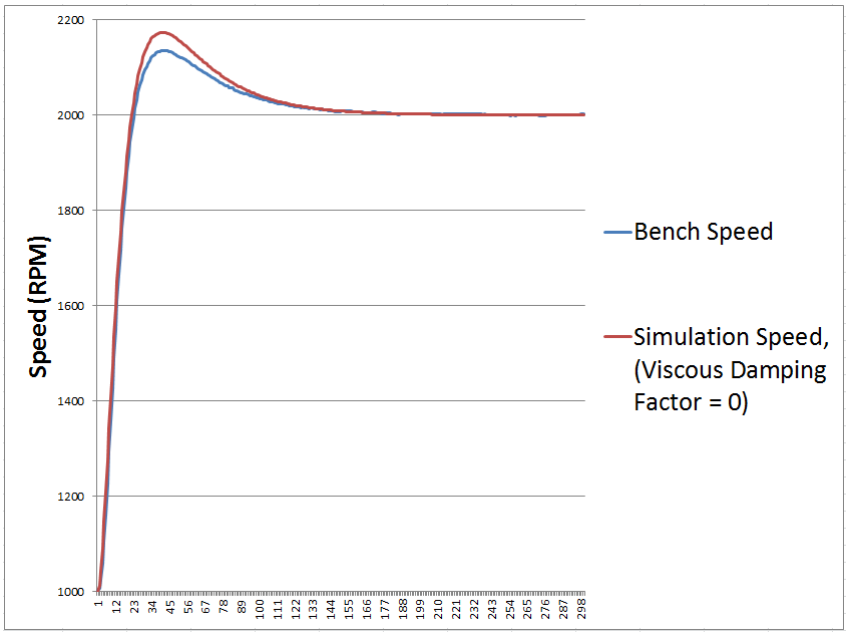


Figure 56: Speed response simulation using the calculated gains

Now if we add a small value of viscous damping factor to the simulation, then we get a perfect match as **Figure 57**.

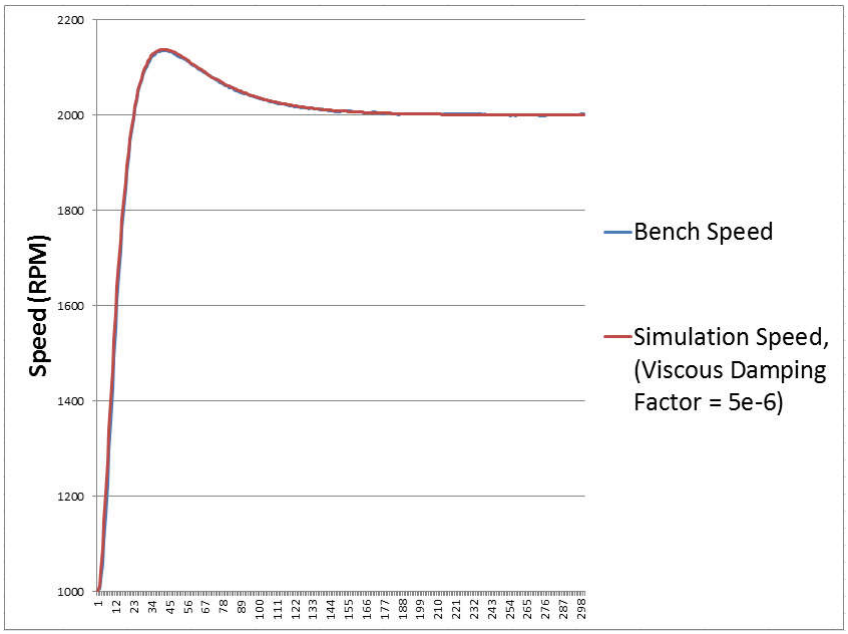


Figure 57: Speed response simulation using a small damping factor

Conclusion

Tuning the speed controller has more unknowns than tuning a current controller. Therefore the first approach to tuning the speed controller in this lab is by using a trial and error approach. It was shown that the parallel speed PI closed loop control correlates to a mass, spring, damper system. If more parameters are known about the mechanical system of the motor controlled system, then the optimum calculated approach can be used. The calculated approach will identify the gains for the speed and current controllers based on the bandwidth and damping selected by the user.

is08_overmodulation – Space Vector Over-Modulation

Abstract

The SVM that is used by InstaSPIN is capable of saturating to a pre-specified duty cycle. When using a duty cycle over 100.0%, the SVM is considered to be in the over-modulation region. When in the over-modulation region, current shunt measurement windows become small or even disappear. This lab will show how to re-create the currents that cannot be measured due to high duty cycles during SVM over-modulation.

Introduction

In a typical three-phase inverter, one of the preferred methods to measure motor currents is with low side shunt resistors as shown in **Figure 56**. This provides an economical solution since the reference of the current measurement is the same as the microcontroller ground. However, it introduces a limitation since the low side shunt resistor carries current only when the low side PWM is ON.

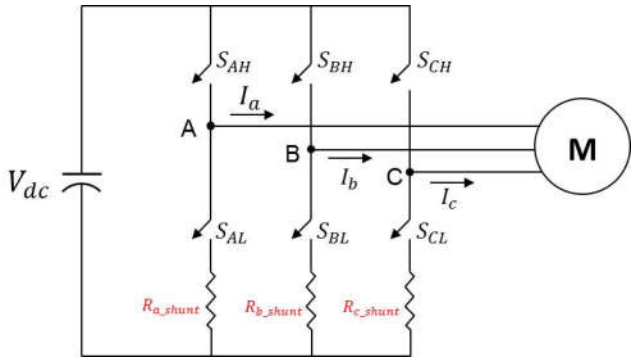


Figure 56: Current shunt layout

Also, when driving a motor with a three-phase inverter, it is desirable to allow full voltage to the motor windings, not only sinusoidal modulated waveforms. This requirement pushes Space Vector Modulation to its limitation and causes extensive periods of time where the low side PWM ON time basically disappears. **Figure 57** shows a scenario where the pulse in PWM1L is too narrow to allow a valid conversion on Phase A.

The method used in this lab utilizes a current reconstruction technique and a set trigger function that allow measuring currents even when narrow pulses like this are generated by the inverter. If we can measure two phase currents at least, the unknown phase can be calculated using current symmetry $I_a + I_b + I_c = 0$. Therefore, we can know all three phases' currents if two phases are always guaranteed to measure current. The current reconstruction method used in this lab is a voltage reconstruction method that relocates phase voltage to guarantee the minimum width in two phases. Even though phase voltages are changed during over-modulation, the line-to-line voltage between these phases is preserved because of the phase compensation method. Being able to reconstruct currents while performing over-modulation allows a field oriented control system to work even during heavy over-modulation or trapezoidal control.

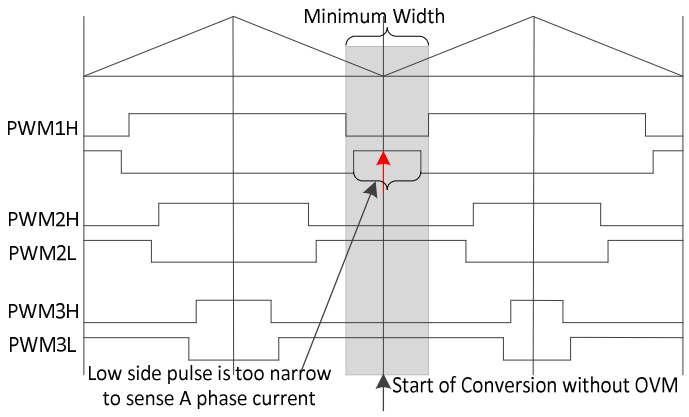


Figure 57: SOC without over-modulation where a low-side pulse is too narrow to sample

Prerequisites

It assumes knowledge of up to project “is07_speed_control”.

Objectives Learned

The objective of this lab is to show users an implementation of current measurement reconstruction when the measurement window is not wide enough.

Detailed Description

In lab “is08_overmodulation“, a new approach to implementing over-modulation is explored. There are four aspects of this algorithm:

- ***Output Voltage Generation.*** This is the ability of space vector modulation to create output waveforms from zero, into sinusoidal waveforms, and then into trapezoidal waveforms, all by just increasing the magnitude of the inputs in alpha/beta coordinates.

- ***Currents Reconstruction.*** Since this is based on a current control algorithm such as FOC, current feedback needs to be always available for Id and Iq current controllers to work. Depending on which current measurements are available, this algorithm reconstructs the three-phase currents when the low side PWM duty cycles fall below a minimum width.

- ***Output Voltage Compensation.*** The output phases are relocated to guarantee minimum duty width in two phases at least. To achieve this minimum duty width, the algorithm analyzes the number of current phases that are able to be measured when compared against a user-defined maximum output voltage limit. When two-phase voltages are bigger than the limit voltage, meaning they cannot be accurately sampled, an offset is found based on the voltage phase with the second-smallest current sample window. The offset is defined as the difference between the magnitude of the “middle” voltage (phase with the second smallest sample window) and the user-defined limiting voltage. Once the offset voltage is defined, all phase voltages are reduced by the offset voltage with keeping line-to-line voltage. Based on the duty cycles loaded on the present and next PWM cycle, this algorithm can also define which currents will be ignored in the next PWM cycle.

- ***Setting the Start of Conversion (SOC) trigger.*** During the creation of PWM outputs, there are several switching events that must be avoided in order to have clean current measurements. This algorithm analyzes the PWM duty cycles and the ignore shunt value previously calculated to properly set the trigger for the next ADC start of conversion signal.

Output Voltage Generation

The implementation of space vector modulation (SVM) allows input amplitudes up to (2/3) pu when in the Alpha-Beta coordinate system. In order to ensure SVM can generate outputs up to (2/3) pu, call the USER_setParams() function to set the maximum modulation index:

Notes: To achieve a pure sinusoidal current waveform, it's recommended to limit the maximum modulation index to $(1.0/\sqrt{3})$.

Define the maximum Vs magnitude limit in "user.h"

```
#define USER_MAX_VS_MAG_PU              (0.55)          // Set to 0.5774 for a pure sinewave
with a peak at 100% duty cycle which need a current reconstruction
```

Set the maximum Vs magnitude to be the DC bus voltage scaled by the per unit limit, and set the Iq and Id limits per this value

```
   // Set the maximum voltage output
   userParams.maxVsMag_V = userParams.maxVsMag_pu * adcData.dcBus_V;
   PI_setMinMax(piHandle_Id,-userParams.maxVsMag_V, userParams.maxVsMag_V);
```

Current Reconstruction

The second aspect of over-modulation is to allow currents to be reconstructed when needed. When sampling the currents in the ISR, currents are read and scaled through the HAL with the following function call:

```
   // read the ADC data with offsets
   HAL_readADCDataWithOffsets(halHandle,&adcData);

   // calculate Vbus scale factor to scale offsets with Vbus
   motorVars.Vbus_sf = adcData.dcBus_V * motorVars.offset_invVbus_invV;

   // remove offsets
   adcData.I_A.value[0] -= motorVars.offsets_I_A.value[0];
   adcData.I_A.value[1] -= motorVars.offsets_I_A.value[1];
   adcData.I_A.value[2] -= motorVars.offsets_I_A.value[2];

   adcData.V_V.value[0] -= motorVars.offsets_V_V.value[0] * motorVars.Vbus_sf;
   adcData.V_V.value[1] -= motorVars.offsets_V_V.value[1] * motorVars.Vbus_sf;
   adcData.V_V.value[2] -= motorVars.offsets_V_V.value[2] * motorVars.Vbus_sf;
```

Following the above code, all current values are stored in the adcData.I_A structure. Some of the values may not be valid depending on how narrow the low side PWM pulse was when the corresponding current was measured. Since we have over-modulation, we make use of an SVM extension module, SVGENCURRENT. This module reconstructs phase currents in a simple way depending on the state of

an enumeration called “SVGENCURRENT_IgnoreShunt_e.” The following logic is implemented as part of the SVGENCURRENT module in “svgen_current.h” in order to reconstruct the currents.

Add two variables to save the previous current sample and PWM output duty for over-modulation

```
// the previous data of ADC result and PWM output
MATH_Vec3 adcDataPrev_A = {0.0, 0.0, 0.0};
MATH_Vec3 pwmDataPrev = {0.0, 0.0, 0.0};
```

The second stage of current reconstruction is added to this lab to account for corner case conditions when two of the three current readings are not valid. This approach makes use of a running average, where the principle is simple: if a current is not valid, use a software approximation with a filter and its past values. The code in “svgen_current.h” shows how this is done.

After this stage of current reconstruction, the measured and estimated currents are as close as possible in order to operate a sensorless FOC system during extreme over-modulation conditions.

Output Voltage Compensation

The third aspect relates to compensating output phase voltage to guarantee minimum duty in at least two phases. This is done by running an SVGENCURRENT function that updates the PWM output value: SVGENCURRENT_compPWMData(). In this function, there are three main functions for output PWM compensation.

- *Define compensation mode*
  All phase voltages are compared with V_{LIM} to check if the phase current is measurable. The limit voltage V_{LIM} is defined as the maximum phase output voltage needed to guarantee the minimum duty width for shunt current measurement. If all phase voltages are less than $V_{LIM,}$ the compensation mode is defined as “*ALL_PHASE_MEASURABLE*” for all three sampled current phases. If two-phase voltages are less than $V_{LIM,}$ the compensation mode is defined as “*TWO_PHASE_MEASURABLE;*” likewise, if only one phase voltage is less than V_{LIM}, the compensation mode is defined as “*ONE_PHASE_MEASURABLE”.* In case of “*ONE_PHASE_MEASURABLE”* mode, voltage compensation is carried out by finding a middle voltage V_{MIDDLE}.

- *Phase voltage compensation*
  In “*ONE_PHASE_MEASURABLE*” mode, all voltage reconstruction should account for offset voltage to guarantee two measurable phases. The offset voltage calculation and subtraction are shown in “svgen_current.h” shows how this is done

- *Ignore Shunts*
  The third module relates to knowing which currents will be ignored in the next interrupt. This function is carried out by comparing the average output voltage and the limit voltage V_{LIMIT}. The IgnoreShunt value is set to two main categories of values. The code from line 501 to line 517 in “svgen_current.h” shows how this is done.

    - *USE_ALL*: If all phase voltages are less than the limit voltage, all currents are sampled, because the width of all pulses is wider than the minimum acceptable width.

- *IGNORE_A, IGNORE_B or IGNORE_C*: Used when the corresponding phase being measured is less than an acceptable measurement window. It also assumes that the difference between the phase being ignored, and the other two, is larger than an acceptable time.

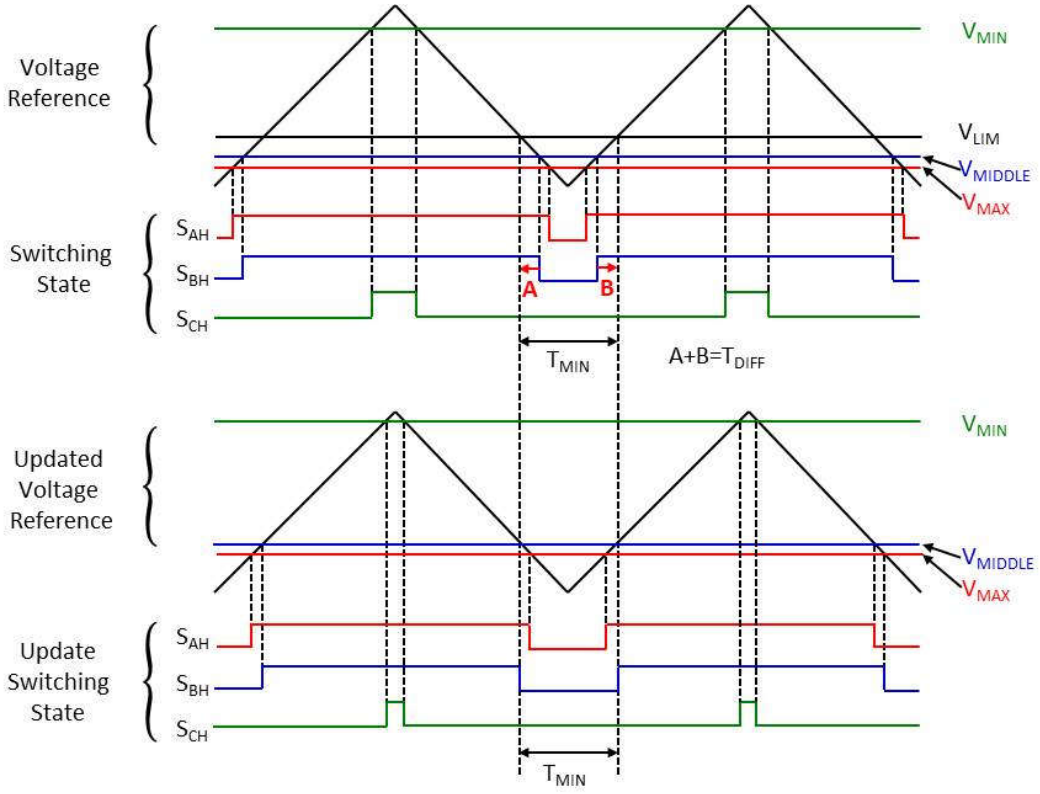


Figure 58: The top diagram shows a scenario where 2 of the 3 phases feature sample windows too narrow to be sampled. The bottom diagram shows the updated switching state, which fixes this problem using offset correction

Setting the Start of Conversion (SOC) trigger

The last aspect of over-modulation is setting the start of conversion trigger in the right spot to ensure the best possible current measurement is taken during the next PWM cycle. This is done through the HAL layer, with function call HAL_setTrigger(). This function sets the trigger of the next conversion based on:

- Next Sample Pulse values: When all shunts are used, the setTrigger function needs to know which pulse is the narrowest one, so the trigger can be placed in the center of the pulse.
- ignoreShunts: Depending on which shunts are ignored, the trigger changes to accommodate the best shunt.
- midVolShunt: When ignoring 1 or 2 shunts, the setTrigger function needs to know which pulse features the second smallest sample window out of three pulses, because this “middle length” pulse is the minimum duty. The PWM trigger is placed in the center of this pulse.

When all the shunts are valid, the trigger is set right in the middle of the narrowest pulse of all three as shown in the following diagram:

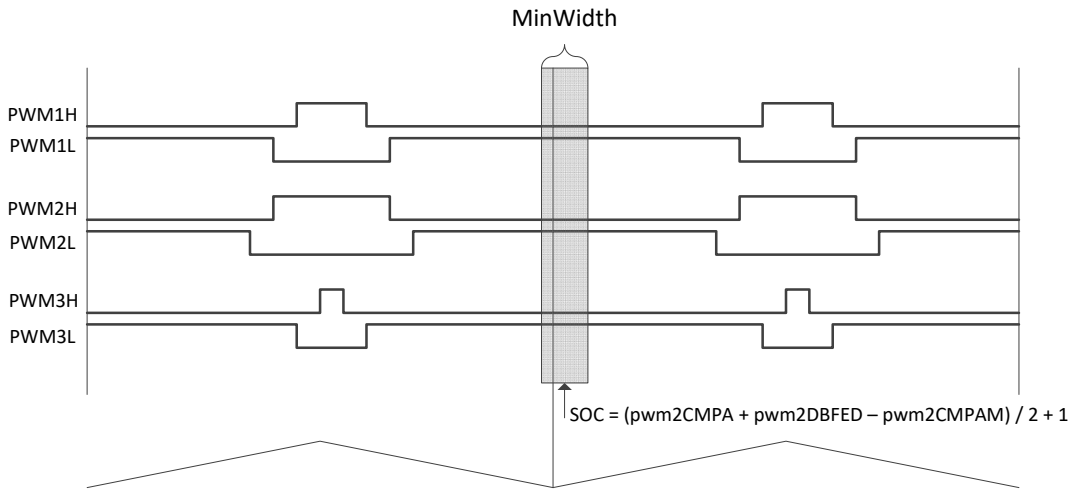


Figure 59: Changing the SOC trigger to successfully sample within the minimum pulse width

When two shunts are valid after output voltage compensation, the trigger is set right in the middle of the middle length out of all three.

Lab Procedure

Step 1.

Set the maximum phase voltage magnitude in "user.h" to be higher than 0.5 and less than 0.667 of Vbus. By default, the value is set to $(1.0/\sqrt{3})$.

```
#define USER_MAX_VS_MAG_PU        (0.570)
```

Step 2.

Open project "is08_overmodulation", build and load

Step 3.

Load variables to your watch window. New variables for this lab are:

- userParams.maxVsMag_pu. This variable will be used to set the limits on the output modulation. A maximum of 2/3 would create a trapezoidal output waveform on the voltage.
- userParams.maxVsMag_V. Use this variable to monitor the real limitation output voltage for the current regulator, which is calculated by multiplying with a scale factor or userParams.maxVsMag_pu by DC bus voltage.

- svgencurrent.minWidth. This variable sets the minimum width for current measurement. This is hardware dependent, but a value corresponding to 2 microseconds is usually good for all applications.
- svgencurrent.ignoreShunt. Use this variable to monitor which shunts are being ignored as the motor spins.
- svgencurrent.compMode. Use this variable to monitor how many phases are able to be measurable.

Step 4.

Run the motor by setting these two flags to true: motorVars.flagEnableSys = 1 and motorVars.flagRunIdentAndOnLine = 1

Step 5.

Increase the speed reference, motorVars.speedRef_Hz, until the ignore shunt value shows that shunts are being ignored as the motor spins

Step 6.

Change maximum modulation value, and monitor the maximum speed you can reach. For example, using the BOOSTXL-DRV8320RS RevA kit and LAUNCHXL_F280049C LaunchPad, and driving an Anaheim motor with 24V under light load, these were the top speeds with each modulation value:

Maximum Output Vs Vector	Top Speed
0.5	360Hz
1/sqrt(3) = 0.5774	400Hz

Table 20: Changing the maxVsMag_pu value to reach different speed during over-modulation

Notes: It is not recommend setting the modulation index higher than $(1.0/\sqrt{3})$

Conclusion

In this lab, several aspects of over-modulation were discussed, allowing the usage of the entire input voltage. Shunt resistor based current sense challenges are also solved using software techniques to reconstruct currents and to set the trigger point at the right spot.

is09_flying_start – Using Flying Start

Abstract

The flying start feature is used to control an already rotating motor and resume normal operation with minimal impact on load or speed. Project “is09_flying_start” shows how to use the flying start function in InstaSPIN-FOC.

Introduction

Lab “is09_flying_start” provides a guideline for applying the flying start feature in InstaSPIN-FOC. Flying start is a feature that allows the drive to determine the speed and direction of a spinning motor and begin the output voltage and frequency at that speed and direction. Without flying start, the drive will begin its output at zero volts and zero speed and attempt to ramp to the commanded speed. If the inertia or direction of rotation of a load requires the motor to produce a large amount of torque, excess current may result and overcurrent trips may occur on the drive. These problems can be eliminated with flying start.

Prerequisites

It assumes knowledge of project “is07_speed_control” or “is08_overmodulation”.

Objectives Learned

The objective of this lab is to learn how to control an already rotating motor using InstaSPIN-FOC with the flying start function.

Detailed Description

Flying start is the capacity to start control at any speed other than **ZERO**, which is an important function in some applications such as traction, washing machine, fan, e-bike, and e-scooter.

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

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

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

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

Project Files

There are no new project files.

Includes

There are no new includes.

Initializing and Setup the Flying Start Module

The code from line 517 to line 530 in “is09_flying_start.c” shows how the flying start module is initialized and configured with default values.

The control program flowchart of flying start is shown in **Figure 60**, the flying start module outputs a flag to enable or disable speed close loop control. A zero reference torque current is set and the speed PI controller output is disabled while flying start is operating.

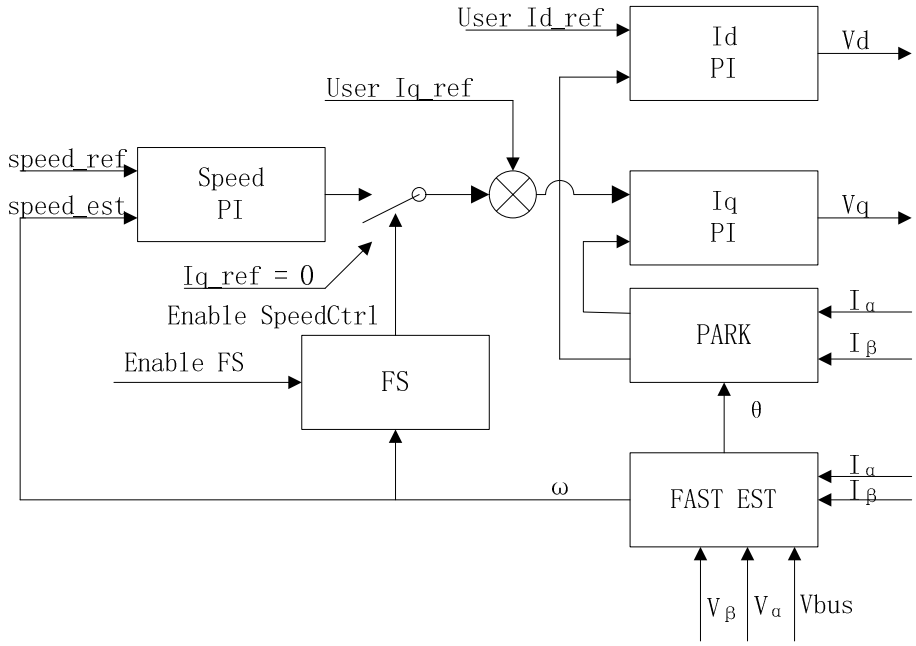


Figure 60: Flying start control program flowchart

The flying start frequency and angle search algorithm function is new to this lab, and must be called in the main ISR.

```
    // run the flying start function
    runFlyingStart(estHandle);
```

The flying start variables and flags are updated in the following new function, which can be called in the main background loop outside of the ISR.

```
      // run motor control function
      runMotorCtrl(estHandle);
```

As shown in **Figure 61**, the module routine disables speed close loop control, sets the reference Iq to zero, and enables the InstaSPIN-FOC module. After the phase currents and voltages are measured, the routine runs InstaSPIN-FOC and the real motor speed can be estimated. The program re-enables speed closed loop control and sets the speed reference value after flying start is completed. The current waveform during restart is shown in **Figure 62** and **Figure 63**.

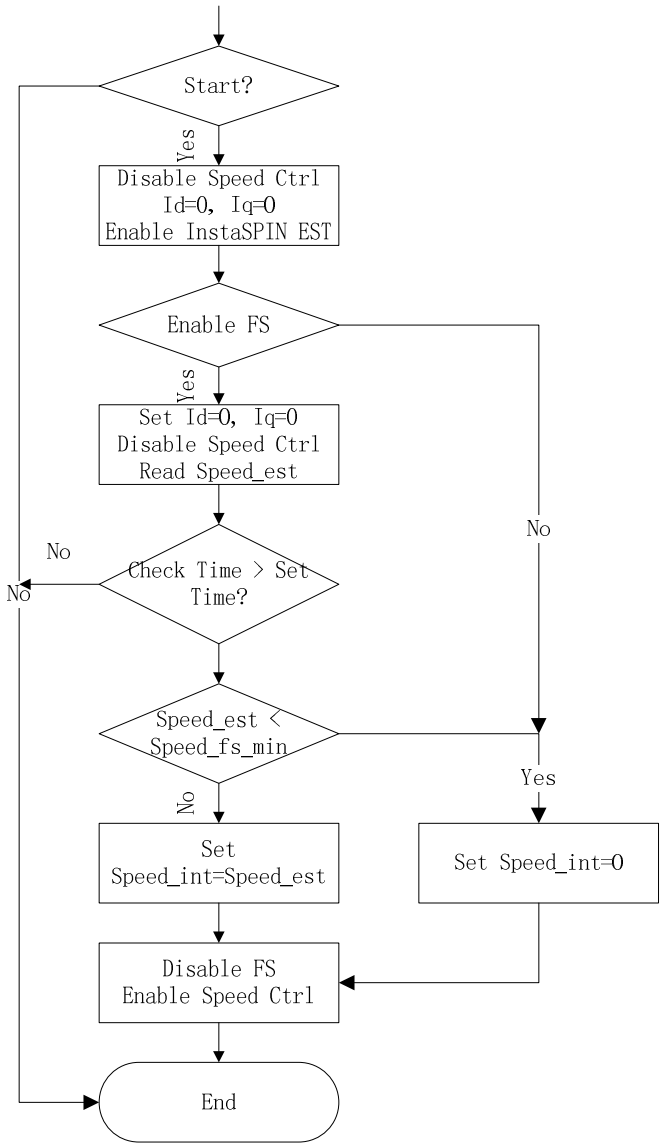


Figure 61: Flying start module program flowchart

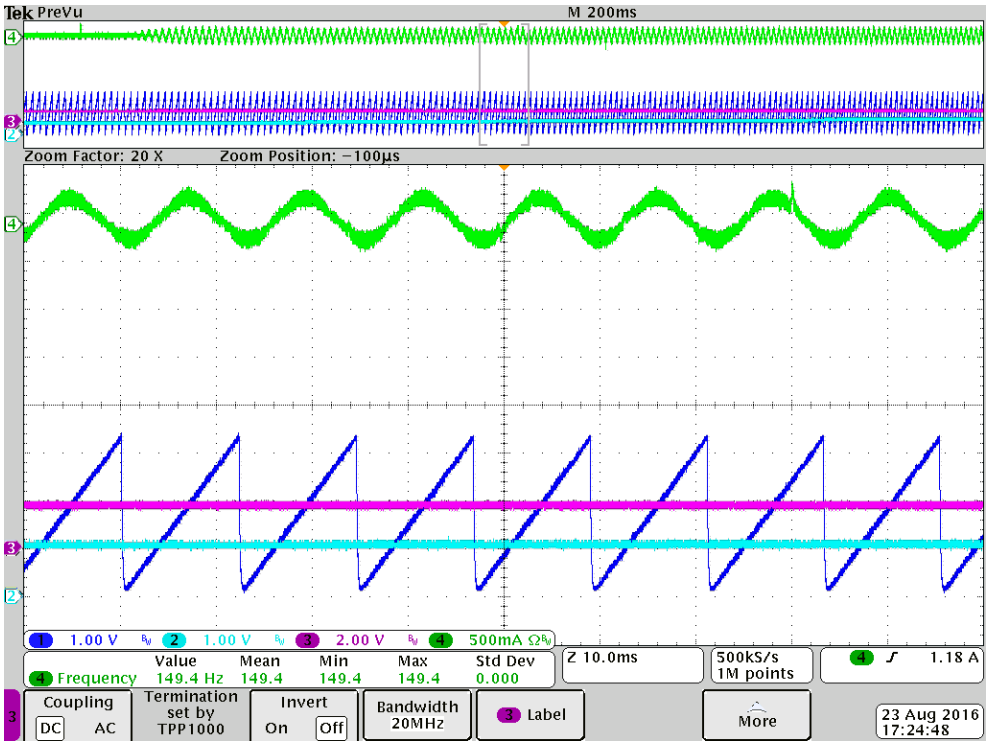


Figure 62: Motor restarting from non-zero speed with flying start function

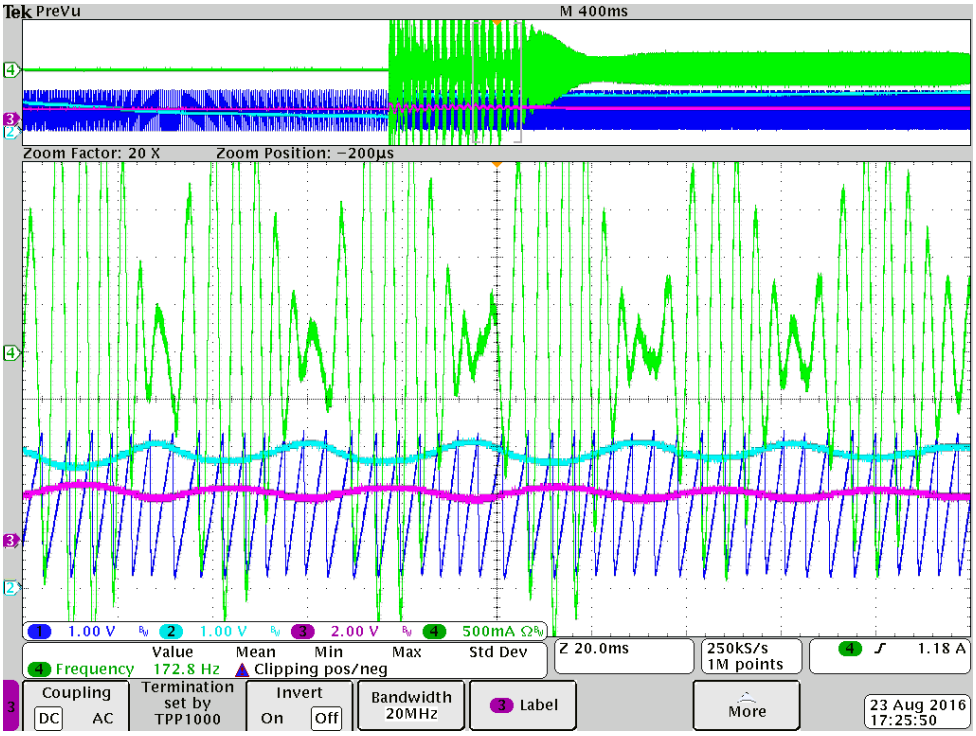


Figure 63: Motor restarting from non-zero speed without flying start function

Lab Procedure

Step 1.
Set up the lab kit, making sure to correctly connect the motor and power supply.

Step 2.
In the “user.h” header file, ensure the motor parameters are known and correctly set. Lab “is09_flying_start” will only work with permanent magnet synchronous motors.

Step 3.
In Code Composer, build project “is09_flying_start”, connect to the target and load the .out file.

- Open the file \solutions\common\sensorless_foc\debug\is09_flying_start.js” via the Scripting Console
  - This will add the variables that we will be using for this project into the watch window
- Enable the realtime debugger
  - This will let the debugger update the watch window variables
- Click the run button.
  - This will run the program on the microcontroller
- Enable continuous refresh on the watch window.
  - This will continuously update the variables in the watch window

Step 4.
To run the motor with flying start function

- To start the project, set the variable “motorVars.flagEnableSys” equal to 1.
- To enable flying start function, set the variable “motorVars.flagEnableFlyingStart” equal to 1.
- To turn on the PWMs to the motor, set the variable “motorVars.flagEnableRunAndIdentify” equal to 1.
- The acceleration can be modified by adjusting the value in “motorVars.accelerationMax_Hzps”.
- Set a reference speed using “motorVars.speedRef_Hz” in order to run the motor at a target speed.

Step 5.
When finished experimenting to stop the motor

- Set the variable “motorVars. flagEnableRunAndIdentify” to ‘0’ to turn off the PWMs to the motor.
- Turn off real-time control and stop the debugger.
- Turn off power supply of drive kit.

Conclusion

Lab “is09_flying_start” adds the flying start functions to an InstaSPIN-FOC example. The flying start feature allows a motor to start at any speed different to zero without over current trips occurring in the drive.

is10_rs_recalc – Using Rs Online Recalibration

Abstract

The stator resistance of the motor's coils, also noted as Rs, can vary drastically depending on the operating temperature of the coils (also known as motor windings). This temperature might increase due to several factors. The following examples list a few of those conditions where the stator coils temperature might be affected:

- Excessive currents through the coils.
- Motor's enclosure does not allow self-cooling.
- Harsh operation environment leading to temperature increase
- Other heating elements in motor's proximity.

As a result of the temperature increase, there is a resistance increase in the motor's windings. This resistance to temperature relationship is well defined depending on the materials used for the windings themselves.

Introduction

Lab "is10_rs_recalc" provides a guideline for applying the Rs online recalibration feature by running a motor.

Prerequisites

It assumes knowledge of project "is07_speed_control" or "is08_overmodulation".

Objectives Learned

- Run Rs Online recalibration feature.
- See the Rs value being updated while motor is running.

Project Files

There are no new project files.

Includes

There are no new includes.

Global Object and Variable Declarations

There are no new global object and variable declarations.

Initialization and Setup

The following functions are new to this lab as shown in **Table 21**. These functions are all bundled in a new function called **runRsOnLine()** which is called from the main background loop. This function contains the following estimator functions:

Background Loop		
	EST	
	EST_getState	Gets the Estimator State to make sure the estimator is running before enabling Rs Online Recalibration
	EST_setFlag_enableRsOnLin	Enables the Rs Online feature. After calling this function with a true as a parameter, a new varying Id reference will be generated to recalibrate Rs
	EST_setRsOnLineId_mag_A	This function sets the level of current to be generated in Id reference. The value needed by this function is in SI units (A)
	EST_setFlag_updateRs	When this function is called with true as a parameter, the internal Rs value used by the estimator will use the Rs Online value. It is recommended to enable this update flag only when the Rs Online value has settled.
	EST_setRsOnLineId_A	It is recommended to call this function with a zero as a parameter to clear the accumulated Id reference when the motor is not running
	EST_setRsOnLine_Ohm	It is recommended to call this function with a zero as a parameter to initialize the Rs Online value to the existing Rs value before enabling Rs Online feature

Table 21: API function calls in runRsOnLine()

The block diagram of InstaSPIN is shown in **Figure 64**, and in red, the function call that enables Rs Online. Also, in red, the Id reference that comes from FAST that allows Rs Online to work when it is enabled.

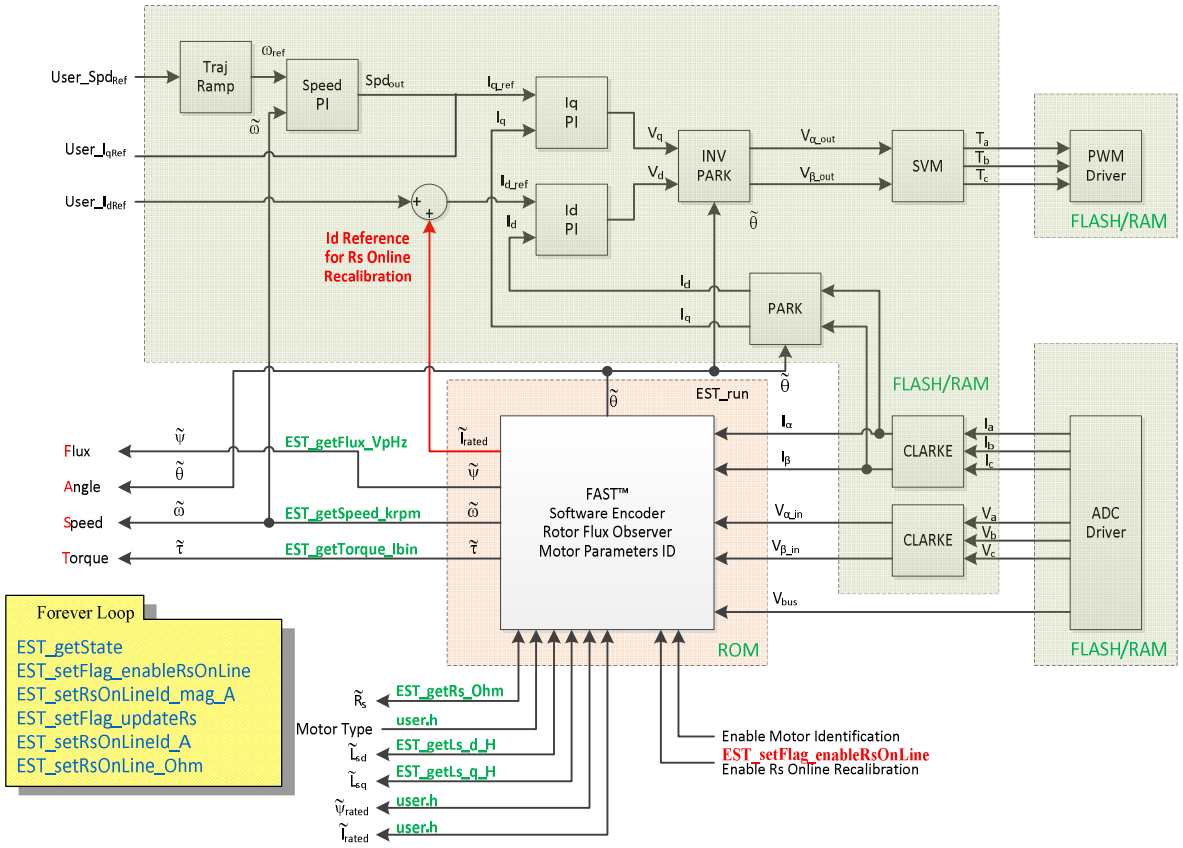


Figure 64: Block diagram of InstaSPIN-FOC with Rs online recalibration

The state machine flowchart of lab“is10_rs_recalc” is shown in **Figure 65** that allows Rs online recalibration to work from the background loop.

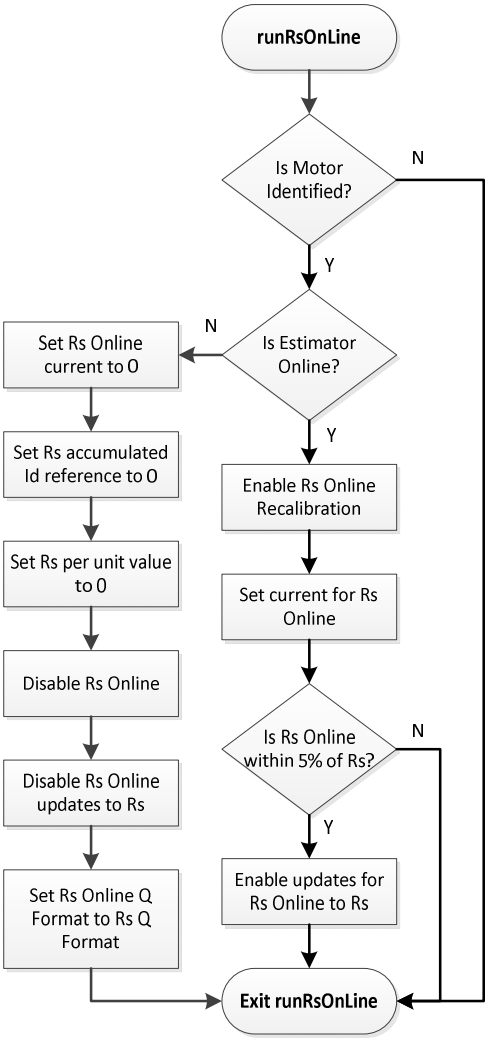


Figure 65: InstaSPIN lab project “is10_rs_recalc” state machine diagram

Lab Procedure

Build project “is10_rs_recalc”, connect to the target and load the .out file.

1. Add the appropriate watch window variables by calling the script “\solutions\common\sensorless_foc\debug\is10_rs_recalc.js”.
2. Enable the real-time debugger.
3. Click the run button.
4. Enable continuous refresh on the watch window.
5. Set both “motorVars.flagEnableSys” and “motorVars.flagEnableRunAndIdentify” flags to “1” to run the motor.
6. Set “motorVars.flagEnableRsOnLine” flag to “1” to enable Rs online.

Once the motor starts running, the current will start looking as shown in **Figure 66** if there is a low frequency component to it. This means that the Rs Online is running and Id reference is being modified

by the algorithm. The following oscilloscope plot was taken while Rs Online was running. There is a command of 0.5 A max amplitude for Rs Online in this case:

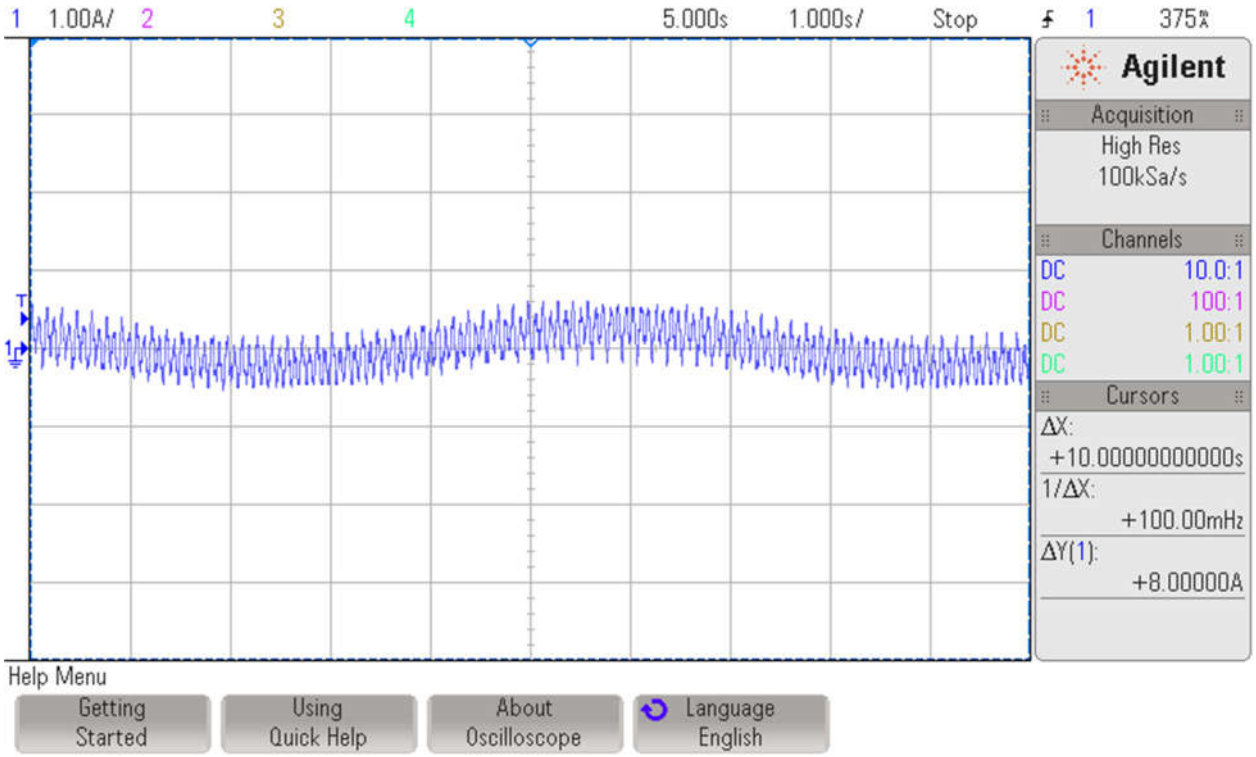


Figure 66: Phase current while running Rs Online with a command of 0.5 A

Now notice both Rs Online and Rs as below.

motorVars.flagEnableRsOnLine	unsigned char	1 '\x01' (Decimal)	0x00000447@Data
motorVars.RsOnLineCurrent_A	float	0.5	0x0000046E@Data
motorVars.RsOnLine_Ohm	float	0.405190706	0x00000476@Data
motorVars.Rs_Ohm	float	0.404209375	0x00000474@Data

7. Now change the maximum amplitude used for Rs Online by changing the following variable to 1.0 (A), the current will start looking as shown in **Figure 67**.

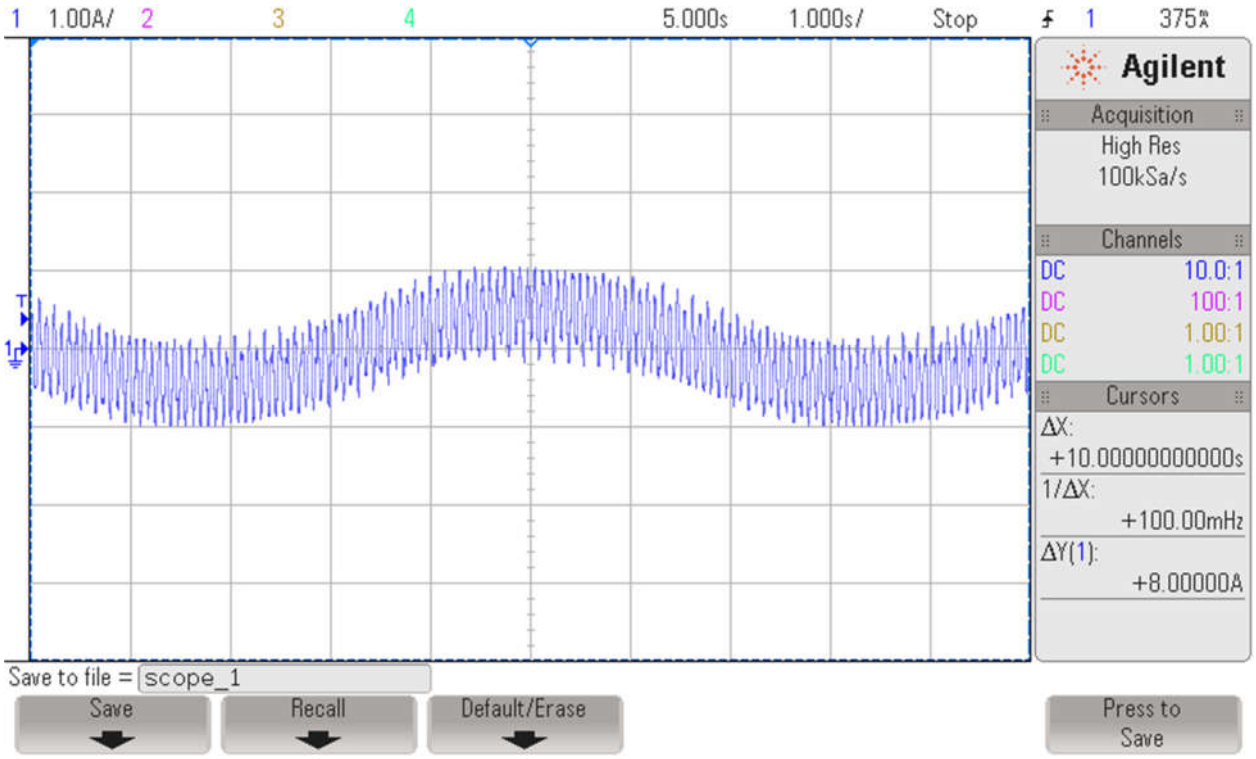


Figure 67: Phase current while running Rs Online with a command of 1.0 A

Now notice both Rs Online and Rs are the same and stable:

motorVars.flagEnableRsOnLine	unsigned char	1 '\x01' (Decimal)	0x00000447@Data
motorVars.RsOnLineCurrent_A	float	1.0	0x0000046E@Data
motorVars.RsOnLine_Ohm	float	0.405000448	0x00000476@Data
motorVars.Rs_Ohm	float	0.40506053	0x00000474@Data

8. When done experimenting, set “motorVars.flagEnableRunAndIdentify” flag to “0” to turn motor off.

Conclusion

In many applications, the motor is subject to overheating conditions. This causes the stator resistance in a motor to change. We have run the Rs Online feature of InstaSPIN, where the motor stator resistance is updated while the motor is running and will update resistance even if resistance goes up or down due to temperature changes.

is11_dual_motor – Dual Motor Sensorless Velocity Control

Abstract

The lab covers how to use InstaSPIN-FOC to control two motors based on one MCU.

Introduction

In lab “is11_dual_motor”, sensorless InstaSPIN-FOC is implemented to control two motors independently by one MCU.

Prerequisites

It assumes knowledge of project “is07_speed_control”

Objectives Learned

- How to use InstaSPIN-FOC to control two motors based on one single MCU.
- How to run two motors synchronously or independently.

Detailed Description

Lab “is11_dual_motor” implements a dual motor control by leveraging two instances of InstaSPIN-FOC on a single C2000 MCU. The same control techniques used to control a single motor are now used for both motors. Lab “is11_dual_motor” continues to use a single interrupt subroutine **`mainISR()`** to run time critical InstaSPIN-FOC code for both motors.

In “is11_dual_motor.c”, initialize parameters for each motor, set up common hardware for both motors, create a new hardware abstraction layer object and set up the drive parameters for each motor, and definine all control objects for each motor.

Phase control is implemented to improve the current sampling for both motor feedback signals. A phase relationship of 90^{o} is established between the respective PWM modules, with motor_1 set as a master module and motor_2 as slave module. Configuration of the PWM modules is performed in HAL_setupPWMs() which is defined in ‘hal_dm.c.’

An example PWM output waveform for dual motor control is shown in **Figure 68**, and the motor current waveforms are shown in **Figure 69**.

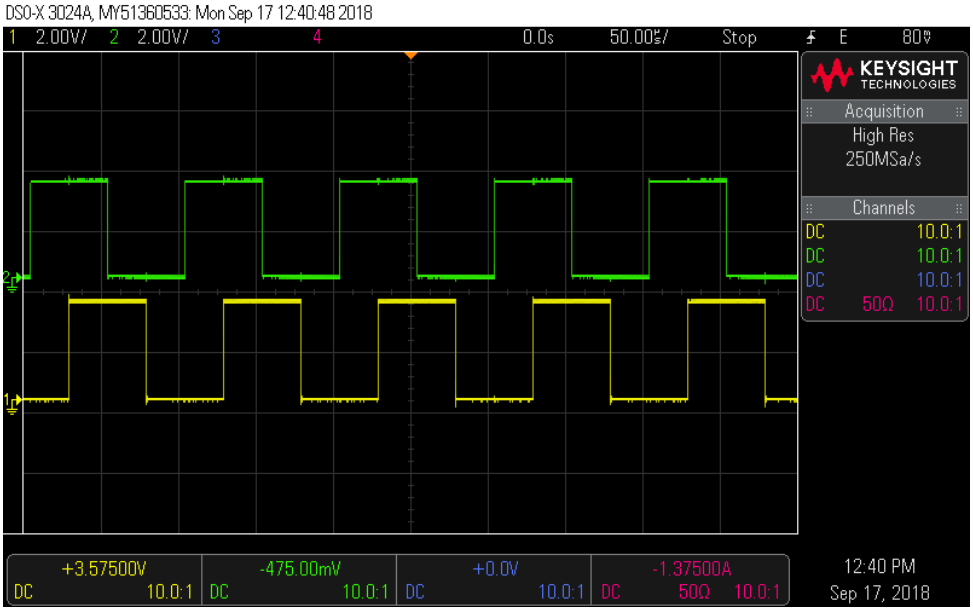


Figure 68: Dual motor PWM output with phase difference of 90º

Where
ch1->PWM_UH for motor_1
ch2-> PWM_UH for motor_2

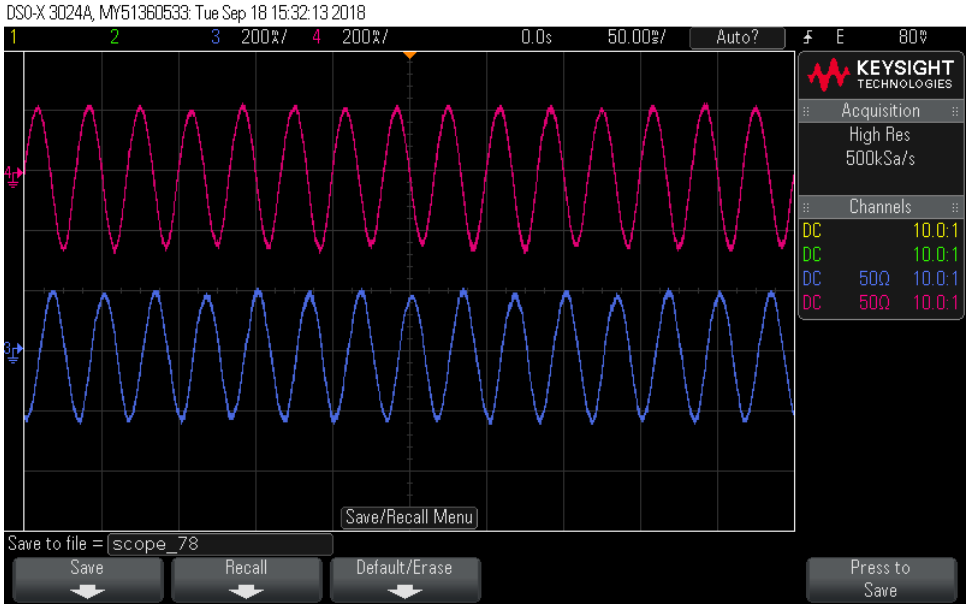


Figure 69: Dual motor phase current waveforms with phase offset of 90º

Where
ch3->Phase U current of motor_1
ch4-> Phase U current of motor_2

CPU Usage Time Calculation

The **`mainISR()`** is time critical. When integrating your code into this ISR, it is important to verify that these two ISRs run in real-time.

The CPU_Time module allows measuring CPU time used by the ISR. Depending on this information, users might want to free up some space to add other functions, or might want to increase the ISR frequency to have a tighter current control. Adding the CPU_Time module to lab11 is described below:

Step 1.

Add “\libraries\utilities\cpu_time\source\cpu_time.c” to the project, as well as include "cpu_time.h" in “labs_dm.h”.

Step 2.

Declare the object and handle for the CPU usage measurement as follows:

```
// define CPU time for performance test
CPU_TIME_Obj     cpuTime;
CPU_TIME_Handle  cpuTimeHandle;
```

Step 3.
Initialize the CPU_Time module and set the parameters to configure the timers and initialize the global variables.

```
   // initialize the CPU usage module
   cpuTimeHandle = CPU_TIME_init(&cpuTime,sizeof(cpuTime));
   CPU_TIME_setParams(cpuTimeHandle,
               EPWM_getTimeBasePeriod(halMtrHandle[HAL_MTR_1]->pwmHandle[0]));
```

Step 4.

In order to measure the ISR execution cycles that it takes to execute the ISR with this module, add the following lines of code at the beginning of the ISR

```
   // read the timer 1 value and update the CPU usage module
   uint32_t timer1Cnt = HAL_readTimerCnt(halHandle, 2);
   CPU_TIME_updateCnts(cpuTimeHandle,timer1Cnt);
```

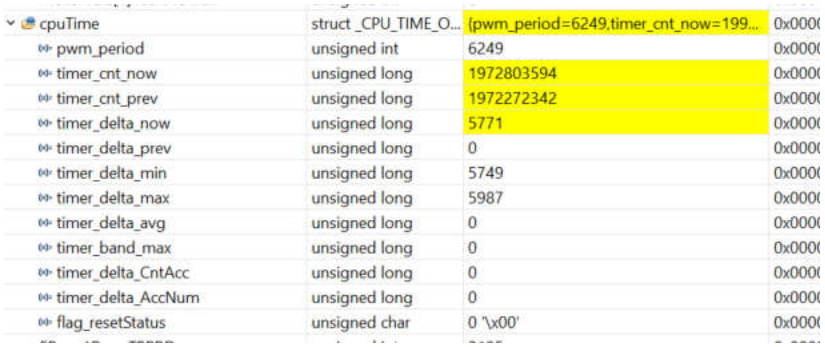
Step 5.

Add the following at the end of the ISR to get the total number of cycles:

```
   // read the timer 2 value and update the CPU usage module
   timer1Cnt = HAL_readTimerCnt(halHandle, 2);
   CPU_TIME_run(cpuTimeHandle, timer1Cnt);
```

The **CPU_TIME_run()** function calculates a maximum, minimum and average ISR CPU usage time. If users want to reset these values and restart the calculation, set the "cpuTime.flag_resetStatus" flag to 1.

The maximum and minimum CPU usage time can be monitored from the watch window as shown in **Figure 70**. Ensure the value of "cpuTime.timer_delta_max" is less than ["cpuTime.pwm_period" – 100] to avoid ISR time overflow.



cpuTime	struct _CPU_TIME_O...	{pwm_period=6249,timer_cnt_now=199...	0x000(
pwm_period	unsigned int	6249	0x000(
timer_cnt_now	unsigned long	1972803594	0x000(
timer_cnt_prev	unsigned long	1972272342	0x000(
timer_delta_now	unsigned long	5771	0x000(
timer_delta_prev	unsigned long	0	0x000(
timer_delta_min	unsigned long	5749	0x000(
timer_delta_max	unsigned long	5987	0x000(
timer_delta_avg	unsigned long	0	0x000(
timer_band_max	unsigned long	0	0x000(
timer_delta_CntAcc	unsigned long	0	0x000(
timer_delta_AccNum	unsigned long	0	0x000(
flag_resetStatus	unsigned char	0 '\x00'	0x000(

Figure 70: Monitor CPU Cycles of mainISR() for Dual Motor Control

Project Files

Compared to lab "is07_speed_control" replaces some of the original files with new files

is11_dual_motor		
	hal_dm.c	Contains the various functions related to the HAL object
	user_dm.c	Contains the function for setting initialization data to the HAL, and EST modules

Table 22: New files that must be included in the project for dual motors control

Includes

A description of the included files for lab "is11_dual_motor", is shown in the below tables. Note that "labs_dm.h" is common across the project so there will be more included files than needed for this lab.

Labs_dm.h	Header file containing all included files used in is11_dual_motor.c, brief Defines the structures, global initialization, and functions used in lab	
	modules	
	math.h	Common math conversions, defines, and shifts
	est.h	Contains the public interface to the estimator (EST) module routines
	platforms	
	hal_dm.h	Contains public interface to various functions related to the HAL object
	hal_obj_dm.h	Defines the structures for the HAL object
	user_com.h	Contains the public interface for user initialization data for user parameters
	user_m1.h	Contains the motor1 initialization parameters for the HAL, and EST modules

	user_m2.h	Contains the motor2 initialization parameters for the HAL, and EST modules

Table 23: Important header files needed for the dual motors control

Global Object and Variable Declarations

Global objects and declarations listed in below table are the objects that are absolutely needed for the drive setup. Other object and variable declarations are used for display or information for the purpose of this lab.

globals			
	EST_Handle	estHandle[2]	The handle to an estimator object. The controller object implements all of the FOC algorithms and calls the FAST observer functions.
	MOTOR_Vars_t	motorVars[2]	Not needed for the implementation of InstaSPIN, but in the example project, this structure contains all of the flags and variables used to turn on and adjust InstaSPIN.
	HAL_PWMData_t	pwmData[2]	The PWM voltage values for the three phases.
	HAL_ADCData_t	adcData[2]	The voltage and current ADC values used by the CTRL controller and the FAST estimator.
	HAL_Handle_mtr	halHandleMtr[2]	The handle for the hardware abstraction layer specific to the motor board.
	HAL_Obj_mtr	halMtr[2]	The hardware abstraction layer object specific to the motor board.
	HAL_Handle	halHandle	The handle for the hardware abstraction layer for common CPU setup
	HAL_Obj	hal	The hardware abstraction layer object
	SYSTEM_Vars_t	systemVars	This structure contains all of the flags and variables to control dual motors synchronously

Table 24: Global object and variable declarations important for the setup

Lab Procedure

Step 1.

Use one LAUNCHXL-F280049C and two sets of BOOSTXL-DRV8320RS to set up the lab kit. Connect the motor and power supply to kit. Import the CCS Project located here: "\solutions\boostxl_drv8320rs\f28004x\ccs\sensorless_foc\is11_dual_motor.projectspec"

Step 2.

In "user_dm.h," "user_com.h," "user_m1.h," and "user_m2.h," ensure motor parameters are known and correctly set. Lab "is11_dual_motor" only works with two PMSM motors.

Step 3.

In Code Composer Studio, build lab "is11_dual_motor", connect to the target and load the .out file:

- Open the command file "\solutions\common\sensorless_foc\debug\is11_dual_motor.js" via the 'Scripting Console'
  - o This will add the variables that we will be using for this project into the watch window

- Enable the real-time debugger
  - o This will let the debugger update the watch window variables
- Click the 'Run' button.
  - o This will run the program on the microcontroller
- Enable continuous refresh on the watch window.
  - o This will continuously update the variables in the watch window

Step 4.

- To start the project, set the variable "systemVars.flagEnableSystem" equal to "1". In this lab, the "motorVars[0].flagEnableSys" and "motorVars[1].flagEnableSys" variables will be set to "1" automatically to complete the current and voltage offset calibration.

To run the two motors synchronously using the same flag and speed reference:

- To enable synchronous control, set the variable "systemVars.flagEnableSynControl" equal to 1.
- To start the current and speed loop controller, set the variable "systemVars.flagEnableRun" equal to 1.
- The acceleration can be modified by adjusting the value in "systemVars. accelerationMaxSet_Hzps".
- Set a reference speed to "systemVars.speedSet_krpm" in order to run the motor at a target speed.

To run the dual motors independently using different flags and speed references:

- To run the first motor (motor_1):
  - o To disable synchronous control, set the variable "systemVars.flagEnableSynControl" to "0".
  - o To enable the PWM output to motor_1, set the variable "motorVars[0].flagEnableRunAndIdentify" to "1".
  - o The acceleration can be modified by adjusting the value in "motorVars[0].accelerationMax_Hzps".
  - o Set a reference speed to "motorVars[0].speedRef_Hz" in order to run the motor_1 at a target speed.

- To run the second motor (motor_2):
  - o To disable synchronous control, set the variable "systemVars.flagEnableSynControl" to 0.
  - o To enable the PWM output to motor_2, set the variable "motorVars[1].flagEnableRunAndIdentify" to "1".
  - o The acceleration can be modified by adjusting the value in "motorVars[1].accelerationMax_Hzps".
  - o Set a reference speed to "motorVars[1]. speedRef_Hz" in order to run the motor_2 at a target speed.

Step 5.

When finished experimenting, stop the motor:

- Set the variable “systemVars.flagEnableRun” to “0” if using synchronous control, or set “motorVars[0].flagEnableRunAndIdentify” or “motorVars[1].flagEnableRunAndIdentify” to “0” if not using synchronous control to disable the PWM output to the motors.
- Turn off real-time control and stop the debugger.
- Turn off power supply of drive kit.

Conclusion

Lab “is11_dual_motor” showed how to use InstaSPIN-FOC in a dual motor system, demonstrating a quickly deployable setup for sensorlessly controlling two motors using a single C2000 device.

is12_variable_pwm_frequency – Online Variable Switching Frequencies

Abstract

In a motor drive application, the thermal consumption and efficiency of the inverter must be considered in the design. The online variable switching frequency can reduce power module loss which in turn drives the motor more efficiently. The lab shows how to use the online variable switching frequencies function in instaSPIN-FOC project.

Introduction

Lab "is12_variable_pwm_frequency" provides an example of the online variable switching frequency feature of InstaSPIN-FOC. Online variable switching frequency is a feature that allows the drive to change a FET's switching frequency to optimize the motor drive efficiency without changing any additional control parameters. Changing the switching frequencies can help minimize the heat generation and loss during low speed operation, reduce the current ripple, and decrease the audible noise in the high speed region by using a higher switching frequency.

Prerequisites

It assumes knowledge of is07_speed_control lab project.

Objectives Learned

The objective of this lab is to learn how to implement online variable switching frequencies in a lab project.

Detailed Description

The total loss of the power module includes Insulated Gate Bipolar Transistor (IGBT) loss and Free Wheeling Diode (FWD) loss. The IGBT loss includes steady state conduction loss and switching loss as shown in **Equation** , and the FWD loss includes steady state conduction loss and reverse recovery loss as shown in **Equation 19**.

$$P_{loss(IGBT)} = P_{cond(IGBT)} + P_{sw(IGBT)} \quad \text{Equation 14}$$

$$P_{loss(FWD)} = P_{cond(FWD)} + P_{rec(FWD)} \quad \text{Equation 15}$$

The conduction losses of the IGBT and freewheeling diode can be calculated as the integration over the conducting period of the current flowing through the collector/anode multiplied by the saturation voltage. In contrast, switching losses arise as a result of energy losses that occur during the transition and switching events. The conduction losses mainly depend on the duty cycle, load current and junction temperature; whereas switching losses depend on the load current, dc link voltage, junction temperature, and switching frequency, as shown in **Equation 16** and **Equation 17**. The IGBT switching loss and FWD reverse recovery loss increase with the switching frequency, while the total loss increases as the current increases. If the switching frequency is higher, then the losses will be higher, leading to more heat generation which requires a larger power module and a cooling system. To solve this problem, an online variable switching frequency scheme is used to reduce power module inefficiencies that arise from switching losses.

$$P_{sw(IGBT)} = (E_{on} + E_{off}) \times f_{sw} \quad \text{Equation 16}$$

$$P_{rec(FWD)} = E_{rec} \times f_{sw}$$ Equation 17

Files Needed for Online Variable Switching Frequencies

Add vsf.c into the project, and add the vsf.h header file include directory in lab.h

1. **user.h**

   Define `USER_EST_FREQ_Hz`, `USER_CTRL_FREQ_Hz`, `USER_CTRL_PERIOD_sec`, and `USER_TRAJ_FREQ_Hz`

2. **user.c**

1). Change the USER_ISR_FREQ_Hz in all xxxWaitTime[xxx] in user.c to the USER_EST_FREQ_Hz as shown in the example code below.

```
    pUserParams->estWaitTime[EST_State_RoverL] =
        (int_least32_t)(5.0 * USER_EST_FREQ_Hz);
```

2). Change the estimator, controller and trajectory frequencies and period as below:

```
    pUserParams->estFreq_Hz = USER_EST_FREQ_Hz;
    pUserParams->ctrlFreq_Hz = USER_CTRL_FREQ_Hz;
    pUserParams->trajFreq_Hz = USER_TRAJ_FREQ_Hz;

    pUserParams->pwmPeriod_usec = USER_PWM_PERIOD_usec;
    pUserParams->ctrlPeriod_sec = USER_CTRL_PERIOD_sec;
```

3. **hal.c**

Add below code in **HAL_enableADCInts()** to enable a dedicated timer for the FAST estimator and controller. In this example project, CPU timer 0 interrupt is used.

```
#ifdef _VSF_EN_
    // enable the cpu timer 0 interrupt for FAST estimator
    Interrupt_enable(INT_TIMER0);
#endif  // _VSF_EN_
```

Add the function **HAL_setupEstTimer()** and call it in **HAL_setParams()** to configure CPU timer 0 to trigger the FAST estimator ISR. Other unused timers can be assigned to trigger the ISR per the system requirement, such as: EPWM4, EPWM5, CPU Timer 1, etc.

```
#ifdef _VSF_EN_
  // setup the timer for estimator
  HAL_setupEstTimer(handle, USER_SYSTEM_FREQ_MHz, USER_EST_FREQ_Hz);
#endif  // _VSF_EN_
```

4. **hal.h**

Initialize the interrupt vector table and point the FAST ISR to the chosen timer interrupt.

```
#ifdef _VSF_EN_
    Interrupt_register(INT_TIMER0, &estISR);
#endif  // _VSF_EN_
```

Variables Needed for Online Variable Switching Frequencies

Add a new handle and object for the online variable switching frequency function in the is12_variable_pwm_frequency source file.

```
VSF_Obj    vsfVars;    //!< the handle for the variable PWM frequency
VSF_Handle vsfHandle;  //!< the variable PWM frequency object

uint16_t pwmFreqSet_Hz = (uint16_t)(USER_PWM_FREQ_kHz * 1000.0);
```

Initialize the Online Variable Switching Frequencies Module

Initialize and setup the parameters for the online variable switching frequency function. The constants are defined in vsf.h and can be changed based on the specification of real system.

```
    vsfHandle = VSF_init(&vsfVars, sizeof(vsfVars));
    VSF_initParameters(vsfHandle, &userParams);

    VSF_setupPwmMode(halHandle);
    VSF_setState(vsfHandle, VSF_IDLE);

    VSF_setFreqMax(vsfHandle, NUM_VSF_MAX_FREQ_HZ);
    VSF_setFreqMin(vsfHandle, NUM_VSF_MIN_FREQ_HZ);

    VSF_setFreqDelta(vsfHandle, NUM_VSF_DELTA);
    VSF_setWaitTime(vsfHandle, NUM_VSF_WAIT_TIME);
```

UpdateOnline Variable Switching Frequencies in the Background Loop

Call the below two functions in the background loop to set the target PWM switching frequency as required.

```
        VSF_setFreq(vsfHandle, pwmFreqSet_Hz);
        VSF_computeFreqParams(vsfHandle);
```

Update the PWM Period in the ISR

Call the below function to set the period register of the ePWM module to update the switching frequency.

```
    VSF_setPeriod(halHandle,vsfHandle);
```

Lab Procedure

Step 1.

Set up the lab kit by connecting the motor and power supply.

Step 2.

In user.h, make sure the motor parameters are known and correctly set.

Step 3.

In Code Composer, build project “is12_variable_pwm_frequency,” connect to the target and load the .out file.

- Open the command file “solutions\common\sensorless_foc\debug\is12_variable_pwm_frequency.js” via the Scripting Console
  - This will add the variables that we will be using for this project into the watch window
- Enable the real-time debugger
  - This will let the debugger update the watch window variables
- Click the run button.
  - This will run the program on the microcontroller
- Enable continuous refresh on the watch window.
  - This will continuously update the variables in the watch window

Step 4.

Run the motor

- To start the project, set the variable “motorVars.flagEnableSys” equal to 1.
- To turn on the PWMs to the motor, set the variable “motorVars.flagEnableRunAndIdentify” equal to 1.
- The acceleration can be modified by adjusting the value in “motorVars.accelerationMax_Hzps”.
- Set a reference speed to “motorVars.speedRef_Hz” in order to run the motor at a target speed.

Step 5.

To run the motor with the variable online switching frequency function

- Set pwmFreqSet_Hz to the target switching frequency; the unit is Hz.
- The NUM_VSF_DELTA and NUM_VSF_WAIT_TIME in vsf.h are used to set the ramp speed of changing switching frequency.

Step 6.

When finished experimenting, stop the motor

- Set the variable “motorVars.flagEnableRunAndIdentify” to 0 to turn off the PWMs to the motor.
- Turn off real-time control and stop the debugger.
- Turn off power supply of drive kit.

Conclusion

This lab adds an online variable switching frequency function to the InstaSPIN-FOC feature set. This feature allows changing the switching frequency at runtime to improve the inverter efficiency by reducing the power loss.

is13_fwc_mtpa – Field Weakening and Maximum Torque per Ampere Control

Abstract

This lab shows an example on how to implement Field Weakening (FW) and Maximum Torque Per Ampere (MTPA) techniques to control three-phase Interior Permanent Magnet Synchronous Motors (IPM) over a wide speed range in an InstaSPIN-FOC project. During this example, the operation mode will automatically change from constant torque region with MTPA control to constant power region with FW control based on the speed command and the input DC-link voltage.

Introduction

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

Lab “is13_fwc_mtpa” builds on top of the “is10_rs_recalc” lab project, adding Field Weakening (FW) and Maximum Torque per Ampere (MTPA) control modules. These two modules can be used simultaneously or can be used separately based on the motor control requirement.

Prerequisites

This lab assumes that the motor’s parameters are known and the “is10_rs_recalc” lab project is understood.

Objectives Learned

- Learn where the FW and MTPA modules are added into the lab “is13_fwc_mtpa”.
- Tune the FW module to perform field weakening control to extend the speed range.
- Tune the MTPA module to perform maximum torque per ampere control to produce a greater amount of electromagnetic torque.

Detailed Description

The voltage equations of the mathematical model of an IPMSM can be described in d-q coordinates as shown in **Equation 18** and **Equation 19**.

$$v_d = L_d\frac{di_d}{dt} + R_s i_d - p\omega_m L_q i_q \qquad \text{Equation 18}$$

$$v_q = L_q\frac{di_q}{dt} + R_s i_q + p\omega_m L_d i_d + p\omega_m \psi_m$$ Equation 19

Where:
v_d, v_q are the d- and q- axis stator voltages;
i_d, i_q are the d- and q- axis stator currents;
R_s, are the stator per phase resistance;
L_d, L_q are the d- and q- axis stator inductances;
ψ_m, is the permanent magnetic flux linkage;
ω_m, is the rotor angle speed in *rad/s*;
p, is the number of the pole pairs;
$\frac{d}{dt}$, is the differential operator;

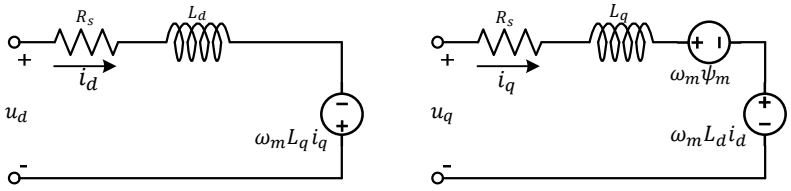


Figure 73 Equivalent Circuit of an IPM Synchronous Motor

The dynamic equivalent circuit of an IPM synchronous motor is shown in **Figure 73**. The total torque T_e generated by the IPMSM can be expressed as **Equation 20** that the produced torque is composed of two distinct terms. The first term corresponds to the mutual reaction torque occurring between torque current i_q and the permanent magnet ψ_m, while the second term corresponds to the reluctance torque due to the differences in d-axis and q-axis inductance.

$$T_e = \frac{3}{2}p\left[\psi_m i_q + (L_d - L_q) i_d i_q\right]$$ Equation 20

Where: T_e, are the electromagnetic torque;

In most applications, IPMSM drives have speed and torque constraints, mainly due to inverter or motor rating currents and available DC link voltage limitations respectively. These constraints can be expressed with the mathematical equations **Equation 21** and **Equation 22**.

$$I_a = \sqrt{i_d^2 + i_q^2} \le I_{max}$$ Equation 21

$$V_a = \sqrt{v_d^2 + v_q^2} \le V_{max}$$ Equation 22

Where V_{max} and I_{max} are the maximum allowable voltage and current of the inverter or motor. In a two-level three-phase Voltage Source Inverter (VSI) fed machine, the maximum achievable phase voltage is limited by the DC link voltage and the PWM strategy. The maximum voltage is limited to the value as shown in**Equation 23** if Space Vector Modulation (SVPWM) is adopted:

$$\sqrt{v_d^2 + v_q^2} \le v_{max} = \frac{v_{dc}}{\sqrt{3}}$$ Equation 23

Usually the resistance R_s is negligible at high speed operation and the derivate of the currents is zero in steady state, thus **Equation 24** is obtained as below.

$$\sqrt{L_d^2(i_d + \frac{\psi_{pm}}{L_d})^2 + L_q^2 L_q^2} \le \frac{V_{max}}{\omega_m}$$ Equation 24

The current limitation of **Equation 21** produces a circle of radius I_{max} in the stator currents' d-q plane, and the voltage limitation of **Equation 24** produces an ellipse whose radius decreases as speed increases. The resultant d-q plane current vector must be controlled to obey the current and voltage constraintssimultaneously. According to these constraints, three operation regions for the IPMSM can be distinguished as shown in **Figure 74**.

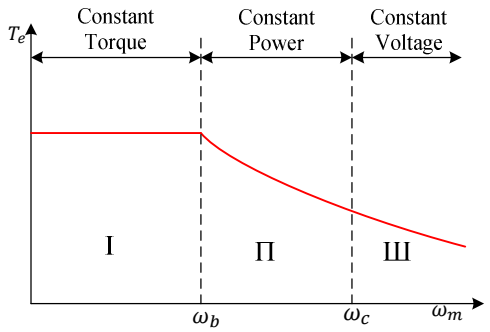


Figure 74 IPMSM control operation regions

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

In the constant torque region, according to **Equation 20**, the total torque of an IPMSM includes the electromagnetic torque from the magnet flux linkage and the reluctance torque from the saliency between L_d and L_q. The electromagnetic torque is proportional to the q-axis current, and the reluctance torque is proportional to the multiplication of the d-axis current, the q-axis current, and the difference between L_d, and L_q.

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

$$I_s = \sqrt{i_d^2 + i_q^2} \qquad \text{Equation 25}$$

$$I_d = I_s \cos\beta \qquad \text{Equation 26}$$

$$I_q = I_s \sin\beta \qquad \text{Equation 27}$$

Where β is the stator current angle in the synchronous (d-q) reference frame.
Equation 20 can be expressed as **Equation 28** where I_s substituted for i_d and i_q.

$$T_e = \frac{3}{2} p I_s \sin\beta \left[\psi_m + (L_d - L_q) I_s \cos\beta \right] \qquad \text{Equation 28}$$

Equation 28 shows that motor torque depends on the angle of the stator current vector; as such, the maximum efficiency point can be calculated when the motor torque differential is equal to zero. The MTPA point can be found when this differential, $\frac{dT_e}{d\beta}$, is zero, as given in **Equation 29**:

$$\frac{dT_e}{d\beta} = \frac{3}{2}p\left[\psi_m I_s \cos\beta + (L_d - L_q) I_s^2 \cos 2\beta\right] = 0 \quad \text{Equation 29}$$

Following, the current angle of the MTPA control can be derived as in **Equation 30**:

$$\beta_{mtpa} = \cos^{-1}\frac{-\psi_m + \sqrt{\psi_m^2 + 8 * \left(L_d - L_q\right)^2 * I_s^2}}{4 * \left(L_d - L_q\right) * I_s} \quad \text{Equation 30}$$

Thus, the effective d-axis and q-axis reference currents can be expressed by **Equation 31** and **Equation 32** using the current angle of the MTPA control.

$$I_d = I_s * \cos\beta_{mtpa} \quad \text{Equation 31}$$

$$I_q = I_s * \sin\beta_{mtpa} \quad \text{Equation 32}$$

However, as shown in **Equation 30**, the angle of the MTPA control is related to d-axis and q-axis inductance. This means that the variation of inductance will impede the ability to find the optimal MTPA point. To improve the efficiency of a motor drive, the d-axis and q-axis inductances should be estimated online, but the parameters L_d and L_q are not easily measured online and are influenced by saturation effects. A robust Look-Up Table (LUT)-ensures controllability under electrical parameter variations. Usually, to simplify the mathematical model, the coupling effect between d-axis and q-axis inductance can be neglected. Thus, L_d changes with i_d only, and L_q changes with i_q only. Consequently, d- and q-axis inductances can be modeled as a function of their d-q currents respectively, as shown in **Equation 33** and **Equation 34**:

$$L_d = f_1(i_d, i_q) = f_1(i_d) \quad \text{Equation 33}$$

$$L_q = f_2(i_q, i_d) = f_2(i_q) \quad \text{Equation 34}$$

The ISR calculation burden can be reduced by simplifying **Equation 30**. Using the motor-parameter-based constant K_{mtpa}, **Equation 30** is expressed instead as **Equation 36**, where K_{mtpa} is computed in the background loop using the updated L_d and L_q.

$$K_{mtpa} = \frac{\psi_m}{4 * \left(L_q - L_d\right)} = 0.25 * \frac{\psi_m}{\left(L_q - L_d\right)} \quad \text{Equation 35}$$

$$\beta_{mtpa} = cos^{-1}\left(K_{mtpa}/I_s - \sqrt{\left(K_{mtpa}/I_s\right)^2 + 0.5}\right) \quad \text{Equation 36}$$

A second intermediate variable G_{mtpa}, described in **Equation 37**, is defined to further simplify the calculation. Using G_{mtpa}, **Equation 36** can be expressed as equation **Equation 38.** The calculation of both equations is performed in the ISR to achieve a real current angle.

$$G_{mtpa} = K_{mtpa}/I_s \quad \text{Equation 37}$$

$$\beta_{mtpa} = cos^{-1}\left(G_{mtpa} - \sqrt{G_{mtpa}^2 + 0.5}\right) \quad \text{Equation 38}$$

In all cases, the magnetic flux can be weakened to extend the achievable speed range by acting on the direct axis current i_d. As a consequence of entering this constant power operating region, field weakening control is chosen instead of the MTPA control used for the constant torque region.

Since the maximum inverter voltage is limited, PMSM motors cannot operate in such speed regions where the back-electromotive force, almost proportional to the permanent magnet field and motor speed, is higher than the maximum output voltage of the inverter. The direct control of magnet flux is not an option in PM motors. However, the air gap flux can be weakened by the demagnetizing effect due to the d-axis armature reaction by adding a negative i_d. Considering the voltage and current constraints, the armature current I_a and the terminal voltage V_a are limited as **Equation 21** and **Equation 22**. The inverter input voltage (DC-Link voltage) variation limits the maximum output of the motor. Furthermore, the maximum fundamental motor voltage also depends on the PWM method used. In **Equation 24**, the IPMSM has two factors: one is a permanent magnet value and the other is made by inductance and current of flux.

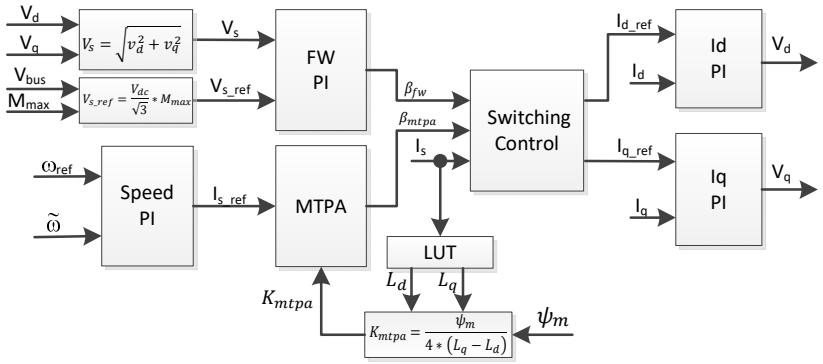


Figure 75 Block diagram of field-weakening and maximum torque per ampere control

Figure 75 shows the typical control structure used to implement field weakening; β_{fw} is the output of the field weakening (FW) PI controller and generates the reference i_d and i_q. Before the voltage magnitude reaches its limit, the input of the PI controller of FW is always positive and therefore the output is always saturated at 0.

A block diagram of **Figure 76** shows the implementation of InstaSPIN-FOC. The block diagram provides an overview of the InstaSPIN-FOC system's functions and variables. The key modules include:

- Trajectory control module for speed
- Clarke forward transform modules for current and voltage
- Park forward and inverse transform module
- Angle and speed estimator module (FAST)
- Proportional integral controller modules for speed and current
- Field weakening (FW) and Maximum torque per ampere (MTPA) module
- Space vector modulation module

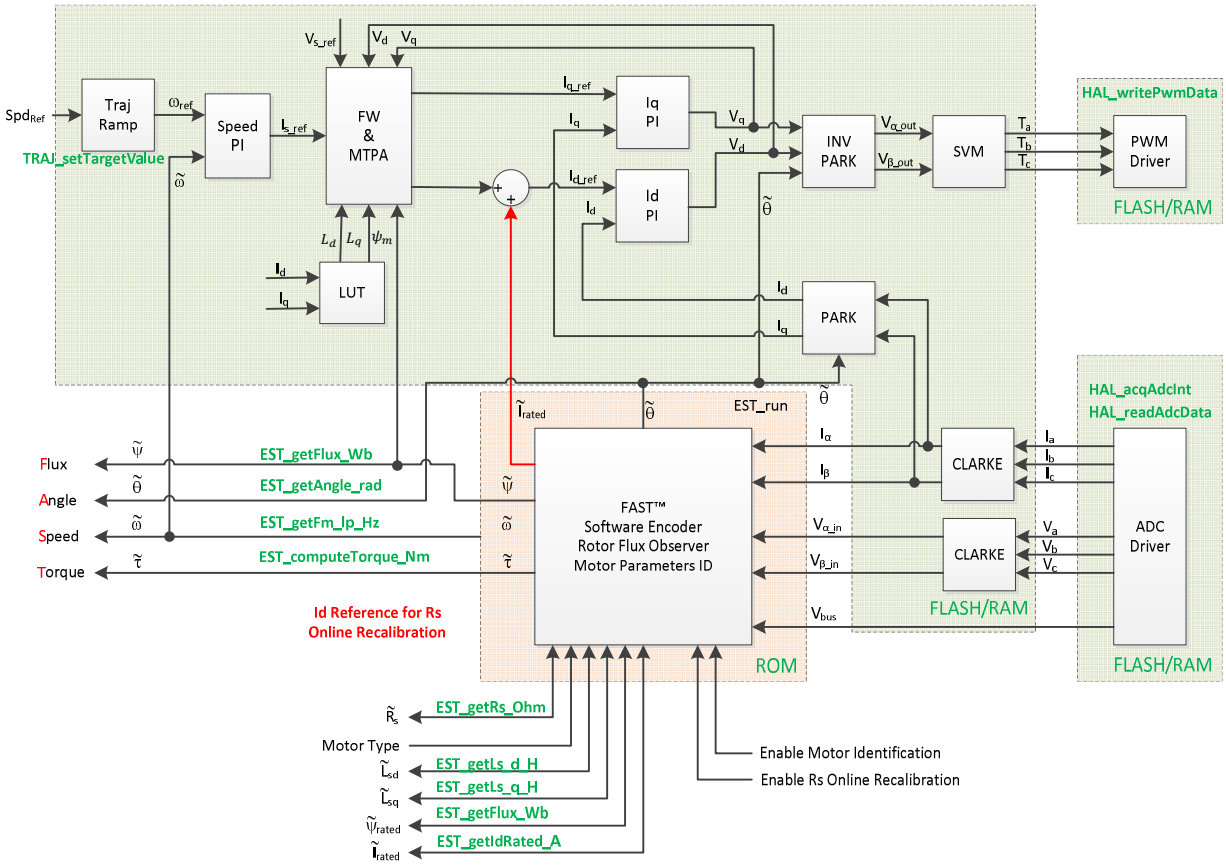


Figure 76 Block Diagram of InstaSPIN-FOC with MTPA and FWC

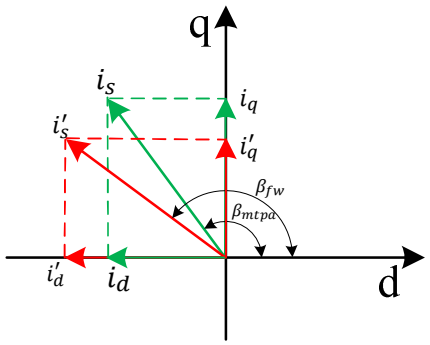


Figure 77 Current phasor diagram of an IPMSM during FW and MTPA

There are two control modules in the InstaSPIN-FOC motor drive system as shown in **Figure 75**: one is MTPA control and the other one is field weakening control. These two modules generate a current angle respectively based on input parameters. The switching control module is used to decide which angle should be applied, and then calculate the reference i_d, and i_q as shown in **Figure 77**. The current angle β is chosen as follows.

- $\beta = \beta_{fw}$ if $\beta_{fw} > \beta_{mtpa}$
- $\beta = \beta_{mtpa}$ if $\beta_{fw} \leq \beta_{mtpa}$

Figure 78 is the flowchart that shows the steps required to run InstaSPIN-FOC with FW and MPTA in the main loop and interrupt.

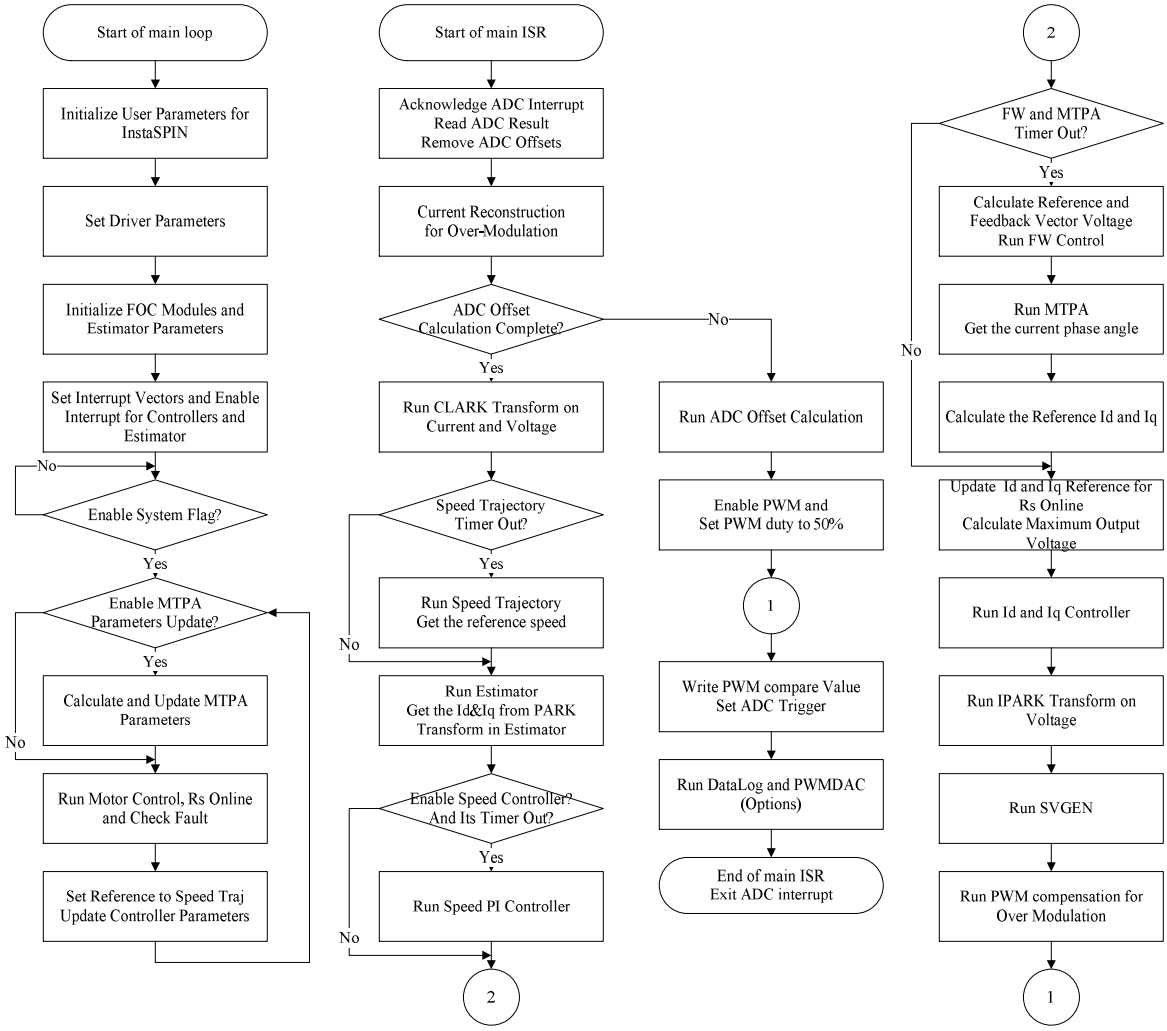


Figure 78 Flowchart for an InstaSPIN-FOC Project with FW and MTPA

Files Needed for FW and MTPA

1). Add fwc.c and mtpa.c into the project and add the include directory of their header files (fwc.h and mtpa.h) in lab.h as demonstrated below.

```
#include "fwc.h"
#include "mtpa.h"
```

2). Add the header file directory to the #include search path by selecting Project->Properties->Build->C2000 Compiler->Include Options as below.

${MCSDK_ROOT}/libraries/control/fwc/include
${MCSDK_ROOT}/libraries/control/mtpa/include

Variables Needed for FW and MTPA modules

Add two new handles and objects for FW and MTPA control in project file.

```
FWC_Handle fwcHandle;      //!< the handle for the Field Weakening Control (FWC)
FWC_Obj    fwc;            //!< the Field Weakening Control (FWC) object

MTPA_Handle mtpaHandle;    //!< the handle for the Maximum torque per ampere (MTPA)
MTPA_Obj    mtpa;          //!< the Maximum torque per ampere (MTPA) object
```

Initializing the FW and MTPA Modules

Initialize and setup parameters for FW and MTPA modules. The constants are defined in user.h that can be changed based on the specification of a real system.

```
    fwcHandle = FWC_init(&fwc, sizeof(fwc));

    FWC_setParams(fwcHandle, USER_FWC_KP, USER_FWC_KI,
                  USER_FWC_MIN_ANGLE_RAD, USER_FWC_MAX_ANGLE_RAD);

    mtpaHandle = MTPA_init(&mtpa, sizeof(mtpa));

    MTPA_computeMotorConstant(mtpaHandle,
                              userParams.motor_Ls_d_H,
                              userParams.motor_Ls_q_H,
                              userParams.motor_ratedFlux_Wb);
```

Updating the control parameters for FW and MTPA modules in the Background Loop

Call FWC and MTPA functions as the codes from line 602 to line 637 during the background loop in “is13_fwc_mtpa.c”, to calculate the control constant of MTPA based on the updated motor parameters. The tables for the calculation of L_d and L_q according to the armature current $\mathrm{I_a}$ are defined in mtpa.h, and the update process is done in the background loop.

Update the FW and MTPA modules in the ISR

Call the FW and MTPA functions as in the code from line 1029 to line 1077 in “is13_fwc_mtpa.c” to calculate current angle, and then compute the reference currents of d-axis and q-axis.

Lab Procedure

Step 1.

Set up hardware kit, connect the motor and apply power supply correctly to the kit.

Step 2.

In user.h, make sure the motor parameters are known and correctly set. In mtpa.h, make sure the tables are set properly for L_d and L_q calculations according to the specification of a motor.

Step 3.

In CCS, build the “is13_fwc_mtpa” project, connect to the target and load the .out file.

- Open the command file “solutions\common\sensorless_foc\debug\is13_fwc_mtpa.js” via the Scripting Console
  - This will add the variables that we will be using for this project into the watch window
- Enable the real-time debugger
  - This will let the debugger update the watch window variables
- Click the run button.
  - This will run the program on the microcontroller
- Enable continuous refresh on the watch window.
  - This will continuously update the variables in the watch window

Step 4.

To run the motor with speed closed loop:

- To start the project, set the variable “motorVars.flagEnableSys” equal to 1.
- To turn on the PWMs to the motor, set the variable “motorVars.flagEnableRunAndIdentify” equal to 1.
- The acceleration can be modified by adjusting the value in “motorVars.accelerationMax_Hzps”.
- Set a reference speed to “motorVars.speedRef_Hz” in order to run the motor at a target speed.

Step 5.

To run the motor with FW and MTPA functions:

- To enable FW, set “motorVars.flagEnableFWC” to “1”.
  - Tune control parameters for FW by setting motorVars.Kp_fwc, motorVars.Ki_fwc, and motorVars.angleMax_fwc_rad.
- To enable MTPA, set “motorVars.flagEnableMTPA” to “1”.

Step 6.

When finished experimenting, to stop the motor:

- Set the variable “motorVars. flagEnableRunAndIdentify” to “0” to turn off the PWMs to the motor.
- Turn off real-time control and stop the debugger.
- Turn off the power supply of drive kit.

Conclusion

This lab adds field weakening (FW) and maximum torque per ampere (MTPA) control features in InstaSPIN-FOC. These features allow improved torque output in the constant torque region with MTPA and to extend the speed range over the base speed with FW with a very smooth transition between MTPA and FW.